ADSP-BF533 Blackfin® Processor Hardware Reference

(Includes ADSP-BF531 and ADSP-BF532 Blackfin Processors)

Revision 3.4, April 2009

Part Number
82-002005-01

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

ANALOG DEVICES

Copyright Information

Printed in the USA.

Disclaimer

Trademark and Service Mark Notice

CONTENTS

OPERATING MODES AND STATES

PROGRAM SEQUENCER

DATA ADDRESS GENERATORS

MEMORY

DYNAMIC POWER MANAGEMENT

SERIAL PORT CONTROLLERS

TIMERS

EXTERNAL BUS INTERFACE UNIT

SYSTEM DESIGN

BLACKFIN PROCESSOR CORE MMR ASSIGNMENTS

SYSTEM MMR ASSIGNMENTS

TEST FEATURES

NUMERIC FORMATS

GLOSSARY

INDEX

PREFACE

Thank you for purchasing and developing systems using Blackfin® processors from Analog Devices, Inc.

Purpose of This Manual

The *ADSP-BF533 Blackfin Processor Hardware Reference* contains information about the DSP architecture for the Blackfin processors. The architectural descriptions cover functional blocks, buses, and ports, including all features and processes that they support.

For programming information, see the *Blackfin Processor Programming Reference*. For timing, electrical, and package specifications, see the *ADSP-BF531/ADSP-BF532/ ADSP-BF533 Embedded Processor Data Sheet*.

Intended Audience

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

Manual Contents

The manual consists of:

- Chapter 1, Introduction
  Provides a high level overview of the processor. Architectural descriptions include functional blocks, buses, and ports, including features and processes they support.
- Chapter 2, Computational Units
  Describes the arithmetic/logic units (ALUs), multiplier/accumulator units (MACs), shifter, and the set of video ALUs. The chapter also discusses data formats, data types, and register files.
- Chapter 3, Operating Modes and States
  Describes the three operating modes of the processor: Emulation mode, Supervisor mode, and User mode. The chapter also describes Idle state and Reset state.
- Chapter 4, Program Sequencer
  Describes the operation of the program sequencer, which controls program flow by providing the address of the next instruction to be executed. The chapter also discusses loops, subroutines, jumps, interrupts, and exceptions.
- Chapter 5, Data Address Generators
  Describes the Data Address Generators (DAGs), addressing modes, how to modify DAG and Pointer registers, memory address alignment, and DAG instructions.
- Chapter 6, Memory
  Describes L1 memories. In particular, details their memory architecture, memory model, memory transaction model, and memory-mapped registers (MMRs). Discusses the instruction, data, and scratchpad memory, which are part of the Blackfin processor core.

- Chapter 7, Chip Bus Hierarchy
  Describes on-chip buses, including how data moves through the system. The chapter also discusses the system memory map, major system components, and the system interconnects.
- Chapter 8, Dynamic Power Management
  Describes system reset and power-up configuration, system clocking and control, and power management.
- Chapter 9, Direct Memory Access
  Describes the peripheral DMA and Memory DMA controllers. The peripheral DMA section discusses direct, block data movements between a peripheral with DMA access and internal or external memory spaces.

  The Memory DMA section discusses memory-to-memory transfer capabilities among the processor memory spaces and the L1, external synchronous, and asynchronous memories.
- Chapter 10, SPI Compatible Port Controllers
  Describes the Serial Peripheral Interface (SPI) port that provides an I/O interface to a variety of SPI compatible peripheral devices.
- Chapter 11, Parallel Peripheral Interface
  Describes the Parallel Peripheral Interface (PPI) of the processor. The PPI is a half-duplex, bidirectional port accommodating up to 16 bits of data and used for digital video and data converter applications.
- Chapter 12, Serial Port Controllers
  Describes the two independent, synchronous Serial Port Controllers (SPORT0 and SPORT1) that provide an I/O interface to a variety of serial peripheral devices.
- Chapter 13, UART Port Controller
  Describes the Universal Asynchronous Receiver/Transmitter (UART) port, which converts data between serial and parallel

formats and includes modem control and interrupt handling hardware. The UART supports the half-duplex IrDA® SIR protocol as a mode-enabled feature.

- Chapter 14, Programmable Flags
  Describes the programmable flags, or general-purpose I/O pins in the processor, including how to configure the pins as inputs and outputs, and how to generate interrupts.
- Chapter 15, Timers
  Describes the three general-purpose timers that can be configured in any of three modes; the core timer that can generate periodic interrupts for a variety of timing functions; and the watchdog timer that can implement software watchdog functions, such as generating events to the Blackfin processor core.
- Chapter 16, Real-Time Clock
  Describes a set of digital watch features of the processor, including time of day, alarm, and stopwatch countdown.
- Chapter 17, External Bus Interface Unit
  Describes the External Bus Interface Unit of the processor. The chapter also discusses the asynchronous memory interface, the SDRAM controller (SDC), related registers, and SDC configuration and commands.
- Chapter 18, System Design
  Describes how to use the processor as part of an overall system. It includes information about interfacing the processor to external memory chips, bus timing and latency numbers, semaphores, and a discussion of the treatment of unused pins.
- Appendix A, Blackfin Processor Core MMR Assignments
  Lists the core memory-mapped registers, their addresses, and cross-references to text.

- Appendix B, System MMR Assignments
  Lists the system memory-mapped registers, their addresses, and cross-references to text.
- Appendix C, Test Features
  Describes test features for the processor; discusses the JTAG standard, boundary-scan architecture, instruction and boundary registers, and public instructions.
- Appendix D, Numeric Formats
  Describes various aspects of the 16-bit data format. The chapter also describes how to implement a block floating-point format in software.
- Appendix G, Glossary
  Contains definitions of terms used in this book, including acronyms.

What's New in This Manual

This is Revision 3.4 of the *ADSP-BF533 Blackfin Processor Hardware Reference*. Changes to this book from Revision 3.3 include corrections of typographic errors and reported document errata.

Technical or Customer Support

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

- Visit the Embedded Processing and DSP products Web site at
  `http://www.analog.com/processors/technical_support`
- E-mail tools questions to
  `processor.tools.support@analog.com`

- E-mail processor questions to
  processor.support@analog.com (World wide support)
  processor.europe@analog.com (Europe support)
  processor.china@analog.com (China support)
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:
  Analog Devices, Inc.
  One Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA

Supported Processors

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

ADSP-BF51x, ADSP-BF52x, ADSP-BF53x, ADSP-BF54x, and ADSP-BF56x

Product Information

Product information can be obtained from the Analog Devices Web site, VisualDSP++ online Help system, and a technical library CD.

Analog Devices Web Site

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

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

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

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

VisualDSP++ Online Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and FLEXnet License Tools software documentation. You can search easily across the entire VisualDSP++ documentation set for any topic of interest.

For easy printing, supplementary Portable Documentation Format (.pdf) files for all manuals are provided on the VisualDSP++ installation CD.

Each documentation file type is described as follows.

File	Description
.chm	Help system files and manuals in Microsoft help format
.htm or .html	Dinkum Abridged C++ library and FLEXnet License Tools software documentation. Viewing and printing the .html files requires a browser, such as Internet Explorer 6.0 (or higher).
.pdf	VisualDSP++ and processor manuals in PDF format. Viewing and printing the .pdf files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

Technical Library CD

The technical library CD contains seminar materials, product highlights, a selection guide, and documentation files of processor manuals, VisualDSP++ software manuals, and hardware tools manuals for the following processor families: Blackfin, SHARC®, TigerSHARC®, ADSP-218x, and ADSP-219x.

To order the technical library CD, go to http://www.analog.com/processors/technical_library, navigate to the manuals page for your processor, click the request CD check mark, and fill out the order form.

Data sheets, which can be downloaded from the Analog Devices Web site, change rapidly, and therefore are not included on the technical library CD. Technical manuals change periodically. Check the Web site for the latest manual revisions and associated documentation errata.

Notation Conventions

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

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
SWRST Software Reset register	Register names appear in UPPERCASE and a special typeface. The descriptive names of registers are in mixed case and regular typeface.
TMR0E, $\overline{\text{RESET}}$	Pin names appear in UPPERCASE and a special typeface. Active low signals appear with an $\overline{\text{OVERBAR}}$.
DRx, I[3:0] $\overline{\text{SMS}}$[3:0]	Register, bit, and pin names in the text may refer to groups of registers or pins: A lowercase x in a register name (DRx) indicates a set of registers (for example, DR2, DR1, and DR0). A colon between numbers within brackets indicates a range of registers or pins (for example, I[3:0] indicates I3, I2, I1, and I0; $\overline{\text{SMS}}$[3:0] indicates $\overline{\text{SMS3}}$, $\overline{\text{SMS2}}$, $\overline{\text{SMS1}}$, and $\overline{\text{SMS0}}$).
0xabcd, b#1111	A 0x prefix indicates hexadecimal; a b# prefix indicates binary.

Example	Description
	**Note:** For correct operation, ... A Note: provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	**Caution:** Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution: identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	**Warning:** Injury to device users may result if ... A Warning: identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

Register Diagram Conventions

Register diagrams use these conventions:

- The descriptive name of the register appears at the top, followed by the short form of the name in parentheses.

Table P-1. Short Form of Register Names

Pattern	Description	Examples
TIMERx_CONFIG	The *x* refers to multiple instances of the peripheral.	TIMER0_CONFIG TIMER1_CONFIG TIMER2_CONFIG
SIC_IARn	The *n* refers to multiple registers within the same peripheral or within the same core component.	SIC_IAR2 ICPLB_DATA15
SPORTx_TCRn	The combination of *x* and *n* indicates multiple instances of the peripheral *and* multiple registers within the same peripheral.	SPORT0_TCR0 SPORT1_TCR1
MDMA_yy_CONFIG	The *yy* represents MemDMA Stream 0 or 1, either Destination or Source.	MDMA_D0_CONFIG MDMA_S0_CONFIG MDMA_D1_CONFIG MDMA_S1_CONFIG

- If the register is read-only (RO), write-1-to-set (W1S), or write-1-to-clear (W1C), this information appears under the name. Read/write is the default and is not noted. Additional descriptive text may follow.
- If any bits in the register do not follow the overall read/write convention, this is noted in the bit description after the bit name.
- If a bit has a short name, the short name appears first in the bit description, followed by the long name in parentheses.

- The MMR assignment appears in hexadecimal to the left of the register or—when multiple addresses are involved—in a table below the register.
- The reset value appears in binary in the individual bits and in hexadecimal to the right of the register.
- Bits marked *x* have an unknown reset value. Consequently, the reset value of registers that contain such bits is undefined or dependent on pin values at reset.
- Shaded bits are reserved.

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

Figure P-1 shows examples of these conventions.

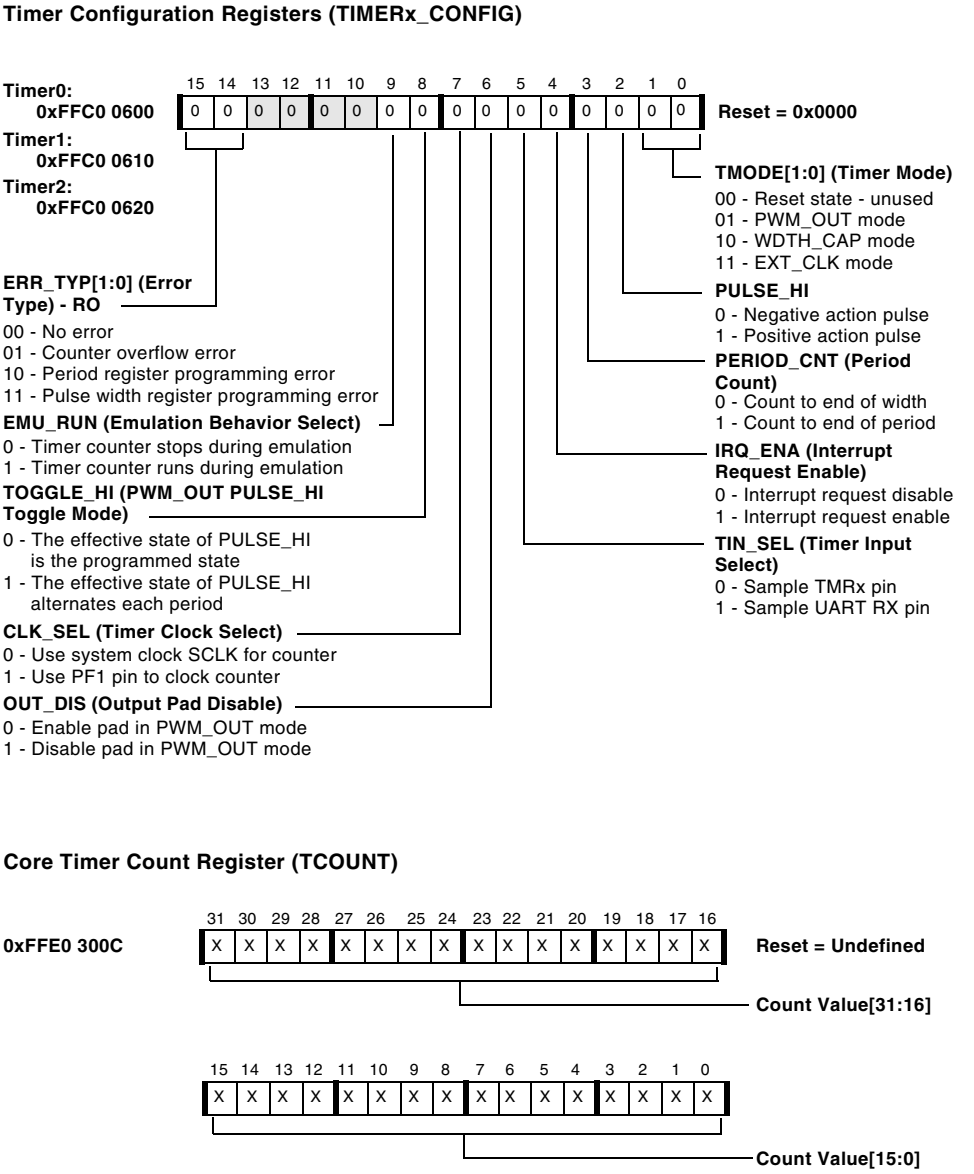


Figure P-1. Register Diagram Examples

1 INTRODUCTION

The ADSP-BF533, ADSP-BF532, and ADSP-BF531 processors are enhanced members of the Blackfin processor family that offer significantly higher performance and lower power than previous Blackfin processors while retaining their ease-of-use and code compatibility benefits. The three new processors are completely pin compatible, differing only in their performance and on-chip memory, mitigating many risks associated with new product development.

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

Blackfin products feature dynamic power management. The ability to vary both the voltage and frequency of operation optimizes the power consumption profile to the specific task.

Peripherals

The processor system peripherals include:

- Parallel Peripheral Interface (PPI)
- Serial Ports (SPORTs)
- Serial Peripheral Interface (SPI)
- General-purpose timers

- Universal Asynchronous Receiver Transmitter (UART)
- Real-Time Clock (RTC)
- Watchdog timer
- General-purpose I/O (programmable flags)

These peripherals are connected to the core via several high bandwidth buses, as shown in Figure 1-1.

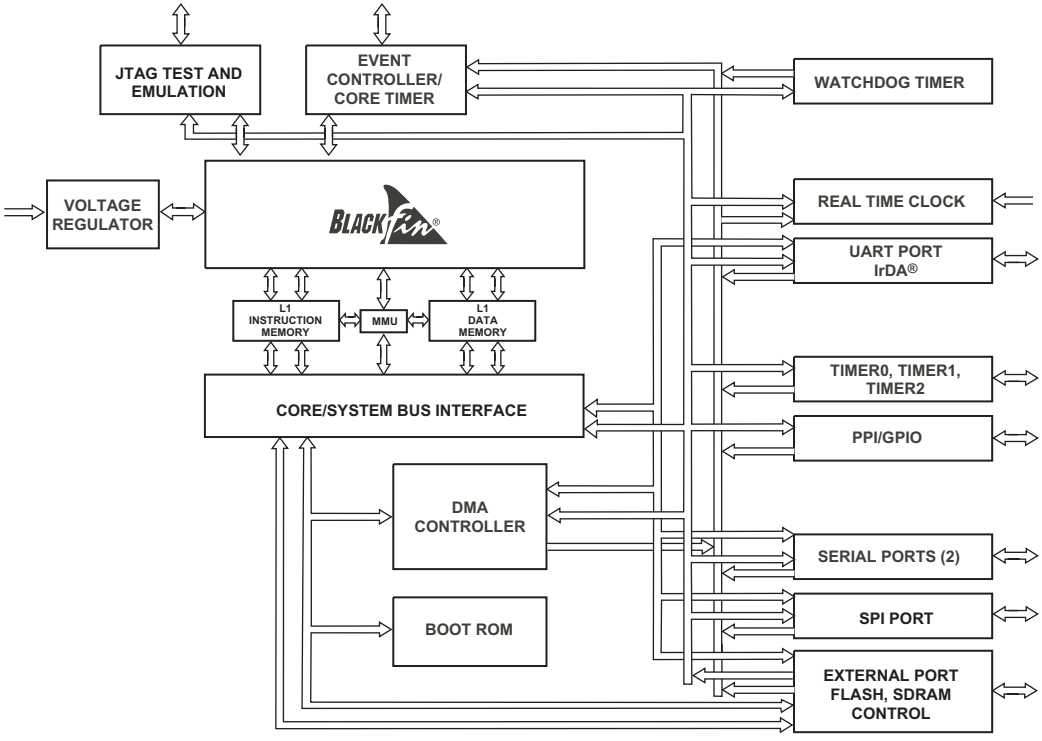


Figure 1-1. Processor Block Diagram

All of the peripherals, except for general-purpose I/O, Real-Time Clock, and Timers, are supported by a flexible DMA structure. There are also two separate memory DMA channels dedicated to data transfers between the processor's memory spaces, which include external SDRAM and asynchronous memory. Multiple on-chip buses provide enough bandwidth to keep the processor core running even when there is also activity on all of the on-chip and external peripherals.

Core Architecture

The processor core contains two 16-bit multipliers, two 40-bit accumulators, two 40-bit arithmetic logic units (ALUs), four 8-bit video ALUs, and a 40-bit shifter, shown in Figure 1-2. The computational units process 8-, 16-, or 32-bit data from the register file.

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

Each MAC can perform a 16- by 16-bit multiply per cycle, with accumulation to a 40-bit result. Signed and unsigned formats, rounding, and saturation are supported.

The ALUs perform a traditional set of arithmetic and logical operations on 16-bit or 32-bit data. Many special instructions are included to accelerate various signal processing tasks. These include bit operations such as field extract and population count, modulo 2^{32} multiply, divide primitives, saturation and rounding, and sign/exponent detection. The set of video instructions include byte alignment and packing operations, 16-bit and 8-bit adds with clipping, 8-bit average operations, and 8-bit subtract/absolute value/accumulate (SAA) operations. Also provided are the compare/select and vector search instructions. For some instructions, two

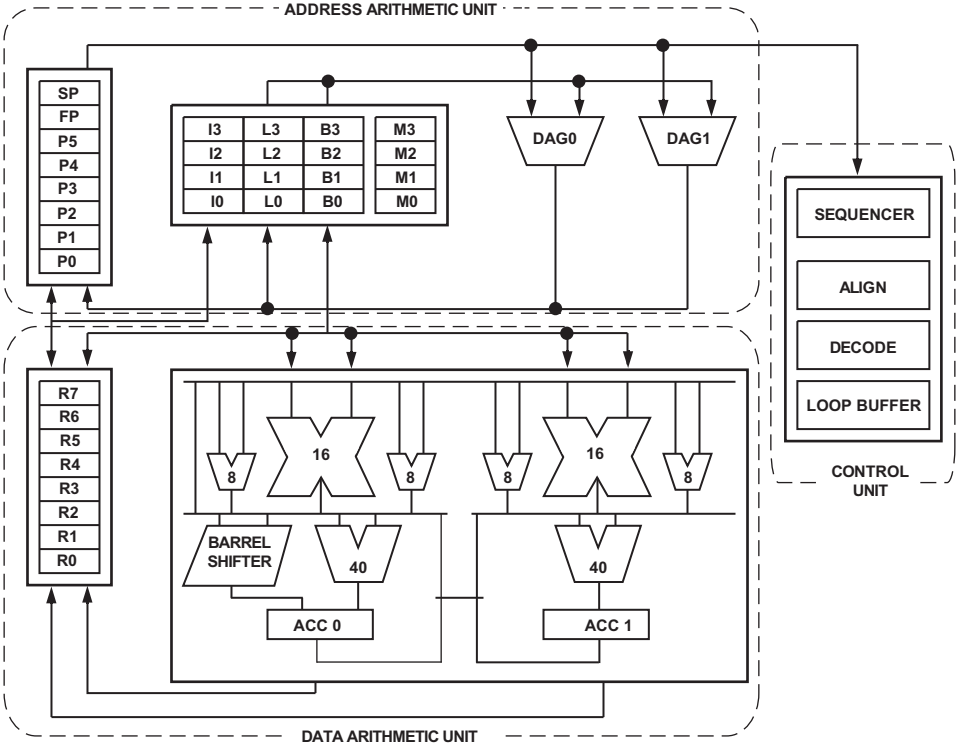


Figure 1-2. Processor Core Architecture

16-bit ALU operations can be performed simultaneously on register pairs (a 16-bit high half and 16-bit low half of a compute register). By also using the second ALU, quad 16-bit operations are possible.

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

A program sequencer controls the instruction execution flow, including instruction alignment and decoding. For program flow control, the sequencer supports PC-relative and indirect conditional jumps (with static branch prediction) and subroutine calls. Hardware is provided to support

zero-overhead looping. The architecture is fully interlocked, meaning there are no visible pipeline effects when executing instructions with data dependencies.

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

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

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

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

The ADSP-BF53x Blackfin processor instruction set is optimized so that 16-bit opcodes represent the most frequently used instructions. Complex DSP instructions are encoded into 32-bit opcodes as multifunction instructions. Blackfin products support a limited multi-issue capability, where a 32-bit instruction can be issued in parallel with two 16-bit instructions. This allows the programmer to use many of the core resources in a single instruction cycle.

The ADSP-BF53x Blackfin processor assembly language uses an algebraic syntax. The architecture is optimized for use with the C compiler.

Memory Architecture

The Blackfin processor architecture structures memory as a single, unified 4G byte address space using 32-bit addresses. All resources, including internal memory, external memory, and I/O control registers, occupy separate sections of this common address space. The memory portions of this address space are arranged in a hierarchical structure to provide a good cost/performance balance of some very fast, low latency on-chip memory as cache or SRAM, and larger, lower cost and lower performance off-chip memory systems. Table 1-1 shows the memory comparison for the ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors.

Table 1-1. Memory Comparison

Type of Memory	ADSP-BF531	ADSP-BF532	ADSP-BF533
Instruction SRAM/Cache	16K byte	16K byte	16K byte
Instruction SRAM	16K byte	32K byte	64K byte
Data SRAM/Cache	16K byte	32K byte	32K byte
Data SRAM	-	-	32K byte
Scratchpad	4K byte	4K byte	4K byte
Total	**84K byte**	**116K byte**	**148K byte**

The L1 memory system is the primary highest performance memory available to the core. The off-chip memory system, accessed through the External Bus Interface Unit (EBIU), provides expansion with SDRAM, flash memory, and SRAM, optionally accessing up to 132M bytes of physical memory.

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

Internal Memory

The processor has three blocks of on-chip memory that provide high bandwidth access to the core:

- L1 instruction memory, consisting of SRAM and a 4-way set-associative cache. This memory is accessed at full processor speed.
- L1 data memory, consisting of SRAM and/or a 2-way set-associative cache. This memory block is accessed at full processor speed.
- L1 scratchpad RAM, which runs at the same speed as the L1 memories but is only accessible as data SRAM and cannot be configured as cache memory.

External Memory

External (off-chip) memory is accessed via the External Bus Interface Unit (EBIU). This 16-bit interface provides a glueless connection to a bank of synchronous DRAM (SDRAM) and as many as four banks of asynchronous memory devices including flash memory, EPROM, ROM, SRAM, and memory-mapped I/O devices.

The PC133-compliant SDRAM controller can be programmed to interface to up to 128M bytes of SDRAM.

The asynchronous memory controller can be programmed to control up to four banks of devices. Each bank occupies a 1M byte segment regardless of the size of the devices used, so that these banks are only contiguous if each is fully populated with 1M byte of memory.

I/O Memory Space

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

Event Handling

The event controller on the processor handles all asynchronous and synchronous events to the processor. The processor event handling supports both nesting and prioritization. Nesting allows multiple event service routines to be active simultaneously. Prioritization ensures that servicing a higher priority event takes precedence over servicing a lower priority event. The controller provides support for five different types of events:

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

- Interrupts – Asynchronous to program flow. These are caused by input pins, timers, and other peripherals.

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

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

Core Event Controller (CEC)

The Core Event Controller supports nine general-purpose interrupts (IVG15–7), in addition to the dedicated interrupt and exception events. Of these general-purpose interrupts, the two lowest priority interrupts (IVG15–14) are recommended to be reserved for software interrupt handlers, leaving seven prioritized interrupt inputs to support peripherals.

System Interrupt Controller (SIC)

The System Interrupt Controller provides the mapping and routing of events from the many peripheral interrupt sources to the prioritized general-purpose interrupt inputs of the CEC. Although the processor provides a default mapping, the user can alter the mappings and priorities of interrupt events by writing the appropriate values into the Interrupt Assignment Registers (IAR).

DMA Support

The processor has multiple, independent DMA controllers that support automated data transfers with minimal overhead for the core. DMA transfers can occur between the internal memories and any of its DMA-capable peripherals. Additionally, DMA transfers can be accomplished between any of the DMA-capable peripherals and external devices connected to the external memory interfaces, including the SDRAM controller and the asynchronous memory controller. DMA-capable peripherals include the SPORTs, SPI port, UART, and PPI. Each individual DMA-capable peripheral has at least one dedicated DMA channel.

The DMA controller supports both one-dimensional (1D) and two-dimensional (2D) DMA transfers. DMA transfer initialization can be implemented from registers or from sets of parameters called descriptor blocks.

The 2D DMA capability supports arbitrary row and column sizes up to 64K elements by 64K elements, and arbitrary row and column step sizes up to +/- 32K elements. Furthermore, the column step size can be less than the row step size, allowing implementation of interleaved data-streams. This feature is especially useful in video applications where data can be de-interleaved on the fly.

Examples of DMA types supported include:

- A single, linear buffer that stops upon completion
- A circular, auto-refreshing buffer that interrupts on each full or fractionally full buffer
- 1D or 2D DMA using a linked list of descriptors
- 2D DMA using an array of descriptors specifying only the base DMA address within a common page

In addition to the dedicated peripheral DMA channels, there is a separate memory DMA channel provided for transfers between the various memories of the system. This enables transfers of blocks of data between any of the memories—including external SDRAM, ROM, SRAM, and flash memory—with minimal processor intervention. Memory DMA transfers can be controlled by a very flexible descriptor-based methodology or by a standard register-based autobuffer mechanism.

External Bus Interface Unit

The External Bus Interface Unit (EBIU) on the processor interfaces with a wide variety of industry-standard memory devices. The controller consists of an SDRAM controller and an asynchronous memory controller.

PC133 SDRAM Controller

The SDRAM controller provides an interface to a single bank of industry-standard SDRAM devices or DIMMs. Fully compliant with the PC133 SDRAM standard, the bank can be configured to contain between 16M and 128M bytes of memory.

A set of programmable timing parameters is available to configure the SDRAM bank to support slower memory devices. The memory bank is 16 bits wide for minimum device count and lower system cost.

Asynchronous Controller

The asynchronous memory controller provides a configurable interface for up to four separate banks of memory or I/O devices. Each bank can be independently programmed with different timing parameters. This allows connection to a wide variety of memory devices, including SRAM, ROM, and flash EPROM, as well as I/O devices that interface with standard memory control lines. Each bank occupies a 1M byte window in the

processor address space, but if not fully populated, these are not made contiguous by the memory controller. The banks are 16 bits wide, for interfacing to a range of memories and I/O devices.

Parallel Peripheral Interface

The processor provides a Parallel Peripheral Interface (PPI) that can connect directly to parallel A/D and D/A converters, ITU-R 601/656 video encoders and decoders, and other general-purpose peripherals. The PPI consists of a dedicated input clock pin, up to 3 frame synchronization pins, and up to 16 data pins. The input clock supports parallel data rates up to half the system clock rate.

In ITU-R 656 modes, the PPI receives and parses a data stream of 8-bit or 10-bit data elements. On-chip decode of embedded preamble control and synchronization information is supported.

Three distinct ITU-R 656 modes are supported:

- Active Video Only – The PPI does not read in any data between the End of Active Video (EAV) and Start of Active Video (SAV) preamble symbols, or any data present during the vertical blanking intervals. In this mode, the control byte sequences are not stored to memory; they are filtered by the PPI.
- Vertical Blanking Only – The PPI only transfers Vertical Blanking Interval (VBI) data, as well as horizontal blanking information and control byte sequences on VBI lines.
- Entire Field – The entire incoming bitstream is read in through the PPI. This includes active video, control preamble sequences, and ancillary data that may be embedded in horizontal and vertical blanking intervals.

Though not explicitly supported, ITU-R 656 output functionality can be achieved by setting up the entire frame structure (including active video, blanking, and control information) in memory and streaming the data out the PPI in a frame sync-less mode. The processor's 2D DMA features facilitate this transfer by allowing the static frame buffer (blanking and control codes) to be placed in memory once, and simply updating the active video information on a per-frame basis.

The general-purpose modes of the PPI are intended to suit a wide variety of data capture and transmission applications. The modes are divided into four main categories, each allowing up to 16 bits of data transfer per `PPI_CLK` cycle:

- Data Receive with Internally Generated Frame Syncs
- Data Receive with Externally Generated Frame Syncs
- Data Transmit with Internally Generated Frame Syncs
- Data Transmit with Externally Generated Frame Syncs

These modes support ADC/DAC connections, as well as video communication with hardware signalling. Many of the modes support more than one level of frame synchronization. If desired, a programmable delay can be inserted between assertion of a frame sync and reception/transmission of data.

Serial Ports (SPORTs)

The processor incorporates two dual-channel synchronous serial ports (SPORT0 and SPORT1) for serial and multiprocessor communications. The SPORTs support these features:

- Bidirectional, I^2S capable operation. Each SPORT has two sets of independent transmit and receive pins, enabling eight channels of I^2S stereo audio.
- Buffered (eight-deep) transmit and receive ports. Each port has a data register for transferring data words to and from other processor components and shift registers for shifting data in and out of the data registers.
- Clocking. Each transmit and receive port can either use an external serial clock or can generate its own in a wide range of frequencies.
- Word length. Each SPORT supports serial data words from 3 to 32 bits in length, transferred in most significant bit first or least significant bit first format.
- Framing. Each transmit and receive port can run with or without frame sync signals for each data word. Frame sync signals can be generated internally or externally, active high or low, and with either of two pulse widths and early or late frame sync.

- Companding in hardware

  Each SPORT can perform A-law or µ-law companding according to ITU recommendation G.711. Companding can be selected on the transmit and/or receive channel of the SPORT without additional latencies.

- DMA operations with single cycle overhead

  Each SPORT can automatically receive and transmit multiple buffers of memory data. The processor can link or chain sequences of DMA transfers between a SPORT and memory.

- Interrupts

  Each transmit and receive port generates an interrupt upon completing the transfer of a data word or after transferring an entire data buffer or buffers through DMA.

- Multichannel capability

  Each SPORT supports 128 channels out of a 1024-channel window and is compatible with the H.100, H.110, MVIP-90, and HMVIP standards.

Serial Peripheral Interface (SPI) Port

The processor has an SPI-compatible port that enables the processor to communicate with multiple SPI-compatible devices.

The SPI interface uses three pins for transferring data: two data pins and a clock pin. An SPI chip select input pin lets other SPI devices select the processor, and seven SPI chip select output pins let the processor select other SPI devices. The SPI select pins are reconfigured Programmable Flag

pins. Using these pins, the SPI port provides a full-duplex, synchronous serial interface, which supports both master and slave modes and multi-master environments.

The SPI port's baud rate and clock phase/polarities are programmable, and it has an integrated DMA controller, configurable to support either transmit or receive datastreams. The SPI's DMA controller can only service unidirectional accesses at any given time.

During transfers, the SPI port simultaneously transmits and receives by serially shifting data in and out of its two serial data lines. The serial clock line synchronizes the shifting and sampling of data on the two serial data lines.

Timers

There are four general-purpose programmable timer units in the processor. Three timers have an external pin that can be configured either as a Pulse Width Modulator (PWM) or timer output, as an input to clock the timer, or as a mechanism for measuring pulse widths of external events. These timer units can be synchronized to an external clock input connected to the `PF1` pin, an external clock input to the `PPI_CLK` pin, or to the internal `SCLK`.

The timer units can be used in conjunction with the UART to measure the width of the pulses in the datastream to provide an autobaud detect function for a serial channel.

The timers can generate interrupts to the processor core to provide periodic events for synchronization, either to the processor clock or to a count of external signals.

In addition to the three general-purpose programmable timers, a fourth timer is also provided. This extra timer is clocked by the internal processor clock and is typically used as a system tick clock for generation of operating system periodic interrupts.

UART Port

The processor provides a full-duplex Universal Asynchronous Receiver/Transmitter (UART) port, which is fully compatible with PC-standard UARTs. The UART port provides a simplified UART interface to other peripherals or hosts, providing full- or half-duplex, DMA-supported, asynchronous transfers of serial data. The UART port includes support for 5 to 8 data bits; 1 or 2 stop bits; and none, even, or odd parity. The UART port supports two modes of operation:

- Programmed I/O. The processor sends or receives data by writing or reading I/O-mapped UART registers. The data is double buffered on both transmit and receive.
- Direct Memory Access (DMA). The DMA controller transfers both transmit and receive data. This reduces the number and frequency of interrupts required to transfer data to and from memory. The UART has two dedicated DMA channels, one for transmit and one for receive. These DMA channels have lower priority than most DMA channels because of their relatively low service rates.

The UART port's baud rate, serial data format, error code generation and status, and interrupts can be programmed to support:

- Wide range of bit rates
- Data formats from 7 to 12 bits per frame
- Generation of maskable interrupts to the processor by both transmit and receive operations

In conjunction with the general-purpose timer functions, autobaud detection is supported.

The capabilities of the UART are further extended with support for the Infrared Data Association (IrDA®) Serial Infrared Physical Layer Link Specification (SIR) protocol.

Real-Time Clock

The processor's Real-Time Clock (RTC) provides a robust set of digital watch features, including current time, stopwatch, and alarm. The RTC is clocked by a 32.768 kHz crystal external to the processor. The RTC peripheral has dedicated power supply pins, so that it can remain powered up and clocked even when the rest of the processor is in a low power state. The RTC provides several programmable interrupt options, including interrupt per second, minute, hour, or day clock ticks, interrupt on programmable stopwatch countdown, or interrupt at a programmed alarm time.

The 32.768 kHz input clock frequency is divided down to a 1 Hz signal by a prescaler. The counter function of the timer consists of four counters: a 60 second counter, a 60 minute counter, a 24 hours counter, and a 32768 day counter.

When enabled, the alarm function generates an interrupt when the output of the timer matches the programmed value in the alarm control register. There are two alarms. The first alarm is for a time of day. The second alarm is for a day and time of that day.

The stopwatch function counts down from a programmed value, with one minute resolution. When the stopwatch is enabled and the counter underflows, an interrupt is generated.

Like the other peripherals, the RTC can wake up the processor from Sleep mode or Deep Sleep mode upon generation of any RTC wakeup event. An RTC wakeup event can also wake up the on-chip internal voltage regulator from a powered down state.

Watchdog Timer

The processor includes a 32-bit timer that can be used to implement a software watchdog function. A software watchdog can improve system availability by forcing the processor to a known state through generation of a hardware reset, nonmaskable interrupt (NMI), or general-purpose interrupt, if the timer expires before being reset by software. The programmer initializes the count value of the timer, enables the appropriate interrupt, then enables the timer. Thereafter, the software must reload the counter before it counts to zero from the programmed value. This protects the system from remaining in an unknown state where software that would normally reset the timer has stopped running due to an external noise condition or software error.

If configured to generate a hardware reset, the watchdog timer resets both the CPU and the peripherals. After a reset, software can determine if the watchdog was the source of the hardware reset by interrogating a status bit in the watchdog control register.

The timer is clocked by the system clock (SCLK), at a maximum frequency of f_{SCLK}.

Programmable Flags

The processor has 16 bidirectional programmable flag (PF) or general-purpose I/O pins, PF[15:0]. Each pin can be individually configured using the flag control, status, and interrupt registers.

- Flag Direction Control register – Specifies the direction of each individual PFx pin as input or output.
- Flag Control and Status registers – The processor employs a "write-1-to-modify" mechanism that allows any combination of individual flags to be modified in a single instruction, without affecting the level of any other flags. Four control registers are provided. One register is written in order to set flag values, one register is written in order to clear flag values, one register is written in order to toggle flag values, and one register is written in order to specify any number of flag values. Reading the Flag Status register allows software to interrogate the sense of the flags.
- Flag Interrupt Mask registers – The two Flag Interrupt Mask registers allow each individual PFx pin to function as an interrupt to the processor. Similar to the two Flag Control registers that are used to set and clear individual flag values, one Flag Interrupt Mask register sets bits to enable interrupt function, and the other Flag Interrupt Mask register clears bits to disable interrupt function. The PFx pins defined as inputs can be configured to generate hardware interrupts, while output PFx pins can be triggered by software interrupts.
- Flag Interrupt Sensitivity registers – The two Flag Interrupt Sensitivity registers specify whether individual PFx pins are level- or edge-sensitive and specify—if edge-sensitive—whether just the rising edge or both the rising and falling edges of the signal are significant. One register selects the type of sensitivity, and one register selects which edges are significant for edge sensitivity.

Clock Signals

The processor can be clocked by an external crystal, a sine wave input, or a buffered, shaped clock derived from an external clock oscillator.

This external clock connects to the processor's CLKIN pin. The CLKIN input cannot be halted, changed, or operated below the specified frequency during normal operation. This clock signal should be a TTL-compatible signal.

The core clock (CCLK) and system peripheral clock (SCLK) are derived from the input clock (CLKIN) signal. An on-chip Phase Locked Loop (PLL) is capable of multiplying the CLKIN signal by a user-programmable (1x to 63x) multiplication factor (bounded by specified minimum and maximum VCO frequencies). The default multiplier is 10x, but it can be modified by a software instruction sequence. On-the-fly frequency changes can be made by simply writing to the PLL_DIV register.

All on-chip peripherals are clocked by the system clock (SCLK). The system clock frequency is programmable by means of the SSEL[3:0] bits of the PLL_DIV register.

Dynamic Power Management

The processor provides four operating modes, each with a different performance/power profile. In addition, Dynamic Power Management provides the control functions to dynamically alter the processor core supply voltage to further reduce power dissipation. Control of clocking to each of the peripherals also reduces power consumption.

Full On Mode (Maximum Performance)

In the Full On mode, the PLL is enabled, not bypassed, providing the maximum operational frequency. This is the normal execution state in which maximum performance can be achieved. The processor core and all enabled peripherals run at full speed.

Active Mode (Moderate Power Savings)

In the Active mode, the PLL is enabled, but bypassed. Because the PLL is bypassed, the processor's core clock (CCLK) and system clock (SCLK) run at the input clock (CLKIN) frequency. In this mode, the CLKIN to VCO multiplier ratio can be changed, although the changes are not realized until the Full On mode is entered. DMA access is available to appropriately configured L1 memories.

In the Active mode, it is possible to disable the PLL through the PLL Control register (PLL_CTL). If disabled, the PLL must be re-enabled before transitioning to the Full On or Sleep modes.

Sleep Mode (High Power Savings)

The Sleep mode reduces power dissipation by disabling the clock to the processor core (CCLK). The PLL and system clock (SCLK), however, continue to operate in this mode. Typically an external event or RTC activity will wake up the processor. When in the Sleep mode, assertion of any interrupt causes the processor to sense the value of the bypass bit (BYPASS) in the PLL Control register (PLL_CTL). If bypass is disabled, the processor transitions to the Full On mode. If bypass is enabled, the processor transitions to the Active mode.

When in the Sleep mode, system DMA access to L1 memory is not supported.

Deep Sleep Mode (Maximum Power Savings)

The Deep Sleep mode maximizes power savings by disabling the processor core and synchronous system clocks (CCLK and SCLK). Asynchronous systems, such as the RTC, may still be running, but cannot access internal resources or external memory. This powered-down mode can only be exited by assertion of the reset interrupt or by an asynchronous interrupt generated by the RTC. When in Deep Sleep mode, an RTC asynchronous interrupt causes the processor to transition to the Active mode. Assertion of $\overline{\text{RESET}}$ while in Deep Sleep mode causes the processor to transition to the Full On mode.

Hibernate State

For lowest possible power dissipation, this state allows the internal supply (V_{DDINT}) to be powered down, while keeping the I/O supply (V_{DDEXT}) running. Although not strictly an operating mode like the four modes detailed above, it is illustrative to view it as such.

Voltage Regulation

The processor provides an on-chip voltage regulator that can generate internal voltage levels (0.8 V to 1.2 V) from an external 2.25 V to 3.6 V supply. Figure 1-3 shows the typical external components required to complete the power management system. The regulator controls the internal logic voltage levels and is programmable with the Voltage Regulator Control register (VR_CTL) in increments of 50 mV. To reduce standby power consumption, the internal voltage regulator can be programmed to remove power to the processor core while keeping I/O power supplied. While in this state, V_{DDEXT} can still be applied, eliminating the need for external buffers. The regulator can also be disabled and bypassed at the user's discretion.

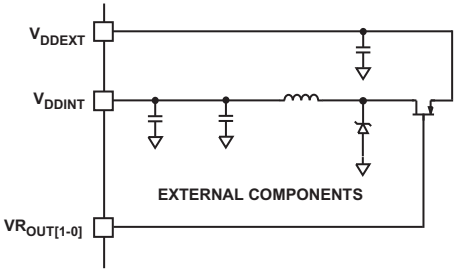


Figure 1-3. Voltage Regulator Circuit

Boot Modes

The processor has two mechanisms for automatically loading internal L1 instruction memory after a reset. A third mode is provided to execute from external memory, bypassing the boot sequence:

- Execute from 16-bit external memory – Execution starts from address 0x2000 0000 with 16-bit packing. The boot ROM is bypassed in this mode. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup).
- Boot from 8-bit or 16-bit external flash memory – The flash boot routine located in boot ROM memory space is set up using Asynchronous Memory Bank 0. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup).

- Boot from SPI serial EEPROM (8-, 16-, or 24-bit addressable) – The SPI uses the PF2 output pin to select a single SPI EEPROM device, submits successive read commands at addresses 0x00, 0x0000, and 0x000000 until a valid 8-, 16-, or 24-bit addressable EEPROM is detected, and begins clocking data into the beginning of L1 instruction memory.
- Boot from SPI host (slave mode) – A user-defined programmable flag pin is an output on the Blackfin processor and an input on the SPI host device. This flag allows the processor to hold off the host device from sending data during certain sections of the boot process. When this flag is de-asserted, the host can continue to send bytes to the processor.

For each of the boot modes, a 10-byte header is first read from an external memory device. The header specifies the number of bytes to be transferred and the memory destination address. Multiple memory blocks may be loaded by any boot sequence. Once all blocks are loaded, program execution commences from the start of L1 instruction SRAM.

In addition, bit 4 of the Reset Configuration register can be set by application code to bypass the normal boot sequence during a software reset. For this case, the processor jumps directly to the beginning of L1 instruction memory.

Instruction Set Description

The ADSP-BF53x processor family assembly language instruction set employs an algebraic syntax designed for ease of coding and readability. The instructions have been specifically tuned to provide a flexible, densely encoded instruction set that compiles to a very small final memory size. The instruction set also provides fully featured multifunction instructions that allow the programmer to use many of the processor core resources in a single instruction. Coupled with many features more often seen on

microcontrollers, this instruction set is very efficient when compiling C and C++ source code. In addition, the architecture supports both user (algorithm/application code) and supervisor (O/S kernel, device drivers, debuggers, ISRs) modes of operation, allowing multiple levels of access to core resources.

The assembly language, which takes advantage of the processor's unique architecture, offers these advantages:

- Seamlessly integrated DSP/CPU features optimized for both 8-bit and 16-bit operations
- A multi-issue load/store modified Harvard architecture, which supports two 16-bit MAC or four 8-bit ALU + two load/store + two pointer updates per cycle
- All registers, I/O, and memory mapped into a unified 4G byte memory space, providing a simplified programming model
- Microcontroller features, such as arbitrary bit and bit field manipulation, insertion, and extraction; integer operations on 8-, 16-, and 32-bit data types; and separate user and supervisor stack pointers.

Code density enhancements include intermixing of 16- and 32-bit instructions with no mode switching or code segregation. Frequently used instructions are encoded in 16 bits.

Development Tools

The processor is supported with a complete set of CrossCore® software and hardware development tools, including Analog Devices emulators and the VisualDSP++ development environment. The same emulator hardware that supports other Analog Devices products also fully emulates the ADSP-BF53x processor family.

The VisualDSP++ project management environment lets programmers develop and debug an application. This environment includes an easy-to-use assembler that is based on an algebraic syntax, an archiver (librarian/library builder), a linker, a loader, a cycle-accurate instruction-level simulator, a C/C++ compiler, and a C/C++ runtime library that includes DSP and mathematical functions. A key point for these tools is C/C++ code efficiency. The compiler has been developed for efficient translation of C/C++ code to Blackfin processor assembly. The Blackfin processor has architectural features that improve the efficiency of compiled C/C++ code.

Debugging both C/C++ and assembly programs with the VisualDSP++ debugger, programmers can:

- View mixed C/C++ and assembly code (interleaved source and object information)
- Insert breakpoints
- Set conditional breakpoints on registers, memory, and stacks
- Trace instruction execution
- Perform linear or statistical profiling of program execution
- Fill, dump, and graphically plot the contents of memory
- Perform source level debugging
- Create custom debugger windows

The VisualDSP++ Integrated Development Environment (IDE) lets programmers define and manage software development. Its dialog boxes and property pages let programmers configure and manage all development tools, including Color Syntax Highlighting in the VisualDSP++ editor. These capabilities permit programmers to:

- Control how the development tools process inputs and generate outputs.
- Maintain a one-to-one correspondence with the tool's command-line switches.

The VisualDSP++ Kernel (VDK) incorporates scheduling and resource management tailored specifically to address the memory and timing constraints of DSP programming. These capabilities enable engineers to develop code more effectively, eliminating the need to start from the very beginning, when developing new application code. The VDK features include threads, critical and unscheduled regions, semaphores, events, and device flags. The VDK also supports priority-based, pre-emptive, cooperative and time-sliced scheduling approaches. In addition, the VDK was designed to be scalable. If the application does not use a specific feature, the support code for that feature is excluded from the target system.

Because the VDK is a library, a developer can decide whether to use it or not. The VDK is integrated into the VisualDSP++ development environment but can also be used with standard command-line tools. The VDK development environment assists in managing system resources, automating the generation of various VDK-based objects, and visualizing the system state during application debug.

Analog Devices emulators use the IEEE 1149.1 JTAG test access port of the processor to monitor and control the target board processor during emulation. The emulator provides full speed emulation, allowing inspection and modification of memory, registers, and processor stacks.

Nonintrusive in-circuit emulation is assured by the use of the processor's JTAG interface—the emulator does not affect target system loading or timing.

In addition to the software and hardware development tools available from Analog Devices, third parties provide a wide range of tools supporting the Blackfin processor family. Hardware tools include the ADSP-BF533 EZ-KIT Lite standalone evaluation/development cards. Third party software tools include DSP libraries, real-time operating systems, and block diagram design tools.

2 COMPUTATIONAL UNITS

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

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

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

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

Figure 2-1 provides a graphical guide to the other topics in this chapter. An examination of each computational unit provides details about its operation and is followed by a summary of computational instructions. Studying the details of the computational units, register files, and data

buses leads to a better understanding of proper data flow for computations. Next, details about the processor's advanced parallelism reveal how to take advantage of multifunction instructions.

Figure 2-1 shows the relationship between the Data Register File and the computational units—multipliers, ALUs, and shifter.

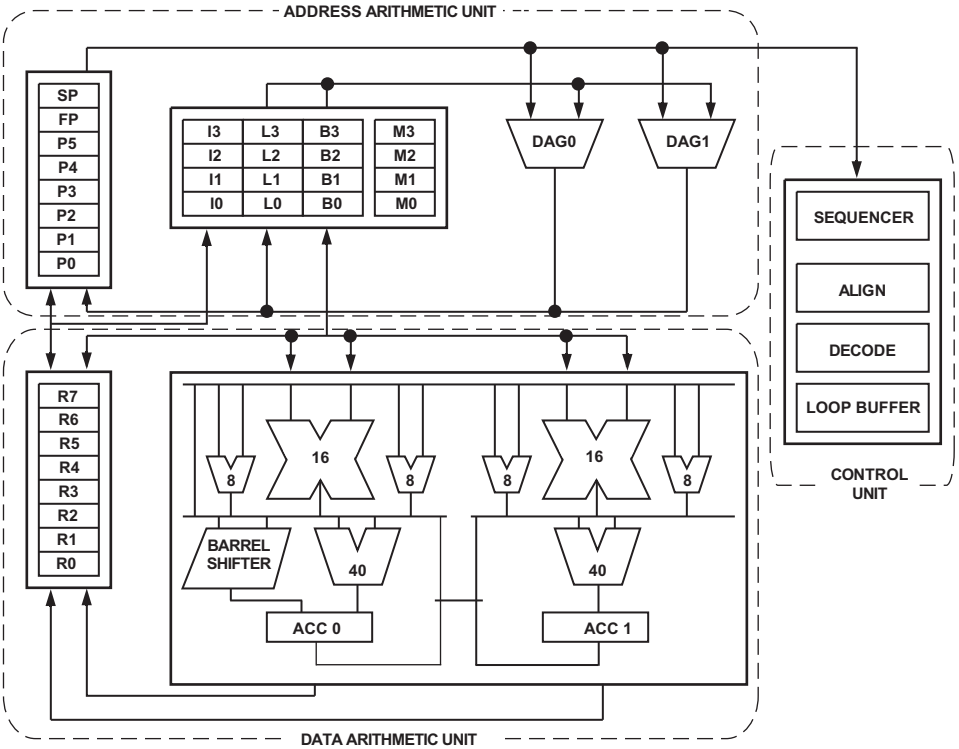


Figure 2-1. Processor Core Architecture

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

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

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

Using Data Formats

ADSP-BF53x processors are primarily 16-bit, fixed-point machines. Most operations assume a two's-complement number representation, while others assume unsigned numbers or simple binary strings. Other instructions support 32-bit integer arithmetic, with further special features supporting 8-bit arithmetic and block floating point. For detailed information about each number format, see Appendix D, "Numeric Formats."

In the ADSP-BF53x processor family arithmetic, signed numbers are always in two's-complement format. These processors do not use signed-magnitude, one's-complement, binary-coded decimal (BCD), or excess-n formats.

Binary String

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

Unsigned

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

Signed Numbers: Two's-Complement

In ADSP-BF53x processor arithmetic, the word *signed* refers to two's-complement numbers. Most ADSP-BF53x processor family operations presume or support two's-complement arithmetic.

Fractional Representation: 1.15

ADSP-BF53x processor arithmetic is optimized for numerical values in a fractional binary format denoted by 1.15 ("one dot fifteen"). In the 1.15 format, 1 sign bit (the Most Significant Bit (MSB)) and 15 fractional bits represent values from –1 to 0.999969.

Figure 2-2 shows the bit weighting for 1.15 numbers as well as some examples of 1.15 numbers and their decimal equivalents.

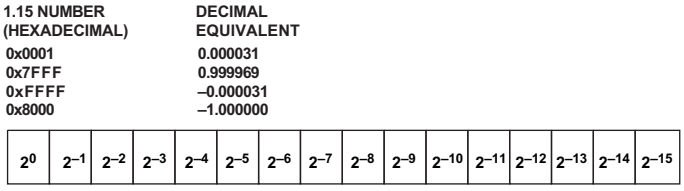


Figure 2-2. Bit Weighting for 1.15 Numbers

Register Files

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

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

For more information, see Chapter 5, "Data Address Generators."

The processor register files appear in Figure 2-3.

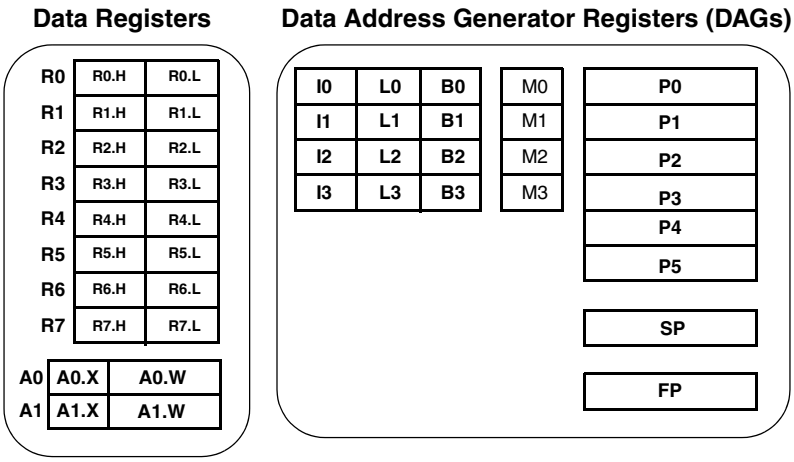


Figure 2-3. Register Files

In the processor, a word is 32 bits long; H denotes the high order 16 bits of a 32-bit register; L denotes the low order 16 bits of a 32-bit register. For example, `A0.W` contains the lower 32 bits of the 40-bit `A0` register; `A0.L` contains the lower 16 bits of `A0.W`, and `A0.H` contains the upper 16 bits of `A0.W`.

Data Register File

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

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

Accumulator Registers

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

Pointer Register File

The general-purpose Address Pointer registers, also called P-registers, are organized as:

- 6-entry, P-register files `P[5:0]`
- Frame Pointers (`FP`) used to point to the current procedure's activation record
- Stack Pointer registers (`SP`) used to point to the last used location on the runtime stack. See mode dependent registers in Chapter 3, "Operating Modes and States."

P-registers are 32 bits wide. Although P-registers are primarily used for address calculations, they may also be used for general integer arithmetic with a limited set of arithmetic operations; for instance, to maintain counters. However, unlike the Data registers, P-register arithmetic does not affect the Arithmetic Status (`ASTAT`) register status flags.

DAG Register Set

DSP instructions primarily use the Data Address Generator (DAG) register set for addressing. The DAG register set consists of these registers:

- `I[3:0]` contain index addresses
- `M[3:0]` contain modify values
- `B[3:0]` contain base addresses
- `L[3:0]` contain length values

All DAG registers are 32 bits wide.

The I (Index) registers and B (Base) registers always contain addresses of 8-bit bytes in memory. The Index registers contain an effective address. The M (Modify) registers contain an offset value that is added to one of the Index registers or subtracted from it.

The B and L (Length) registers define circular buffers. The B register contains the starting address of a buffer, and the L register contains the length in bytes. Each L and B register pair is associated with the corresponding I register. For example, `L0` and `B0` are always associated with `I0`. However, any M register may be associated with any I register. For example, `I0` may be modified by `M3`. For more information, see Chapter 5, "Data Address Generators."

Register File Instruction Summary

Table 2-1 lists the register file instructions. For more information about assembly language syntax, see the *Blackfin Processor Programming Reference*.

In Table 2-1, note the meaning of these symbols:

- Allreg denotes: `R[7:0]`, `P[5:0]`, `SP`, `FP`, `I[3:0]`, `M[3:0]`, `B[3:0]`, `L[3:0]`, `A0.X`, `A0.W`, `A1.X`, `A1.W`, `ASTAT`, `RETS`, `RETI`, `RETX`, `RETN`, `RETE`, `LC[1:0]`, `LT[1:0]`, `LB[1:0]`, `USP`, `SEQSTAT`, `SYSCFG`, `CYCLES`, and `CYCLES2`.
- A*n* denotes either ALU Result register `A0` or `A1`.
- Dreg denotes any Data Register File register.
- Sysreg denotes the system registers: `ASTAT`, `SEQSTAT`, `SYSCFG`, `RETI`, `RETX`, `RETN`, `RETE`, or `RETS`, `LC[1:0]`, `LT[1:0]`, `LB[1:0]`, `CYCLES`, and `CYCLES2`.
- Preg denotes any Pointer register, `FP`, or `SP` register.
- Dreg_even denotes `R0`,`R2`,`R4`, or `R6`.

- Dreg_odd denotes `R1,R3,R5,` or `R7.`
- DPreg denotes any Data Register File register or any Pointer register, `FP`, or `SP` register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- A*n*.L denotes the lower 16 bits of Accumulator `A0.W` or `A1.W`.
- A*n*.H denotes the upper 16 bits of Accumulator `A0.W` or `A1.W`.
- Dreg_byte denotes the low order 8 bits of each Data register.
- Option (X) denotes sign extended.
- Option (Z) denotes zero extended.
- * Indicates the flag may be set or cleared, depending on the result of the instruction.
- ** Indicates the flag is cleared.
- – Indicates no effect.

Table 2-1. Register File Instruction Summary

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AVS	AV1 AV1S	CC	V V_COPY VS
allreg = allreg ; [1]	–	–	–	–	–	–	–
A*n* = A*n* ;	–	–	–	–	–	–	–
A*n* = Dreg ;	–	–	–	–	–	–	–
Dreg_even = A0 ;	*	*	–	–	–	–	*

Table 2-1. Register File Instruction Summary (Cont'd)

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

1 Warning: Not all register combinations are allowed. For details, see the functional description of the Move Register instruction in the *Blackfin Processor Programming Reference*.

Data Types

The processor supports 32-bit words, 16-bit half words, and bytes. The 32- and 16-bit words can be integer or fractional, but bytes are always integers. Integer data types can be signed or unsigned, but fractional data types are always signed.

Table 2-3 illustrates the formats for data that resides in memory, in the register file, and in the accumulators. In the table, the letter *d* represents one bit, and the letter *s* represents one signed bit.

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

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

Other instructions manipulate data as 32-bit numbers. In addition, two 16-bit half words or four 8-bit bytes can be manipulated as 32-bit values. For details, refer to the instructions in the *Blackfin Processor Programming Reference*.

In Table 2-2, note the meaning of these symbols:

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

Table 2-2. Data Formats

Format	Representation in Memory	Representation in 32-bit Register
32.0 Unsigned Word	dddd dddd dddd dddd dddd dddd dddd dddd	dddd dddd dddd dddd dddd dddd dddd dddd
32.0 Signed Word	sddd dddd dddd dddd dddd dddd dddd dddd	sddd dddd dddd dddd dddd dddd dddd dddd
16.0 Unsigned Half Word	dddd dddd dddd dddd	0000 0000 0000 0000 dddd dddd dddd dddd
16.0 Signed Half Word	sddd dddd dddd dddd	ssss ssss ssss ssss sddd dddd dddd dddd
8.0 Unsigned Byte	dddd dddd	0000 0000 0000 0000 0000 0000 dddd dddd
8.0 Signed Byte	sddd dddd	ssss ssss ssss ssss ssss ssss sddd dddd
0.16 Unsigned Fraction	.dddd dddd dddd dddd	0000 0000 0000 0000 .dddd dddd dddd dddd
1.15 Signed Fraction	s.ddd dddd dddd dddd	ssss ssss ssss ssss s.ddd dddd dddd dddd
0.32 Unsigned Fraction	.dddd dddd dddd dddd dddd dddd dddd dddd	.dddd dddd dddd dddd dddd dddd dddd dddd
1.31 Signed Fraction	s.ddd dddd dddd dddd dddd dddd dddd dddd	s.ddd dddd dddd dddd dddd dddd dddd dddd
Packed 8.0 Unsigned Byte	dddd dddd *dddd dddd* dddd dddd *dddd dddd*	dddd dddd *dddd dddd* dddd dddd *dddd dddd*
Packed 0.16 Unsigned Fraction	.dddd dddd dddd dddd *.dddd dddd dddd dddd*	.dddd dddd dddd dddd *.dddd dddd dddd dddd*
Packed 1.15 Signed Fraction	s.ddd dddd dddd dddd *s.ddd dddd dddd dddd*	s.ddd dddd dddd dddd *s.ddd dddd dddd dddd*

Endianess

Both internal and external memory are accessed in little endian byte order. For more information, see "Memory Transaction Model" on page 6-65.

ALU Data Types

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

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

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

ALU results generate status information. For more information about using ALU status, see "ALU Instruction Summary" on page 2-29.

Multiplier Data Types

Each multiplier produces results that are binary strings. The inputs are interpreted according to the information given in the instruction itself (whether it is signed multiplied by signed, unsigned multiplied by unsigned, a mixture, or a rounding operation). The 32-bit result from the multipliers is assumed to be signed; it is sign-extended across the full 40-bit width of the A0 or A1 registers.

The processor supports two modes of format adjustment: the fractional mode for fractional operands (1.15 format with 1 sign bit and 15 fractional bits) and the integer mode for integer operands (16.0 format).

When the processor multiplies two 1.15 operands, the result is a 2.30 (2 sign bits and 30 fractional bits) number. In the fractional mode, the multiplier automatically shifts the multiplier product left one bit before transferring the result to the multiplier result register (A0, A1). This shift of the redundant sign bit causes the multiplier result to be in 1.31 format, which can be rounded to 1.15 format. The resulting format appears in Figure 2-4.

In the integer mode, the left shift does not occur. For example, if the operands are in the 16.0 format, the 32-bit multiplier result would be in 32.0 format. A left shift is not needed and would change the numerical representation. This result format appears in Figure 2-5.

Multiplier results generate status information when they update accumulators or when they are transferred to a destination register in the register file. For more information, see "Multiplier Instruction Summary" on page 2-40.

Shifter Data Types

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

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

Shifter results generate status information. For more information about using shifter status, see "Shifter Instruction Summary" on page 2-55.

Arithmetic Formats Summary

Table 2-3, Table 2-4, Table 2-5, and Table 2-6 summarize some of the arithmetic characteristics of computational operations.

Table 2-3. ALU Arithmetic Formats

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

Table 2-4. Multiplier Fractional Modes Formats

Operation	Operand Formats	Result Formats
Multiplication	1.15 explicitly signed or unsigned	2.30 shifted to 1.31
Multiplication/Addition	1.15 explicitly signed or unsigned	2.30 shifted to 1.31
Multiplication/Subtraction	1.15 explicitly signed or unsigned	2.30 shifted to 1.31

Table 2-5. Multiplier Arithmetic Integer Modes Formats

Operation	Operand Formats	Result Formats
Multiplication	16.0 explicitly signed or unsigned	32.0 not shifted
Multiplication/Addition	16.0 explicitly signed or unsigned	32.0 not shifted
Multiplication/Subtraction	16.0 explicitly signed or unsigned	32.0 not shifted

Table 2-6. Shifter Arithmetic Formats

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

Using Multiplier Integer and Fractional Formats

For multiply-and-accumulate functions, the processor provides two choices—fractional arithmetic for fractional numbers (1.15) and integer arithmetic for integers (16.0).

For fractional arithmetic, the 32-bit product output is format adjusted—sign-extended and shifted one bit to the left—before being added to accumulator A0 or A1. For example, bit 31 of the product lines up with bit 32 of A0 (which is bit 0 of A0.X), and bit 0 of the product lines up with bit 1 of A0 (which is bit 1 of A0.W). The Least Significant Bit (LSB) is zero filled. The fractional multiplier result format appears in Figure 2-4.

For integer arithmetic, the 32-bit product register is not shifted before being added to A0 or A1. Figure 2-5 shows the integer mode result placement.

With either fractional or integer operations, the multiplier output product is fed into a 40-bit adder/subtracter which adds or subtracts the new product with the current contents of the A0 or A1 register to produce the final 40-bit result.

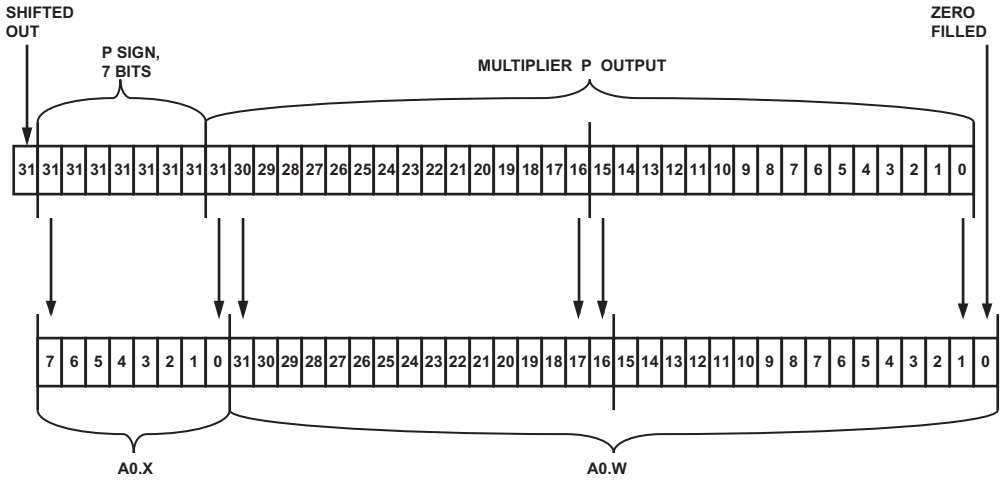


Figure 2-4. Fractional Multiplier Results Format

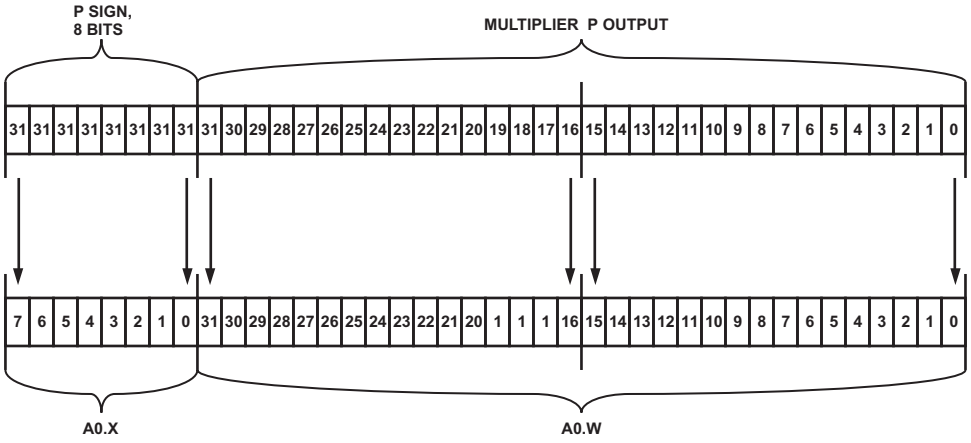


Figure 2-5. Integer Multiplier Results Format

Rounding Multiplier Results

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

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

For most algorithms, unbiased rounding is preferred.

Unbiased Rounding

The *convergent* rounding method returns the number closest to the original. In cases where the original number lies exactly halfway between two numbers, this method returns the nearest even number, the one containing an LSB of 0. For example, when rounding the 3-bit, two's-complement fraction 0.25 (binary 0.01) to the nearest 2-bit, two's-complement fraction, the result would be 0.0, because that is the even-numbered choice of 0.5 and 0.0. Since it rounds up and down based on the surrounding values, this method is called *unbiased* rounding.

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

The accumulator uses an unbiased rounding scheme. The conventional method of biased rounding adds a 1 into bit position 15 of the adder chain. This method causes a net positive bias because the midway value (when `A0.L/A1.L = 0x8000`) is always rounded upward.

The accumulator eliminates this bias by forcing bit 16 in the result output to 0 when it detects this midway point. Forcing bit 16 to 0 has the effect of rounding odd `A0.L/A1.L` values upward and even values downward, yielding a large sample bias of 0, assuming uniformly distributed values.

The following examples use *x* to represent any bit pattern (not all zeros). The example in Figure 2-6 shows a typical rounding operation for `A0`; the example also applies for `A1`.

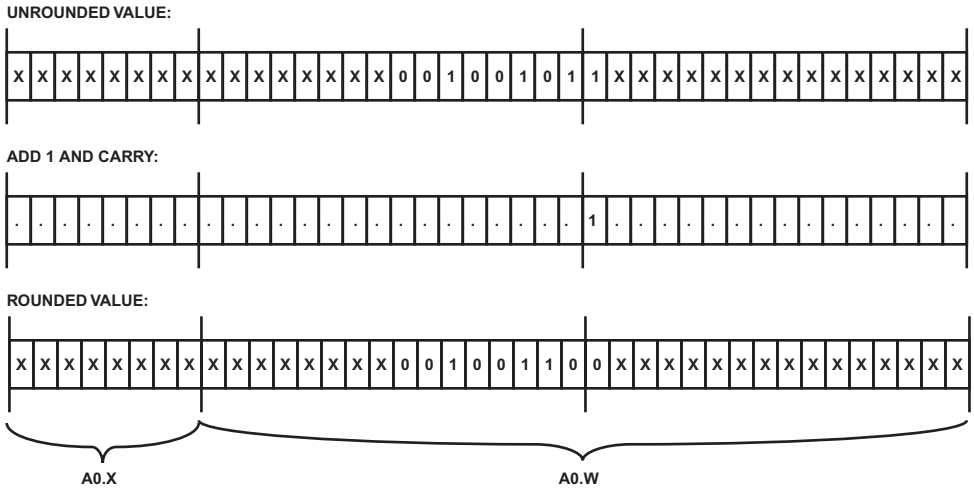


Figure 2-6. Typical Unbiased Multiplier Rounding

The compensation to avoid net bias becomes visible when all lower 15 bits are 0 and bit 15 is 1 (the midpoint value) as shown in Figure 2-7.

In Figure 2-7, `A0` bit 16 is forced to 0. This algorithm is employed on every rounding operation, but is evident only when the bit patterns shown in the lower 16 bits of the next example are present.

Biased Rounding

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

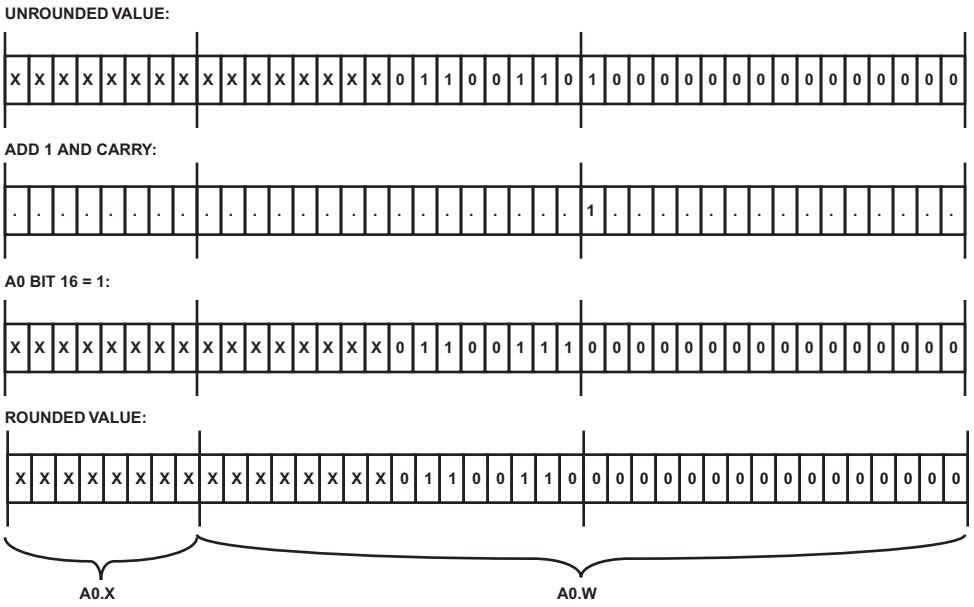


Figure 2-7. Avoiding Net Bias in Unbiased Multiplier Rounding

The RND_MOD bit in the ASTAT register enables biased rounding. When the RND_MOD bit is cleared, the RND option in multiplier instructions uses the normal, unbiased rounding operation, as discussed in "Unbiased Rounding" on page 2-19.

When the RND_MOD bit is set (=1), the processor uses biased rounding instead of unbiased rounding. When operating in biased rounding mode, all rounding operations with A0.L/A1.L set to 0x8000 round up, rather than only rounding odd values up. For an example of biased rounding, see Table 2-7.

Table 2-7. Biased Rounding in Multiplier Operation

A0/A1 Before RND	Biased RND Result	Unbiased RND Result
0x00 0000 8000	0x00 0001 8000	0x00 0000 0000
0x00 0001 8000	0x00 0002 0000	0x00 0002 0000
0x00 0000 8001	0x00 0001 0001	0x00 0001 0001
0x00 0001 8001	0x00 0002 0001	0x00 0002 0001
0x00 0000 7FFF	0x00 0000 FFFF	0x00 0000 FFFF
0x00 0001 7FFF	0x00 0001 FFFF	0x00 0001 FFFF

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

Truncation

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

Special Rounding Instructions

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

For example:

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

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

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

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

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

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

Using Computational Status

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

ASTAT Register

Figure 2-8 describes the Arithmetic Status (ASTAT) register. The processor updates the status bits in ASTAT, indicating the status of the most recent ALU, multiplier, or shifter operation.

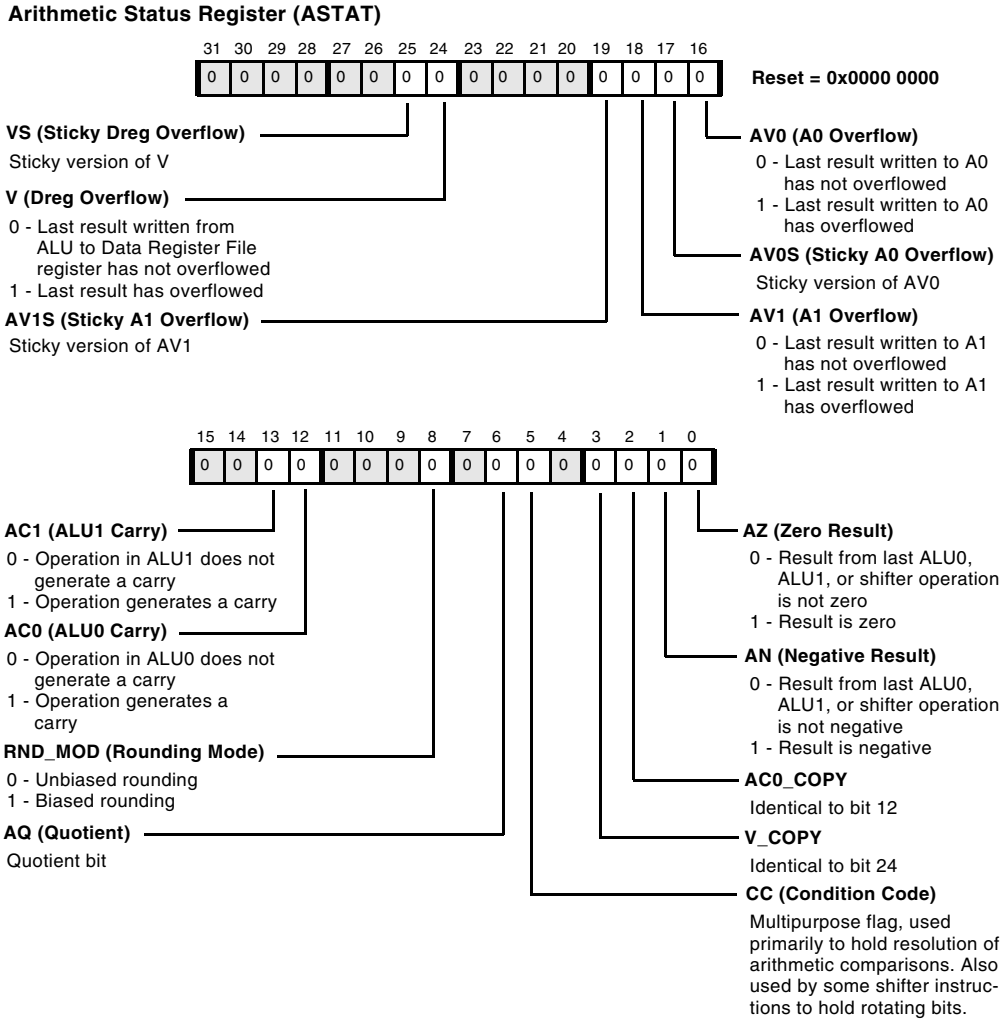


Figure 2-8. Arithmetic Status Register

Arithmetic Logic Unit (ALU)

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

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

ALU Operations

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

Table 2-8 describes the possible inputs and outputs of each ALU.

Table 2-8. Inputs and Outputs of Each ALU

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

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

Single 16-Bit Operations

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

For example:

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

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

Dual 16-Bit Operations

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

For example:

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

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

The instruction also subtracts the 16-bit contents of `R2.L` (`R2` low half) from the contents of `R1.L` (`R1` low half) and deposits the result in `R3.L` (`R3` low half) with saturation (see Figure 2-10).

Quad 16-Bit Operations

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

For example:

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

performs four operations:

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

Explicitly, the four equivalent instructions are:

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

Single 32-Bit Operations

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

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

Instructions may not intermingle Pointer registers with Data registers.

For example:

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

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

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

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

Dual 32-Bit Operations

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

For example:

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

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

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

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

For example:

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

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

ALU Instruction Summary

Table 2-9 lists the ALU instructions. For more information about assembly language syntax and the effect of ALU instructions on the status flags, see the *Blackfin Processor Programming Reference*.

In Table 2-9, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Preg denotes any Pointer register, FP, or SP register.
- Dreg_lo_hi denotes any 16-bit register half in any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.

- imm7 denotes a signed, 7-bit wide, immediate value.
- A*n* denotes either ALU Result register `A0` or `A1`.
- DIVS denotes a Divide Sign primitive.
- DIVQ denotes a Divide Quotient primitive.
- MAX denotes the maximum, or most positive, value of the source registers.
- MIN denotes the minimum value of the source registers.
- ABS denotes the absolute value of the upper and lower halves of a single 32-bit register.
- RND denotes rounding a half word.
- RND12 denotes saturating the result of an addition or subtraction and rounding the result on bit 12.
- RND20 denotes saturating the result of an addition or subtraction and rounding the result on bit 20.
- SIGNBITS denotes the number of sign bits in a number, minus one.
- EXPADJ denotes the lesser of the number of sign bits in a number minus one, and a threshold value.
- * Indicates the flag may be set or cleared, depending on the results of the instruction.
- ** Indicates the flag is cleared.
- – Indicates no effect.
- *d* indicates `AQ` contains the dividend MSB Exclusive-OR divisor MSB.

Table 2-9. ALU Instruction Summary

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
Preg = Preg + Preg ;	–	–	–	–	–	–	–
Preg += Preg ;	–	–	–	–	–	–	–
Preg –= Preg ;	–	–	–	–	–	–	–
Dreg = Dreg + Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg – Dreg (S) ;	*	*	*	–	–	*	–
Dreg = Dreg + Dreg, Dreg = Dreg – Dreg ;	*	*	*	–	–	*	–
Dreg_lo_hi = Dreg_lo_hi + Dreg_lo_hi ;	*	*	*	–	–	*	–
Dreg_lo_hi = Dreg_lo_hi – Dreg_lo_hi (S) ;	*	*	*	–	–	*	–
Dreg = Dreg +|+ Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg +|– Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg –|+ Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg –|– Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg +|+Dreg, Dreg = Dreg –|– Dreg ;	*	*	–	–	–	*	–
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg ;	*	*	–	–	–	*	–
Dreg = A*n* + A*n*, Dreg = A*n* – A*n* ;	*	*	*	–	–	*	–
Dreg += imm7 ;	*	*	*	–	–	*	–
Preg += imm7 ;	–	–	–	–	–	–	–
Dreg = (A0 += A1) ;	*	*	*	*	–	*	–
Dreg_lo_hi = (A0 += A1) ;	*	*	*	*	–	*	–

Table 2-9. ALU Instruction Summary (Cont'd)

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
A0 += A1 ;	*	*	*	*	–	–	–
A0 –= A1 ;	*	*	*	*	–	–	–
DIVS (Dreg, Dreg) ;	*	*	*	*	–	–	d
DIVQ (Dreg, Dreg) ;	*	*	*	*	–	–	d
Dreg = MAX (Dreg, Dreg) (V) ;	*	*	–	–	–	**/–	–
Dreg = MIN (Dreg, Dreg) (V) ;	*	*	–	–	–	**/–	–
Dreg = ABS Dreg (V) ;	*	**	–	–	–	*	–
A*n* = ABS A*n* ;	*	**	–	*	*	*	–
A*n* = ABS A*n*, A*n* = ABS A*n* ;	*	**	–	*	*	*	–
A*n* = –A*n* ;	*	*	*	*	*	*	–
A*n* = –A*n*, A*n* =– A*n* ;	*	*	*	*	*	*	–
A*n* = A*n* (S) ;	*	*	–	*	*	–	–
A*n* = A*n* (S), A*n* = A*n* (S) ;	*	*	–	*	*	–	–
Dreg_lo_hi = Dreg (RND) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg + Dreg (RND12) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg – Dreg (RND12) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg + Dreg (RND20) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg – Dreg (RND20) ;	*	*	–	–	–	*	–

Table 2-9. ALU Instruction Summary (Cont'd)

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

ALU Data Flow Details

Figure 2-9 shows a more detailed diagram of the Arithmetic Units and the Data Register File, which appears in Figure 2-1.

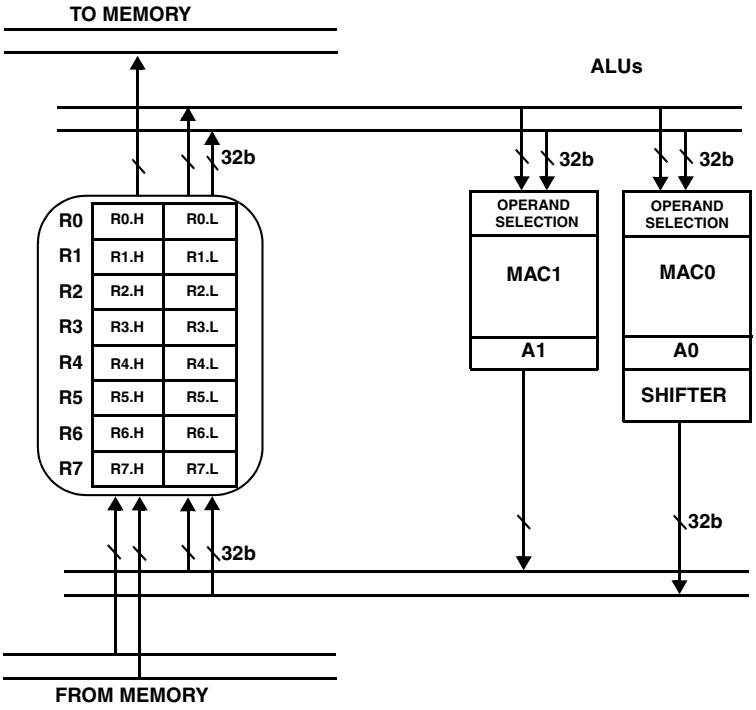


Figure 2-9. Register Files and ALUs

ALU0 is described here for convenience. ALU1 is very similar—a subset of ALU0.

Each ALU performs 40-bit addition for the accumulation of the multiplier results, as well as 32-bit and dual 16-bit operations. Each ALU has two 32-bit input ports that can be considered a pair of 16-bit operands or a

single 32-bit operand. For single 16-bit operations, any of the four possible 16-bit operands may be used with any of the other 16-bit operands presented at the input to the ALU.

As shown in Figure 2-10, for dual 16-bit operations, the high halves and low halves are paired, providing four possible combinations of addition and subtraction.

```
(A) H + H, L + L    (B) H + H, L - L
(C) H - H, L + L    (D) H - H, L - L
```

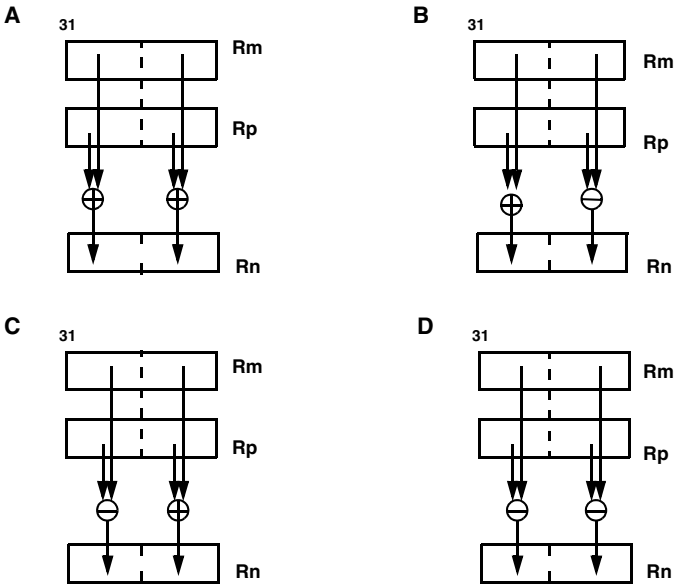


Figure 2-10. Dual 16-Bit ALU Operations

Dual 16-Bit Cross Options

For dual 16-bit operations, the results may be crossed. “Crossing the results” changes the location in the result register for the result of a calculation. Usually, the result from the high side calculation is placed in the high half of the result register, and the result from the low side calculation is placed in the low half of the result register. With the cross option, the high result is placed in the low half of the destination register, and the low result is placed in the high half of the destination register (see Figure 2-11). This is particularly useful when dealing with complex math and portions of the Fast Fourier Transform (FFT). The cross option applies to ALU0 only.

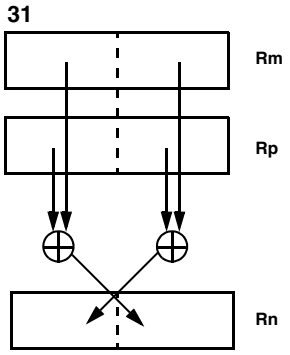


Figure 2-11. Cross Options for Dual 16-Bit ALU Operations

ALU Status Signals

Each ALU generates six status signals: the zero (AZ) status, the negative (AN) status, the carry (ACn) status, the sticky overflow (AVnS) status, the immediate overflow (AVn) status, and the quotient (AQ) status. All arithmetic status signals are latched into the arithmetic status register (ASTAT) at the end of the cycle. For the effect of ALU instructions on the status flags, see Table 2-9.

Depending on the instruction, the inputs can come from the Data Register File, the Pointer Register File, or the Arithmetic Result registers. Arithmetic on 32-bit operands directly support multiprecision operations in the ALU.

ALU Division Support Features

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

The division can be either signed or unsigned, but both the dividend and divisor must be of the same type. Details about using division and programming examples are available in the *Blackfin Processor Programming Reference*.

Special SIMD Video ALU Operations

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

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

For more information about the operation of these instructions, see the *Blackfin Processor Programming Reference*.

Multiply Accumulators (Multipliers)

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

Multiplier fixed-point instructions operate on 16-bit fixed-point data and produce 32-bit results that may be added or subtracted from a 40-bit accumulator.

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

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

Multiplier Operation

Each multiplier has two 32-bit inputs from which it derives the two 16-bit operands. For single multiply and accumulate instructions, these operands can be any Data registers in the Data Register File. Each multiplier can accumulate results in its Accumulator register, `A1` or `A0`. The accumulator results can be saturated to 32 or 40 bits. The multiplier result can also be written directly to a 16- or 32-bit destination register with optional rounding.

Each multiplier instruction determines whether the inputs are either both in integer format or both in fractional format. The format of the result matches the format of the inputs. In MAC0, both inputs are treated as signed or unsigned. In MAC1, there is a mixed-mode option.

If both inputs are fractional and signed, the multiplier automatically shifts the result left one bit to remove the redundant sign bit. Unsigned fractional, integer, and mixed modes do not perform a shift for sign bit correction. Multiplier instruction options specify the data format of the inputs. See "Multiplier Instruction Options" on page 2-42 for more information.

Placing Multiplier Results in Multiplier Accumulator Registers

As shown in Figure 2-9, each multiplier has a dedicated accumulator, `A0` or `A1`. Each Accumulator register is divided into three sections—`A0.L/A1.L` (bits 15:0), `A0.H/A1.H` (bits 31:16), and `A0.X/A1.X` (bits 39:32).

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

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

Rounding or Saturating Multiplier Results

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

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

adding 0x8000 to the accumulator (for MAC) or multiply result (for mult) and then saturating to 32-bits. For more information, see “Rounding Multiplier Results” on page 2-18.

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

Saturating Multiplier Results on Overflow

The following bits in ASTAT indicate multiplier overflow status:

- Bit 16 (AV0) and bit 18 (AV1) record overflow condition (whether the result has overflowed 32 bits) for the A0 and A1 accumulators, respectively. If the bit is cleared (=0), no overflow or underflow has occurred. If the bit is set (=1), an overflow or underflow has occurred. The AV0S and AV1S bits are sticky bits.
- Bit 24 (V) and bit 25 (VS) are set if overflow occurs in extracting the accumulator result to a register.

Multiplier Instruction Summary

Table 2-10 lists the multiplier instructions. For more information about assembly language syntax and the effect of multiplier instructions on the status flags, see the *Blackfin Processor Programming Reference*.

In Table 2-10, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Dreg_lo_hi denotes any 16-bit register half in any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.

- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- A*n* denotes either MAC Accumulator register `A0` or `A1`.
- * Indicates the flag may be set or cleared, depending on the results of the instruction.
- – Indicates no effect.

Multiplier instruction options are described on page 2-42.

Table 2-10. Multiplier Instruction Summary

Instruction	ASTAT Status Flags		
	AV0 AV0S	AV1 AV1S	V V_COPY VS
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ;	–	–	*
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi ;	–	–	*
Dreg = Dreg_lo_hi * Dreg_lo_hi ;	–	–	*
A*n* = Dreg_lo_hi * Dreg_lo_hi ;	*	*	–
A*n* += Dreg_lo_hi * Dreg_lo_hi ;	*	*	–
An –= Dreg_lo_hi * Dreg_lo_hi ;	*	*	–
Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (A*n* = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (A*n* += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*

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

Instruction	ASTAT Status Flags		
	AV0 AV0S	AV1 AV1S	V V_COPY VS
Dreg = (A*n* –= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg *= Dreg ;	–	–	–

Multiplier Instruction Options

The following descriptions of multiplier instruction options provide an overview. Not all options are available for all instructions. For information about how to use these options with their respective instructions, see the *Blackfin Processor Programming Reference*.

default — No option; input data is signed fraction.

(IS) — Input data operands are signed integer. No shift correction is made.

(FU) — Input data operands are unsigned fraction. No shift correction is made.

(IU) — Input data operands are unsigned integer. No shift correction is made.

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

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

(ISS2) If multiplying and accumulating to a register:

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

If multiplying and accumulating to a half register:

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

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

(W32) Input data operands are signed fraction with no extension bits in the Accumulators at 32 bits. Left-shift correction of the product is performed, as required. This option is used for legacy GSM speech vocoder algorithms written for 32-bit Accumulators. For this option only, this special case applies: `0x8000 x 0x8000 = 0x7FFF`.

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

Multiplier Data Flow Details

Figure 2-12 shows the Register files and ALUs, along with the multiplier/accumulators.

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

One of the operands must be selected from the low half or the high half of one 32-bit word. The other operand must be selected from the low half or the high half of the other 32-bit word. Thus, each MAC is presented with four possible input operand combinations. The two 32-bit words can pcontain the same register information, giving the options for squaring and multiplying the high half and low half of the same register. Figure 2-13 show these possible combinations.

The 32-bit product is passed to a 40-bit adder/subtracter, which may add or subtract the new product from the contents of the Accumulator Result register or pass the new product directly to the Data Register File Results

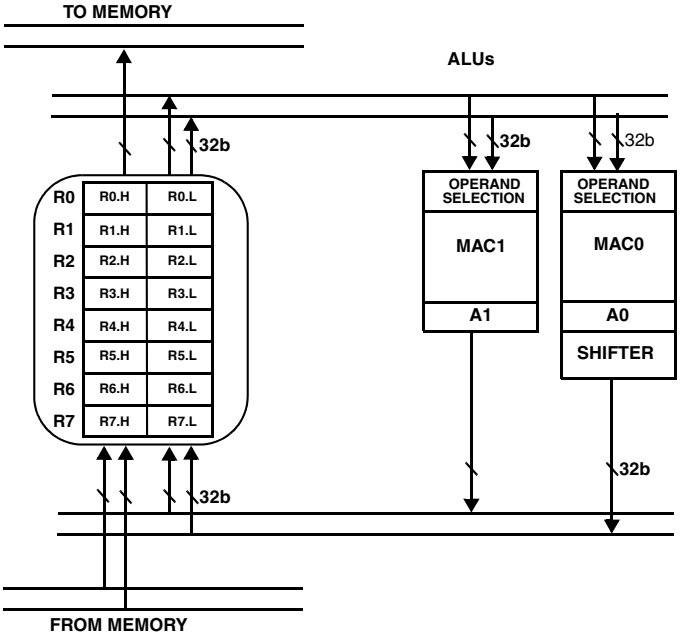


Figure 2-12. Register Files and ALUs

register. For results, the A0 and A1 registers are 40 bits wide. Each of these registers consists of smaller 32- and 8-bit registers—A0.W, A1.W, A0.X, and A1.X.

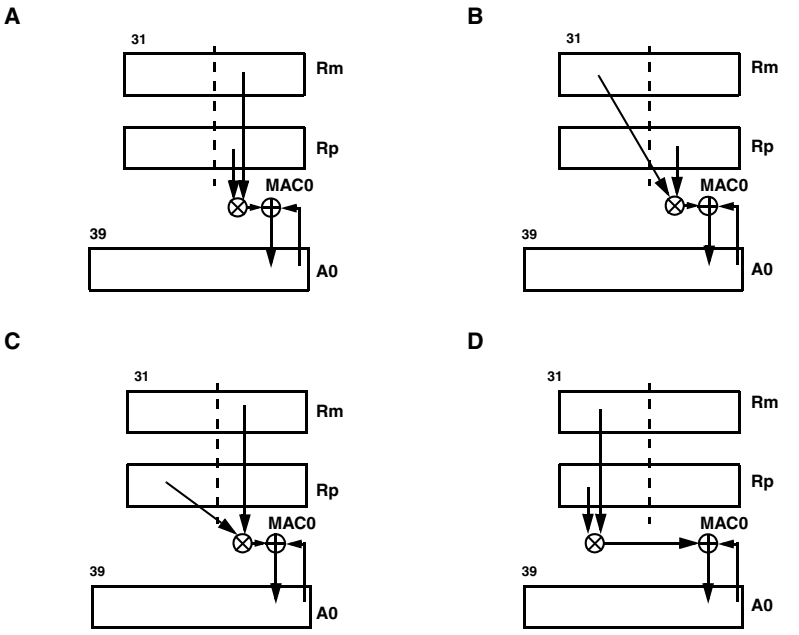


Figure 2-13. Four Possible Combinations of MAC Operations

Some example instructions:

```
A0 = R3.L * R4.H ;
```

In this instruction, the `MAC0` multiplier/accumulator performs a multiply and puts the result in the Accumulator register.

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

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

Multiply Without Accumulate

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

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

- If the multiplication uses fractional operands or the `IH` option, then the high half of the result is extracted and stored in the 16-bit destination registers (see Figure 2-14).
- If the multiplication uses integer operands, then the low half of the result is extracted and stored in the 16-bit destination registers. These extractions provide the most useful information in the resultant 16-bit word for the data type chosen (see Figure 2-15).

For example, this instruction uses fractional, unsigned operands:

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

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

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

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

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

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

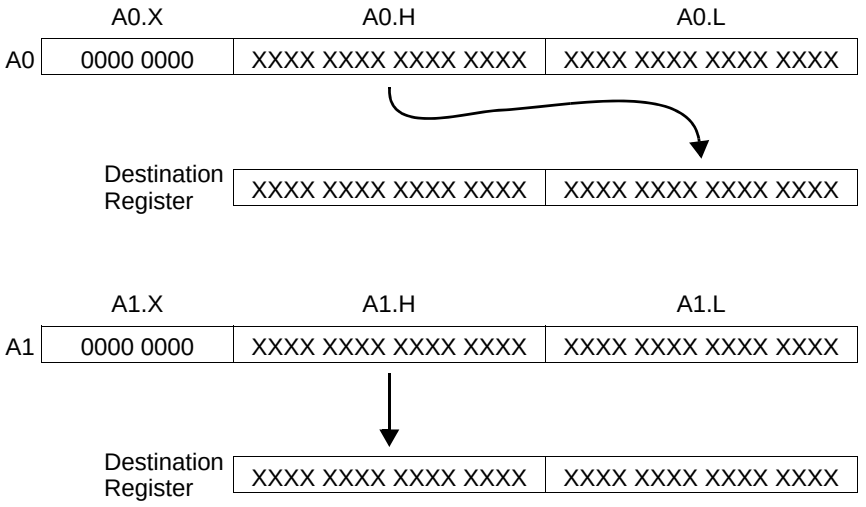


Figure 2-14. Multiplication of Fractional Operands

Special 32-Bit Integer MAC Instruction

The processor supports a multicycle 32-bit MAC instruction:

```
Dreg *= Dreg
```

The single instruction multiplies two 32-bit integer operands and provides a 32-bit integer result, destroying one of the input operands.

The instruction takes multiple cycles to execute. Refer to the product data sheet and the *Blackfin Processor Programming Reference* for more information about the exact operation of this instruction. This macro function is interruptable and does not modify the data in either Accumulator register `A0` or `A1`.

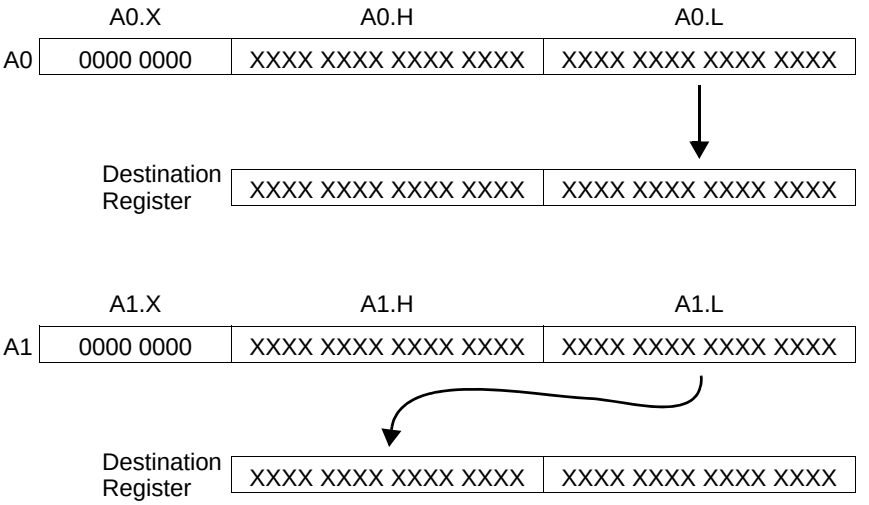


Figure 2-15. Multiplication of Integer Operands

Dual MAC Operations

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

An example of a dual multiply and accumulate instruction is

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

This instruction represents two multiply and accumulate operations.

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

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

For example:

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

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

The operand type determines the correct bits to extract from the Accumulator and deposit in the 16-bit destination register. See "Multiply Without Accumulate" on page 2-47.

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

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

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

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

Barrel Shifter (Shifter)

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

Shifter Operations

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

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

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

Arithmetic shift, logical shift, and rotate instructions can obtain the shift argument from a register or directly from an immediate value in the instruction. For details about shifter related instructions, see “Shifter Instruction Summary” on page 2-55.

Two-Operand Shifts

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

Immediate Shifts

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

The following example shows the input value downshifted.

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

results in

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

The following example shows the input value upshifted.

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

results in

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

Register Shifts

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

The following example shows the input value upshifted.

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

results in

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

Three-Operand Shifts

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

Immediate Shifts

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

The following example shows the input value downshifted.

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

results in

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

The following example shows the input value upshifted.

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

results in

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

Register Shifts

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

The following example shows the input value upshifted.

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

results in

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

The following example shows the input value rotated. Assume the Condition Code (`CC`) bit is set to 0. For more information about `CC`, see "Condition Code Flag" on page 4-13.

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

results in

```
R1 contains BCDE F125 ;
```

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

Bit Test, Set, Clear, Toggle

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

The following examples show a variety of operations.

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

Field Extract and Field Deposit

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

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

Shifter Instruction Summary

Table 2-11 lists the shifter instructions. For more information about assembly language syntax and the effect of shifter instructions on the status flags, see the *Blackfin Processor Programming Reference*.

In Table 2-11, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.

- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- * Indicates the flag may be set or cleared, depending on the results of the instruction.
- * 0 Indicates versions of the instruction that send results to Accumulator `A0` set or clear `AV0`.
- * 1 Indicates versions of the instruction that send results to Accumulator `A1` set or clear `AV1`.
- ** Indicates the flag is cleared.
- *** Indicates `CC` contains the latest value shifted into it.
- – Indicates no effect.

Table 2-11. Shifter Instruction Summary

Instruction	ASTAT Status Flag						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
BITCLR (Dreg, uimm5) ;	*	*	**	–	–	–	**/–
BITSET (Dreg, uimm5) ;	**	*	**	–	–	–	**/–
BITTGL (Dreg, uimm5) ;	*	*	**	–	–	–	**/–
CC = BITTST (Dreg, uimm5) ;	–	–	–	–	–	*	–
CC = !BITTST (Dreg, uimm5) ;	–	–	–	–	–	*	–
Dreg = DEPOSIT (Dreg, Dreg) ;	*	*	**	–	–	–	**/–
Dreg = EXTRACT (Dreg, Dreg) ;	*	*	**	–	–	–	**/–

Table 2-11. Shifter Instruction Summary (Cont'd)

Instruction	ASTAT Status Flag						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
BITMUX (Dreg, Dreg, A0) ;	–	–	–	–	–	–	–
Dreg_lo = ONES Dreg ;	–	–	–	–	–	–	–
Dreg = PACK (Dreg_lo_hi, Dreg_lo_hi);	–	–	–	–	–	–	–
Dreg >>>= uimm5 ;	*	*	–	–	–	–	**/–
Dreg >>= uimm5 ;	*	*	–	–	–	–	**/–
Dreg <<= uimm5 ;	*	*	–	–	–	–	**/–
Dreg = Dreg >>> uimm5 ;	*	*	–	–	–	–	**/–
Dreg = Dreg >> uimm5 ;	*	*	–	–	–	–	**/–
Dreg = Dreg << uimm5 ;	*	*	–	–	–	–	*
Dreg = Dreg >>> uimm4 (V) ;	*	*	–	–	–	–	**/–
Dreg = Dreg >> uimm4 (V) ;	*	*	–	–	–	–	**/–
Dreg = Dreg << uimm4 (V) ;	*	*	–	–	–	–	*
A*n* = A*n* >>> uimm5 ;	*	*	–	** 0/–	** 1/–	–	–
A*n* = A*n* >> uimm5 ;	*	*	–	** 0/–	** 1/–	–	–
A*n* = A*n* << uimm5 ;	*	*	–	* 0	* 1	–	–
Dreg_lo_hi = Dreg_lo_hi >>> uimm4 ;	*	*	–	–	–	–	**/–
Dreg_lo_hi = Dreg_lo_hi >> uimm4 ;	*	*	–	–	–	–	**/–
Dreg_lo_hi = Dreg_lo_hi << uimm4 ;	*	*	–	–	–	–	*
Dreg >>>= Dreg ;	*	*	–	–	–	–	**/–

Table 2-11. Shifter Instruction Summary (Cont'd)

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

Table 2-11. Shifter Instruction Summary (Cont'd)

Instruction	ASTAT Status Flag						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
Preg = Preg + (Preg << 1) ;	–	–	–	–	–	–	–
Preg = Preg + (Preg << 2) ;	–	–	–	–	–	–	–

3 OPERATING MODES AND STATES

The processor supports the following three processor modes:

- User mode
- Supervisor mode
- Emulation mode

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

User mode is considered the domain of application programs. Supervisor mode and Emulation mode are usually reserved for the kernel code of an operating system.

The processor mode is determined by the Event Controller. When servicing an interrupt, a nonmaskable interrupt (NMI), or an exception, the processor is in Supervisor mode. When servicing an emulation event, the processor is in Emulation mode. When not servicing any events, the processor is in User mode.

The current processor mode may be identified by interrogating the `IPEND` memory-mapped register (MMR), as shown in Table 3-1.

MMRs cannot be read while the processor is in User mode.

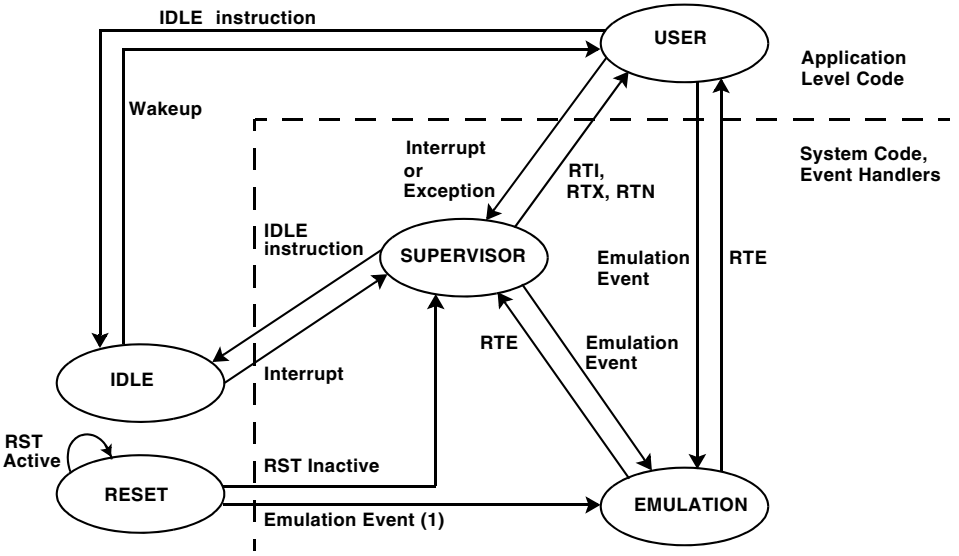
Table 3-1. Identifying the Current Processor Mode

Event	Mode	IPEND
Interrupt	Supervisor	≥ 0x10 but IPEND[0], IPEND[1], IPEND[2], and IPEND[3] = 0.
Exception	Supervisor	≥ 0x08 The core is processing an exception event if IPEND[0] = 0, IPEND[1] = 0, IPEND[2] = 0, IPEND[3] = 1, and IPEND[15:4] are 0's or 1's.
NMI	Supervisor	≥ 0x04 The core is processing an NMI event if IPEND[0] = 0, IPEND[1] = 0, IPEND[2] = 1, and IPEND[15:2] are 0's or 1's.
Reset	Supervisor	= 0x02 As the reset state is exited, IPEND is set to 0x02, and the reset vector runs in Supervisor mode.
Emulation	Emulator	= 0x01 The processor is in Emulation mode if IPEND[0] = 1, regardless of the state of the remaining bits IPEND[15:1].
None	User	= 0x00

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

- Idle state
- Reset state

Figure 3-1 illustrates the processor modes and states as well as the transition conditions between them.



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

Figure 3-1. Processor Modes and States

User Mode

The processor is in User mode when it is not in Reset or Idle state, and when it is not servicing an interrupt, NMI, exception, or emulation event. User mode is used to process application level code that does not require explicit access to system registers. Any attempt to access restricted system registers causes an exception event. Table 3-2 lists the registers that may be accessed in User mode.

Table 3-2. Registers Accessible in User Mode

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

Protected Resources and Instructions

System resources consist of a subset of processor registers, all MMRs, and a subset of protected instructions. These system and core MMRs are located starting at address 0xFFC0 0000. This region of memory is protected from User mode access. Any attempt to access MMR space in User mode causes an exception.

A list of protected instructions appears in Table 3-3. Any attempt to issue any of the protected instructions from User mode causes an exception event.

Table 3-3. Protected Instructions

Instruction	Description
RTI	Return from Interrupt
RTX	Return from Exception
RTN	Return from NMI
CLI	Disable Interrupts
STI	Enable Interrupts
RAISE	Force Interrupt/Reset
RTE	Return from Emulation Causes an exception only if executed outside Emulation mode

Protected Memory

Additional memory locations can be protected from User mode access. A Cacheability Protection Lookaside Buffer (CPLB) entry can be created and enabled. See "Memory Management Unit" on page 6-46 for further information.

Entering User Mode

When coming out of reset, the processor is in Supervisor mode because it is servicing a reset event. To enter User mode from the Reset state, two steps must be performed. First, a return address must be loaded into the `RETI` register. Second, an `RTI` must be issued. The following example code shows how to enter User mode upon reset.

Example Code to Enter User Mode Upon Reset

Listing 3-1 provides code for entering User mode from reset.

Listing 3-1. Entering User Mode from Reset

```
P1.L = START ;   /* Point to start of user code */
P1.H = START ;
RETI = P1 ;
RTI ;   /* Return from Reset Event */

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

Return Instructions That Invoke User Mode

Table 3-4 provides a summary of return instructions that can be used to invoke User mode from various processor event service routines. When these instructions are used in service routines, the value of the return address must be first stored in the appropriate event `RETx` register. In the

case of an interrupt routine, if the service routine is interruptible, the return address is stored on the stack. For this case, the address can be found by popping the value from the stack into RETI. Once RETI has been loaded, the RTI instruction can be issued.

Note the stack pop is optional. If the RETI register is not pushed/popped, then the interrupt service routine becomes non-interruptible, because the return address is not saved on the stack.

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

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

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

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

Supervisor Mode

The processor services all interrupt, NMI, and exception events in Supervisor mode.

Supervisor mode has full, unrestricted access to all processor system resources, including all emulation resources, unless a CPLB has been configured and enabled. See "Memory Management Unit" on page 6-46 for a further description. Only Supervisor mode can use the register alias USP, which references the User Stack Pointer in memory. This register alias is necessary because in Supervisor mode, SP refers to the kernel stack pointer rather than to the user stack pointer.

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

Non-OS Environments

For non-OS environments, application code should remain in Supervisor mode so that it can access all core and system resources. When $\overline{\text{RESET}}$ is deasserted, the processor initiates operation by servicing the reset event. Emulation is the only event that can pre-empt this activity. Therefore, lower priority events cannot be processed.

One way of keeping the processor in Supervisor mode and still allowing lower priority events to be processed is to set up and force the lowest priority interrupt (IVG15). Events and interrupts are described further in "Events and Sequencing" on page 4-18. After the low priority interrupt has been forced using the RAISE 15 instruction, RETI can be loaded with a return address that points to user code that can execute until IVG15 is issued. After RETI has been loaded, the RTI instruction can be issued to return from the reset event.

The interrupt handler for IVG15 can be set to jump to the application code starting address. An additional RTI is not required. As a result, the processor remains in Supervisor mode because IPEND[15] remains set. At this point, the processor is servicing the lowest priority interrupt. This ensures that higher priority interrupts can be processed.

Example Code for Supervisor Mode Coming Out of Reset

To remain in Supervisor mode when coming out of the Reset state, use code as shown in Listing 3-2.

Listing 3-2. Staying in Supervisor Mode Coming Out of Reset

```
P0.L = LO(EVT15) ;   /* Point to IVG15 in Event Vector Table */
P0.H = HI(EVT15) ;
P1.L = START ;   /* Point to start of User code */

P1.H = START ;
[P0] = P1 ;   /* Place the address of start code in IVG15 of EVT
*/

P0.L = LO(IMASK) ;

R0 = [P0] ;
R1.L = EVT_IVG15 & 0xFFFF ;

R0 = R0 | R1 ;
[P0] = R0 ;   /* Set (enable) IVG15 bit in Interrupt Mask Register
*/

RAISE 15 ;   /* Invoke IVG15 interrupt */
P0.L = WAIT_HERE ;
P0.H = WAIT_HERE ;
RETI = P0 ;   /* RETI loaded with return address */
```

```
RTI ;   /* Return from Reset Event */
WAIT_HERE :   /* Wait here till IVG15 interrupt is serviced */

JUMP WAIT_HERE ;

START:   /* IVG15 vectors here */
[--SP] = RETI ;   /* Enables interrupts and saves return address
to stack */
```

Emulation Mode

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

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

The processor remains in Emulation mode until the emulation service routine executes an `RTE` instruction. If no interrupts are pending when the `RTE` instruction executes, the processor switches to User mode. Otherwise, the processor switches to Supervisor mode to service the interrupt.

Emulation mode is the highest priority mode, and the processor has unrestricted access to all system resources.

Idle State

Idle state stops all processor activity at the user's discretion, usually to conserve power during lulls in activity. No processing occurs during the Idle state. The Idle state is invoked by a sequential `IDLE` instruction. The `IDLE` instruction notifies the processor hardware that the Idle state is requested.

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

In Listing 3-3, core interrupts are disabled and the IDLE instruction is executed. When all the pending processes have completed, the core disables its clocks. Since interrupts are disabled, Idle state can be terminated only by asserting a WAKEUP signal. For more information, see "SIC_IWR Register" on page 4-26. (While not required, an interrupt could also be enabled in conjunction with the WAKEUP signal.)

When the WAKEUP signal is asserted, the processor wakes up, and the STI instruction enables interrupts again.

Example Code for Transition to Idle State

To transition to the Idle state, use code shown in Listing 3-3.

Listing 3-3. Transitioning to Idle State

```
CLI R0 ;   /* disable interrupts */
IDLE ;   /* drain pipeline and send core into IDLE state */
STI R0 ;   /* re-enable interrupts after wakeup */
```

Reset State

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

The processor remains in the Reset state as long as external logic asserts the external $\overline{\text{RESET}}$ signal. Upon deassertion, the processor completes the reset sequence and switches to Supervisor mode, where it executes code found at the reset event vector.

Software in Supervisor or Emulation mode can invoke the Reset state without involving the external $\overline{\texttt{RESET}}$ signal. This can be done by issuing the Reset version of the `RAISE` instruction.

Application programs in User mode cannot invoke the Reset state, except through a system call provided by an operating system kernel. Table 3-5 summarizes the state of the processor upon reset.

Table 3-5. Processor State Upon Reset

Item	Description of Reset State
Core	
Operating Mode	Supervisor mode in reset event, clocks stopped
Rounding Mode	Unbiased rounding
Cycle Counters	Disabled, zero
DAG Registers (I, L, B, M)	Random values (must be cleared at initialization)
Data and Address Registers	Random values (must be cleared at initialization)
IPEND, IMASK, ILAT	Cleared, interrupts globally disabled with IPEND bit 4
CPLBs	Disabled
L1 Instruction Memory	SRAM (cache disabled)
L1 Data Memory	SRAM (cache disabled)
Cache Validity Bits	Invalid
System	
Booting Methods	Determined by the values of BMODE pins at reset
MSEL Clock Frequency	Reset value = 10
PLL Bypass Mode	Disabled
VCO/Core Clock Ratio	Reset value = 1
VCO/System Clock Ratio	Reset value = 5
Peripheral Clocks	Disabled

System Reset and Powerup

Table 3-6 describes the five types of resets. Note all resets, except System Software, reset the core.

Table 3-6. Resets

Reset	Source	Result
Hardware Reset	The $\overline{\text{RESET}}$ pin causes a hardware reset.	Resets both the core and the peripherals, including the Dynamic Power Management Controller (DPMC). Resets the No Boot on Software Reset bit in SYSCR. For more information, see "SYSCR Register" on page 3-14.
System Software Reset	Writing b#111 to bits [2:0] in the system MMR SWRST at address 0xFFC0 0100 causes a System Software reset.	Resets only the peripherals, excluding the RTC (Real-Time Clock) block and most of the DPMC. The DPMC resets only the No Boot on Software Reset bit in SYSCR. Does not reset the core. Does not initiate a boot sequence.
Watchdog Timer Reset	Programming the watchdog timer appropriately causes a Watchdog Timer reset.	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The Software Reset register (SWRST) can be read to determine whether the reset source was the watchdog timer.

Table 3-6. Resets (Cont'd)

Reset	Source	Result
Core Double-Fault Reset	If the core enters a double-fault state, and the Core Double Fault Reset Enable bit (DOUBLE_FAULT) is set in the SWRST register, then a software reset occurs.	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The SWRST register can be read to determine whether the reset source was Core Double Fault.
Core-Only Software Reset	This reset is caused by executing a RAISE1 instruction or by setting the Software Reset (SYSRST) bit in the core Debug Control register (DBGCTL) via emulation software through the JTAG port. The DBGCTL register is not visible to the memory map.	Resets only the core. The peripherals do not recognize this reset.

Hardware Reset

The processor chip reset is an asynchronous reset event. The $\overline{\texttt{RESET}}$ input pin must be deasserted to perform a hardware reset. For more information, see the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet*.

A hardware-initiated reset results in a system-wide reset that includes both core and peripherals. After the $\overline{\texttt{RESET}}$ pin is deasserted, the processor ensures that all asynchronous peripherals have recognized and completed a reset. After the reset, the processor transitions into the Boot mode sequence configured by the BMODE state.

The BMODE[1:0] pins are dedicated mode control pins. No other functions are shared with these pins, and they may be permanently strapped by tying them directly to either V_{DD} or V_{SS}. The pins and the corresponding bits

in SYSCR configure the Boot mode that is employed after hardware reset or System Software reset. See "Reset" on page 4-39, and Table 4-11, "Events That Cause Exceptions," on page 4-43 for further information.

SYSCR Register

The values sensed from the BMODE[1:0] pins are latched into the System Reset Configuration register (SYSCR) upon the deassertion of the RESET pin. The values are made available for software access and modification after the hardware reset sequence. Software can modify only the No Boot on Software Reset bit.

The various configuration parameters are distributed to the appropriate destinations from SYSCR (see Figure 3-2).

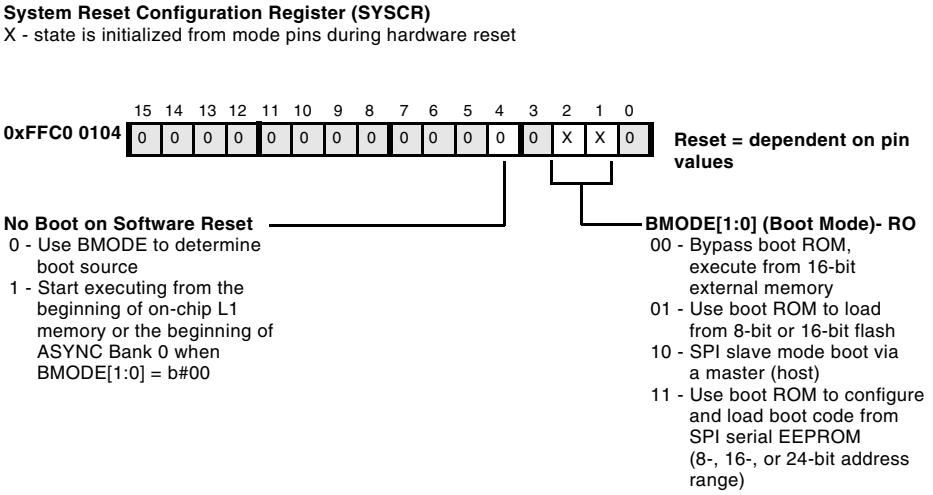


Figure 3-2. System Reset Configuration Register

Software Resets and Watchdog Timer

A software reset may be initiated in three ways:

- By the watchdog timer, if appropriately configured
- By setting the System Software Reset field in the Software Reset register (see Figure 3-3)
- By the `RAISE1` instruction

The watchdog timer resets both the core and the peripherals. A System Software reset results in a reset of the peripherals without resetting the core and without initiating a booting sequence.

The System Software reset must be performed while executing from Level 1 memory (either as cache or as SRAM).

When L1 instruction memory is configured as cache, make sure the System Software reset sequence has been read into the cache.

After either the watchdog or System Software reset is initiated, the processor ensures that all asynchronous peripherals have recognized and completed a reset.

For a reset generated by the watchdog timer, the processors transitions into the Boot mode sequence. The Boot mode is configured by the state of the `BMODE` and the No Boot on Software Reset control bits.

If the No Boot on Software Reset bit in `SYSCR` is cleared, the reset sequence is determined by the `BMODE[1:0]` control bits.

SWRST Register

A software reset can be initiated by setting the System Software Reset field in the Software Reset register (`SWRST`). Bit 15 indicates whether a software reset has occurred since the last time `SWRST` was read. Bit 14 and Bit 13,

respectively, indicate whether the Software Watchdog Timer or a Core Double Fault has generated a software reset. Bits [15:13] are read-only and cleared when the register is read. Bits [3:0] are read/write.

When the BMODE pins are not set to b#00 and the No Boot on Software Reset bit in SYSCR is set, the processor starts executing from the start of on-chip L1 memory. In this configuration, the core begins fetching instructions from the beginning of on-chip L1 memory.

When the BMODE pins are set to b#00 the core begins fetching instructions from address 0x2000 0000 (the beginning of ASYNC Bank 0).

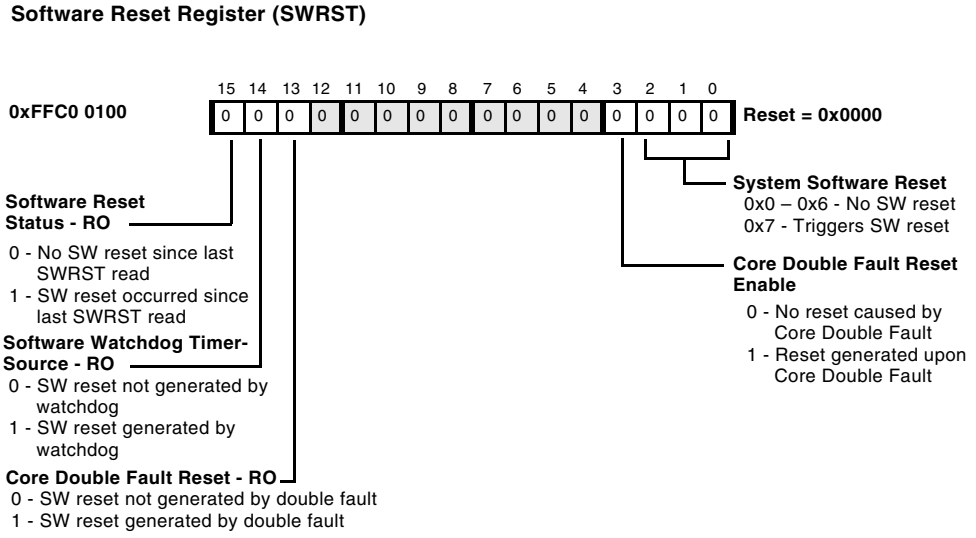


Figure 3-3. Software Reset Register

Core-Only Software Reset

A Core-Only Software reset is initiated by executing the `RAISE 1` instruction or by setting the Software Reset (`SYSRST`) bit in the core Debug Control register (`DBGCTL`) via emulation software through the JTAG port. (`DBGCTL` is not visible to the memory map.)

A Core-Only Software reset affects only the state of the core. Note the system resources may be in an undetermined or even unreliable state, depending on the system activity during the reset period.

Core and System Reset

To perform a system and core reset, use the code sequence shown in Listing 3-4. As described in the code comments, the system soft reset takes five system clock cycles to complete, so a delay loop is needed. This code must reside in L1 memory for the system soft reset to work properly.

Listing 3-4. Core and System Reset

```
/* Issue soft reset */
P0.L = LO(SWRST) ;
P0.H = HI(SWRST) ;
R0.L = 0x0007 ;
W[P0] = R0 ;
SSYNC ;

/* ********
Wait for system reset to complete (needs to be five SCLKs).
Assuming a worst case CCLK:SCLK ratio (15:1), use 5*15 = 75 as
the loop count.
/* ********
```

```
P1 = 75 ;
LSETUP(start, end) LC0 = P1 ;
   start:
   end:
     NOP ;

/* Clear soft reset */
P0.L = LO(SWRST) ;
P0.H = HI(SWRST) ;
R0.L = 0x0000 ;
W[P0] = R0 ;
SSYNC ;

/* Core reset - forces reboot */

RAISE 1 ;
```

Booting Methods

The internal boot ROM includes a small boot kernel that can either be bypassed or used to load user code from an external memory device. See Table 4-10, “Reset Vector Addresses,” on page 4-40 for further information. The boot kernel reads the `BMODE[1:0]` pin state at reset to identify the download source (see Table 4-7 on page 4-24). When in Boot Mode 0, the processor is set to execute from 16-bit wide external memory at address 0x2000 0000 (ASYNC Bank 0).

Several boot methods are available in which user code can be loaded from an external memory device or a host device (as in the case of SPI slave mode booting). For these modes, the boot kernel sets up the selected peripheral based on the `BMODE[1:0]` pin settings.

For each Boot mode, user code read in from the memory device is placed at the starting location of L1 memory. Additional sections are read into internal memory as specified within headers in the loader file. The boot

kernel terminates the boot process with a jump to the start of the L1 instruction memory space. The processor then begins execution from this address.

If booting from Serial Peripheral Interface (SPI), general-purpose flag pin 2 is used as the SPI-chip select. This line must be connected for proper operation.

A Core-Only Software reset also vectors the core to the boot ROM. Only the core is reset with the Core-Only Software reset; this reset does not affect the rest of the system. The boot ROM kernel detects a No Boot on Software Reset condition in SYSCR to avoid initiating a download. If this bit is set on a software reset, the processor skips the normal boot sequence and jumps to the beginning of L1 memory and begins execution.

The boot kernel assumes these conditions for the Flash Boot mode (BMODE = 01):

- Asynchronous Memory Bank (AMB) 0 enabled
- 16-bit packing for AMB 0 enabled
- Bank 0 RDY is set to active high
- Bank 0 hold time (read/write deasserted to $\overline{\text{AOE}}$ deasserted) = 3 cycles
- Bank 0 read/write access times = 15 cycles

For SPI master mode boot (BMODE = 11), the boot kernel assumes that the SPI baud rate is 500 kHz. SPI serial EEPROMs that are 8-bit, 16-bit, and 24-bit addressable are supported. The SPI uses the PF2 output pin to select a single SPI EEPROM device. The SPI controller submits successive read

commands at addresses 0x00, 0x0000, and 0x000000 until a valid 8-, 16-, or 24-bit addressable EEPROM is detected. It then begins clocking data into the beginning of L1 instruction memory.

The MISO pin must be pulled high for SPI master mode booting (BMODE = 11).

For each of the boot modes, 10-byte headers are first read from an external memory device. The header specifies the number of bytes to be transferred and the memory destination address. Once all blocks are loaded, program execution commences from the start of L1 instruction SRAM.

For SPI slave mode boot (BMODE = 10), the hardware configuration shown in Figure 3-4 is assumed.

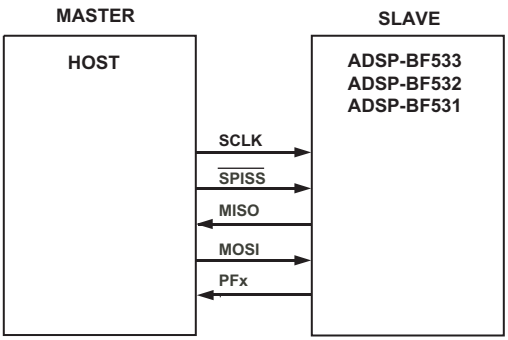


Figure 3-4. SPI Slave Boot Mode

The user defined programmable flag PFx is an output on the Blackfin processor and an input on the host device. This flag allows the processor to hold off the host device from sending data during certain sections of the boot process. When this flag is de-asserted, the host can continue to send bytes to the processor.

4 PROGRAM SEQUENCER

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

The linear flow varies occasionally when the program uses nonsequential program structures, such as those illustrated in Figure 4-1. Nonsequential structures direct the processor to execute an instruction that is not at the next sequential address. These structures include:

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

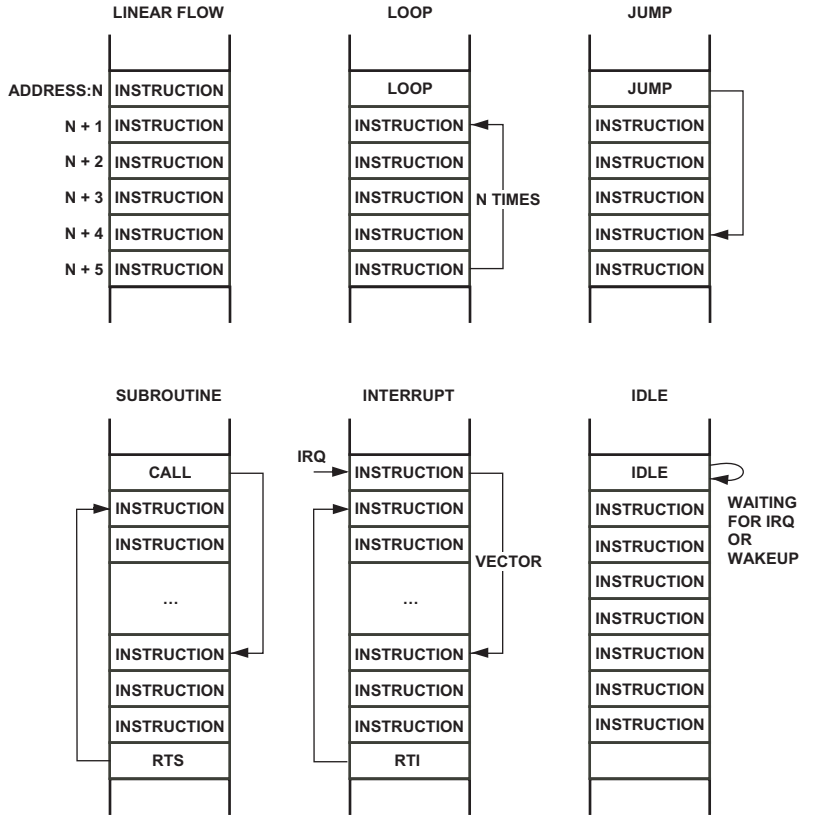


Figure 4-1. Program Flow Variations

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

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

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

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

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

Sequencer Related Registers

Table 4-1 lists the registers within the processor that are related to the sequencer. Except for the `PC` and `SEQSTAT` registers, all sequencer-related registers are directly readable and writable. Manually pushing or popping registers to or from the stack is done using the explicit instructions:

- `[--SP] = Rn` (for push)
- `Rn = [SP++]` (for pop)

Table 4-1. Sequencer-Related Registers

Register Name	Description
SEQSTAT	Sequencer Status register
RETX RETN RETI RETE RETS	Return Address registers: See "Events and Sequencing" on page 4-18. Exception Return NMI Return Interrupt Return Emulation Return Subroutine Return
LC0, LC1 LT0, LT1 LB0, LB1	Zero-Overhead Loop registers: Loop Counters Loop Tops Loop Bottoms
FP, SP	Frame Pointer and Stack Pointer: See "Frame and Stack Pointers" on page 5-5.
SYSCFG	System Configuration register
CYCLES, CYCLES2	Cycle Counters
PC	Program Counter

SEQSTAT Register

The Sequencer Status register (SEQSTAT) contains information about the current state of the sequencer as well as diagnostic information from the last event. SEQSTAT is a read-only register and is accessible only in Supervisor mode.

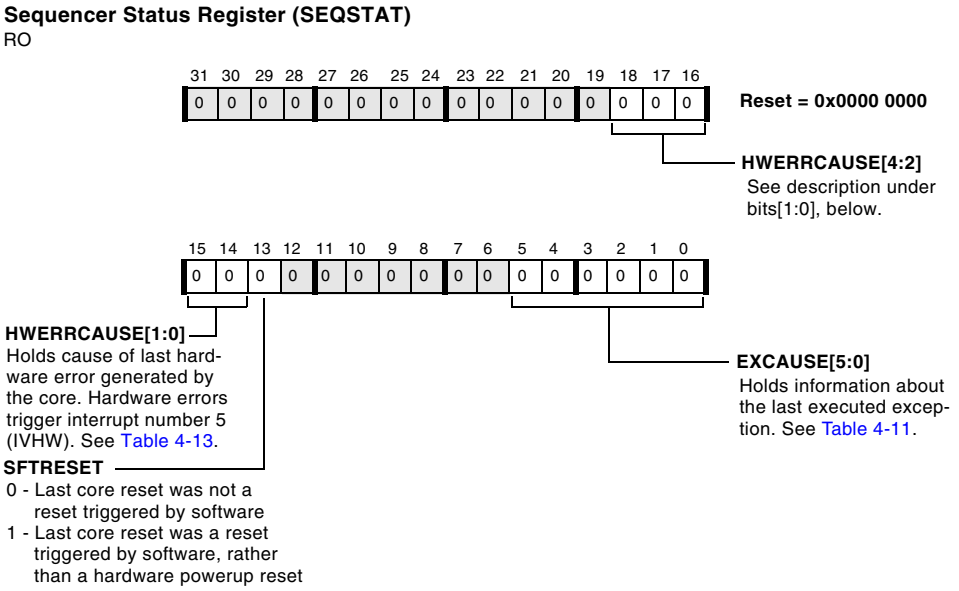


Figure 4-2. Sequencer Status Register

Zero-Overhead Loop Registers (LC, LT, and LB)

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

Table 4-2 describes the 32-bit loop register sets.

Table 4-2. Loop Registers

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

SYSCFG Register

The System Configuration register (SYSCFG) controls the configuration of the processor. This register is accessible only from the Supervisor mode.

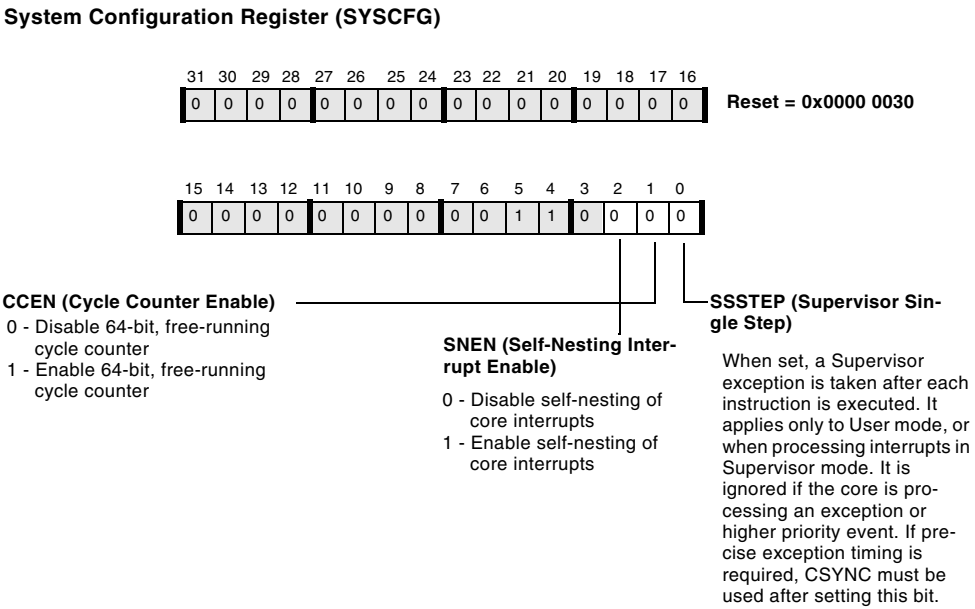


Figure 4-3. System Configuration Register

Instruction Pipeline

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

The processor has a ten-stage instruction pipeline, shown in Table 4-3.

Table 4-3. Stages of Instruction Pipeline

Pipeline Stage	Description
Instruction Fetch 1 (IF1)	Issue instruction address to IAB bus, start compare tag of instruction cache
Instruction Fetch 2 (IF2)	Wait for instruction data
Instruction Fetch 3 (IF3)	Read from IDB bus and align instruction
Instruction Decode (DEC)	Decode instructions
Address Calculation (AC)	Calculation of data addresses and branch target address
Data Fetch 1 (DF1)	Issue data address to DA0 and DA1 bus, start compare tag of data cache
Data Fetch 2 (DF2)	Read register files
Execute 1 (EX1)	Read data from LD0 and LD1 bus, start multiply and video instructions
Execute 2 (EX2)	Execute/Complete instructions (shift, add, logic, and so on)
Write Back (WB)	Writes back to register files, SD bus, and pointer updates (also referred to as the “commit” stage)

Figure 4-4 shows a diagram of the pipeline.

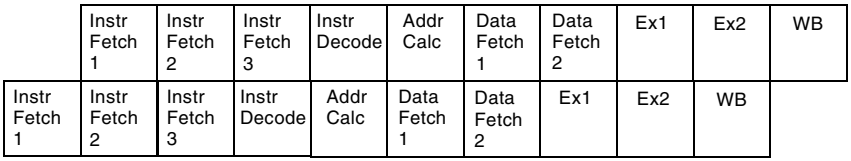


Figure 4-4. Processor Pipeline

The instruction fetch and branch logic generates 32-bit fetch addresses for the Instruction Memory Unit. The Instruction Alignment Unit returns instructions and their width information at the end of the IF3 stage.

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

The sequencer holds the fetch address until it receives a request from the alignment logic or until a change of flow occurs. The sequencer always increments the previous fetch address by 8 (the next 8 bytes). If a change of flow occurs, such as a branch or an interrupt, data in the Instruction Alignment Unit is invalidated. The sequencer decodes and distributes instruction data to the appropriate locations such as the register file and data memory.

The Execution Unit contains two 16-bit multipliers, two 40-bit ALUs, two 40-bit accumulators, one 40-bit shifter, a video unit (which adds 8-bit ALU support), and an 8-entry 32-bit Data Register File.

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

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

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

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

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

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

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

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

Branches and Sequencing

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

- A JUMP or a CALL instruction transfers program flow to another memory location. The difference between a JUMP and a CALL is that a CALL automatically loads the return address into the RETS register. The return address is the next sequential address after the CALL instruction. This push makes the address available for the CALL instruction's matching return instruction, allowing easy return from the subroutine.
- A return instruction causes the sequencer to fetch the instruction at the return address, which is stored in the RETS register (for subroutine returns). The types of return instructions include: return from subroutine (RTS), return from interrupt (RTI), return from exception (RTX), return from emulation (RTE), and return from nonmaskable interrupt (RTN). Each return type has its own register for holding the return address.
- A JUMP instruction can be conditional, depending on the status of the CC bit of the ASTAT register. These instructions are immediate and may not be delayed. The program sequencer can evaluate the CC status bit to decide whether to execute a branch. If no condition is specified, the branch is always taken.
- Conditional JUMP instructions use static branch prediction to reduce the branch latency caused by the length of the pipeline.

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

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

Direct Short and Long Jumps

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

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

If, at the time of writing the program, the destination is known to be less than a 13-bit offset from the current PC value, then the `JUMP.S 0xnnnn` instruction may be used. If the destination requires more than a 13-bit offset, then the `JUMP.L 0xnnnnnnn` instruction must be used. If the destination offset is unknown and development tools must evaluate the offset, then use the instruction `JUMP 0xnnnnnnn`. Upon disassembly, the instruction is replaced by the appropriate `JUMP.S` or `JUMP.L` instruction.

Direct Call

The `CALL` instruction is a branch instruction that copies the address of the instruction which would have executed next (had the `CALL` instruction not executed) into the `RETS` register. The direct `CALL` instruction has a 25-bit,

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

Indirect Branch and Call

The indirect JUMP and CALL instructions get their destination address from a data address generator (DAG) P-register. For the CALL instruction, the RETS register is loaded with the address of the instruction which would have executed next in the absence of the CALL instruction.

For example:

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

PC-Relative Indirect Branch and Call

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

For example:

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

Condition Code Flag

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

- A conditional branch is resolved by the value in CC.
- A Data register value may be copied into CC, and the value in CC may be copied to a Data register.
- The BITTST instruction accesses the CC flag.
- A status flag may be copied into CC, and the value in CC may be copied to a status flag.
- The CC flag bit may be set to the result of a Pointer register comparison.
- The CC flag bit may be set to the result of a Data register comparison.
- Some shifter instructions (rotate or BXOR) use CC as a portion of the shift operand/result.
- Test and set instructions can set and clear the CC bit.

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

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

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

Conditional Branches

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

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

```
IF CC JUMP dest_address ;
```

Conditional Register Move

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

Example code:

```
IF CC R0 = P0 ;
```

Branch Prediction

The sequencer supports static branch prediction to accelerate execution of conditional branches. These branches are executed based on the state of the CC bit.

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

The branch latency for conditional branches is as follows.

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

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

Consider the example in Table 4-4.

Table 4-4. Branch Prediction

Instruction	Description
If CC JUMP dest (bp)	This instruction tests the CC flag, and if it is set, jumps to a location, identified by the label, dest. If the CC flag is set, the branch is correctly predicted and the branch latency is reduced. Otherwise, the branch is incorrectly predicted and the branch latency increases.

Loops and Sequencing

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

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

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

Since there are two loop units, loop unit 1 is assigned higher priority so it can be used as the inner loop in a nested loop structure. In other words, a loopback caused by loop unit 1 on a particular instruction (PC == LB1, LC1 >= 2) will prevent loop unit 0 from looping back on that same instruction, even if the address matches. Loop unit 0 is allowed to loop back only after the loop count 1 is exhausted.

The LSETUP instruction can be used to load all three registers of a loop unit at once. Each loop register can also be loaded individually with a register transfer, but this incurs a significant overhead if the loop count is nonzero (the loop is active) at the time of the transfer.

The following code example shows a loop that contains two instructions and iterates 32 times.

Listing 4-1. Loop Example

```
P5 = 0x20 ;
LSETUP ( lp_start, lp_end ) LC0 = P5 ;
lp_start:
R5 = R0 + R1(ns) || R2 = [P2++] || R3 = [I1++] ;

lp_end:  R5 = R5 + R2 ;
```

Two sets of loop registers are used to manage two nested loops:

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

Table 4-5. Loop Registers

First/Last Address of the Loop	PC-Relative Offset Used to Compute the Loop Start Address	Effective Range of the Loop Start Instruction
Top / First	5-bit signed immediate; must be a multiple of 2.	0 to 30 bytes away from LSETUP instruction.
Bottom / Last	11-bit signed immediate; must be a multiple of 2.	0 to 2046 bytes away from LSETUP instruction (the defined loop can be 2046 bytes long).

When executing an `LSETUP` instruction, the program sequencer loads the address of the loop's last instruction into `LBx` and the address of the loop's first instruction into `LTx`. The top and bottom addresses of the loop are computed as PC-relative addresses from the `LSETUP` instruction, plus an offset. In each case, the offset value is added to the location of the `LSETUP` instruction.

The LC0 and LC1 registers are unsigned 32-bit registers, each supporting 2^{32} –1 iterations through the loop.

When LCx = 0, the loop is disabled, and a single pass of the code executes.

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

A four-cycle latency occurs on the first loopback when the LSETUP specifies a nonzero start offset (lp_start). Therefore, zero start offsets are preferred.

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

Events and Sequencing

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

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

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

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

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

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

The CEC supports nine general-purpose interrupts (`IVG7` – `IVG15`) in addition to the dedicated interrupt and exception events that are described in Table 4-6. It is recommended that the two lowest priority interrupts (`IVG14` and `IVG15`) be reserved for software interrupt handlers, leaving seven prioritized interrupt inputs (`IVG7` – `IVG13`) to support the system.

Refer to Table 4-6.

Table 4-6. System and Core Event Mapping

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

Table 4-6. System and Core Event Mapping (Cont'd)

	Event Source	Core Event Name
System Interrupts	PLL Wakeup Interrupt DMA Error (generic) PPI Error Interrupt SPORT0 Error Interrupt SPORT1 Error Interrupt SPI Error Interrupt UART Error Interrupt	IVG7
	Real-Time Clock Interrupts DMA0 Interrupt (PPI)	IVG8
	DMA1 Interrupt (SPORT0 RX) DMA2 Interrupt (SPORT0 TX) DMA3 Interrupt (SPORT1 RX) DMA4 Interrupt (SPORT1 TX)	IVG9
	DMA5 Interrupt (SPI) DMA6 Interrupt (UART RX) DMA7 Interrupt (UART TX)	IVG10
	Timer0, Timer1, Timer2 Interrupts	IVG11
	Programmable Flags Interrupt A/B	IVG12
	DMA8/9 Interrupt (Memory DMA Stream 0) DMA10/11 Interrupt (Memory DMA Stream 1) Software Watchdog Timer	IVG13
	Software Interrupt 1	IVG14
	Software Interrupt 2 (lowest priority)	IVG15

Note the System Interrupt to Core Event mappings shown are the default values at reset and can be changed by software.

System Interrupt Processing

Referring to Figure 4-5, note when an interrupt (Interrupt A) is generated by an interrupt-enabled peripheral:

1. SIC_ISR logs the request and keeps track of system interrupts that are asserted but not yet serviced (that is, an interrupt service routine hasn't yet cleared the interrupt).
2. SIC_IWR checks to see if it should wake up the core from an idled state based on this interrupt request.
3. SIC_IMASK masks off or enables interrupts from peripherals at the system level. If Interrupt A is not masked, the request proceeds to Step 4.
4. The SIC_IARx registers, which map the peripheral interrupts to a smaller set of general-purpose core interrupts (IVG7 - IVG15), determine the core priority of Interrupt A.
5. ILAT adds Interrupt A to its log of interrupts latched by the core but not yet actively being serviced.
6. IMASK masks off or enables events of different core priorities. If the IVGx event corresponding to Interrupt A is not masked, the process proceeds to Step 7.
7. The Event Vector Table (EVT) is accessed to look up the appropriate vector for Interrupt A's interrupt service routine (ISR).
8. When the event vector for Interrupt A has entered the core pipeline, the appropriate IPEND bit is set, which clears the respective ILAT bit. Thus, IPEND tracks all pending interrupts, as well as those being presently serviced.

9. When the interrupt service routine (ISR) for Interrupt A has been executed, the RTI instruction clears the appropriate IPEND bit. However, the relevant SIC_ISR bit is not cleared unless the interrupt service routine clears the mechanism that generated Interrupt A, or if the process of servicing the interrupt clears this bit.

It should be noted that emulation, reset, NMI, and exception events, as well as hardware error (IVHW) and core timer (IVTMR) interrupt requests, enter the interrupt processing chain at the ILAT level and are not affected by the system-level interrupt registers (SIC_IWR, SIC_ISR, SIC_IMASK, SIC_IARx).

If multiple interrupt sources share a single core interrupt, then the interrupt service routine (ISR) must identify the peripheral that generated the interrupt. The ISR may then need to interrogate the peripheral to determine the appropriate action to take.

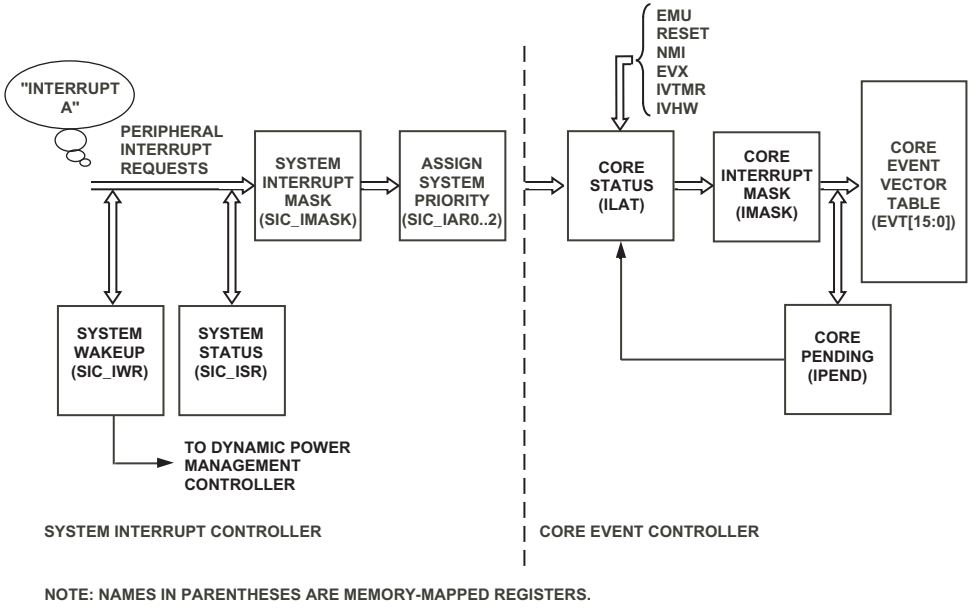


Figure 4-5. Interrupt Processing Block Diagram

System Peripheral Interrupts

The processor system has numerous peripherals, which therefore require many supporting interrupts. Table 4-7 lists:

- The Peripheral Interrupt source
- The Peripheral Interrupt ID used in the System Interrupt Assignment registers (SIC_IARx). See "System Interrupt Assignment Registers (SIC_IARx)" on page 4-30.
- The general-purpose interrupt of the core to which the interrupt maps at reset
- The Core Interrupt ID used in the System Interrupt Assignment registers (SIC_IARx). See "System Interrupt Assignment Registers (SIC_IARx)" on page 4-30.

Table 4-7. Peripheral Interrupt Source Reset State

Peripheral Interrupt Source	Peripheral Interrupt ID	General-purpose Interrupt (Assignment at Reset)	Core Interrupt ID
PLL Wakeup Interrupt	0	IVG7	0
DMA Error (generic)	1	IVG7	0
PPI Error Interrupt	2	IVG7	0
SPORT0 Error Interrupt	3	IVG7	0
SPORT1 Error Interrupt	4	IVG7	0
SPI Error Interrupt	5	IVG7	0
UART Error Interrupt	6	IVG7	0
Real-Time Clock Interrupts (alarm, second, minute, hour, countdown)	7	IVG8	1
DMA 0 Interrupt (PPI)	8	IVG8	1
DMA 1 Interrupt (SPORT0 RX)	9	IVG9	2

Table 4-7. Peripheral Interrupt Source Reset State (Cont'd)

Peripheral Interrupt Source	Peripheral Interrupt ID	General-purpose Interrupt (Assignment at Reset)	Core Interrupt ID
DMA 2 Interrupt (SPORT0 TX)	10	IVG9	2
DMA 3 Interrupt (SPORT1 RX)	11	IVG9	2
DMA 4 Interrupt (SPORT1 TX)	12	IVG9	2
DMA 5 Interrupt (SPI)	13	IVG10	3
DMA 6 Interrupt (UART RX)	14	IVG10	3
DMA 7 Interrupt (UART TX)	15	IVG10	3
Timer0 Interrupt	16	IVG11	4
Timer1 Interrupt	17	IVG11	4
Timer2 Interrupt	18	IVG11	4
PF Interrupt A	19	IVG12	5
PF Interrupt B	20	IVG12	5
DMA 8/9 Interrupt (Memory DMA Stream 0)	21	IVG13	6
DMA 10/11 Interrupt (Memory DMA Stream 1)	22	IVG13	6
Software Watchdog Timer Interrupt	23	IVG13	6
Reserved	24-31	-	-

The peripheral interrupt structure of the processor is flexible. By default upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core, as shown in Table 4-7.

An interrupt service routine that supports multiple interrupt sources must interrogate the appropriate system memory mapped registers (MMRs) to determine which peripheral generated the interrupt.

If the default assignments shown in Table 4-7 are acceptable, then interrupt initialization involves only:

- Initialization of the core Event Vector Table (EVT) vector address entries
- Initialization of the IMASK register
- Unmasking the specific peripheral interrupts in SIC_IMASK that the system requires

SIC_IWR Register

The System Interrupt Wakeup-Enable register (SIC_IWR) provides the mapping between the peripheral interrupt source and the Dynamic Power Management Controller (DPMC). Any of the peripherals can be configured to wake up the core from its idled state to process the interrupt, simply by enabling the appropriate bit in the System Interrupt Wakeup-enable register (SIC_IWR, refer to Figure 4-6). If a peripheral interrupt source is enabled in SIC_IWR and the core is idled, the interrupt causes the DPMC to initiate the core wakeup sequence in order to process the interrupt. Note this mode of operation may add latency to interrupt processing, depending on the power control state. For further discussion of power modes and the idled state of the core, see Chapter 8, "Dynamic Power Management."

By default, all interrupts generate a wakeup request to the core. However, for some applications it may be desirable to disable this function for some peripherals, such as for a SPORTx Transmit Interrupt.

The SIC_IWR register has no effect unless the core is idled. The bits in this register correspond to those of the System Interrupt Mask (SIC_IMASK) and Interrupt Status (SIC_ISR) registers.

After reset, all valid bits of this register are set to 1, enabling the wakeup function for all interrupts that are not masked. Before enabling interrupts, configure this register in the reset initialization sequence. The SIC_IWR register can be read from or written to at any time. To prevent spurious or lost interrupt activity, this register should be written to only when all peripheral interrupts are disabled.

Note the wakeup function is independent of the interrupt mask function. If an interrupt source is enabled in SIC_IWR but masked off in SIC_IMASK, the core wakes up if it is idled, but it does not generate an interrupt.

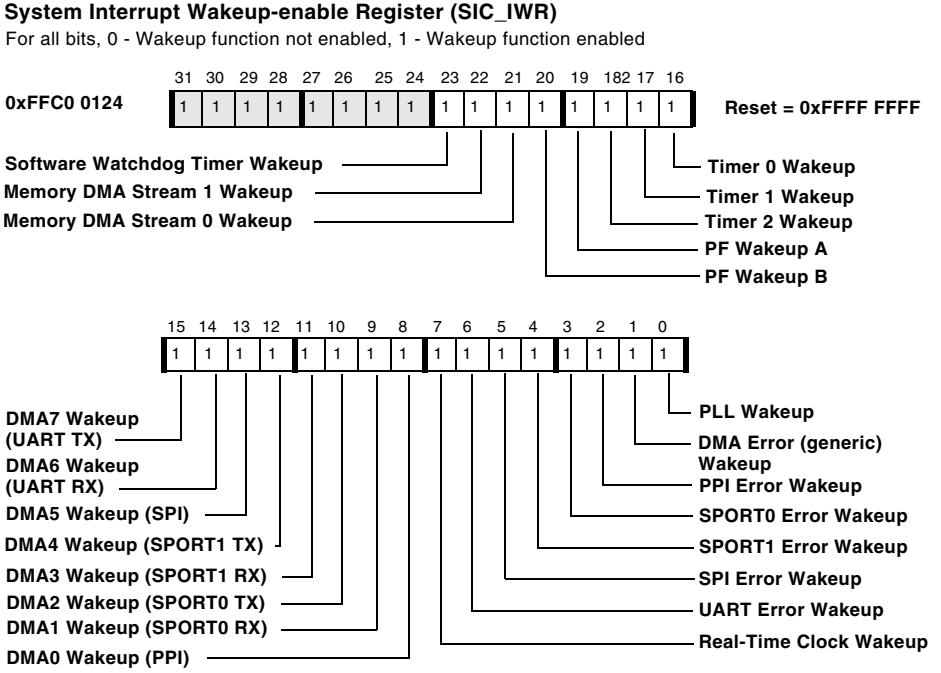


Figure 4-6. System Interrupt Wakeup-enable Register

SIC_ISR Register

The System Interrupt Controller (SIC) includes a read-only status register, the System Interrupt Status register (SIC_ISR), shown in Figure 4-7. Each valid bit in this register corresponds to one of the peripheral interrupt sources. The bit is set when the SIC detects the interrupt is asserted and cleared when the SIC detects that the peripheral interrupt input has been deasserted. Note for some peripherals, such as programmable flag asynchronous input interrupts, many cycles of latency may pass from the time an interrupt service routine initiates the clearing of the interrupt (usually by writing a system MMR) to the time the SIC senses that the interrupt has been deasserted.

Depending on how interrupt sources map to the general-purpose interrupt inputs of the core, the interrupt service routine may have to interrogate multiple interrupt status bits to determine the source of the interrupt. One of the first instructions executed in an interrupt service routine should read SIC_ISR to determine whether more than one of the peripherals sharing the input has asserted its interrupt output. The service routine should fully process all pending, shared interrupts before executing the RTI, which enables further interrupt generation on that interrupt input.

When an interrupt's service routine is finished, the RTI instruction clears the appropriate bit in the IPEND register. However, the relevant SIC_ISR bit is not cleared unless the service routine clears the mechanism that generated the interrupt.

Many systems need relatively few interrupt-enabled peripherals, allowing each peripheral to map to a unique core priority level. In these designs, SIC_ISR will seldom, if ever, need to be interrogated.

The SIC_ISR register is not affected by the state of the System Interrupt Mask register (SIC_IMASK) and can be read at any time. Writes to the SIC_ISR register have no effect on its contents.

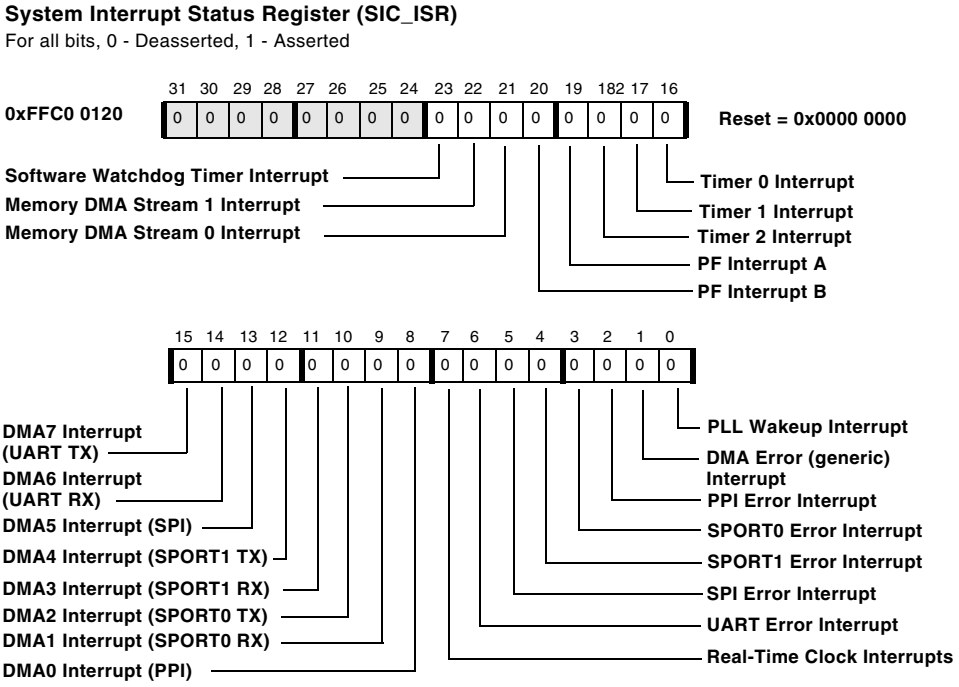


Figure 4-7. System Interrupt Status Register

SIC_IMASK Register

The System Interrupt Mask register (SIC_IMASK, shown in Figure 4-8) allows masking of any peripheral interrupt source at the System Interrupt Controller (SIC), independently of whether it is enabled at the peripheral itself.

A reset forces the contents of SIC_IMASK to all 0s to mask off all peripheral interrupts. Writing a 1 to a bit location turns off the mask and enables the interrupt.

Although this register can be read from or written to at any time (in Supervisor mode), it should be configured in the reset initialization sequence before enabling interrupts.

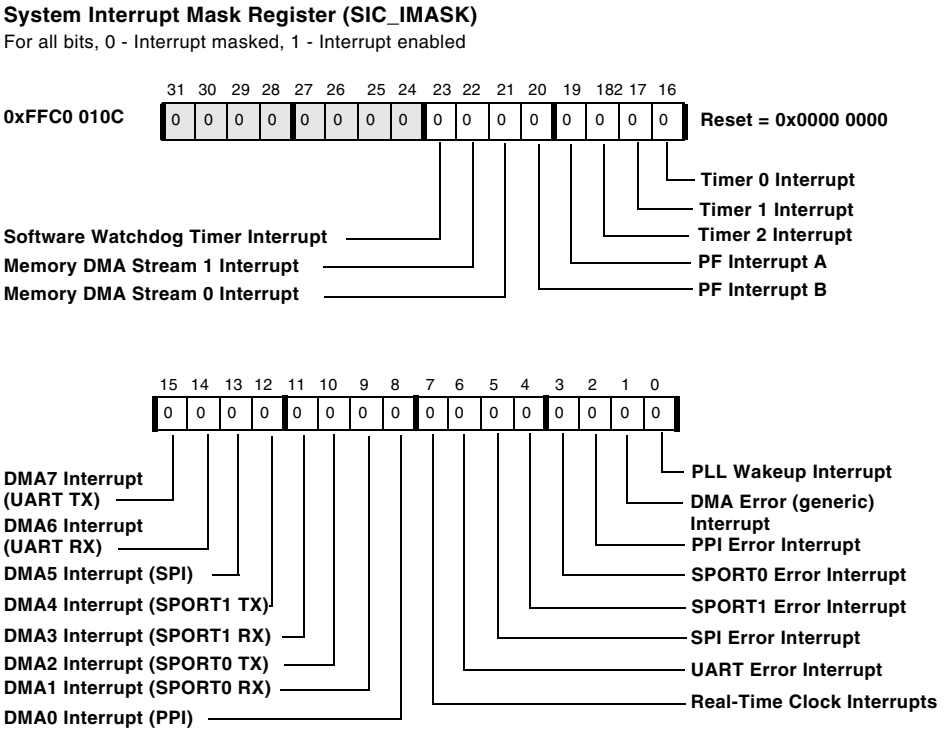


Figure 4-8. System Interrupt Mask Register

System Interrupt Assignment Registers (SIC_IARx)

The relative priority of peripheral interrupts can be set by mapping the peripheral interrupt to the appropriate general-purpose interrupt level in the core. The mapping is controlled by the System Interrupt Assignment register settings, as detailed in Figure 4-9, Figure 4-10, and Figure 4-11.

If more than one interrupt source is mapped to the same interrupt, they are logically OR'ed, with no hardware prioritization. Software can prioritize the interrupt processing as required for a particular system application.

For general-purpose interrupts with multiple peripheral interrupts assigned to them, take special care to ensure that software correctly processes all pending interrupts sharing that input. Software is responsible for prioritizing the shared interrupts.

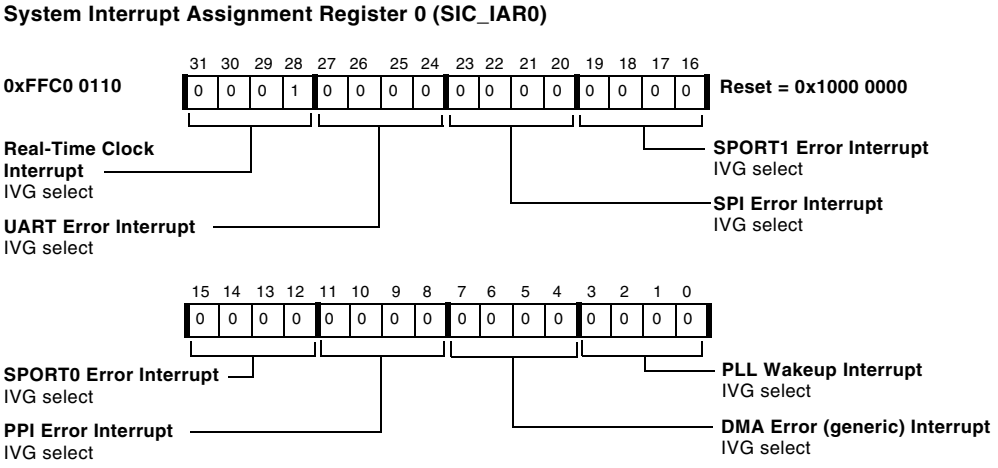


Figure 4-9. System Interrupt Assignment Register 0

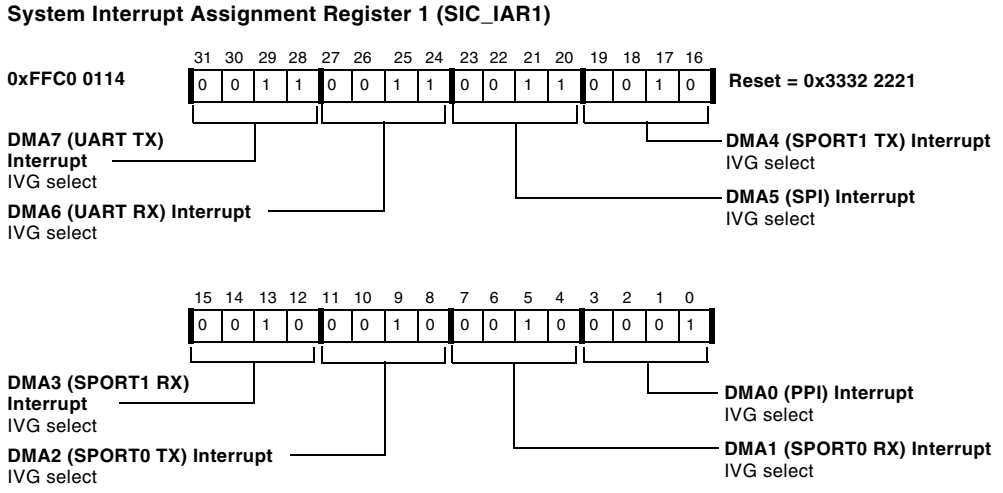


Figure 4-10. System Interrupt Assignment Register 1

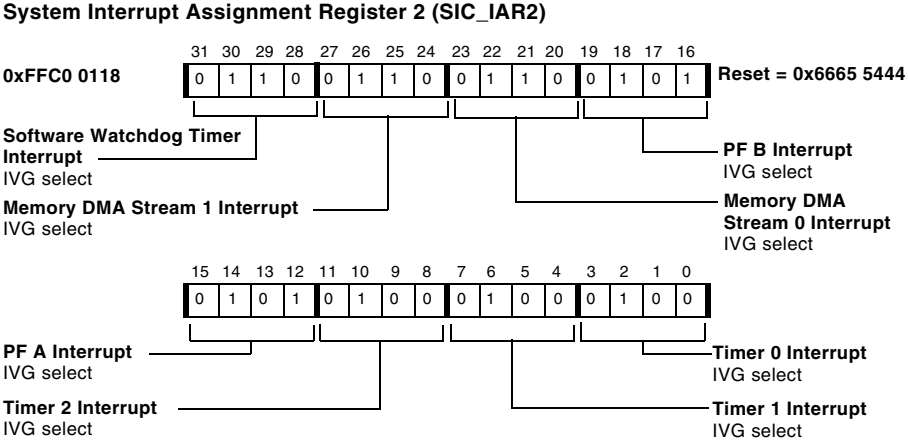


Figure 4-11. System Interrupt Assignment Register 2

These registers can be read from or written to at any time in Supervisor mode. It is advisable, however, to configure them in the Reset interrupt service routine before enabling interrupts. To prevent spurious or lost interrupt activity, these registers should be written to only when all peripheral interrupts are disabled.

Table 4-8 defines the value to write in SIC_IARx to configure a peripheral for a particular IVG priority.

Table 4-8. IVG Select Definitions

General-purpose Interrupt	Value in SIC_IAR
IVG7	0
IVG8	1
IVG9	2
IVG10	3
IVG11	4
IVG12	5
IVG13	6
IVG14	7
IVG15	8

Core Event Controller Registers

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

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

These three registers are accessible in Supervisor mode only.

IMASK Register

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

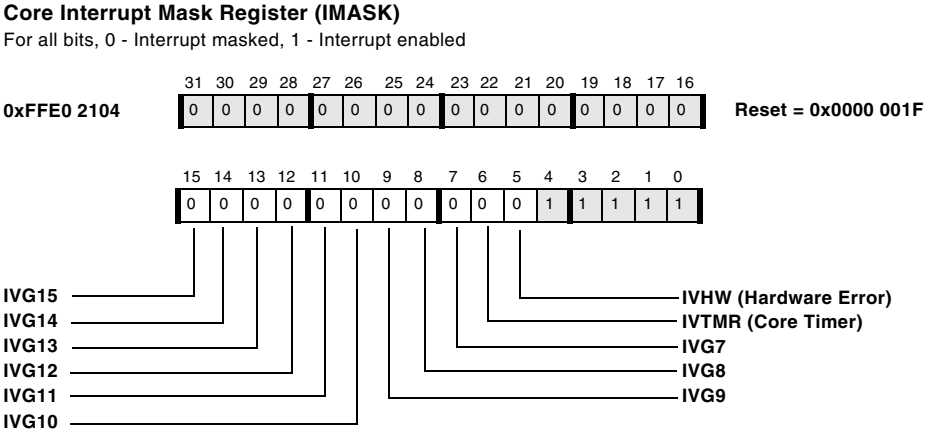


Figure 4-12. Core Interrupt Mask Register

ILAT Register

Each bit in the Core Interrupt Latch register (`ILAT`) indicates that the corresponding event is latched, but not yet accepted into the processor (see Figure 4-13). The bit is reset before the first instruction in the corresponding ISR is executed. At the point the interrupt is accepted, `ILAT[N]` will be cleared and `IPEND[N]` will be set simultaneously. The `ILAT` register can be read in Supervisor mode. Writes to `ILAT` are used to clear bits only (in Supervisor mode). To clear bit `N` from `ILAT`, first make sure that `IMASK[N] == 0`, and then write `ILAT[N] = 1`. This write functionality to `ILAT` is provided for cases where latched interrupt requests need to be cleared (cancelled) instead of serviced.

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

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

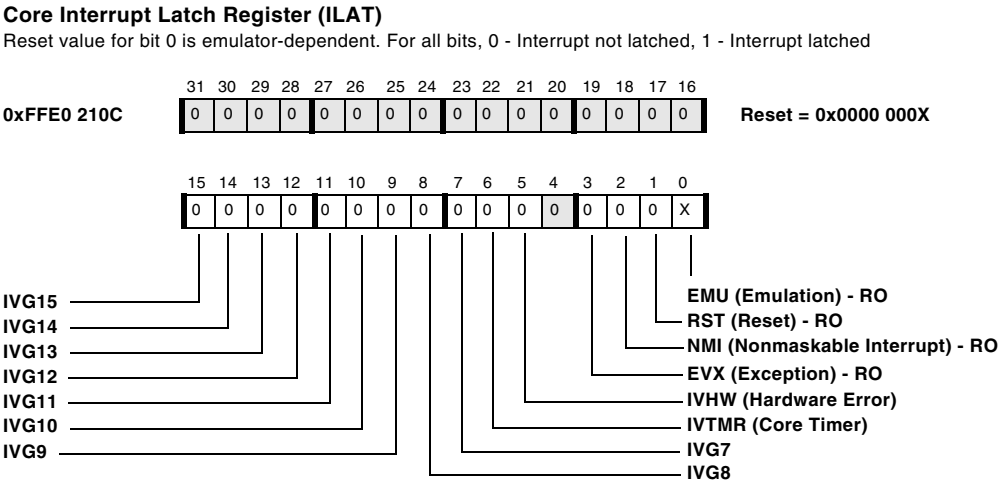


Figure 4-13. Core Interrupt Latch Register

IPEND Register

The Core Interrupt Pending register (`IPEND`) keeps track of all currently nested interrupts (see Figure 4-14). Each bit in `IPEND` indicates that the corresponding interrupt is currently active or nested at some level. It may be read in Supervisor mode, but not written. The `IPEND[4]` bit is used by the Event Controller to temporarily disable interrupts on entry and exit to an interrupt service routine.

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

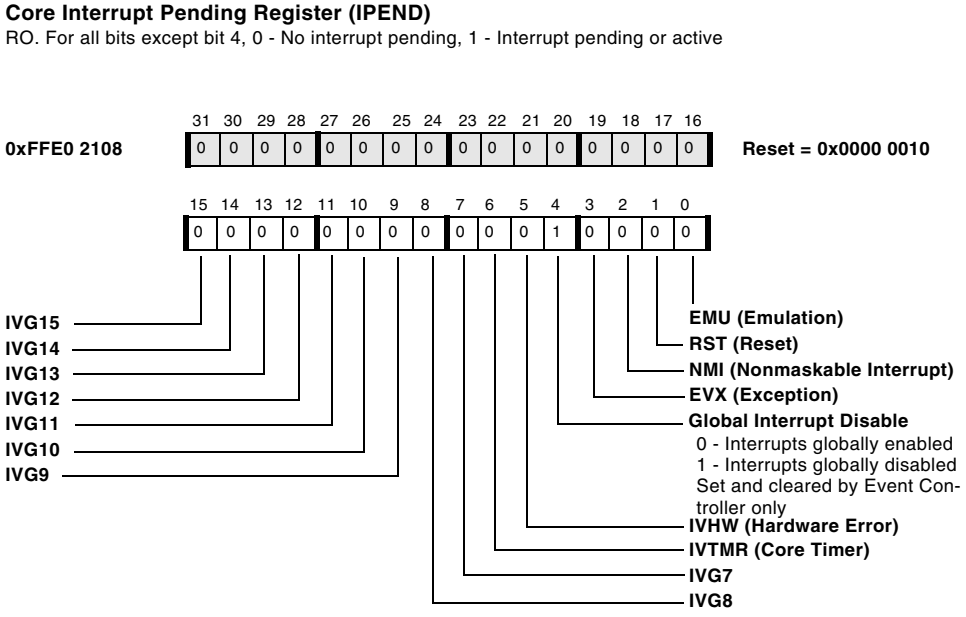


Figure 4-14. Core Interrupt Pending Register

Global Enabling/Disabling of Interrupts

General-purpose interrupts can be globally disabled with the `CLI Dreg` instruction and re-enabled with the `STI Dreg` instruction, both of which are only available in Supervisor mode. Reset, NMI, emulation, and exception events cannot be globally disabled. Globally disabling interrupts clears `IMASK[15:5]` after saving `IMASK`'s current state. See "Enable Interrupts" and "Disable Interrupts" in the External Event Management chapter of the *Blackfin Processor Programming Reference*.

When program code is too time critical to be delayed by an interrupt, disable the general-purpose interrupts, but be sure to re-enable them at the conclusion of the code sequence.

Event Vector Table

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

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

Table 4-9 lists events by priority. Each event has a corresponding bit in the event state registers `ILAT`, `IMASK`, and `IPEND`.

Table 4-9. Core Event Vector Table

Event Number	Event Class	Name	MMR Location	Notes
EVT0	Emulation	EMU	0xFFE0 2000	Highest priority. Vector address is provided by JTAG.
EVT1	Reset	RST	0xFFE0 2004	
EVT2	NMI	NMI	0xFFE0 2008	
EVT3	Exception	EVX	0xFFE0 200C	
EVT4	Reserved	Reserved	0xFFE0 2010	Reserved vector
EVT5	Hardware Error	IVHW	0xFFE0 2014	
EVT6	Core Timer	IVTMR	0xFFE0 2018	
EVT7	Interrupt 7	IVG7	0xFFE0 201C	
EVT8	Interrupt 8	IVG8	0xFFE0 2020	
EVT9	Interrupt 9	IVG9	0xFFE0 2024	

Table 4-9. Core Event Vector Table (Cont'd)

Event Number	Event Class	Name	MMR Location	Notes
EVT10	Interrupt 10	IVG10	0xFFE0 2028	
EVT11	Interrupt 11	IVG11	0xFFE0 202C	
EVT12	Interrupt 12	IVG12	0xFFE0 2030	
EVT13	Interrupt 13	IVG13	0xFFE0 2034	
EVT14	Interrupt 14	IVG14	0xFFE0 2038	
EVT15	Interrupt 15	IVG15	0xFFE0 203C	Lowest priority

Emulation

An emulation event causes the processor to enter Emulation mode, where instructions are read from the JTAG interface. It is the highest priority interrupt to the core.

For detailed information about emulation, see Chapter 21, "Debug", in the *Blackfin Processor Programming Reference*.

Reset

The reset interrupt (RST) can be initiated via the $\overline{\text{RESET}}$ pin or through expiration of the watchdog timer. The reset vector can be reconfigured to another address during runtime and therefore, an application can vector to an address other than 0xFFA0 0000 (ADSP-BF533) or 0xFFA0 8000 (ADSP-BF531/ADSP-BF532) after a software reset. If the reset vector is modified during runtime, ensure that the reset vector address within the EVT1 register is a valid instruction address. This location differs from that of other interrupts in that its content is read-only. Writes to this address change the register but do not change where the processor vectors upon reset. The processor always vectors to the reset vector address upon reset. For more information, see "Reset State" on page 3-10 and "Booting Methods" on page 3-18.

The core has an output that indicates that a double fault has occurred. This is a nonrecoverable state. The system (via the SWRST register) can be programmed to send a reset request if a double fault condition is detected. Subsequently, the reset request forces a system reset for core and peripherals.

The reset vector is determined by the processor system. It points to the start of the on-chip boot ROM, or to the start of external asynchronous memory, depending on the state of the BMODE[1:0] pins. Refer to Table 4-10.

Table 4-10. Reset Vector Addresses

Boot Source	BMODE[1:0]	Execution Start Address
Bypass boot ROM; execute from 16-bit wide external memory (Async Bank 0)	00	0x2000 0000
Use boot ROM to boot from 8-bit or 16-bit flash	01	0xEF00 0000
Use boot ROM to boot from 8-bit SPI host device	10	0xEF00 0000
Use boot ROM to configure and load boot code from SPI serial EEPROM (8-, 16-, or 24-bit address range)	11	0xEF00 0000

If the BMODE[1:0] pins indicate either booting from flash, SPI flash, SPI host, or serial EEPROM, the reset vector points to the start of the internal boot ROM, where a small bootstrap kernel resides. The bootstrap code reads the System Reset Configuration register (SYSCR) to determine the value of the BMODE[1:0] pins, which determine the appropriate boot sequence. For information about the boot ROM, see "Booting Methods" on page 3-18.

If the BMODE[1:0] pins indicate to bypass boot ROM, the reset vector points to the start of the external asynchronous memory region. In this mode, the internal boot ROM is not used. To support reads from this memory region, the External Bus Interface Unit (EBIU) uses the default external memory configuration that results from hardware reset.

NMI (Nonmaskable Interrupt)

The NMI entry is reserved for a nonmaskable interrupt, which can be generated by the Watchdog timer or by the NMI input signal to the processor. NMI is a level-sensitive pin; when not used, it should always be pulled *low* for ADSP-BF531/2/3 processors. Only events that require immediate processor attention are appropriate as an NMI entry. For example, a powerdown warning is an appropriate NMI event.

If an exception occurs in an event handler that is already servicing an Exception, NMI, Reset, or Emulation event, this will trigger a double fault condition, and the address of the excepting instruction will be written to RETX.

Exceptions

Exceptions are synchronous to the instruction stream. In other words, a particular instruction causes an exception when it attempts to finish execution. No instructions after the offending instruction are executed before the exception handler takes effect.

Many of the exceptions are memory related. For example, an exception is given when a misaligned access is attempted, or when a cacheability protection lookaside buffer (CPLB) miss or protection violation occurs. Exceptions are also given when illegal instructions or illegal combinations of registers are executed.

An excepting instruction may or may not commit before the exception event is taken, depending on if it is a service type or an error type exception.

An instruction causing a service type event will commit, and the address written to the RETX register will be the next instruction after the excepting one. An example of a service type exception is the single step.

An instruction causing an error type event cannot commit, so the address written to the RETX register will be the address of the offending instruction. An example of an error type event is a CPLB miss.

Usually the RETX register contains the correct address to return to. To skip over an excepting instruction, take care in case the next address is not simply the next linear address. This could happen when the excepting instruction is a loop end. In that case, the proper next address would be the loop top.

The EXCAUSE[5:0] field in the Sequencer Status register (SEQSTAT) is written whenever an exception is taken, and indicates to the exception handler which type of exception occurred. Refer to Table 4-11 for a list of events that cause exceptions.

If an exception occurs in an event handler that is already servicing an Exception, NMI, Reset, or Emulation event, this will trigger a double fault condition, and the address of the excepting instruction will be written to RETX.

Table 4-11. Events That Cause Exceptions

Exception	EXCAUSE [5:0]	Type: (E) Error (S) Service[1]	Notes/Examples
Force Exception instruction EXCPT with 4-bit m field	m field	S	Instruction provides 4 bits of EXCAUSE.
Single step	0x10	S	When the processor is in single step mode, every instruction generates an exception. Primarily used for debugging.
Exception caused by a trace buffer full condition	0x11	S	The processor takes this exception when the trace buffer overflows (only when enabled by the Trace Unit Control register).
Undefined instruction	0x21	E	May be used to emulate instructions that are not defined for a particular processor implementation.
Illegal instruction combination	0x22	E	See section for multi-issue rules in the *Blackfin Processor Programming Reference*.
Data access CPLB protection violation	0x23	E	Attempted read or write to Supervisor resource, or illegal data memory access. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions. (A simultaneous, dual access to two MMRs using the data address generators generates this type of exception.) In addition, this entry is used to signal a protection violation caused by disallowed memory access, and it is defined by the Memory Management Unit (MMU) cacheability protection lookaside buffer (CPLB).
Data access misaligned address violation	0x24	E	Attempted misaligned data memory or data cache access.

Table 4-11. Events That Cause Exceptions (Cont'd)

Exception	EXCAUSE [5:0]	Type: (E) Error (S) Service[1]	Notes/Examples
Unrecoverable event	0x25	E	For example, an exception generated while processing a previous exception.
Data access CPLB miss	0x26	E	Used by the MMU to signal a CPLB miss on a data access.
Data access multiple CPLB hits	0x27	E	More than one CPLB entry matches data fetch address.
Exception caused by an emulation watchpoint match	0x28	E	There is a watchpoint match, and one of the EMUSW bits in the Watchpoint Instruction Address Control register (WPIACTL) is set.
Instruction fetch misaligned address violation	0x2A	E	Attempted misaligned instruction cache fetch. On a misaligned instruction fetch exception, the return address provided in RETX is the destination address which is misaligned, rather than the address of the offending instruction. For example, if an indirect branch to a misaligned address held in P0 is attempted, the return address in RETX is equal to P0, rather than to the address of the branch instruction. (Note this exception can never be generated from PC-relative branches, only from indirect branches.)
Instruction fetch CPLB protection violation	0x2B	E	Illegal instruction fetch access (memory protection violation).
Instruction fetch CPLB miss	0x2C	E	CPLB miss on an instruction fetch.

Table 4-11. Events That Cause Exceptions (Cont'd)

Exception	EXCAUSE [5:0]	Type: (E) Error (S) Service[1]	Notes/Examples
Instruction fetch multiple CPLB hits	0x2D	E	More than one CPLB entry matches instruction fetch address.
Illegal use of supervisor resource	0x2E	E	Attempted to use a Supervisor register or instruction from User mode. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions.

1 For services (S), the return address is the address of the instruction that follows the exception. For errors (E), the return address is the address of the excepting instruction.

If an instruction causes multiple exceptions, only the exception with the highest priority is taken. Table 4-12 ranks exceptions by descending priority.

Table 4-12. Exceptions by Descending Priority

Priority	Exception	EXCAUSE
1	Unrecoverable Event	0x25
2	I-Fetch Multiple CPLB Hits	0x2D
3	I-Fetch Misaligned Access	0x2A
4	I-Fetch Protection Violation	0x2B
5	I-Fetch CPLB Miss	0x2C
6	I-Fetch Access Exception	0x29
7	Watchpoint Match	0x28
8	Undefined Instruction	0x21
9	Illegal Combination	0x22
10	Illegal Use of Protected Resource	0x2E

Table 4-12. Exceptions by Descending Priority (Cont'd)

Priority	Exception	EXCAUSE
11	DAG0 Multiple CPLB Hits	0x27
12	DAG0 Misaligned Access	0x24
13	DAG0 Protection Violation	0x23
14	DAG0 CPLB Miss	0x26
15	DAG1 Multiple CPLB Hits	0x27
16	DAG1 Misaligned Access	0x24
17	DAG1 Protection Violation	0x23
18	DAG1 CPLB Miss	0x26
19	EXCPT Instruction	m field
20	Single Step	0x10
21	Trace Buffer	0x11

Exceptions While Executing an Exception Handler

While executing the exception handler, avoid issuing an instruction that generates another exception. If an exception is caused while executing code within the exception handler, the NMI handler, the reset vector, or in emulator mode:

- The excepting instruction is not committed. All writebacks from the instruction are prevented.
- The generated exception is not taken.
- The EXCAUSE field in SEQSTAT is updated with an unrecoverable event code.

- The address of the offending instruction is saved in RETX. Note if the processor were executing, for example, the NMI handler, the RETN register would not have been updated; the excepting instruction address is always stored in RETX.

To determine whether an exception occurred while an exception handler was executing, check SEQSTAT at the end of the exception handler for the code indicating an "unrecoverable event" (EXCAUSE = 0x25). If an unrecoverable event occurred, register RETX holds the address of the most recent instruction to cause an exception. This mechanism is not intended for recovery, but rather for detection.

Hardware Error Interrupt

The Hardware Error Interrupt indicates a hardware error or system malfunction. Hardware errors occur when logic external to the core, such as a memory bus controller, is unable to complete a data transfer (read or write) and asserts the core's error input signal. Such hardware errors invoke the Hardware Error Interrupt (interrupt IVHW in the Event Vector Table (EVT) and ILAT, IMASK, and IPEND registers). The Hardware Error Interrupt service routine can then read the cause of the error from the 5-bit HWERRCAUSE field appearing in the Sequencer Status register (SEQSTAT) and respond accordingly.

The Hardware Error Interrupt is generated by:

- Bus parity errors
- Internal error conditions within the core, such as Performance Monitor overflow
- Peripheral errors
- Bus timeout errors

The list of supported hardware conditions, with their related HWERRCAUSE codes, appears in Table 4-13. The bit code for the most recent error appears in the HWERRCAUSE field. If multiple hardware errors occur simultaneously, only the last one can be recognized and serviced. The core does not support prioritizing, pipelining, or queuing multiple error codes. The Hardware Error Interrupt remains active as long as any of the error conditions remain active.

Table 4-13. Hardware Conditions Causing Hardware Error Interrupts

Hardware Condition	HWERRCAUSE (Binary)	HWERRCAUSE (Hexadecimal)	Notes / Examples
System MMR Error	0b00010	0x02	An error can occur if an invalid System MMR location is accessed, if a 32-bit register is accessed with a 16-bit instruction, or if a 16-bit register is accessed with a 32-bit instruction.
External Memory Addressing Error	0b00011	0x03	
Performance Monitor Overflow	0b10010	0x12	
RAISE 5 instruction	0b11000	0x18	Software issued a RAISE 5 instruction to invoke the Hardware Error Interrupt (IVHW).
Reserved	All other bit combinations.	All other values.	

Core Timer

The Core Timer Interrupt (IVTMR) is triggered when the core timer value reaches zero. See Chapter 15, “Timers.”

General-Purpose Interrupts (IVG7-IVG15)

General-purpose interrupts are used for any event that requires processor attention. For instance, a DMA controller may use them to signal the end of a data transmission, or a serial communications device may use them to signal transmission errors.

Software can also trigger general-purpose interrupts by using the RAISE instruction. The RAISE instruction forces events for interrupts IVG15-IVG7, IVTMR, IVHW, NMI, and RST, but not for exceptions and emulation (EVX and EMU, respectively).

It is recommended to reserve the two lowest priority interrupts (IVG15 and IVG14) for software interrupt handlers.

Servicing Interrupts

The Core Event Controller (CEC) has a single interrupt queueing element per event—a bit in the ILAT register. The appropriate ILAT bit is set when an interrupt rising edge is detected (which takes two core clock cycles) and cleared when the respective IPEND register bit is set. The IPEND bit indicates that the event vector has entered the core pipeline. At this point, the CEC recognizes and queues the next rising edge event on the corresponding interrupt input. The minimum latency from the rising edge transition of the general-purpose interrupt to the IPEND output assertion is three core clock cycles. However, the latency can be much higher, depending on the core's activity level and state.

To determine when to service an interrupt, the controller logically ANDs the three quantities in ILAT, IMASK, and the current processor priority level.

Servicing the highest priority interrupt involves these actions:

1. The interrupt vector in the Event Vector Table (EVT) becomes the next fetch address. On an interrupt, most instructions currently in the pipeline are aborted. On a service exception, all instructions after the excepting instruction are aborted. On an error exception, the excepting instruction and all instructions after it are aborted.
2. The return address is saved in the appropriate return register. The return register is `RETI` for interrupts, `RETX` for exceptions, `RETN` for NMIs, and `RETE` for debug emulation. The return address is the address of the instruction after the last instruction executed from normal program flow.
3. Processor mode is set to the level of the event taken. If the event is an NMI, exception, or interrupt, the processor mode is Supervisor. If the event is an emulation exception, the processor mode is Emulation.
4. Before the first instruction starts execution, the corresponding interrupt bit in `ILAT` is cleared and the corresponding bit in `IPEND` is set. Bit `IPEND[4]` is also set to disable all interrupts until the return address in `RETI` is saved.

Nesting of Interrupts

Interrupts are handled either with or without nesting.

Non-Nested Interrupts

If interrupts do not require nesting, all interrupts are disabled during the interrupt service routine. Note, however, that emulation, NMI, and exceptions are still accepted by the system.

When the system does not need to support nested interrupts, there is no need to store the return address held in RETI. Only the portion of the machine state used in the interrupt service routine must be saved in the Supervisor stack. To return from a non-nested interrupt service routine, only the RTI instruction must be executed, because the return address is already held in the RETI register.

Figure 4-15 shows an example of interrupt handling where interrupts are globally disabled for the entire interrupt service routine.

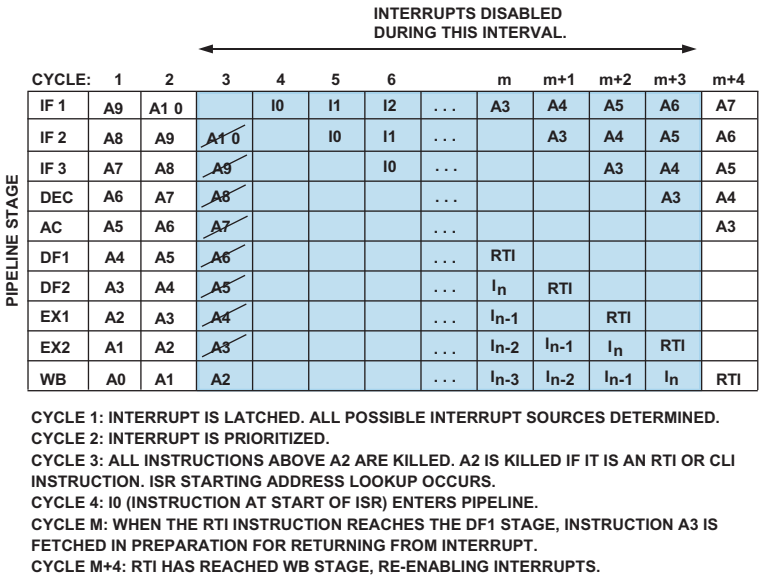


Figure 4-15. Non-Nested Interrupt Handling

Nested Interrupts

If interrupts require nesting, the return address to the interrupted point in the original interrupt service routine must be explicitly saved and subsequently restored when execution of the nested interrupt service routine

has completed. The first instruction in an interrupt service routine that supports nesting must save the return address currently held in `RETI` by pushing it onto the Supervisor stack (`[--SP] = RETI`). This clears the global interrupt disable bit `IPEND[4]`, enabling interrupts. Next, all registers that are modified by the interrupt service routine are saved onto the Supervisor stack. Processor state is stored in the Supervisor stack, not in the User stack. Hence, the instructions to push `RETI` (`[--SP] = RETI`) and pop `RETI` (`RETI = [SP++]`) use the Supervisor stack.

Figure 4-16 illustrates that by pushing `RETI` onto the stack, interrupts can be re-enabled during an interrupt service routine, resulting in a short duration where interrupts are globally disabled.

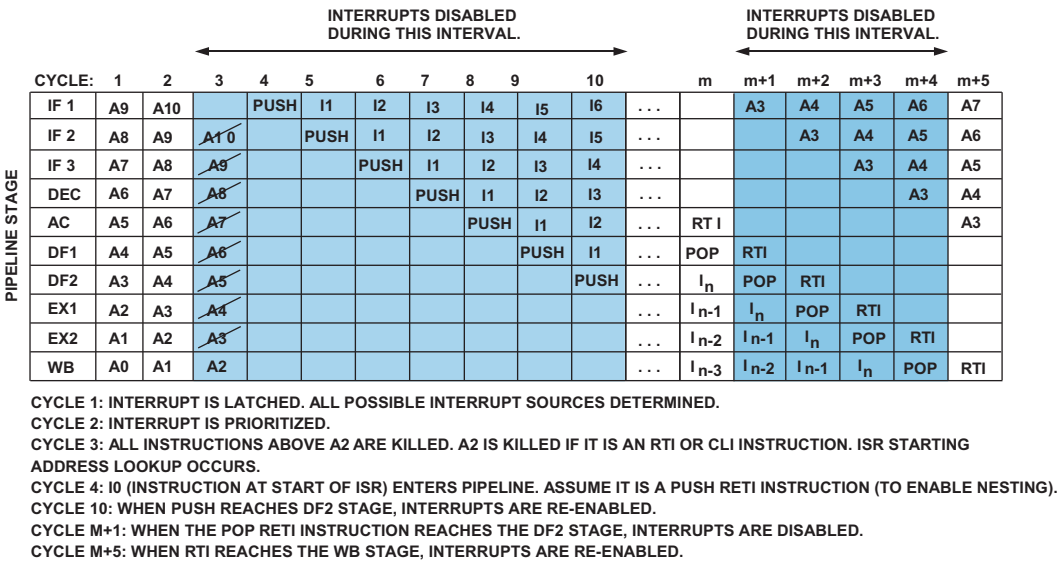


Figure 4-16. Nested Interrupt Handling

Example Prolog Code for Nested Interrupt Service Routine

Listing 4-2. Prolog Code for Nested ISR

```
/* Prolog code for nested interrupt service routine.
Push return address in RETI into Supervisor stack, ensuring that
interrupts are back on. Until now, interrupts have been
suspended.*/
ISR:
[--SP] = RETI ; /* Enables interrupts and saves return address to
stack */
[--SP] = ASTAT ;
[--SP] = FP ;
[-- SP] = (R7:0, P5:0) ;
/* Body of service routine. Note none of the processor resources
(accumulators, DAGs, loop counters and bounds) have been saved.
It is assumed this interrupt service routine does not use the
processor resources. */
```

Example Epilog Code for Nested Interrupt Service Routine

Listing 4-3. Epilog Code for Nested ISR

```
/* Epilog code for nested interrupt service routine.
Restore ASTAT, Data and Pointer registers. Popping RETI from
Supervisor stack ensures that interrupts are suspended between
load of return address and RTI. */
(R7:0, P5:0) = [SP++] ;
FP    = [SP++] ;
ASTAT = [SP++] ;
RETI  = [SP++] ;
```

```
/* Execute RTI, which jumps to return address, re-enables inter-
rupts, and switches to User mode if this is the last nested
interrupt in service. */
RTI;
```

The RTI instruction causes the return from an interrupt. The return address is popped into the RETI register from the stack, an action that suspends interrupts from the time that RETI is restored until RTI finishes executing. The suspension of interrupts prevents a subsequent interrupt from corrupting the RETI register.

Next, the RTI instruction clears the highest priority bit that is currently set in IPEND. The processor then jumps to the address pointed to by the value in the RETI register and re-enables interrupts by clearing IPEND[4].

Logging of Nested Interrupt Requests

The System Interrupt Controller (SIC) detects level-sensitive interrupt requests from the peripherals. The Core Event Controller (CEC) provides edge-sensitive detection for its general-purpose interrupts (IVG7-IVG15). Consequently, the SIC generates a synchronous interrupt pulse to the CEC and then waits for interrupt acknowledgement from the CEC. When the interrupt has been acknowledged by the core (via assertion of the appropriate IPEND output), the SIC generates another synchronous interrupt pulse to the CEC if the peripheral interrupt is still asserted. This way, the system does not lose peripheral interrupt requests that occur during servicing of another interrupt.

Multiple interrupt sources can map to a single core processor general-purpose interrupt. Because of this, multiple pulse assertions from the SIC can occur simultaneously, before, or during interrupt processing for an interrupt event that is already detected on this interrupt input. For a shared interrupt, the IPEND interrupt acknowledge mechanism described above re-enables all shared interrupts. If any of the shared interrupt sources are

still asserted, at least one pulse is again generated by the SIC. The Interrupt Status registers indicate the current state of the shared interrupt sources.

Exception Handling

Interrupts and exceptions treat instructions in the pipeline differently.

- When an interrupt occurs, all instructions in the pipeline are aborted.
- When an exception occurs, all instructions in the pipeline after the excepting instruction are aborted. For error exceptions, the excepting instruction is also aborted.

Because exceptions, NMIs, and emulation events have a dedicated return register, guarding the return address is optional. Consequently, the `PUSH` and `POP` instructions for exceptions, NMIs, and emulation events do not affect the interrupt system.

Note, however, the return instructions for exceptions (`RTX`, `RTN`, and `RTE`) do clear the Least Significant Bit (LSB) currently set in `IPEND`.

Deferring Exception Processing

Exception handlers are usually long routines, because they must discriminate among several exception causes and take corrective action accordingly. The length of the routines may result in long periods during which the interrupt system is, in effect, suspended.

To avoid lengthy suspension of interrupts, write the exception handler to identify the exception cause, but defer the processing to a low priority interrupt. To set up the low priority interrupt handler, use the Force Interrupt / Reset instruction (`RAISE`).

When deferring the processing of an exception to lower priority interrupt IVGx, the system must guarantee that IVGx is entered before returning to the application-level code that issued the exception. If a pending interrupt of higher priority than IVGx occurs, it is acceptable to enter the high priority interrupt before IVGx.

Example Code for an Exception Handler

The following code is for an exception routine handler with deferred processing.

Listing 4-4. Exception Routine Handler With Deferred Processing

```
/* Determine exception cause by examining EXCAUSE field in
SEQSTAT (first save contents of R0, P0, P1 and ASTAT in Supervi-
sor SP) */
[--SP] = R0 ;
[--SP] = P0 ;
[--SP] = P1 ;
[--SP] = ASTAT ;
R0 = SEQSTAT ;
/* Mask the contents of SEQSTAT, and leave only EXCAUSE in R0 */
R0 <<= 26 ;
R0 >>= 26 ;
/* Using jump table EVTABLE, jump to the event pointed to by R0
*/
P0 = R0 ;
P1 = _EVTABLE ;
P0 = P1 + ( P0 << 1 ) ;
R0 = W [ P0 ] (Z) ;
P1 = R0 ;
JUMP (PC + P1) ;
```

```
/* The entry point for an event is as follows. Here, processing
is deferred to low priority interrupt IVG15. Also, parameter
passing would typically be done here. */
_EVENT1:
RAISE 15 ;
JUMP.S _EXIT ;
/* Entry for event at IVG14 */
_EVENT2:
RAISE 14 ;
JUMP.S _EXIT ;
/* Comments for other events */
/* At the end of handler, restore R0, P0, P1 and ASTAT, and
return. */
_EXIT:
ASTAT = [SP++] ;
P1 = [SP++] ;
P0 = [SP++] ;
R0 = [SP++] ;
RTX ;
_EVTABLE:
.byte2 addr_event1;
.byte2 addr_event2;
...
.byte2 addr_eventN;
/* The jump table EVTABLE holds 16-bit address offsets for each
event. With offsets, this code is position independent and the
table is small.
+--------------+
| addr_event1  | _EVTABLE
+--------------+
| addr_event2  | _EVTABLE + 2
+--------------+
|     . . .    |
+--------------+
```

```
| addr_eventN  | _EVTABLE + 2N
+--------------+
*/
```

Example Code for an Exception Routine

The following code provides an example framework for an interrupt routine jumped to from an exception handler such as that described above.

Listing 4-5. Interrupt Routine for Handling Exception

```
[--SP] = RETI ;   /* Push return address on stack. */

/* Put body of routine here.*/

RETI = [SP++] ;   /* To return, pop return address and jump. */

RTI ;   /* Return from interrupt. */
```

Example Code for Using Hardware Loops in an ISR

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

Listing 4-6. Saving and Restoring With Hardware Loops

```
lhandler:
<Save other registers here>
[--SP] = LC0;  /* save loop 0 */
[--SP] = LB0;
[--SP] = LT0;

<Handler code here>
```

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

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

<Restore other registers here>

RTI;
```

Additional Usability Issues

The following sections describe additional usability issues.

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

Instructions `RTX`, `RTN`, and `RTE` are designed to return from an exception, NMI, or emulator event, respectively. Do not use them to return from a lower priority event. To return from an interrupt, use the `RTI` instruction. Failure to use the correct instruction may produce unintended results.

In the case of `RTX`, bit `IPEND[3]` is cleared. In the case of `RTI`, the bit of the highest priority interrupt in `IPEND` is cleared.

Allocating the System Stack

The software stack model for processing exceptions implies that the Supervisor stack must never generate an exception while the exception handler is saving its state. However, if the Supervisor stack grows past a CPLB entry or SRAM block, it may, in fact, generate an exception.

To guarantee that the Supervisor stack never generates an exception—never overflows past a CPLB entry or SRAM block while executing the exception handler—calculate the maximum space that all interrupt service routines and the exception handler occupy while they are active, and then allocate this amount of SRAM memory.

Latency in Servicing Events

In some processor architectures, if instructions are executed from external memory and an interrupt occurs while the instruction fetch operation is underway, then the interrupt is held off from being serviced until the current fetch operation has completed. Consider a processor operating at 300 MHz and executing code from external memory with 100 ns access times. Depending on when the interrupt occurs in the instruction fetch operation, the interrupt service routine may be held off for around 30 instruction clock cycles. When cache line fill operations are taken into account, the interrupt service routine could be held off for many hundreds of cycles.

In order for high priority interrupts to be serviced with the least latency possible, the processor allows any high latency fill operation to be completed at the system level, while an interrupt service routine executes from L1 memory. See Figure 4-17.

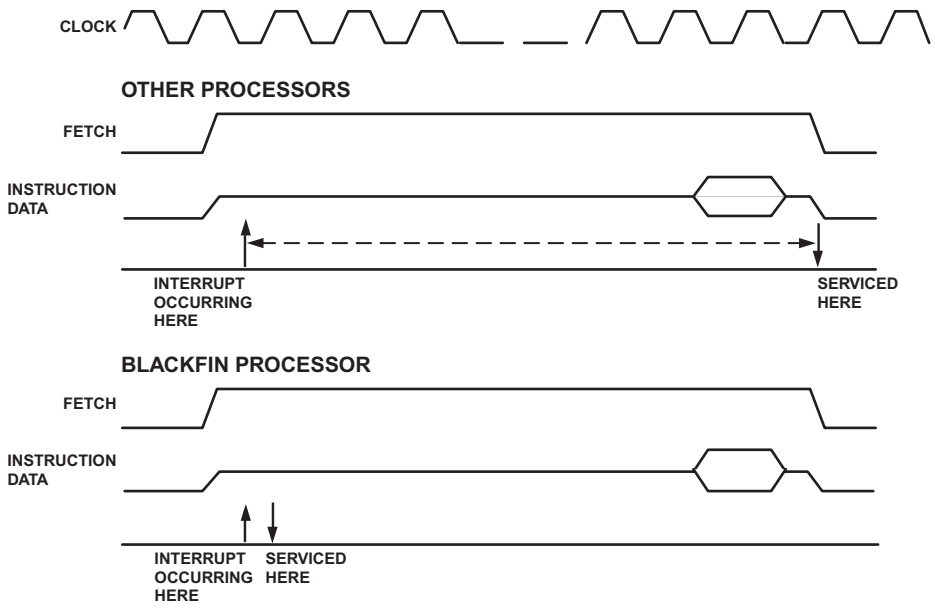


Figure 4-17. Minimizing Latency in Servicing an Interrupt Service Routine

If an instruction load operation misses the L1 instruction cache and generates a high latency line fill operation, then when an interrupt occurs, it is not held off until the fill has completed. Instead, the processor executes the interrupt service routine in its new context, and the cache fill operation completes in the background.

Note the interrupt service routine must reside in L1 cache or SRAM memory and must not generate a cache miss, an L2 memory access, or a peripheral access, as the processor is already busy completing the original cache line fill operation. If a load or store operation is executed in the

interrupt service routine requiring one of these accesses, then the interrupt service routine is held off while the original external access is completed, before initiating the new load or store.

If the interrupt service routine finishes execution before the load operation has completed, then the processor continues to stall, waiting for the fill to complete.

This same behavior is also exhibited for stalls involving reads of slow data memory or peripherals.

Writes to slow memory generally do not show this behavior, as the writes are deemed to be single cycle, being immediately transferred to the write buffer for subsequent execution.

For detailed information about cache and memory structures, see Chapter 6, “Memory.”

5 DATA ADDRESS GENERATORS

The Data Address Generators (DAGs) generate addresses for data moves to and from memory. By generating addresses, the DAGs let programs refer to addresses indirectly, using a DAG register instead of an absolute address.

The DAG architecture, shown in Figure 5-1, supports several functions that minimize overhead in data access routines. These functions include:

- Supply address – Provides an address during a data access
- Supply address and post-modify – Provides an address during a data move and auto-increments/decrements the stored address for the next move
- Supply address with offset – Provides an address from a base with an offset without incrementing the original address pointer
- Modify address – Increments or decrements the stored address without performing a data move
- Bit-reversed carry address – Provides a bit-reversed carry address during a data move without reversing the stored address

The DAG subsystem comprises two DAG Arithmetic units, nine Pointer registers, four Index registers and four complete sets of related Modify, Base, and Length registers. These registers hold the values that the DAGs use to generate addresses. The types of registers are:

- Index registers, `I[3:0]`. Unsigned 32-bit Index registers hold an address pointer to memory. For example, the instruction `R3 = [I0]` loads the data value found at the memory location pointed to by the register `I0`. Index registers can be used for 16- and 32-bit memory accesses.
- Modify registers, `M[3:0]`. Signed 32-bit Modify registers provide the increment or step size by which an Index register is post-modified during a register move. For example, the `R0 = [I0 ++ M1]` instruction directs the DAG to:

  – Output the address in register `I0`
  – Load the contents of the memory location pointed to by `I0` into `R0`
  – Modify the contents of `I0` by the value contained in the `M1` register
- Base and Length registers, `B[3:0]` and `L[3:0]`. Unsigned 32-bit Base and Length registers set up the range of addresses and the starting address of a circular buffer. Each `B`, `L` pair is always coupled with a corresponding I-register, for example, `I3`, `B3`, `L3`. For more information on circular buffers, see "Addressing Circular Buffers" on page 5-6.
- Pointer registers, `P[5:0]`, `FP`, `USP`, and `SP`. 32-bit Pointer registers hold an address pointer to memory. The `P[5:0]` field, `FP` (Frame Pointer) and `SP/USP` (Stack Pointer/User Stack Pointer) can be manipulated and used in various instructions. For example, the instruction `R3 = [P0]` loads the register `R3` with the data value found at the memory location pointed to by the register `P0`. The Pointer registers have no effect on circular buffer addressing. They

can be used for 8-, 16-, and 32-bit memory accesses. For added mode protection, SP is accessible only in Supervisor mode, while USP is accessible in User mode.

Do not assume the L-registers are automatically initialized to zero for linear addressing. The I-, M-, L-, and B-registers contain random values after reset. For each I-register used, programs must initialize the corresponding L-registers to zero for linear addressing or to the buffer length for circular buffer addressing.

Note all DAG registers must be initialized individually. Initializing a B-register does not automatically initialize the I-register.

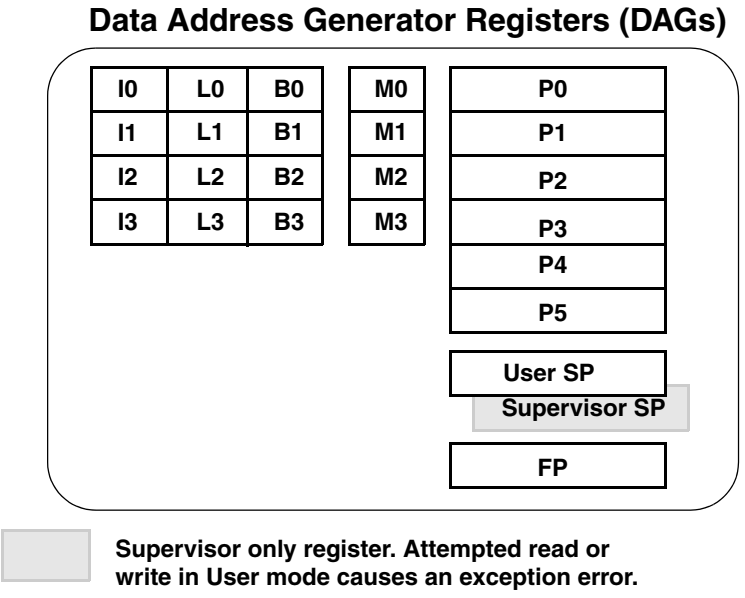


Figure 5-1. Processor DAG Registers

Addressing With DAGs

The DAGs can generate an address that is incremented by a value or by a register. In post-modify addressing, the DAG outputs the I-register value unchanged; then the DAG adds an M-register or immediate value to the I-register.

In indexed addressing, the DAG adds a small offset to the value in the P-register, but does not update the P-register with this new value, thus providing an offset for that particular memory access.

The processor is byte addressed. All data accesses must be aligned to the data size. In other words, a 32-bit fetch must be aligned to 32 bits, but an 8-bit store can be aligned to any byte. Depending on the type of data used, increments and decrements to the DAG registers can be by 1, 2, or 4 to match the 8-, 16-, or 32-bit accesses.

For example, consider the following instruction:

```
R0 = [ P3++ ];
```

This instruction fetches a 32-bit word, pointed to by the value in `P3`, and places it in `R0`. It then post-increments `P3` by *four*, maintaining alignment with the 32-bit access.

```
R0.L = W [ I3++ ];
```

This instruction fetches a 16-bit word, pointed to by the value in `I3`, and places it in the low half of the destination register, `R0.L`. It then post-increments `I3` by *two*, maintaining alignment with the 16-bit access.

```
R0 = B [ P3++ ] (Z) ;
```

This instruction fetches an 8-bit word, pointed to by the value in `P3`, and places it in the destination register, `R0`. It then post-increments `P3` by one, maintaining alignment with the 8-bit access. The byte value may be zero extended (as shown) or sign extended into the 32-bit data register.

Instructions using Index registers use an M-register or a small immediate value (+/– 2 or 4) as the modifier. Instructions using Pointer registers use a small immediate value or another P-register as the modifier. For details, see Table 5-3, “DAG Instruction Summary,” on page 5-18.

Frame and Stack Pointers

In many respects, the Frame and Stack Pointer registers perform like the other P-registers, `P[5:0]`. They can act as general pointers in any of the load/store instructions, for example, `R1 = B[SP] (Z)`. However, `FP` and `SP` have additional functionality.

The Stack Pointer registers include:

- a User Stack Pointer (`USP` in Supervisor mode, `SP` in User mode)
- a Supervisor Stack Pointer (`SP` in Supervisor mode)

The User Stack Pointer register and the Supervisor Stack Pointer register are accessed using the register alias `SP`. Depending on the current processor operating mode, only one of these registers is active and accessible as `SP`:

- In User mode, any reference to `SP` (for example, stack pop `R0 = [ SP++ ] ;`) implicitly uses the `USP` as the effective address.
- In Supervisor mode, the same reference to `SP` (for example, `R0 = [ SP++ ] ;`) implicitly uses the Supervisor Stack Pointer as the effective address.

  To manipulate the User Stack Pointer for code running in Supervisor mode, use the register alias `USP`. When in Supervisor mode, a register move from `USP` (for example, `R0 = USP ;`) moves the current User Stack Pointer into `R0`. The register alias `USP` can only be used in Supervisor mode.

Some load/store instructions use FP and SP implicitly:

- FP-indexed load/store, which extends the addressing range for 16-bit encoded load/stores
- Stack push/pop instructions, including those for pushing and popping multiple registers
- Link/unlink instructions, which control stack frame space and manage the Frame Pointer register (FP) for that space

Addressing Circular Buffers

The DAGs support addressing circular buffers. Circular buffers are a range of addresses containing data that the DAG steps through repeatedly, wrapping around to repeat stepping through the same range of addresses in a circular pattern.

The DAGs use four types of DAG registers for addressing circular buffers. For circular buffering, the registers operate this way:

- The Index (I) register contains the value that the DAG outputs on the address bus.
- 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. Any M-register can be used with any I-register. The modify value can also be an immediate value instead of an M-register. The size of the modify value must be less than or equal to the length (L-register) of the circular buffer.

- The Length (L) register sets the size of the circular buffer and the address range through which the DAG circulates the I-register. L is positive and cannot have a value greater than $2^{32} - 1$. If an L-register's value is zero, its circular buffer operation is disabled.
- The Base (B) register or the B-register plus the L-register is the value with which the DAG compares the modified I-register value after each access.

To address a circular buffer, the DAG steps the Index pointer (I-register) through the buffer values, post-modifying and updating the index on each access with a positive or negative modify value from the M-register.

If the Index pointer falls outside the buffer range, the DAG subtracts the length of the buffer (L-register) from the value or adds the length of the buffer to the value, wrapping the Index pointer back to a point inside the buffer.

The starting address that the DAG wraps around is called the buffer's base address (B-register). There are no restrictions on the value of the base address for circular buffers that contains 8-bit data. Circular buffers that contain 16- and 32-bit data must be 16-bit aligned and 32-bit aligned, respectively. Exceptions can be made for video operations. For more information, see "Memory Address Alignment" on page 5-13. Circular buffering uses post-modify addressing.

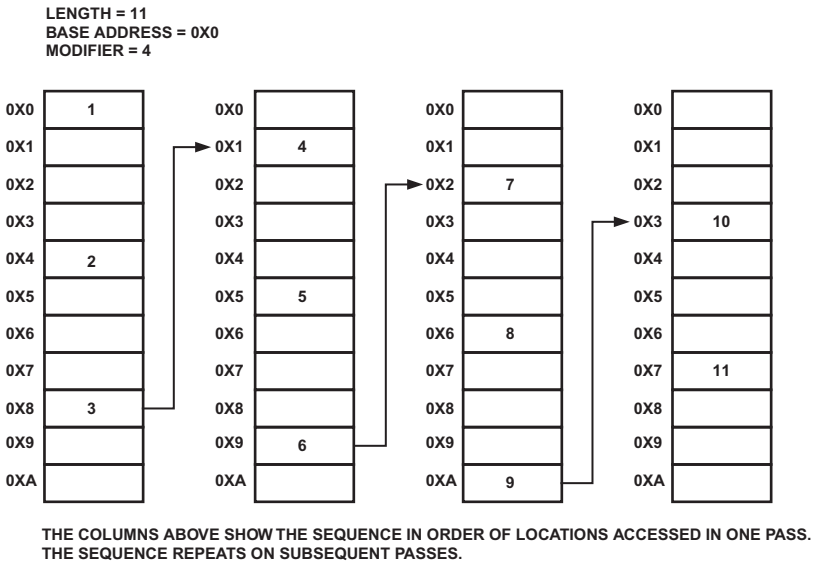


Figure 5-2. Circular Data Buffers

As seen in Figure 5-2, 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 index value exceeds the buffer length, the DAG subtracts (for a positive modify value) or adds (for a negative modify value) 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, shown for "I+M" operations.

- If M is positive:

  $I_{new} = I_{old} + M$
  if $I_{old} + M <$ buffer base + length (end of buffer)

  $I_{new} = I_{old} + M - L$
  if $I_{old} + M \geq$ buffer base + length (end of buffer)

- If M is negative:

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

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

Addressing With Bit-Reversed Addresses

To obtain results in sequential order, programs need bit-reversed carry addressing for some algorithms, particularly Fast Fourier Transform (FFT) calculations. To satisfy the requirements of these algorithms, the DAG's bit-reversed addressing feature permits repeatedly subdividing data sequences and storing this data in bit-reversed order. For detailed information about bit-reversed addressing, see the Modify-Increment instruction in the *Blackfin Processor Programming Reference*.

Indexed Addressing With Index and Pointer Registers

Indexed addressing uses the value in the Index or Pointer register as an effective address. This instruction can load or store 16- or 32-bit values. The default is a 32-bit transfer. If a 16-bit transfer is required, then the *W* designator is used to preface the load or store.

For example:

```
R0 = [ I2 ];
```

loads a 32-bit value from an address pointed to by `I2` and stores it in the destination register `R0`.

```
R0.H = W [ I2 ];
```

loads a 16-bit value from an address pointed to by `I2` and stores it in the 16-bit destination register `R0.H`.

```
[ P1 ] = R0 ;
```

is an example of a 32-bit store operation.

Pointer registers can be used for 8-bit loads and stores.

For example:

```
B [ P1++] = R0 ;
```

stores the 8-bit value from the `R0` register in the address pointed to by the `P1` register, then increments the `P1` register.

Auto-Increment and Auto-Decrement Addressing

Auto-increment addressing updates the Pointer and Index registers after the access. The amount of increment depends on the word size. An access of 32-bit words results in an update of the Pointer by 4. A 16-bit word

access updates the Pointer by 2, and an access of an 8-bit word updates the Pointer by 1. Both 8- and 16-bit read operations may specify either to sign-extend or zero-extend the contents into the destination register. Pointer registers may be used for 8-, 16-, and 32-bit accesses while Index registers may be used only for 16- and 32-bit accesses.

For example:

```
R0 = W [ P1++ ](Z);
```

loads a 16-bit word into a 32-bit destination register from an address pointed to by the `P1` Pointer register. The Pointer is then incremented by 2 and the word is zero extended to fill the 32-bit destination register.

Auto-decrement works the same way by decrementing the address after the access.

For example:

```
R0 = [ I2-- ] ;
```

loads a 32-bit value into the destination register and decrements the Index register by 4.

Pre-Modify Stack Pointer Addressing

The only pre-modify instruction in the processor uses the Stack Pointer register, `SP`. The address in `SP` is decremented by 4 and then used as an effective address for the store. The instruction `[ --SP ] = R0` ; is used for stack push operations and can support only a 32-bit word transfer.

Indexed Addressing With Immediate Offset

Indexed addressing allows programs to obtain values from data tables, with reference to the base of that table. The Pointer register is modified by the immediate field and then used as the effective address. The value of the Pointer register is not updated.

Alignment exceptions are triggered when a final address is unaligned.

For example, if `P1 = 0x13`, then `[P1 + 0x11]` would effectively be equal to `[0x24]`, which is aligned for all accesses.

Post-Modify Addressing

Post-modify addressing uses the value in the Index or Pointer registers as the effective address and then modifies it by the contents of another register. Pointer registers are modified by other Pointer registers. Index registers are modified by Modify registers. Post-modify addressing does not support the Pointer registers as destination registers, nor does it support byte-addressing.

For example:

```
R5 = [ P1++P2 ] ;
```

loads a 32-bit value into the `R5` register, found in the memory location pointed to by the `P1` register.

The value in the `P2` register is then added to the value in the `P1` register.

For example:

```
R2 = W [ P4++P5 ] (Z) ;
```

loads a 16-bit word into the low half of the destination register `R2` and zero-extends it to 32 bits. The value of the pointer `P4` is incremented by the value of the pointer `P5`.

For example:

```
R2 = [ I2++M1 ];
```

loads a 32-bit word into the destination register `R2`. The value in the Index register, `I2`, is updated by the value in the Modify register, `M1`.

Modifying DAG and Pointer Registers

The DAGs support operations that modify an address value in an Index register without outputting an address. The operation, address-modify, is useful for maintaining pointers.

The address-modify operation modifies addresses in any DAG Index and Pointer register (`I[3:0]`, `P[5:0]`, `FP`, `SP`) without accessing memory. If the Index register's corresponding B- and L-registers are set up for circular buffering, the address-modify operation performs the specified buffer wraparound (if needed).

The syntax is similar to post-modify addressing (index += modifier). For Index registers, an M-register is used as the modifier. For Pointer registers, another P-register is used as the modifier.

Consider the example, `I1 += M2 ;`

This instruction adds `M2` to `I1` and updates `I1` with the new value.

Memory Address Alignment

The processor requires proper memory alignment to be maintained for the data size being accessed. Unless exceptions are disabled, violations of memory alignment cause an alignment exception. Some instructions—for example, many of the Video ALU instructions—automatically disable

alignment exceptions because the data may not be properly aligned when stored in memory. Alignment exceptions may be disabled by issuing the DISALGNEXPT instruction in parallel with a load/store operation.

Normally, the memory system requires two address alignments:

- 32-bit word load/stores are accessed on four-byte boundaries, meaning the two least significant bits of the address are b#00.
- 16-bit word load/stores are accessed on two-byte boundaries, meaning the least significant bit of the address must be b#0.

Table 5-1 summarizes the types of transfers and transfer sizes supported by the addressing modes.

Be careful when using the DISALGNEXPT instruction, because it disables automatic detection of memory alignment errors. The DISALGNEXPT instruction only affects misaligned loads that use I-register indirect addressing. Misaligned loads using P-register addressing will still cause an exception.

Table 5-1. Types of Transfers Supported and Transfer Sizes

Addressing Mode	Types of Transfers Supported	Transfer Sizes
Auto-increment Auto-decrement Indirect Indexed	To and from Data Registers	LOADS: 32-bit word 16-bit, zero extended half word 16-bit, sign extended half word 8-bit, zero extended byte 8-bit, sign extended byte STORES: 32-bit word 16-bit half word 8-bit byte
	To and from Pointer Registers	LOAD: 32-bit word STORE: 32-bit word
Post-increment	To and from Data Registers	LOADS: 32-bit word 16-bit half word to Data Register high half 16-bit half word to Data Register low half 16-bit, zero extended half word 16-bit, sign extended half word STORES: 32-bit word 16-bit half word from Data Register high half 16-bit half word from Data Register low half

Table 5-2 summarizes the addressing modes. In the table, an asterisk (*) indicates the processor supports the addressing mode.

Table 5-2. Addressing Modes

	32-bit word	16-bit half-word	8-bit byte	Sign/zero extend	Data Register	Pointer register	Data Register Half
P Auto-inc [P0++]	*	*	*	*	*	*	
P Auto-dec [P0--]	*	*	*	*	*	*	
P Indirect [P0]	*	*	*	*	*	*	*
P Indexed [P0+im]	*	*	*	*	*	*	
FP indexed [FP+im]	*				*	*	
P Post-inc [P0++P1]	*	*		*	*		*
I Auto-inc [I0++]	*	*			*		*
I Auto-dec [I0--]	*	*			*		*
I Indirect [I0]	*	*			*		*
I Post-inc [I0++M0]	*				*		

DAG Instruction Summary

Table 5-3 lists the DAG instructions. For more information on assembly language syntax, see the *Blackfin Processor Programming Reference*. In Table 5-3, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- Preg denotes any Pointer register, `FP`, or `SP` register.
- Ireg denotes any DAG Index register.
- Mreg denotes any DAG Modify register.
- W denotes a 16-bit wide value.
- B denotes an 8-bit wide value.
- immA denotes a signed, A-bits wide, immediate value.
- uimmAmB denotes an unsigned, A-bits wide, immediate value that is an even multiple of B.
- Z denotes the zero-extension qualifier.
- X denotes the sign-extension qualifier.
- BREV denotes the bit-reversal qualifier.

The *Blackfin Processor Programming Reference* more fully describes the options that may be applied to these instructions and the sizes of immediate fields.

DAG instructions do not affect the ASTAT Status flags.

Table 5-3. DAG Instruction Summary

Instruction
Preg = [Preg] ;
Preg = [Preg ++] ;
Preg = [Preg --] ;
Preg = [Preg + uimm6m4] ;
Preg = [Preg + uimm17m4] ;
Preg = [Preg – uimm17m4] ;
Preg = [FP – uimm7m4] ;
Dreg = [Preg] ;
Dreg = [Preg ++] ;
Dreg = [Preg --] ;
Dreg = [Preg + uimm6m4] ;
Dreg = [Preg + uimm17m4] ;
Dreg = [Preg – uimm17m4] ;
Dreg = [Preg ++ Preg] ;
Dreg = [FP – uimm7m4] ;
Dreg = [Ireg] ;
Dreg = [Ireg ++] ;
Dreg = [Ireg --] ;
Dreg = [Ireg ++ Mreg] ;
Dreg =W [Preg] (Z) ;
Dreg =W [Preg ++] (Z) ;
Dreg =W [Preg --] (Z) ;
Dreg =W [Preg + uimm5m2] (Z) ;

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
Dreg =W [Preg + uimm16m2] (Z) ;
Dreg =W [Preg – uimm16m2] (Z) ;
Dreg =W [Preg ++ Preg] (Z) ;
Dreg = W [Preg] (X) ;
Dreg = W [Preg ++] (X) ;
Dreg = W [Preg --] (X) ;
Dreg =W [Preg + uimm5m2] (X) ;
Dreg =W [Preg + uimm16m2] (X) ;
Dreg =W [Preg – uimm16m2] (X) ;
Dreg =W [Preg ++ Preg] (X) ;
Dreg_hi = W [Ireg] ;
Dreg_hi = W [Ireg ++] ;
Dreg_hi = W [Ireg --] ;
Dreg_hi = W [Preg] ;
Dreg_hi = W [Preg ++ Preg] ;
Dreg_lo = W [Ireg] ;
Dreg_lo = W [Ireg ++] ;
Dreg_lo = W [Ireg --] ;
Dreg_lo = W [Preg] ;
Dreg_lo = W [Preg ++ Preg] ;
Dreg = B [Preg] (Z) ;
Dreg = B [Preg ++] (Z) ;
Dreg = B [Preg --] (Z) ;
Dreg = B [Preg + uimm15] (Z) ;

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
Dreg = B [Preg – uimm15] (Z) ;
Dreg = B [Preg] (X) ;
Dreg = B [Preg ++] (X) ;
Dreg = B [Preg --] (X) ;
Dreg = B [Preg + uimm15] (X) ;
Dreg = B [Preg – uimm15] (X) ;
[Preg] = Preg ;
[Preg ++] = Preg ;
[Preg --] = Preg ;
[Preg + uimm6m4] = Preg ;
[Preg + uimm17m4] = Preg ;
[Preg – uimm17m4] = Preg ;
[FP – uimm7m4] = Preg ;
[Preg] = Dreg ;
[Preg ++] = Dreg ;
[Preg --] = Dreg ;
[Preg + uimm6m4] = Dreg ;
[Preg + uimm17m4] = Dreg ;
[Preg – uimm17m4] = Dreg ;
[Preg ++ Preg] = Dreg ;
[FP – uimm7m4] = Dreg ;
[Ireg] = Dreg ;
[Ireg ++] = Dreg ;
[Ireg --] = Dreg ;

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
[Ireg ++ Mreg] = Dreg ;
W [Ireg] = Dreg_hi ;
W [Ireg ++] = Dreg_hi ;
W [Ireg --] = Dreg_hi ;
W [Preg] = Dreg_hi ;
W [Preg ++ Preg] = Dreg_hi ;
W [Ireg] = Dreg_lo ;
W [Ireg ++] = Dreg_lo ;
W [Ireg --] = Dreg_lo ;
W [Preg] = Dreg_lo ;
W [Preg] = Dreg ;
W [Preg ++] = Dreg ;
W [Preg --] = Dreg ;
W [Preg + uimm5m2] = Dreg ;
W [Preg + uimm16m2] = Dreg ;
W [Preg – uimm16m2] = Dreg ;
W [Preg ++ Preg] = Dreg_lo ;
B [Preg] = Dreg ;
B [Preg ++] = Dreg ;
B [Preg --] = Dreg ;
B [Preg + uimm15] = Dreg ;
B [Preg – uimm15] = Dreg ;
Preg = imm7 (X) ;
Preg = imm16 (X) ;

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
Preg += Preg (BREV) ;
Ireg += Mreg (BREV) ;
Preg = Preg << 2 ;
Preg = Preg >> 2 ;
Preg = Preg >> 1 ;
Preg = Preg + Preg << 1 ;
Preg = Preg + Preg << 2 ;
Preg –= Preg ;
Ireg –= Mreg ;

6 MEMORY

The processor supports a hierarchical memory model with different performance and size parameters, depending on the memory location within the hierarchy. Level 1 (L1) memories are located on the chip and are faster than the Level 2 (L2) memory systems. The Level 2 (L2) memories are off-chip and have longer access latencies. The faster L1 memories, which are typically small scratchpad memory or cache memories, are found within the core itself.

Memory Architecture

The processor has a unified 4G byte address range that spans a combination of on-chip and off-chip memory and memory-mapped I/O resources. Of this range, some of the address space is dedicated to internal, on-chip resources. The processor populates portions of this internal memory space with:

- L1 Static Random Access Memories (SRAM)
- A set of memory-mapped registers (MMRs)
- A boot Read-Only Memory (ROM)

A portion of the internal L1 SRAM can also be configured to run as cache. The processor also provides support for an external memory space that includes asynchronous memory space and synchronous DRAM (SDRAM) space. See Chapter 17, “External Bus Interface Unit,” for a detailed discussion of each of these memory regions and the controllers that support them.

Figure 6-1 provides an overview of the ADSP-BF533 processor system memory map. Figure 6-2 shows this information for the ADSP-BF532 processor, and Figure 6-3 for the ADSP-BF531 processor. Note the architecture does not define a separate I/O space. All resources are mapped through the flat 32-bit address space. The memory is byte-addressable.

As shown in Table 6-1, the ADSP-BF533, ADSP-BF532, and ADSP-BF531 processors offer a variety of instruction and data memory configurations.

Table 6-1. Memory Configurations

Type of Memory	ADSP-BF531	ADSP-BF532	ADSP-BF533
Instruction SRAM/Cache, lockable by Way or line	16K byte	16K byte	16K byte
Instruction SRAM	16K byte	32K byte	64K byte
Data SRAM/Cache	16K byte	32K byte	32K byte
Data SRAM	-	-	32K byte
Data Scratchpad SRAM	4K byte	4K byte	4K byte
Total	**52K byte**	**84K byte**	**148K byte**

The upper portion of internal memory space is allocated to the core and system MMRs. Accesses to this area are allowed only when the processor is in Supervisor or Emulation mode (see Chapter 3, “Operating Modes and States”).

The lowest 1K byte of internal memory space is occupied by the boot ROM. Depending on the booting option selected, the appropriate boot program is executed from this memory space when the processor is reset (see “Booting Methods” on page 3-18.)

Within the external memory map, four banks of asynchronous memory space and one bank of SDRAM memory are available. Each of the asynchronous banks is 1M byte and the SDRAM bank is up to 128M byte.

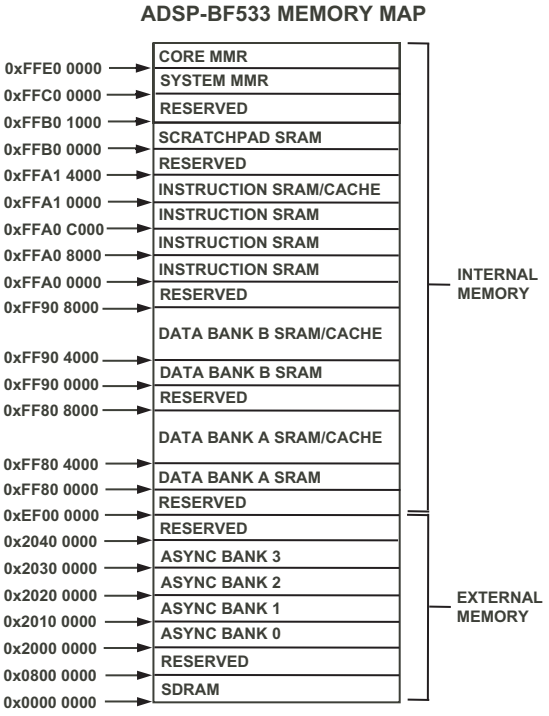


Figure 6-1. ADSP-BF533 Memory Map

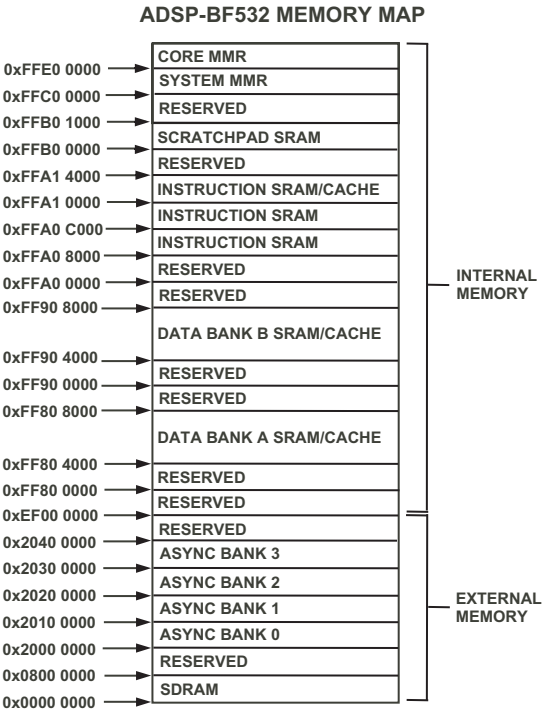


Figure 6-2. ADSP-BF532 Memory Map

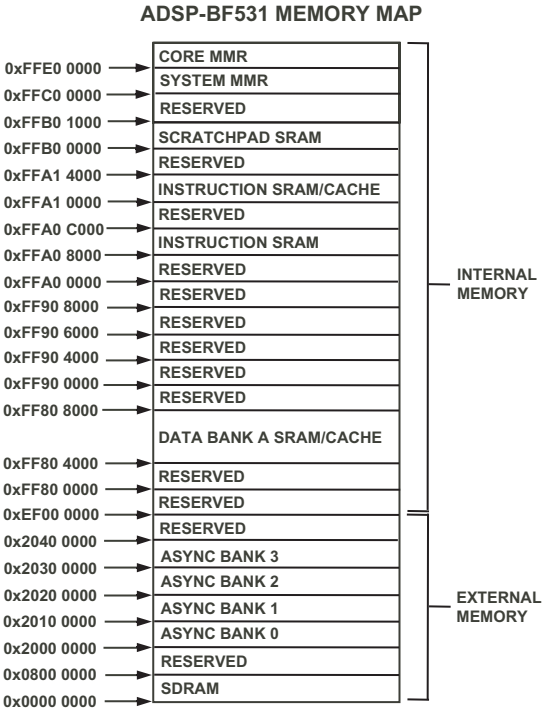


Figure 6-3. ADSP-BF531 Memory Map

Overview of Internal Memory

The L1 memory system performance provides high bandwidth and low latency. Because SRAMs provide deterministic access time and very high throughput, DSP systems have traditionally achieved performance improvements by providing fast SRAM on the chip.

The addition of instruction and data caches (SRAMs with cache control hardware) provides both high performance and a simple programming model. Caches eliminate the need to explicitly manage data movement

into and out of L1 memories. Code can be ported to or developed for the processor quickly without requiring performance optimization for the memory organization.

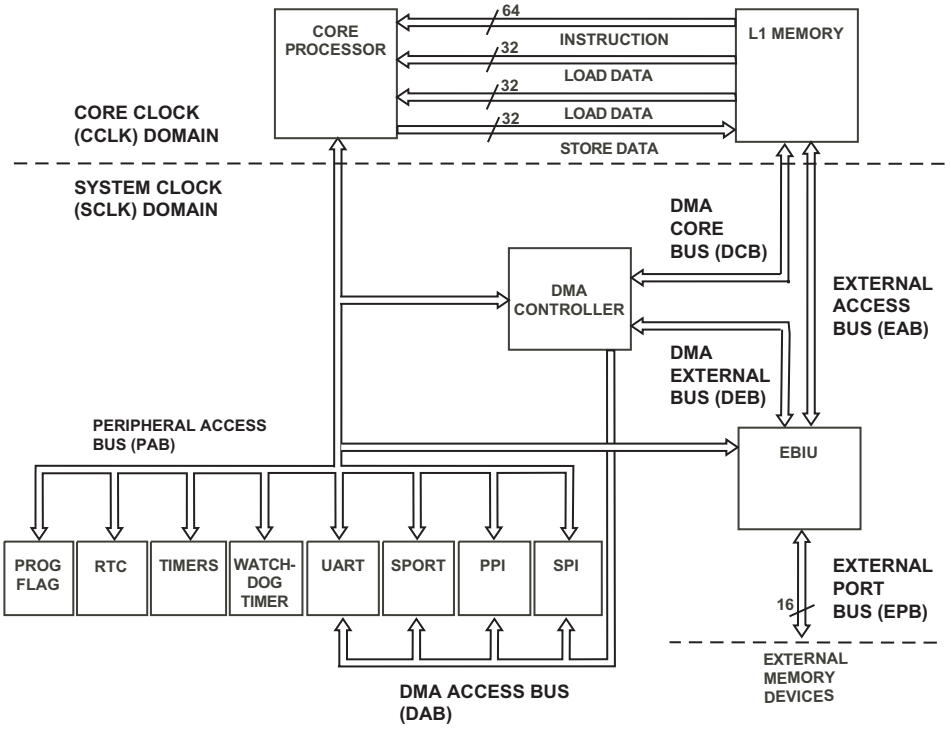


Figure 6-4. Processor Memory Architecture

The L1 memory provides:

- A modified Harvard architecture, allowing up to four core memory accesses per clock cycle (one 64-bit instruction fetch, two 32-bit data loads, and one pipelined 32-bit data store)
- Simultaneous system DMA, cache maintenance, and core accesses

- SRAM access at processor clock rate (CCLK) for critical DSP algorithms and fast context switching
- Instruction and data cache options for microcontroller code, excellent High Level Language (HLL) support, and ease of programming cache control instructions, such as PREFETCH and FLUSH
- Memory protection

The L1 memories operate at the core clock frequency (CCLK).

Overview of Scratchpad Data SRAM

The processor provides a dedicated 4K byte bank of scratchpad data SRAM. The scratchpad is independent of the configuration of the other L1 memory banks and cannot be configured as cache or targeted by DMA. Typical applications use the scratchpad data memory where speed is critical. For example, the User and Supervisor stacks should be mapped to the scratchpad memory for the fastest context switching during interrupt handling.

The L1 memories operate at the core clock frequency (CCLK).

Scratchpad data SRAM cannot be accessed by the DMA controller.

L1 Instruction Memory

L1 Instruction Memory consists of a combination of dedicated SRAM and banks which can be configured as SRAM or cache. For the 16K byte bank that can be either cache or SRAM, control bits in the IMEM_CONTROL register can be used to organize all four subbanks of the L1 Instruction Memory as:

- A simple SRAM
- A 4-Way, set associative instruction cache
- A cache with as many as four locked Ways

L1 Instruction Memory can be used only to store instructions.

IMEM_CONTROL Register

The Instruction Memory Control register (IMEM_CONTROL) contains control bits for the L1 Instruction Memory. By default after reset, cache and Cacheability Protection Lookaside Buffer (CPLB) address checking is disabled (see "L1 Instruction Cache" on page 6-12).

When the LRUPRIORST bit is set to 1, the cached states of all CPLB_LRUPRIO bits (see "ICPLB_DATAx Registers" on page 6-55) are cleared. This simultaneously forces all cached lines to be of equal (low) importance. Cache replacement policy is based first on line importance indicated by the cached states of the CPLB_LRUPRIO bits, and then on LRU (least recently used). See "Instruction Cache Locking by Line" on page 6-19 for complete details. This bit must be 0 to allow the state of the CPLB_LRUPRIO bits to be stored when new lines are cached.

The ILOC[3:0] bits provide a useful feature only after code has been manually loaded into cache. See "Instruction Cache Locking by Way" on page 6-20. These bits specify which Ways to remove from the cache replacement policy. This has the effect of locking code present in

nonparticipating Ways. Code in nonparticipating Ways can still be removed from the cache using an IFLUSH instruction. If an ILOC[3:0] bit is 0, the corresponding Way is not locked and that Way participates in cache replacement policy. If an ILOC[3:0] bit is 1, the corresponding Way is locked and does not participate in cache replacement policy.

The IMC bit reserves a portion of L1 instruction SRAM to serve as cache. Note reserving memory to serve as cache will not alone enable L2 memory accesses to be cached. CPLBs must also be enabled using the EN_ICPLB bit and the CPLB descriptors (ICPLB_DATAx and ICPLB_ADDRx registers) must specify desired memory pages as cache-enabled.

Instruction CPLBs are disabled by default after reset. When disabled, only minimal address checking is performed by the L1 memory interface. This minimal checking generates an exception to the processor whenever it attempts to fetch an instruction from:

- Reserved (nonpopulated) L1 instruction memory space
- L1 data memory space
- MMR space

CPLBs must be disabled using this bit prior to updating their descriptors (DCPLB_DATAx and DCPLB_ADDRx registers). Note since load store ordering is weak (see "Ordering of Loads and Stores" on page 6-67), disabling of CPLBs should be proceeded by a CSYNC.

When enabling or disabling cache or CPLBs, immediately follow the write to IMEM_CONTROL with a CSYNC to ensure proper behavior.

To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.

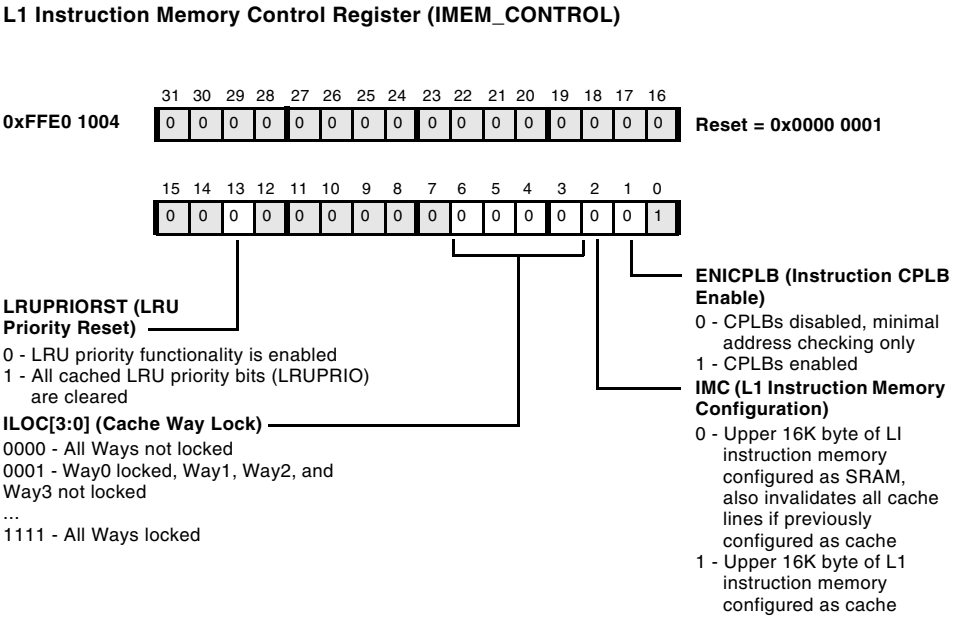


Figure 6-5. L1 Instruction Memory Control Register

L1 Instruction SRAM

The processor core reads the instruction memory through the 64-bit wide instruction fetch bus. All addresses from this bus are 64-bit aligned. Each instruction fetch can return any combination of 16-, 32- or 64-bit instructions (for example, four 16-bit instructions, two 16-bit instructions and one 32-bit instruction, or one 64-bit instruction).

The DAGs, which are described in Chapter 5, cannot access L1 Instruction Memory directly. A DAG reference to instruction memory SRAM space generates an exception (see “Exceptions” on page 4-41).

Write access to the L1 Instruction SRAM Memory must be made through the 64-bit wide system DMA port. Because the SRAM is implemented as a collection of single ported subbanks, the instruction memory is effectively dual ported.

Table 6-2 lists the memory start locations of the L1 Instruction Memory subbanks.

Table 6-2. L1 Instruction Memory Subbanks

Memory Subbank	Memory Start Location, ADSP-BF533	Memory Start Location, ADSP-BF532	Memory Start Location, ADSP-BF531
0	0xFFA0 0000	0xFFA0 8000	0xFFA0 8000
1	0xFFA0 1000	0xFFA0 9000	0xFFA0 9000
2	0xFFA0 2000	0xFFA0 A000	0xFFA0 A000
3	0xFFA0 3000	0xFFA0 B000	0xFFA0 B000
4	0xFFA0 4000	0xFFA0 C000	
5	0xFFA0 5000	0xFFA0 D000	
6	0xFFA0 6000	0xFFA0 E000	
7	0xFFA0 7000	0xFFA0 F000	
8	0xFFA0 8000		
9	0xFFA0 9000		
10	0xFFA0 A000		
11	0xFFA0 B000		
12	0xFFA0 C000		

Table 6-2. L1 Instruction Memory Subbanks (Cont'd)

Memory Subbank	Memory Start Location, ADSP-BF533	Memory Start Location, ADSP-BF532	Memory Start Location, ADSP-BF531
13	0xFFA0 D000		
14	0xFFA0 E000		
15	0xFFA0 F000		

Figure 6-6 describes the bank architecture of the L1 Instruction Memory. As the figure shows, each 16K byte bank is made up of four 4K byte subbanks.

L1 Instruction Cache

For information about cache terminology, see "Terminology" on page 6-74.

The L1 Instruction Memory may also be configured to contain a, 4-Way set associative instruction 16K byte cache. To improve the average access latency for critical code sections, each Way or line of the cache can be locked independently. When the memory is configured as cache, it cannot be accessed directly.

When cache is enabled, only memory pages further specified as cacheable by the CPLBs will be cached. When CPLBs are enabled, any memory location that is accessed must have an associated page definition available, or a CPLB exception is generated. CPLBs are described in "Memory Protection and Properties" on page 6-46.

Figure 6-7 shows the overall Blackfin processor instruction cache organization.

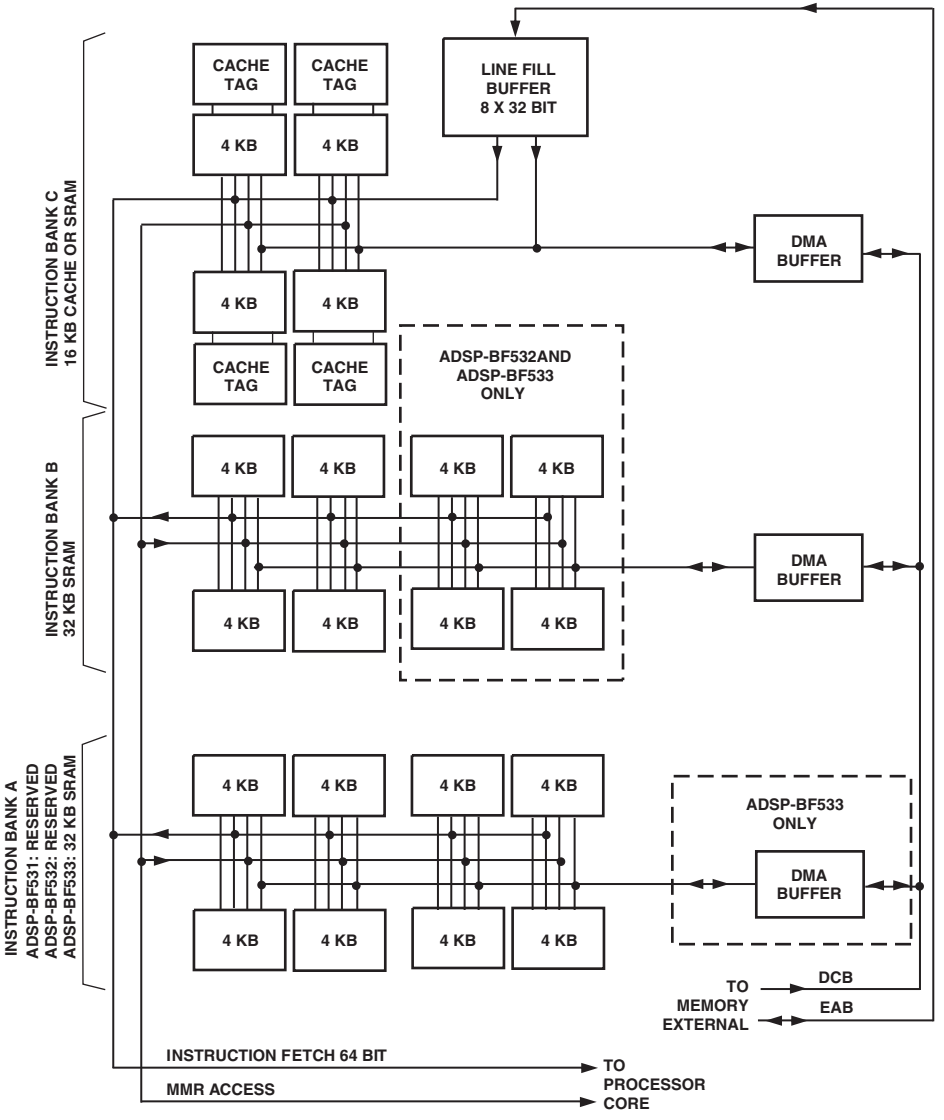


Figure 6-6. L1 Instruction Memory Bank Architecture

Cache Lines

As shown in Figure 6-7, the cache consists of a collection of cache lines. Each cache line is made up of a *tag* component and a *data* component.

- The tag component incorporates a 20-bit address tag, least recently used (LRU) bits, a Valid bit, and a Line Lock bit.
- The data component is made up of four 64-bit words of instruction data.

The tag and data components of cache lines are stored in the tag and data memory arrays, respectively.

The address tag consists of the upper 18 bits plus bits 11 and 10 of the physical address. Bits 12 and 13 of the physical address are not part of the address tag. Instead, these bits are used to identify the 4K byte memory subbank targeted for the access.

The LRU bits are part of an LRU algorithm used to determine which cache line should be replaced if a cache miss occurs.

The Valid bit indicates the state of a cache line. A cache line is always valid or invalid.

- Invalid cache lines have their Valid bit cleared, indicating the line will be ignored during an address-tag compare operation.
- Valid cache lines have their Valid bit set, indicating the line contains valid instruction/data that is consistent with the source memory.

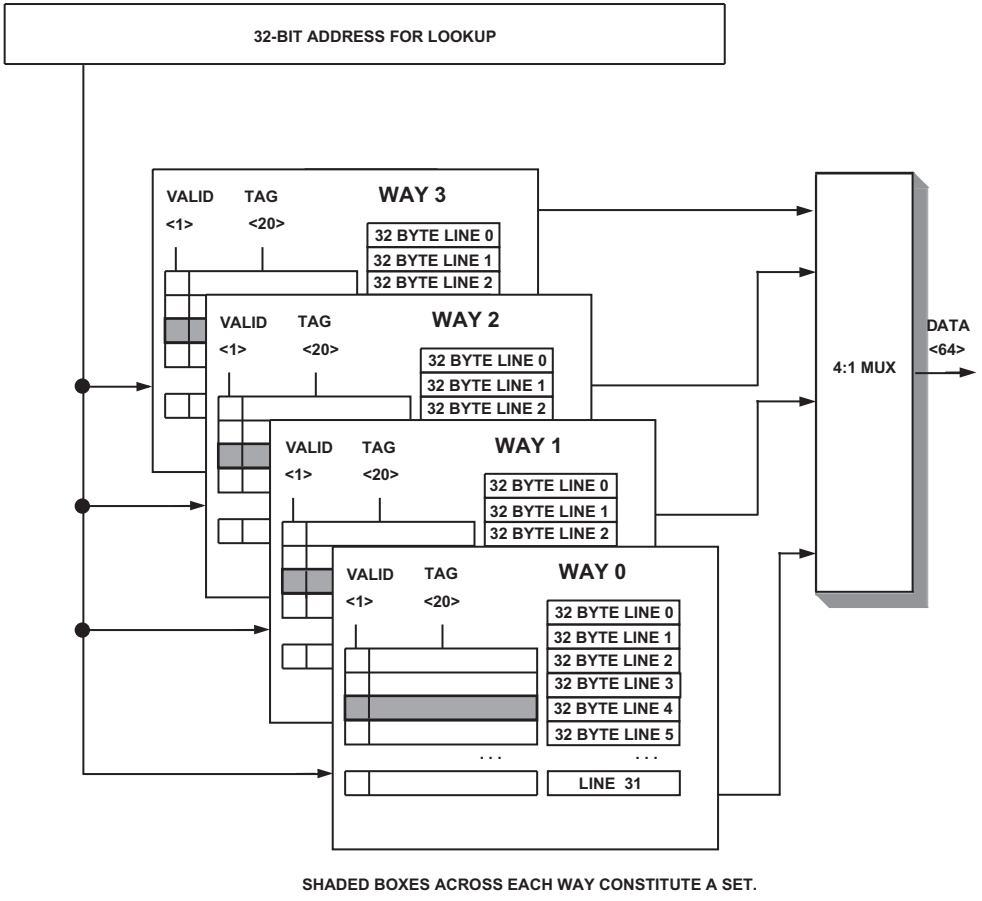


Figure 6-7. Blackfin Processor Instruction Cache Organization

The tag and data components of a cache line are illustrated in Figure 6-8.

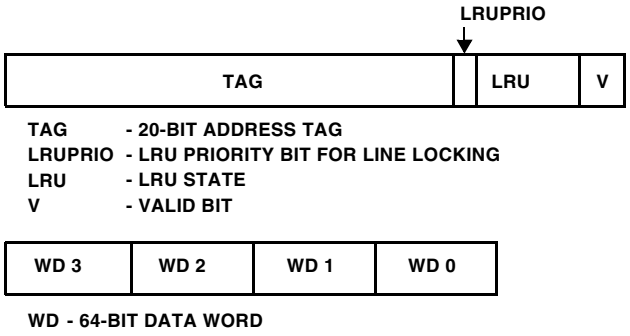


Figure 6-8. Cache Line – Tag and Data Portions

Cache Hits and Misses

A cache hit occurs when the address for an instruction fetch request from the core matches a valid entry in the cache. Specifically, a cache hit is determined by comparing the upper 18 bits and bits 11 and 10 of the instruction fetch address to the address tags of valid lines currently stored in a cache set. The cache set is selected, using bits 9 through 5 of the instruction fetch address. If the address-tag compare operation results in a match, a cache hit occurs. If the address-tag compare operation does not result in a match, a cache miss occurs.

When a cache miss occurs, the instruction memory unit generates a cache line fill access to retrieve the missing cache line from memory that is external to the core. The address for the external memory access is the address of the target instruction word. When a cache miss occurs, the core halts until the target instruction word is returned from external memory.

Cache Line Fills

A cache line fill consists of fetching 32 bytes of data from memory. The operation starts when the instruction memory unit requests a line-read data transfer (a burst of four 64-bit words of data) on its external read-data port. The address for the read transfer is the address of the target instruction word. When responding to a line-read request from the instruction memory unit, the external memory returns the target instruction word first. After it has returned the target instruction word, the next three words are fetched in sequential address order. This fetch wraps around if necessary, as shown in Table 6-3.

Table 6-3. Cache Line Word Fetching Order

Target Word	Fetching Order for Next Three Words
WD0	WD0, WD1, WD2, WD3
WD1	WD1, WD2, WD3, WD0
WD2	WD2, WD3, WD0, WD1
WD3	WD3, WD0, WD1, WD2

Line Fill Buffer

As the new cache line is retrieved from external memory, each 64-bit word is buffered in a four-entry line fill buffer before it is written to a 4K byte memory bank within L1 memory. The line fill buffer allows the core to access the data from the new cache line as the line is being retrieved from external memory, rather than having to wait until the line has been written into the cache.

Cache Line Replacement

When the instruction memory unit is configured as cache, bits 9 through 5 of the instruction fetch address are used as the index to select the cache set for the tag-address compare operation. If the tag-address compare

operation results in a cache miss, the Valid and LRU bits for the selected set are examined by a cache line replacement unit to determine the entry to use for the new cache line, that is, whether to use Way0, Way1, Way2, or Way3. See Figure 6-7, "Blackfin Processor Instruction Cache Organization," on page 6-15.

The cache line replacement unit first checks for invalid entries (that is, entries having its Valid bit cleared). If only a single invalid entry is found, that entry is selected for the new cache line. If multiple invalid entries are found, the replacement entry for the new cache line is selected based on the following priority:

- Way0 first
- Way1 next
- Way2 next
- Way3 last

For example:

- If Way3 is invalid and Ways0, 1, 2 are valid, Way3 is selected for the new cache line.
- If Ways0 and 1 are invalid and Ways2 and 3 are valid, Way0 is selected for the new cache line.
- If Ways2 and 3 are invalid and Ways0 and 1 are valid, Way2 is selected for the new cache line.

When no invalid entries are found, the cache replacement logic uses an LRU algorithm.

Instruction Cache Management

The system DMA controller and the core DAGs cannot access the instruction cache directly. By a combination of instructions and the use of core MMRs, it is possible to initialize the instruction tag and data arrays indirectly and provide a mechanism for instruction cache test, initialization, and debug.

The coherency of instruction cache must be explicitly managed. To accomplish this and ensure that the instruction cache fetches the latest version of any modified instruction space, invalidate instruction cache line entries, as required.

See "Instruction Cache Invalidation" on page 6-21.

Instruction Cache Locking by Line

The CPLB_LRUPRIO bits in the ICPLB_DATAx registers (see "Memory Protection and Properties" on page 6-46) are used to enhance control over which code remains resident in the instruction cache. When a cache line is filled, the state of this bit is stored along with the line's tag. It is then used in conjunction with the LRU (least recently used) policy to determine which Way is victimized when all cache Ways are occupied when a new cacheable line is fetched. This bit indicates that a line is of either "low" or "high" importance. In a modified LRU policy, a high can replace a low, but a low cannot replace a high. If all Ways are occupied by highs, an otherwise cacheable low will still be fetched for the core, but will not be cached. Fetched highs seek to replace unoccupied Ways first, then least recently used lows next, and finally other highs using the LRU policy. Lows can only replace unoccupied Ways or other lows, and do so using the LRU policy. If *all* previously cached highs ever become less important, they may be simultaneously transformed into lows by writing to the LRUPRIRST bit in the IMEM_CONTROL register (see page 6-8).

Instruction Cache Locking by Way

The instruction cache has four independent lock bits (`ILOC[3:0]`) that control each of the four Ways of the instruction cache. When the cache is enabled, L1 Instruction Memory has four Ways available. Setting the lock bit for a specific Way prevents that Way from participating in the LRU replacement policy. Thus, a cached instruction with its Way locked can only be removed using an `IFLUSH` instruction, or a "back door" MMR assisted manipulation of the tag array.

An example sequence is provided below to demonstrate how to lock down Way0:

- If the code of interest may already reside in the instruction cache, invalidate the entire cache first (for an example, see "Instruction Cache Management" on page 6-19).
- Disable interrupts, if required, to prevent interrupt service routines (ISRs) from potentially corrupting the locked cache.
- Set the locks for the other Ways of the cache by setting `ILOC[3:1]`. Only Way0 of the instruction cache can now be replaced by new code.
- Execute the code of interest. Any cacheable exceptions, such as exit code, traversed by this code execution are also locked into the instruction cache.
- Upon exit of the critical code, clear `ILOC[3:1]` and set `ILOC[0]`. The critical code (and the instructions which set `ILOC[0]`) is now locked into Way0.
- Re-enable interrupts, if required.

If all four Ways of the cache are locked, then further allocation into the cache is prevented.

Instruction Cache Invalidation

The instruction cache can be invalidated by address, cache line, or complete cache. The IFLUSH instruction can explicitly invalidate cache lines based on their line addresses. The target address of the instruction is generated from the P-registers. Because the instruction cache should not contain modified (dirty) data, the cache line is simply invalidated.

In the following example, the P2 register contains the address of a valid memory location. If this address has been brought into cache, the corresponding cache line is invalidated after the execution of this instruction.

Example of ICACHE instruction:

```
iflush [ p2 ] ;   /* Invalidate cache line containing address
that P2 points to */
```

Because the IFLUSH instruction is used to invalidate a specific address in the memory map, it is impractical to use this instruction to invalidate an entire Way or bank of cache. A second technique can be used to invalidate larger portions of the cache directly. This second technique directly invalidates Valid bits by setting the Invalid bit of each cache line to the invalid state. To implement this technique, additional MMRs (ITEST_COMMAND and ITEST_DATA[1:0]) are available to allow arbitrary read/write of all the cache entries directly. This method is explained in the next section.

For invalidating the complete instruction cache, a third method is available. By clearing the IMC bit in the IMEM_CONTROL register (see Figure 6-5, “L1 Instruction Memory Control Register,” on page 6-10), all Valid bits in the instruction cache are set to the invalid state. A second write to the IMEM_CONTROL register to set the IMC bit configures the instruction memory as cache again. An SSYNC instruction should be run before invalidating the cache and a CSYNC instruction should be inserted after each of these operations.

Instruction Test Registers

The Instruction Test registers allow arbitrary read/write of all L1 cache entries directly. They make it possible to initialize the instruction tag and data arrays and to provide a mechanism for instruction cache test, initialization, and debug.

When the Instruction Test Command register (`ITEST_COMMAND`) is used, the L1 cache data or tag arrays are accessed, and data is transferred through the Instruction Test Data registers (`ITEST_DATA[1:0]`). The `ITEST_DATAx` registers contain either the 64-bit data that the access is to write to or the 64-bit data that was read during the access. The lower 32 bits are stored in the `ITEST_DATA[0]` register, and the upper 32 bits are stored in the `ITEST_DATA[1]` register. When the tag arrays are accessed, `ITEST_DATA[0]` is used. Graphical representations of the `ITEST` registers begin with Figure 6-9.

The following figures describe the `ITEST` registers:

- Figure 6-9, “Instruction Test Command Register,” on page 6-23
- Figure 6-10, “Instruction Test Data 1 Register,” on page 6-24
- Figure 6-11, “Instruction Test Data 0 Register,” on page 6-25

Access to these registers is possible only in Supervisor or Emulation mode. When writing to `ITEST` registers, always write to the `ITEST_DATAx` registers first, then the `ITEST_COMMAND` register. When reading from `ITEST` registers, reverse the sequence—read the `ITEST_COMMAND` register first, then the `ITEST_DATAx` registers.

ITEST_COMMAND Register

When the Instruction Test Command register (ITEST_COMMAND) is written to, the L1 cache data or tag arrays are accessed, and the data is transferred through the Instruction Test Data registers (ITEST_DATA[1:0]).

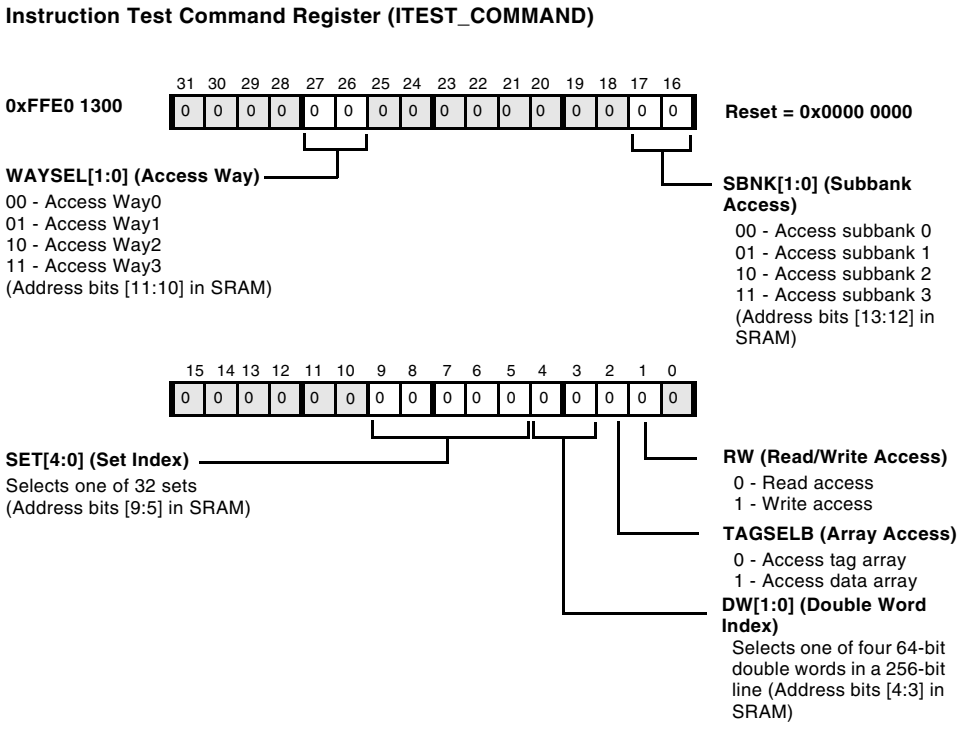


Figure 6-9. Instruction Test Command Register

ITEST_DATA1 Register

Instruction Test Data registers (ITEST_DATA[1:0]) are used to access L1 cache data arrays. They contain either the 64-bit data that the access is to write to or the 64-bit data that the access is to read from. The Instruction Test Data 1 register (ITEST_DATA1) stores the upper 32 bits.

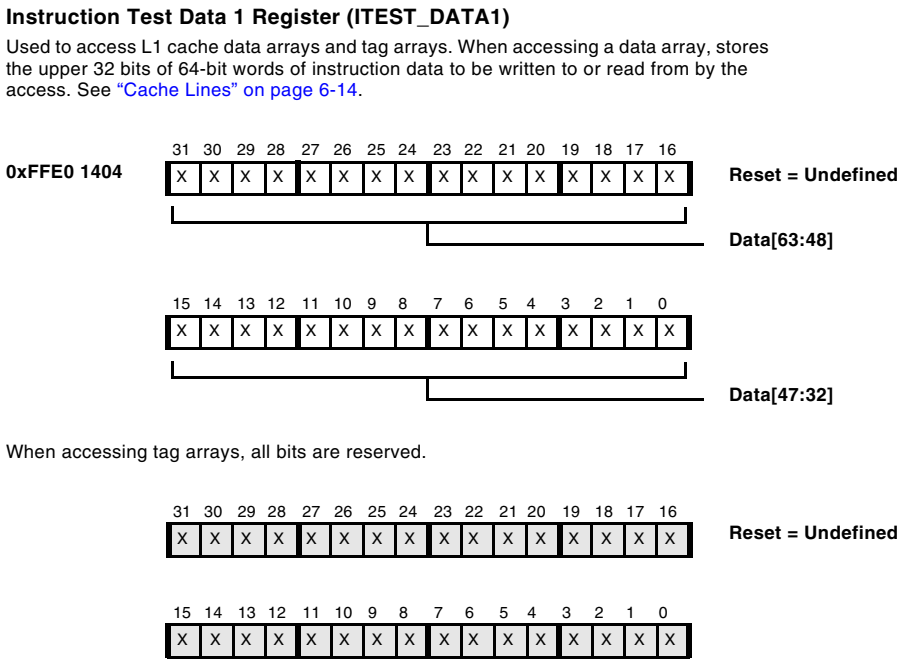


Figure 6-10. Instruction Test Data 1 Register

ITEST_DATA0 Register

The Instruction Test Data 0 register (ITEST_DATA0) stores the lower 32 bits of the 64-bit data to be written to or read from by the access. The ITEST_DATA0 register is also used to access tag arrays. This register also contains the Valid and Dirty bits, which indicate the state of the cache line.

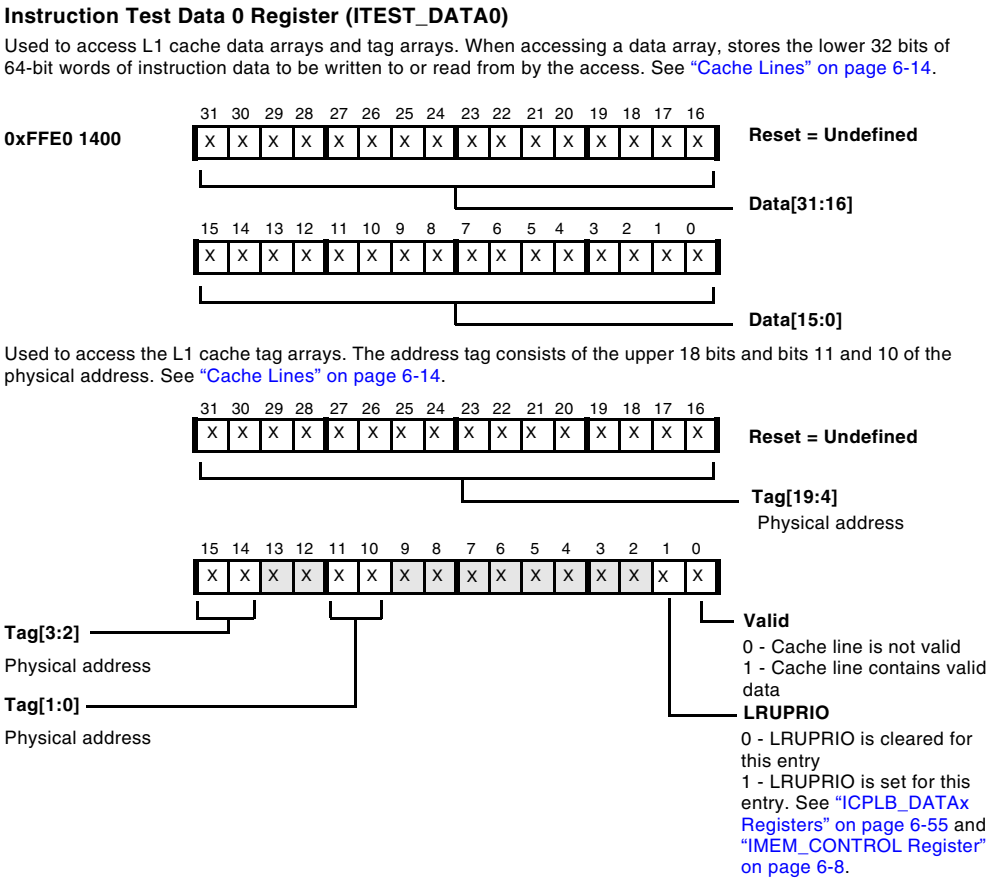


Figure 6-11. Instruction Test Data 0 Register

L1 Data Memory

The L1 data SRAM/cache is constructed from single-ported subsections, but organized to reduce the likelihood of access collisions. This organization results in apparent multi-ported behavior. When there are no collisions, this L1 data traffic could occur in a single core clock cycle:

- Two 32-bit DAG loads
- One pipelined 32-bit DAG store
- One 64-bit DMA IO
- One 64-bit cache fill/victim access

L1 Data Memory can be used only to store data.

DMEM_CONTROL Register

The Data Memory Control register (`DMEM_CONTROL`) contains control bits for the L1 Data Memory.

Data Memory Control Register (DMEM_CONTROL)

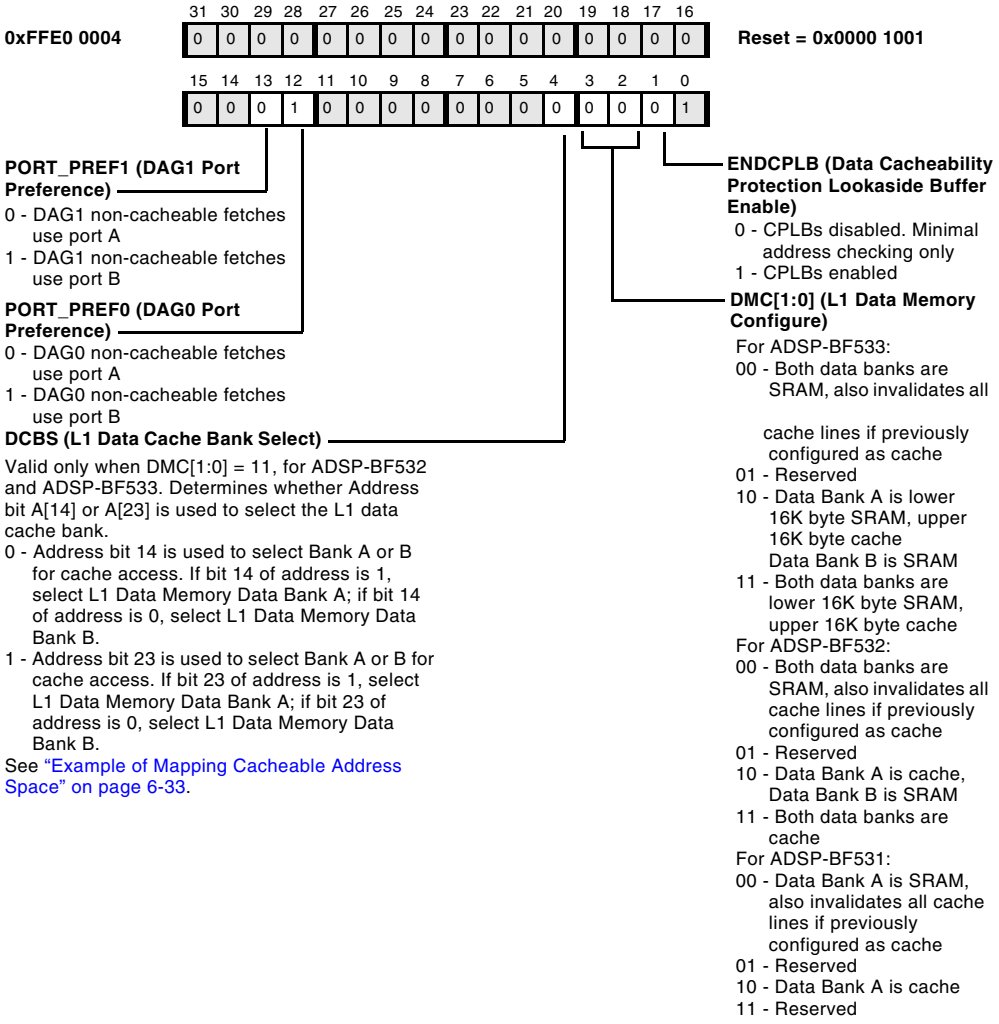


Figure 6-12. L1 Data Memory Control Register

The PORT_PREF1 bit selects the data port used to process DAG1 non-cacheable L2 fetches. Cacheable fetches are always processed by the data port physically associated with the targeted cache memory. Steering DAG0, DAG1, and cache traffic to different ports optimizes performance by keeping the queue to L2 memory full.

The PORT_PREF0 bit selects the data port used to process DAG0 non-cacheable L2 fetches. Cacheable fetches are always processed by the data port physically associated with the targeted cache memory. Steering DAG0, DAG1, and cache traffic to different ports optimizes performance by keeping the queue to L2 memory full.

For optimal performance with dual DAG reads, DAG0 and DAG1 should be configured for different ports. For example, if PORT_PREF0 is configured as 1, then PORT_PREF1 should be programmed to 0.

The DCBS bit provides some control over which addresses alias into the same set. This bit can be used to affect which addresses tend to remain resident in cache by avoiding victimization of repetitively used sets. It has no affect unless both Data Bank A and Data Bank B are serving as cache (bits DMC[1:0] in this register are set to 11).

The ENDCPLB bit is used to enable/disable the 16 Cacheability Protection Lookaside Buffers (CPLBs) used for data (see "L1 Data Cache" on page 6-31). Data CPLBs are disabled by default after reset. When disabled, only minimal address checking is performed by the L1 memory interface. This minimal checking generates an exception when the processor:

- Addresses nonexistent (reserved) L1 memory space
- Attempts to perform a nonaligned memory access
- Attempts to access MMR space either using DAG1 or when in User mode

CPLBs must be disabled using this bit prior to updating their descriptors (registers DCPLB_DATAx and DCPLB_ADDRx). Note that since load store ordering is weak (see "Ordering of Loads and Stores" on page 6-67), disabling CPLBs should be preceded by a CSYNC instruction.

When enabling or disabling cache or CPLBs, immediately follow the write to DMEM_CONTROL with a SSYNC to ensure proper behavior.

By default after reset, all L1 Data Memory serves as SRAM. The DMC[1:0] bits can be used to reserve portions of this memory to serve as cache instead. Reserving memory to serve as cache does not enable L2 memory accesses to be cached. To do this, CPLBs must also be enabled (using the ENDCPLB bit) and CPLB descriptors (registers DCPLB_DATAx and DCPLB_ADDRx) must specify chosen memory pages as cache-enabled.

By default after reset, cache and CPLB address checking is disabled.

To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.

L1 Data SRAM

Accesses to SRAM do not collide unless all of the following are true: the accesses are to the same 32-bit word polarity (address bits 2 match), the same 4K byte subbank (address bits 13 and 12 match), the same 16K byte half bank (address bits 16 match), and the same bank (address bits 21 and 20 match). When an address collision is detected, access is nominally granted first to the DAGs, then to the store buffer, and finally to the DMA and cache fill/victim traffic. To ensure adequate DMA bandwidth, DMA is given highest priority if it has been blocked for more than 16 sequential core clock cycles, or if a second DMA I/O is queued before the first DMA I/O is processed.

Table 6-4 shows how the subbank organization is mapped into memory.

Table 6-4. L1 Data Memory SRAM Subbank Start Addresses

Memory Bank and Subbank	ADSP-BF533	ADSP-BF532	ADSP-BF531
Data Bank A, Subbank 0	0xFF80 0000	-	-
Data Bank A, Subbank 1	0xFF80 1000	-	-
Data Bank A, Subbank 2	0xFF80 2000	-	-
Data Bank A, Subbank 3	0xFF80 3000	-	-
Data Bank A, Subbank 4	0xFF80 4000	0xFF80 4000	0xFF80 4000
Data Bank A, Subbank 5	0xFF80 5000	0xFF80 5000	0xFF80 5000
Data Bank A, Subbank 6	0xFF80 6000	0xFF80 6000	0xFF80 6000
Data Bank A, Subbank 7	0xFF80 7000	0xFF80 7000	0xFF80 7000
Data Bank B, Subbank 0	0xFF90 0000	-	-
Data Bank B, Subbank 1	0xFF90 1000	-	-
Data Bank B, Subbank 2	0xFF90 2000	-	-
Data Bank B, Subbank 3	0xFF90 3000	-	-
Data Bank B, Subbank 4	0xFF90 4000	0xFF90 4000	-
Data Bank B, Subbank 5	0xFF90 5000	0xFF90 5000	-

Table 6-4. L1 Data Memory SRAM Subbank Start Addresses (Cont'd)

Memory Bank and Subbank	ADSP-BF533	ADSP-BF532	ADSP-BF531
Data Bank B, Subbank 6	0xFF90 6000	0xFF90 6000	-
Data Bank B, Subbank 7	0xFF90 7000	0xFF90 7000	-

Figure 6-13 shows the L1 Data Memory architecture.

L1 Data Cache

For definitions of cache terminology, see "Terminology" on page 6-74.

When data cache is enabled (controlled by bits `DMC[1:0]` in the `DMEM_CONTROL` register), either 16K byte of Data Bank A or 16K byte of both Data Bank A and Data Bank B can be set to serve as cache. For the ADSP-BF533, the upper 16K byte is used. For the ADSP-BF531, only Data Bank A is available. Unlike instruction cache, which is 4-Way set associative, data cache is 2-Way set associative. When two banks are available and enabled as cache, additional sets rather than Ways are created. When both Data Bank A and Data Bank B have memory serving as cache, the `DCBS` bit in the `DMEM_CONTROL` register may be used to control which half of all address space is handled by which bank of cache memory. The `DCBS` bit selects either address bit 14 or 23 to steer traffic between the cache banks. This provides some control over which addresses alias into the same set. It may therefore be used to affect which addresses tend to remain resident in cache by avoiding victimization of repetitively used sets.

Accesses to cache do not collide unless they are to the same 4K byte subbank, the same half bank, and to the same bank. Cache has less apparent multi-ported behavior than SRAM due to the overhead in maintaining

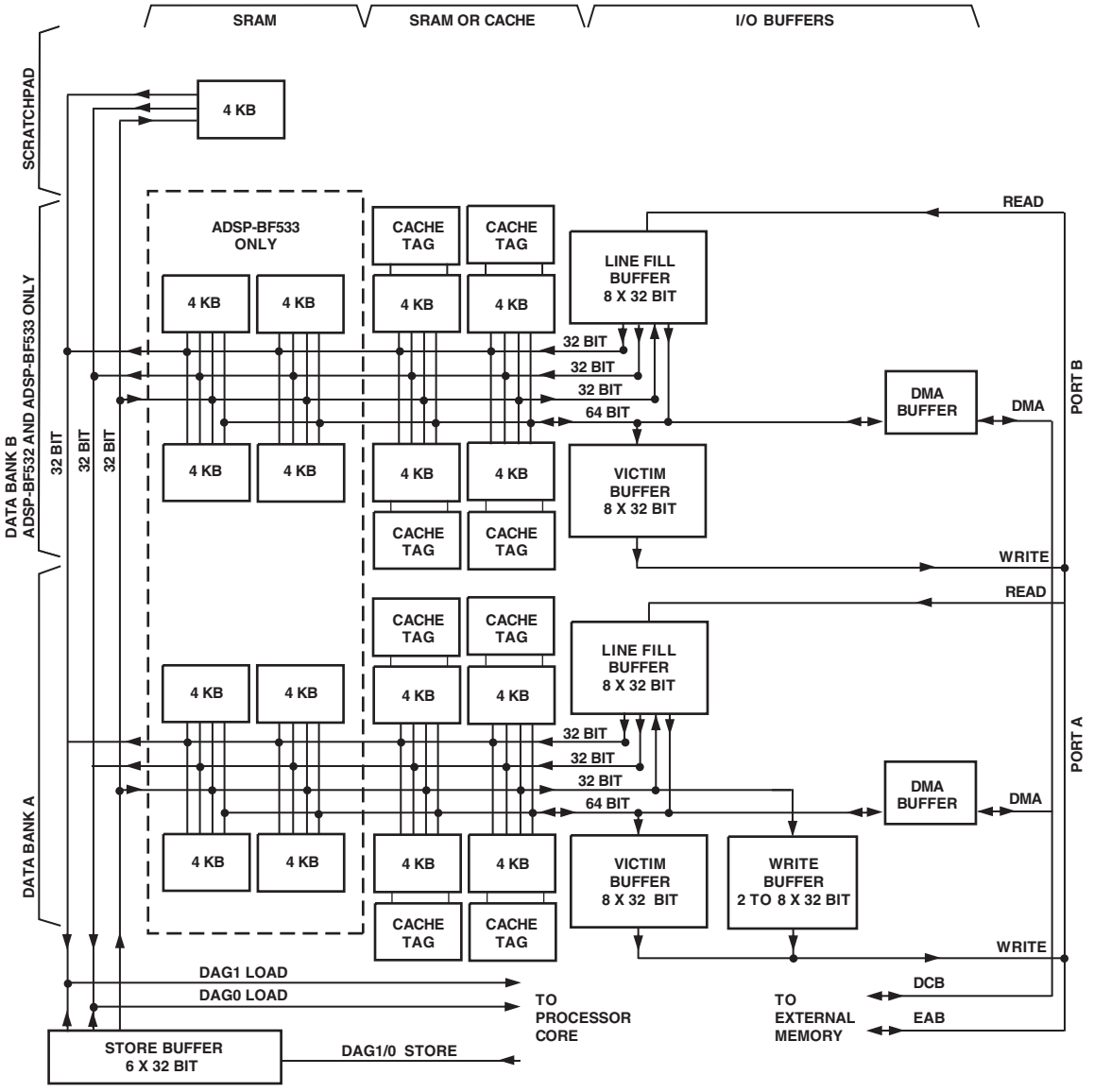


Figure 6-13. L1 Data Memory Architecture

tags. When cache addresses collide, access is granted first to the DTEST register accesses, then to the store buffer, and finally to cache fill/victim traffic.

Three different cache modes are available.

- Write-through with cache line allocation only on reads
- Write-through with cache line allocation on both reads and writes
- Write-back which allocates cache lines on both reads and writes

Cache mode is selected by the DCPLB descriptors (see "Memory Protection and Properties" on page 6-46). Any combination of these cache modes can be used simultaneously since cache mode is selectable for each memory page independently.

If cache is enabled (controlled by bits DMC[1:0] in the DMEM_CONTROL register), data CPLBs should also be enabled (controlled by ENDCPLB bit in the DMEM_CONTROL register). Only memory pages specified as cacheable by data CPLBs will be cached. The default behavior when data CPLBs are disabled is for nothing to be cached.

Erroneous behavior can result when MMR space is configured as cacheable by data CPLBs, or when data banks serving as L1 SRAM are configured as cacheable by data CPLBs.

Example of Mapping Cacheable Address Space

An example of how the cacheable address space maps into two data banks follows.

When both banks are configured as cache on the ADSP-BF533 or ADSP-BF532, they operate as two independent, 16K byte, 2-Way set associative caches that can be independently mapped into the Blackfin processor address space.

If both data banks are configured as cache, the DCBS bit in the DMEM_CONTROL register designates Address bit A[14] or A[23] as the cache selector. Address bit A[14] or A[23] selects the cache implemented by Data Bank A or the cache implemented by Data Bank B.

- If DCBS = 0, then A[14] is part of the address index, and all addresses in which A[14] = 0 use Data Bank B. All addresses in which A[14] = 1 use Data Bank A.

  In this case, A[23] is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.

- If DCBS = 1, then A[23] is part of the address index, and all addresses where A[23] = 0 use Data Bank B. All addresses where A[23] = 1 use Data Bank A.

  In this case, A[14] is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.

The result of choosing DCBS = 0 or DCBS = 1 is:

- If DCBS = 0, A[14] selects Data Bank A instead of Data Bank B.

  Alternating 16K byte pages of memory map into each of the two 16K byte caches implemented by the two data banks. Consequently:

  Any data in the first 16K byte of memory could be stored only in Data Bank B.

  Any data in the next address range (16K byte through 32K byte) – 1 could be stored only in Data Bank A.

Any data in the next range (32K byte through 48K byte) – 1 would be stored in Data Bank B.

Alternate mapping would continue.

As a result, the cache operates as if it were a single, contiguous, 2-Way set associative 32K byte cache. Each Way is 16K byte long, and all data elements with the same first 14 bits of address index to a unique set in which up to two elements can be stored (one in each Way).

- If `DCBS = 1`, `A[23]` selects Data Bank A instead of Data Bank B.

  With `DCBS = 1`, the system functions more like two independent caches, each a 2-Way set associative 16K byte cache. Each Bank serves an alternating set of 8M byte blocks of memory.

  For example, Data Bank B caches all data accesses for the first 8M byte of memory address range. That is, every 8M byte of range vies for the two line entries (rather than every 16K byte repeat). Likewise, Data Bank A caches data located above 8M byte and below 16M byte.

  For example, if the application is working from a data set that is 1M byte long and located entirely in the first 8M byte of memory, it is effectively served by only half the cache, that is, by Data Bank B (a 2-Way set associative 16K byte cache). In this instance, the application never derives any benefit from Data Bank A.

For most applications, it is best to operate with `DCBS = 0`.

However, if the application is working from two data sets, located in two memory spaces at least 8M byte apart, closer control over how the cache maps to the data is possible. For example, if the program is doing a series

of dual MAC operations in which both DAGs are accessing data on every cycle, by placing DAG0's data set in one block of memory and DAG1's data set in the other, the system can ensure that:

- DAG0 gets its data from Data Bank A for all of its accesses and
- DAG1 gets its data from Data Bank B.

This arrangement causes the core to use both data buses for cache line transfer and achieves the maximum data bandwidth between the cache and the core.

Figure 6-14 shows an example of how mapping is performed when `DCBS = 1`.

The `DCBS` selection can be changed dynamically; however, to ensure that no data is lost, first flush and invalidate the entire cache.

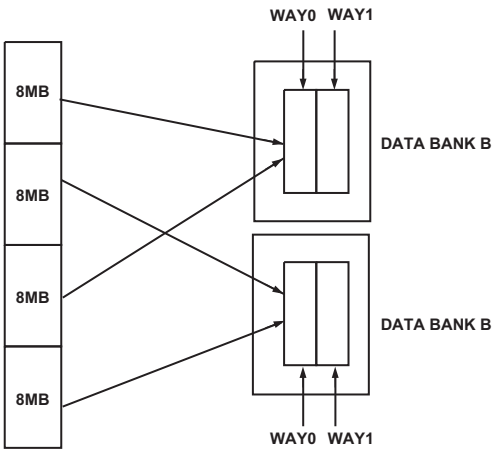


Figure 6-14. Data Cache Mapping When DCBS = 1

Data Cache Access

The Cache Controller tests the address from the DAGs against the tag bits. If the logical address is present in L1 cache, a cache hit occurs, and the data is accessed in L1. If the logical address is not present, a cache miss occurs, and the memory transaction is passed to the next level of memory via the system interface. The line index and replacement policy for the Cache Controller determines the cache tag and data space that are allocated for the data coming back from external memory.

A data cache line is in one of three states: invalid, exclusive (valid and clean), and modified (valid and dirty). If valid data already occupies the allocated line and the cache is configured for write-back storage, the controller checks the state of the cache line and treats it accordingly:

- If the state of the line is exclusive (clean), the new tag and data write over the old line.
- If the state of the line is modified (dirty), then the cache contains the only valid copy of the data. If the line is dirty, the current contents of the cache are copied back to external memory before the new data is written to the cache.

The processor provides victim buffers and line fill buffers. These buffers are used if a cache load miss generates a victim cache line that should be replaced. The line fill operation goes to external memory. The data cache performs the line fill request to the system as critical (or requested) word first, and forwards that data to the waiting DAG as it updates the cache line. In other words, the cache performs critical word forwarding.

The data cache supports hit-under-a-store miss, and hit-under-a-prefetch miss. In other words, on a write-miss or execution of a `PREFETCH` instruction that misses the cache (and is to a cacheable region), the instruction pipeline incurs a minimum of a 4-cycle stall. Furthermore, a subsequent load or store instruction can hit in the L1 cache while the line fill completes.

Interrupts of sufficient priority (relative to the current context) cancel a stalled load instruction. Consequently, if the load operation misses the L1 Data Memory cache and generates a high latency line fill operation on the system interface, it is possible to interrupt the core, causing it to begin processing a different context. The system access to fill the cache line is not cancelled, and the data cache is updated with the new data before any further cache miss operations to the respective data bank are serviced. For more information see "Exceptions" on page 4-41.

Cache Write Method

Cache write memory operations can be implemented by using either a write-through method or a write-back method:

- For each store operation, write-through caches initiate a write to external memory immediately upon the write to cache.

  If the cache line is replaced or explicitly flushed by software, the contents of the cache line are invalidated rather than written back to external memory.

- A write-back cache does not write to external memory until the line is replaced by a load operation that needs the line.

The L1 Data Memory employs a full cache line width copyback buffer on each data bank. In addition, a two-entry write buffer in the L1 Data Memory accepts all stores with cache inhibited or store-through protection. An `SSYNC` instruction flushes the write buffer.

IPRIO Register and Write Buffer Depth

The Interrupt Priority register (`IPRIO`) can be used to control the size of the write buffer on Port A (see "L1 Data Memory Architecture" on page 6-32).

The IPRIO[3:0] bits can be programmed to reflect the low priority interrupt watermark. When an interrupt occurs, causing the processor to vector from a low priority interrupt service routine to a high priority interrupt service routine, the size of the write buffer increases from two to eight 32-bit words deep. This allows the interrupt service routine to run and post writes without an initial stall, in the case where the write buffer was already filled in the low priority interrupt routine. This is most useful when posted writes are to a slow external memory device. When returning from a high priority interrupt service routine to a low priority interrupt service routine or user mode, the core stalls until the write buffer has completed the necessary writes to return to a two-deep state. By default, the write buffer is a fixed two-deep FIFO.

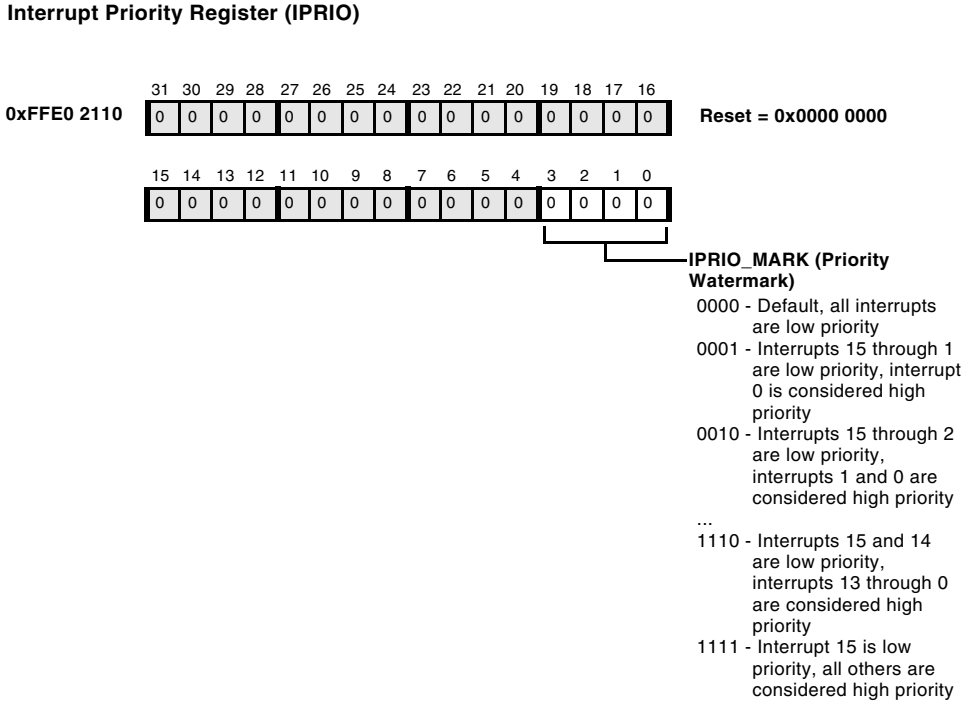


Figure 6-15. Interrupt Priority Register

Data Cache Control Instructions

The processor defines three data cache control instructions that are accessible in User and Supervisor modes. The instructions are PREFETCH, FLUSH, and FLUSHINV.

- PREFETCH (Data Cache Prefetch) attempts to allocate a line into the L1 cache. If the prefetch hits in the cache, generates an exception, or addresses a cache inhibited region, PREFETCH functions like a NOP.
- FLUSH (Data Cache Flush) causes the data cache to synchronize the specified cache line with external memory. If the cached data line is dirty, the instruction writes the line out and marks the line clean in the data cache. If the specified data cache line is already clean or does not exist, FLUSH functions like a NOP.
- FLUSHINV (Data Cache Line Flush and Invalidate) causes the data cache to perform the same function as the FLUSH instruction and then invalidate the specified line in the cache. If the line is in the cache and dirty, the cache line is written out to external memory. The Valid bit in the cache line is then cleared. If the line is not in the cache, FLUSHINV functions like a NOP.

If software requires synchronization with system hardware, place an SSYNC instruction after the FLUSH instruction to ensure that the flush operation has completed. If ordering is desired to ensure that previous stores have been pushed through all the queues, place an SSYNC instruction before the FLUSH.

Data Cache Invalidation

Besides the FLUSHINV instruction, explained in the previous section, two additional methods are available to invalidate the data cache when flushing is not required. The first technique directly invalidates Valid bits by setting the Invalid bit of each cache line to the invalid state. To implement

this technique, additional MMRs (DTEST_COMMAND and DTEST_DATA[1:0]) are available to allow arbitrary reads/writes of all the cache entries directly. This method is explained in the next section.

For invalidating the complete data cache, a second method is available. By clearing the DMC[1:0] bits in the DMEM_CONTROL register (see Figure 6-12, "L1 Data Memory Control Register," on page 6-27), all Valid bits in the data cache are set to the invalid state. A second write to the DMEM_CONTROL register to set the DMC[1:0] bits to their previous state then configures the data memory back to its previous cache/SRAM configuration. An SSYNC instruction should be run before invalidating the cache and a CSYNC instruction should be inserted after each of these operations.

Data Test Registers

Like L1 Instruction Memory, L1 Data Memory contains additional MMRs to allow arbitrary reads/writes of all cache entries directly. The registers provide a mechanism for data cache test, initialization, and debug.

When the Data Test Command register (DTEST_COMMAND) is written to, the L1 cache data or tag arrays are accessed and data is transferred through the Data Test Data registers (DTEST_DATA[1:0]). The DTEST_DATA[1:0] registers contain the 64-bit data to be written, or they contain the destination for the 64-bit data read. The lower 32 bits are stored in the DTEST_DATA[0] register and the upper 32 bits are stored in the DTEST_DATA[1] register. When the tag arrays are being accessed, then the DTEST_DATA[0] register is used.

A CSYNC instruction is required after writing the DTEST_COMMAND MMR.

These figures describe the DTEST registers.

- Figure 6-16, “Data Test Command Register,” on page 6-43
- Figure 6-17, “Data Test Data 1 Register,” on page 6-44
- Figure 6-18, “Data Test Data 0 Register,” on page 6-45

Access to these registers is possible only in Supervisor or Emulation mode. When writing to DTEST registers, always write to the DTEST_DATA registers first, then the DTEST_COMMAND register.

DTEST_COMMAND Register

When the Data Test Command register (DTEST_COMMAND) is written to, the L1 cache data or tag arrays are accessed, and the data is transferred through the Data Test Data registers (DTEST DATA[1:0]).

The Data/Instruction Access bit allows direct access via the DTEST_COMMAND MMR to L1 instruction SRAM.

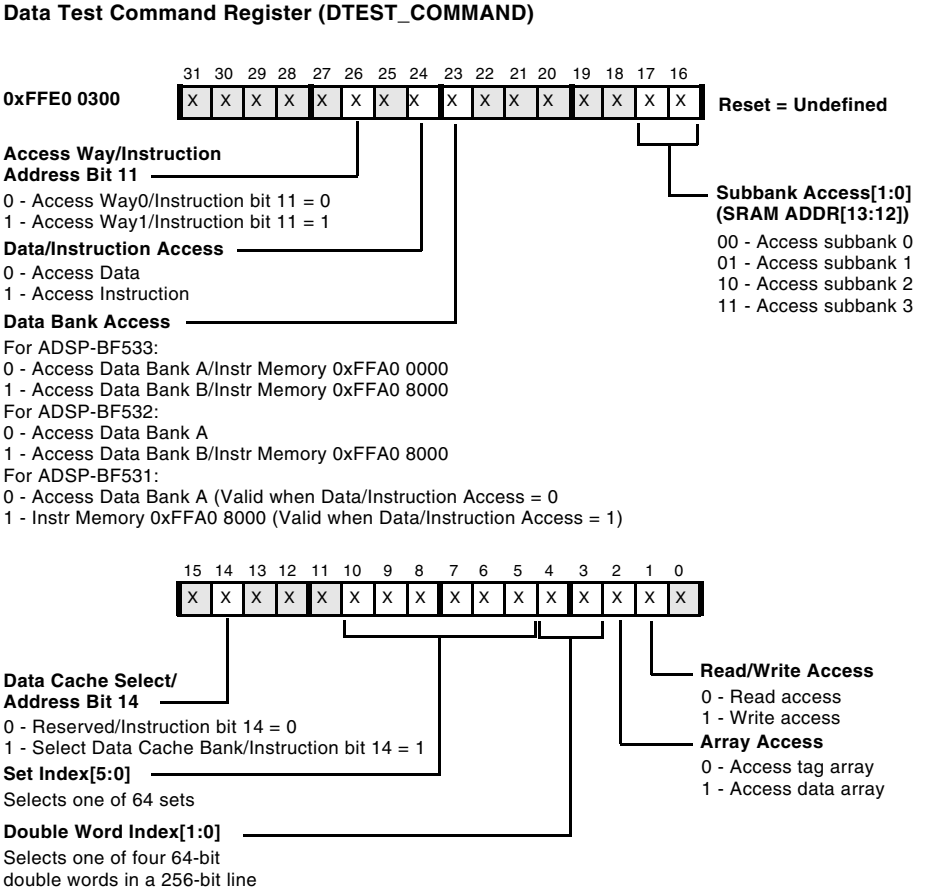


Figure 6-16. Data Test Command Register

DTEST_DATA1 Register

Data Test Data registers (DTEST_DATA[1:0]) contain the 64-bit data to be written, or they contain the destination for the 64-bit data read. The Data Test Data 1 register (DTEST_DATA1) stores the upper 32 bits.

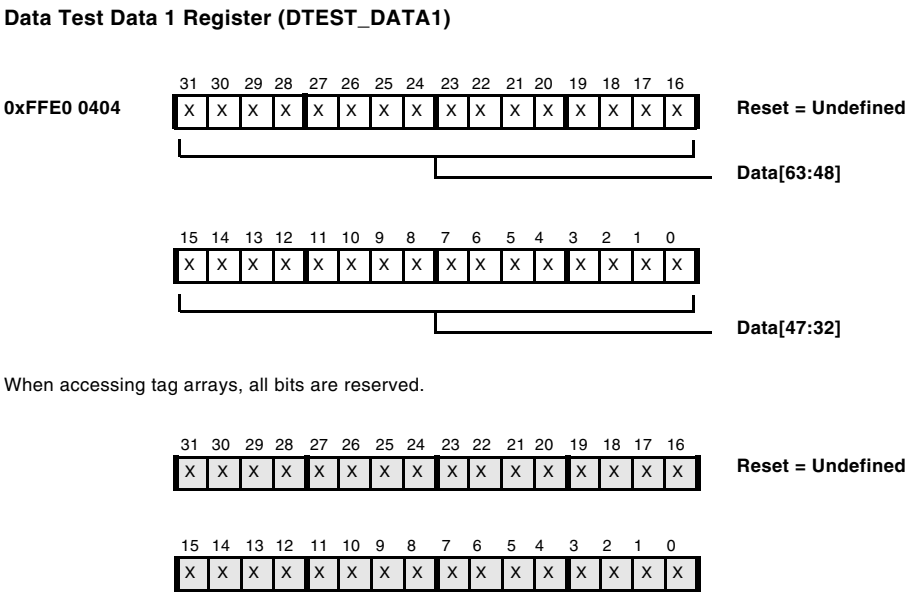


Figure 6-17. Data Test Data 1 Register

DTEST_DATA0 Register

The Data Test Data 0 register (DTEST_DATA0) stores the lower 32 bits of the 64-bit data to be written, or it contains the lower 32 bits of the destination for the 64-bit data read. The DTEST_DATA0 register is also used to access the tag arrays and contains the Valid and Dirty bits, which indicate the state of the cache line.

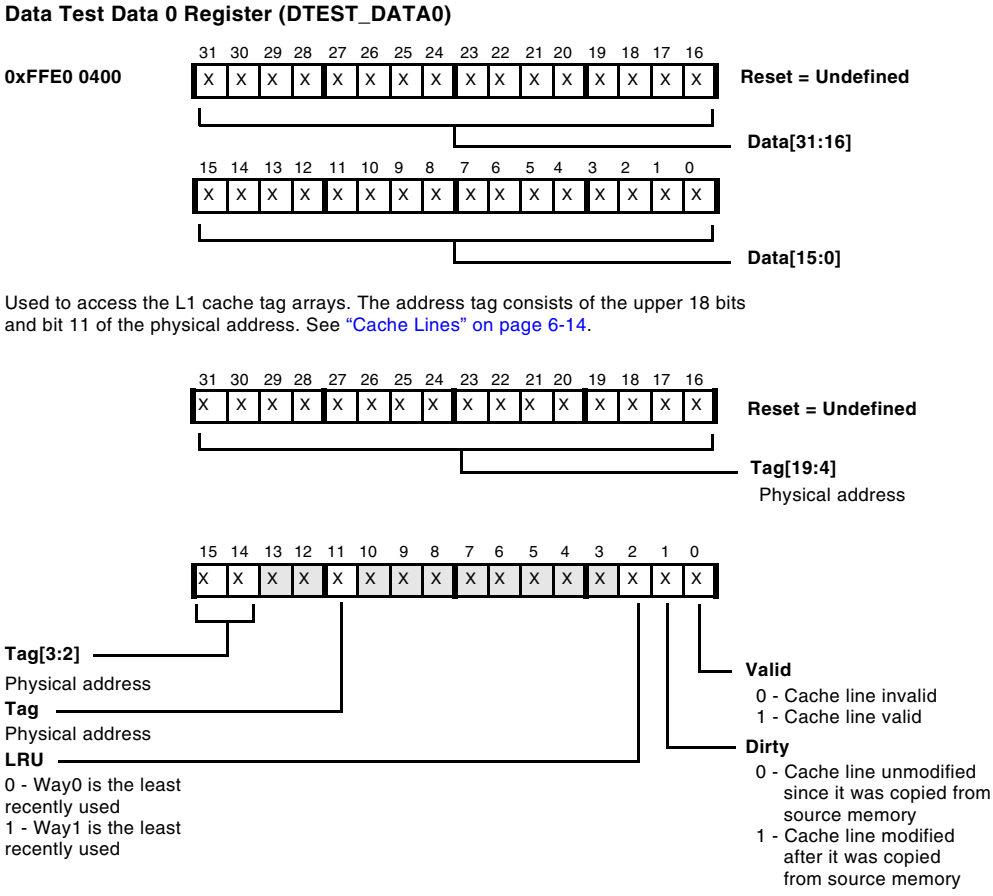


Figure 6-18. Data Test Data 0 Register

External Memory

The external memory space is shown in Figure 6-1. One of the memory regions is dedicated to SDRAM support. The size of the SDRAM bank is programmable and can range in size from 16M byte to 128M byte. The start address of the bank is 0x0000 0000.

Each of the next four banks contains 1M byte and is dedicated to support asynchronous memories. The start address of the asynchronous memory bank is 0x2000 0000.

Memory Protection and Properties

This section describes the Memory Management Unit (MMU), memory pages, CPLB management, MMU management, and CPLB registers.

Memory Management Unit

The Blackfin processor contains a page based Memory Management Unit (MMU). This mechanism provides control over cacheability of memory ranges, as well as management of protection attributes at a page level. The MMU provides great flexibility in allocating memory and I/O resources between tasks, with complete control over access rights and cache behavior.

The MMU is implemented as two 16-entry Content Addressable Memory (CAM) blocks. Each entry is referred to as a Cacheability Protection Lookaside Buffer (CPLB) descriptor. When enabled, every valid entry in the MMU is examined on any fetch, load, or store operation to determine whether there is a match between the address being requested and the page described by the CPLB entry. If a match occurs, the cacheability and protection attributes contained in the descriptor are used for the memory transaction with no additional cycles added to the execution of the instruction.

Because the L1 memories are separated into instruction and data memories, the CPLB entries are also divided between instruction and data CPLBs. Sixteen CPLB entries are used for instruction fetch requests; these are called *ICPLBs*. Another sixteen CPLB entries are used for data transactions; these are called *DCPLBs*. The ICPLBs and DCPLBs are enabled by setting the appropriate bits in the L1 Instruction Memory Control (`IMEM_CONTROL`) and L1 Data Memory Control (`DMEM_CONTROL`) registers, respectively. These registers are shown in Figure 6-5 and Figure 6-12, respectively.

Each CPLB entry consists of a pair of 32-bit values. For instruction fetches:

- `ICPLB_ADDR[n]` defines the start address of the page described by the CPLB descriptor.
- `ICPLB_DATA[n]` defines the properties of the page described by the CPLB descriptor.

For data operations:

- `DCPLB_ADDR[m]` defines the start address of the page described by the CPLB descriptor.
- `DCPLB_DATA[m]` defines the properties of the page described by the CPLB descriptor.

There are two default CPLB descriptors for data accesses to the scratchpad data memory and to the system and core MMR space. These default descriptors define the above space as non-cacheable, so that additional CPLBs do not need to be set up for these regions of memory.

If valid CPLBs are set up for this space, the default CPLBs are ignored.

Memory Pages

The 4G byte address space of the processor can be divided into smaller ranges of memory or I/O referred to as memory pages. Every address within a page shares the attributes defined for that page. The architecture supports four different page sizes:

- 1K byte
- 4K byte
- 1M byte
- 4M byte

Different page sizes provide a flexible mechanism for matching the mapping of attributes to different kinds of memory and I/O.

Memory Page Attributes

Each page is defined by a two-word descriptor, consisting of an address descriptor word `xCPLB_ADDR[n]` and a properties descriptor word `xCPLB_DATA[n]`. The address descriptor word provides the base address of the page in memory. Pages must be aligned on page boundaries that are an integer multiple of their size. For example, a 4M byte page must start on an address divisible by 4M byte; whereas a 1K byte page can start on any 1K byte boundary. The second word in the descriptor specifies the other properties or attributes of the page. These properties include:

- Page size. 1K byte, 4K byte, 1M byte, 4M byte
- Cacheable/non-cacheable: Accesses to this page use the L1 cache or bypass the cache.

- If cacheable: write-through/write-back. Data writes propagate directly to memory or are deferred until the cache line is reallocated. If write-through, allocate on read only, or read and write.
- Dirty/modified. The data in this page in memory has changed since the CPLB was last loaded.
- Supervisor write access permission. Enables or disables writes to this page when in Supervisor mode, for data pages only.
- User write access permission. Enables or disables writes to this page when in User mode, for data pages only.
- User read access permission. Enables or disables reads from this page when in User mode.
- Valid. Check this bit to determine whether this is valid CPLB data.
- Lock. Keep this entry in MMR; do not participate in CPLB replacement policy.

Page Descriptor Table

For memory accesses to utilize the cache when CPLBs are enabled for instruction access, data access, or both, a valid CPLB entry must be available in an MMR pair. The MMR storage locations for CPLB entries are limited to 16 descriptors for instruction fetches and 16 descriptors for data load and store operations.

For small and/or simple memory models, it may be possible to define a set of CPLB descriptors that fit into these 32 entries, cover the entire addressable space, and never need to be replaced. This type of definition is referred to as a *static* memory management model.

However, operating environments commonly define more CPLB descriptors to cover the addressable memory and I/O spaces than will fit into the available on-chip CPLB MMRs. When this happens, a memory-based data structure, called a Page Descriptor Table, is used; in it can be stored all the potentially required CPLB descriptors. The specific format for the Page Descriptor Table is not defined as part of the Blackfin processor architecture. Different operating systems, which have different memory management models, can implement Page Descriptor Table structures that are consistent with the OS requirements. This allows adjustments to be made between the level of protection afforded versus the performance attributes of the memory-management support routines.

CPLB Management

When the Blackfin processor issues a memory operation for which no valid CPLB (cacheability protection lookaside buffer) descriptor exists in an MMR pair, an exception occurs that places the processor into Supervisor mode and vectors to the MMU exception handler

(see "Exceptions" on page 4-41 for more information). The handler is typically part of the operating system (OS) kernel that implements the CPLB replacement policy.

Before CPLBs are enabled, valid CPLB descriptors must be in place for both the Page Descriptor Table and the MMU exception handler. The LOCK bits of these CPLB descriptors are commonly set so they are not inadvertently replaced in software.

The handler uses the faulting address to index into the Page Descriptor Table structure to find the correct CPLB descriptor data to load into one of the on-chip CPLB register pairs. If all on-chip registers contain valid CPLB entries, the handler selects one of the descriptors to be replaced, and the new descriptor information is loaded. Before loading new descriptor data into any CPLBs, the corresponding group of sixteen CPLBs must be disabled using:

- The Enable DCPLB (ENDCPLB) bit in the DMEM_CONTROL register for data descriptors, or
- The Enable ICPLB (ENICPLB) bit in the IMEM_CONTROL register for instruction descriptors

The CPLB replacement policy and algorithm to be used are the responsibility of the system MMU exception handler. This policy, which is dictated by the characteristics of the operating system, usually implements a modified LRU (Least Recently Used) policy, a round robin scheduling method, or pseudo random replacement.

After the new CPLB descriptor is loaded, the exception handler returns, and the faulting memory operation is restarted. this operation should now find a valid CPLB descriptor for the requested address, and it should proceed normally.

A single instruction may generate an instruction fetch as well as one or two data accesses. It is possible that more than one of these memory operations references data for which there is no valid CPLB descriptor in an MMR pair. In this case, the exceptions are prioritized and serviced in this order:

- Instruction page miss
- A page miss on DAG0
- A page miss on DAG1

MMU Application

Memory management is an optional feature in the Blackfin processor architecture. Its use is predicated on the system requirements of a given application. Upon reset, all CPLBs are disabled, and the Memory Management Unit (MMU) is not used.

If all L1 memory is configured as SRAM, then the data and instruction MMU functions are optional, depending on the application's need for protection of memory spaces either between tasks or between User and Supervisor modes. To protect memory between tasks, the operating system can maintain separate tables of instruction and/or data memory pages available for each task and make those pages visible only when the relevant task is running. When a task switch occurs, the operating system can ensure the invalidation of any CPLB descriptors on chip that should not be available to the new task. It can also preload descriptors appropriate to the new task.

For many operating systems, the application program is run in User mode while the operating system and its services run in Supervisor mode. It is desirable to protect code and data structures used by the operating system from inadvertent modification by a running User mode application. This protection can be achieved by defining CPLB descriptors for protected memory ranges that allow write access only when in Supervisor mode. If a

write to a protected memory region is attempted while in User mode, an exception is generated before the memory is modified. Optionally, the User mode application may be granted read access for data structures that are useful to the application. Even Supervisor mode functions can be blocked from writing some memory pages that contain code that is not expected to be modified. Because CPLB entries are MMRs that can be written only while in Supervisor mode, user programs cannot gain access to resources protected in this way.

If either the L1 Instruction Memory or the L1 Data Memory is configured partially or entirely as cache, the corresponding CPLBs must be enabled. When an instruction generates a memory request and the cache is enabled, the processor first checks the ICPLBs to determine whether the address requested is in a cacheable address range. If no valid ICPLB entry in an MMR pair corresponds to the requested address, an MMU exception is generated to obtain a valid ICPLB descriptor to determine whether the memory is cacheable or not. As a result, if the L1 Instruction Memory is enabled as cache, then any memory region that contains instructions must have a valid ICPLB descriptor defined for it. These descriptors must either reside in MMRs at all times or be resident in a memory-based Page Descriptor Table that is managed by the MMU exception handler. Likewise, if either or both L1 data banks are configured as cache, all potential data memory ranges must be supported by DCPLB descriptors.

Before caches are enabled, the MMU and its supporting data structures must be set up and enabled.

Examples of Protected Memory Regions

In Figure 6-19, a starting point is provided for basic CPLB allocation for Instruction and Data CPLBs. Note some ICPLBs and DCPLBs have common descriptors for the same address space.

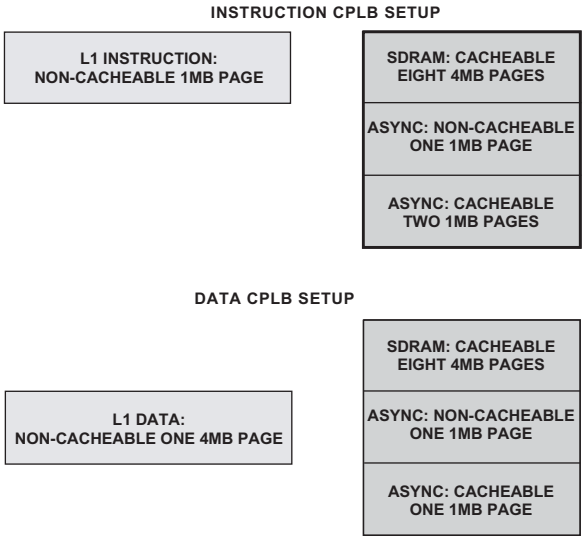


Figure 6-19. Examples of Protected Memory Regions

ICPLB_DATAx Registers

Figure 6-20 describes the ICPLB Data registers (ICPLB_DATAx).

To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.

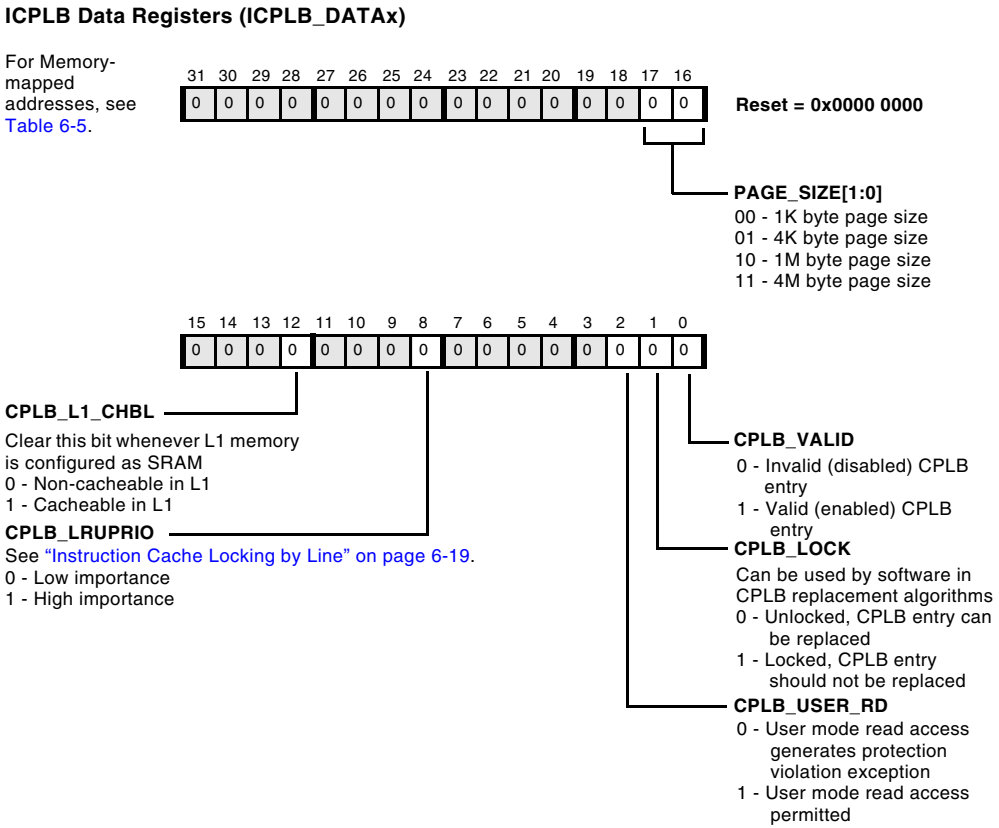


Figure 6-20. ICPLB Data Registers

Table 6-5. ICPLB Data Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
ICPLB_DATA0	0xFFE0 1200
ICPLB_DATA1	0xFFE0 1204
ICPLB_DATA2	0xFFE0 1208
ICPLB_DATA3	0xFFE0 120C
ICPLB_DATA4	0xFFE0 1210
ICPLB_DATA5	0xFFE0 1214
ICPLB_DATA6	0xFFE0 1218
ICPLB_DATA7	0xFFE0 121C
ICPLB_DATA8	0xFFE0 1220
ICPLB_DATA9	0xFFE0 1224
ICPLB_DATA10	0xFFE0 1228
ICPLB_DATA11	0xFFE0 122C
ICPLB_DATA12	0xFFE0 1230
ICPLB_DATA13	0xFFE0 1234
ICPLB_DATA14	0xFFE0 1238
ICPLB_DATA15	0xFFE0 123C

DCPLB_DATAx Registers

Figure 6-21 shows the DCPLB Data registers (DCPLB_DATAx).

To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.

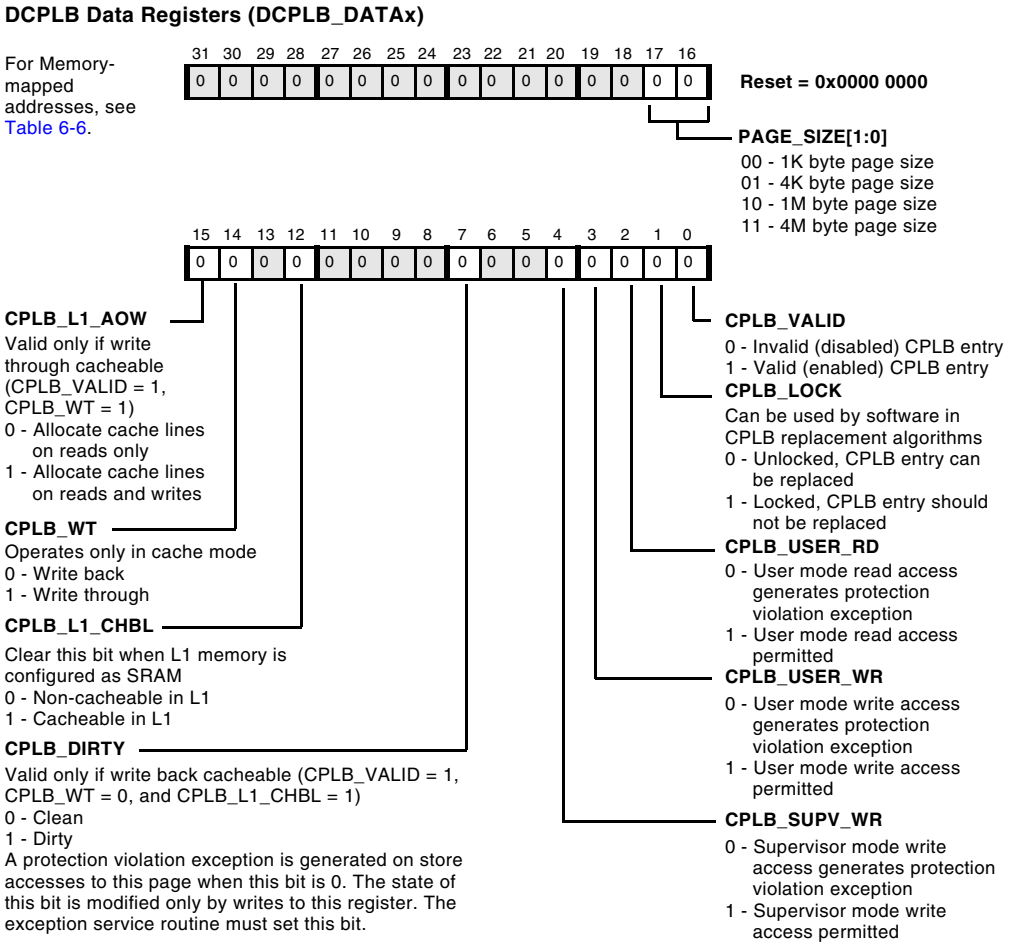


Figure 6-21. DCPLB Data Registers

Table 6-6. DCPLB Data Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DCPLB_DATA0	0xFFE0 0200
DCPLB_DATA1	0xFFE0 0204
DCPLB_DATA2	0xFFE0 0208
DCPLB_DATA3	0xFFE0 020C
DCPLB_DATA4	0xFFE0 0210
DCPLB_DATA5	0xFFE0 0214
DCPLB_DATA6	0xFFE0 0218
DCPLB_DATA7	0xFFE0 021C
DCPLB_DATA8	0xFFE0 0220
DCPLB_DATA9	0xFFE0 0224
DCPLB_DATA10	0xFFE0 0228
DCPLB_DATA11	0xFFE0 022C
DCPLB_DATA12	0xFFE0 0230
DCPLB_DATA13	0xFFE0 0234
DCPLB_DATA14	0xFFE0 0238
DCPLB_DATA15	0xFFE0 023C

DCPLB_ADDRx Registers

Figure 6-22 shows the DCPLB Address registers (DCPLB_ADDRx).

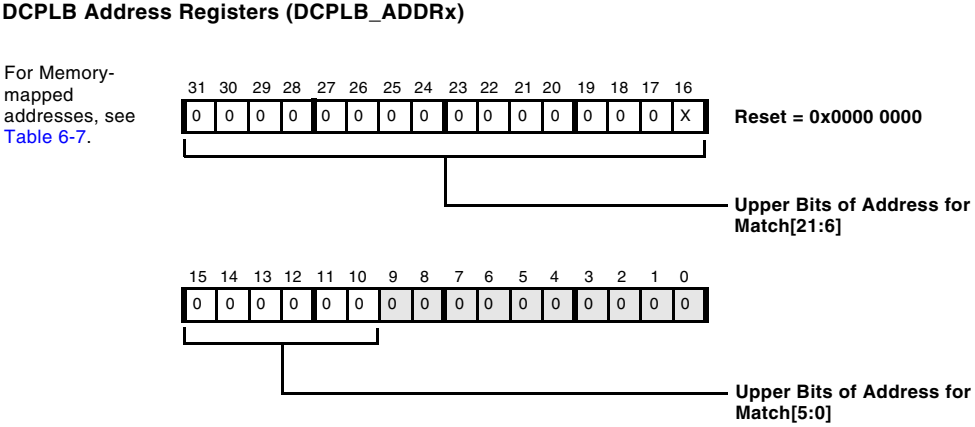


Figure 6-22. DCPLB Address Registers

Table 6-7. DCPLB Address Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DCPLB_ADDR0	0xFFE0 0100
DCPLB_ADDR1	0xFFE0 0104
DCPLB_ADDR2	0xFFE0 0108
DCPLB_ADDR3	0xFFE0 010C
DCPLB_ADDR4	0xFFE0 0110
DCPLB_ADDR5	0xFFE0 0114
DCPLB_ADDR6	0xFFE0 0118
DCPLB_ADDR7	0xFFE0 011C
DCPLB_ADDR8	0xFFE0 0120
DCPLB_ADDR9	0xFFE0 0124

Table 6-7. DCPLB Address Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DCPLB_ADDR10	0xFFE0 0128
DCPLB_ADDR11	0xFFE0 012C
DCPLB_ADDR12	0xFFE0 0130
DCPLB_ADDR13	0xFFE0 0134
DCPLB_ADDR14	0xFFE0 0138
DCPLB_ADDR15	0xFFE0 013C

ICPLB_ADDRx Registers

Figure 6-23 shows the ICPLB Address registers (ICPLB_ADDRx).

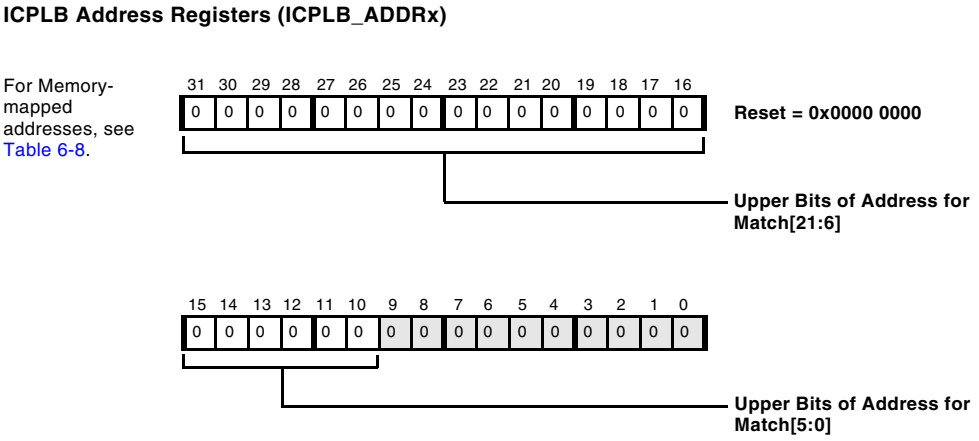


Figure 6-23. ICPLB Address Registers

Table 6-8. ICPLB Address Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
ICPLB_ADDR0	0xFFE0 1100
ICPLB_ADDR1	0xFFE0 1104
ICPLB_ADDR2	0xFFE0 1108
ICPLB_ADDR3	0xFFE0 110C
ICPLB_ADDR4	0xFFE0 1110
ICPLB_ADDR5	0xFFE0 1114
ICPLB_ADDR6	0xFFE0 1118
ICPLB_ADDR7	0xFFE0 111C
ICPLB_ADDR8	0xFFE0 1120
ICPLB_ADDR9	0xFFE0 1124
ICPLB_ADDR10	0xFFE0 1128
ICPLB_ADDR11	0xFFE0 112C
ICPLB_ADDR12	0xFFE0 1130
ICPLB_ADDR13	0xFFE0 1134
ICPLB_ADDR14	0xFFE0 1138
ICPLB_ADDR15	0xFFE0 113C

DCPLB_STATUS and ICPLB_STATUS Registers

Bits in the DCPLB Status register (`DCPLB_STATUS`) and ICPLB Status register (`ICPLB_STATUS`) identify the CPLB entry that has triggered CPLB-related exceptions. The exception service routine can infer the cause of the fault by examining the CPLB entries.

The `DCPLB_STATUS` and `ICPLB_STATUS` registers are valid only while in the faulting exception service routine.

Bits FAULT_DAG, FAULT_USERSUPV and FAULT_RW in the DCPLB Status register (DCPLB_STATUS) are used to identify the CPLB entry that has triggered the CPLB-related exception (see Figure 6-24).

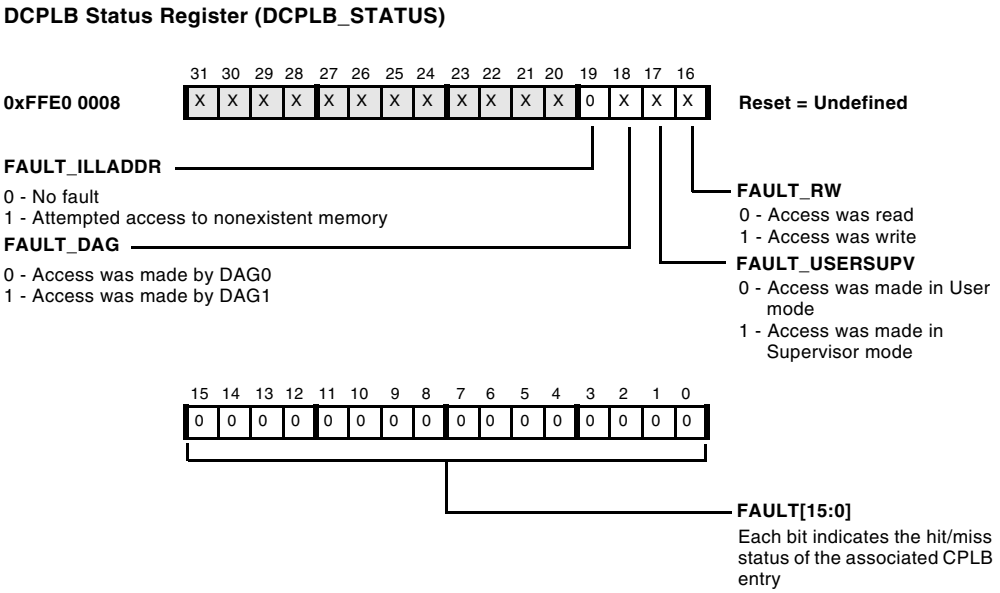


Figure 6-24. DCPLB Status Register

Bit FAULT_USERSUPV in the ICPLB Status register (ICPLB_STATUS) is used to identify the CPLB entry that has triggered the CPLB-related exception (see Figure 6-25).

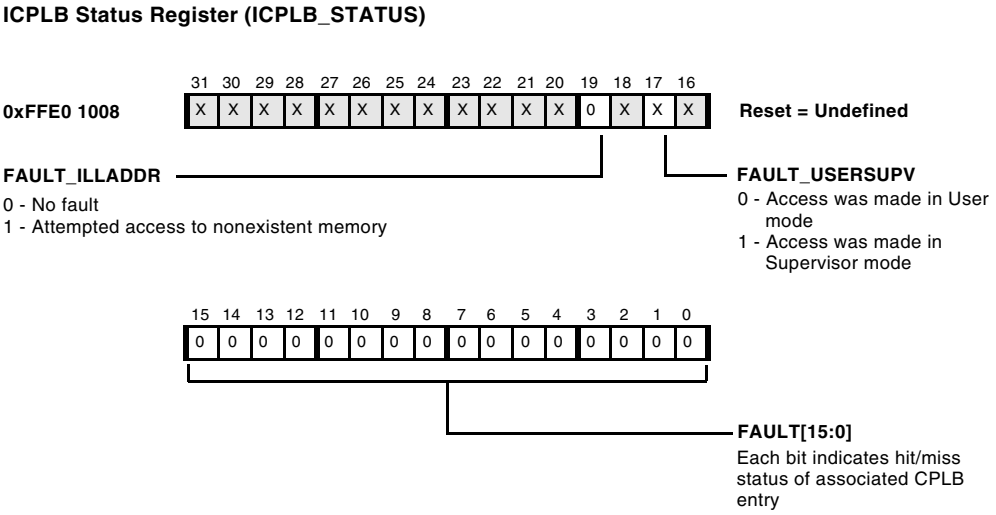


Figure 6-25. ICPLB Status Register

DCPLB_FAULT_ADDR and ICPLB_FAULT_ADDR Registers

The DCPLB Address register (`DCPLB_FAULT_ADDR`) and ICPLB Fault Address register (`ICPLB_FAULT_ADDR`) hold the address that has caused a fault in the L1 Data Memory or L1 Instruction Memory, respectively. See Figure 6-26 and Figure 6-27.

The `DCPLB_FAULT_ADDR` and `ICPLB_FAULT_ADDR` registers are valid only while in the faulting exception service routine.

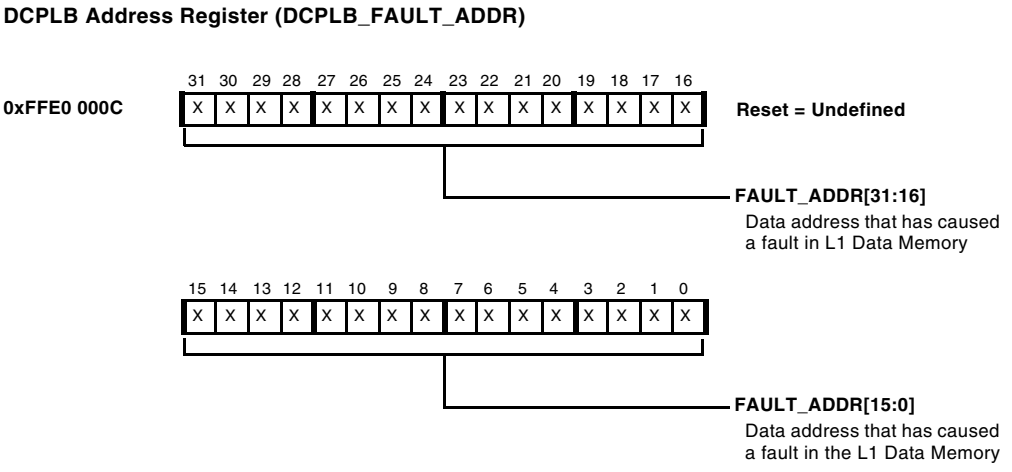


Figure 6-26. DCPLB Address Register

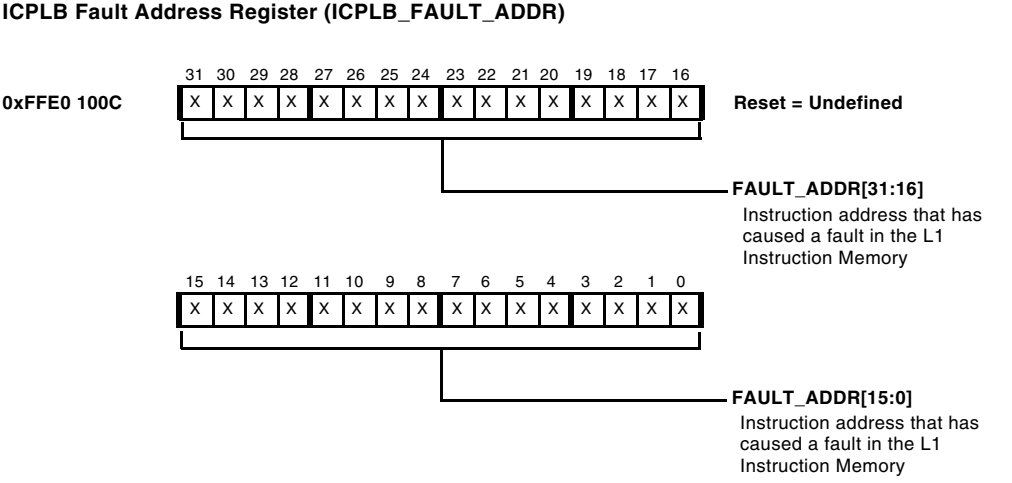


Figure 6-27. ICPLB Fault Address Register

Memory Transaction Model

Both internal and external memory locations are accessed in little endian byte order. Figure 6-28 shows a data word stored in register R0 and in memory at address location *addr*. B0 refers to the least significant byte of the 32-bit word.

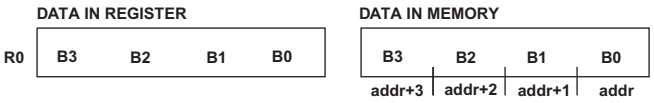


Figure 6-28. Data Stored in Little Endian Order

Figure 6-29 shows 16- and 32-bit instructions stored in memory. The diagram on the left shows 16-bit instructions stored in memory with the most significant byte of the instruction stored in the high address (byte B1 in *addr+1*) and the least significant byte in the low address (byte B0 in *addr*).

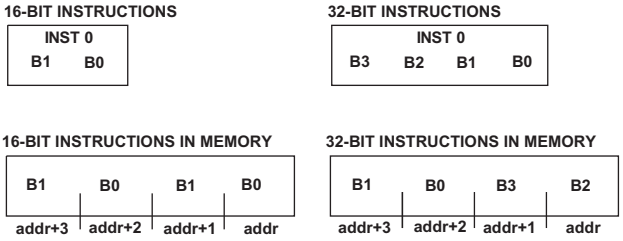


Figure 6-29. Instructions Stored in Little Endian Order

The diagram on the right shows 32-bit instructions stored in memory. Note the most significant 16-bit half word of the instruction (bytes B3 and B2) is stored in the low addresses (*addr+1* and *addr*), and the least significant half word (bytes B1 and B0) is stored in the high addresses (*addr+3* and *addr+2*).

Load/Store Operation

The Blackfin processor architecture supports the RISC concept of a Load/Store machine. This machine is the characteristic in RISC architectures whereby memory operations (loads and stores) are intentionally separated from the arithmetic functions that use the targets of the memory operations. The separation is made because memory operations, particularly instructions that access off-chip memory or I/O devices, often take multiple cycles to complete and would normally halt the processor, preventing an instruction execution rate of one instruction per cycle.

Separating load operations from their associated arithmetic functions allows compilers or assembly language programmers to place unrelated instructions between the load and its dependent instructions. If the value is returned before the dependent operation reaches the execution stage of the pipeline, the operation completes in one cycle.

In write operations, the store instruction is considered complete as soon as it executes, even though many cycles may execute before the data is actually written to an external memory or I/O location. This arrangement allows the processor to execute one instruction per clock cycle, and it implies that the synchronization between when writes complete and when subsequent instructions execute is not guaranteed. Moreover, this synchronization is considered unimportant in the context of most memory operations.

Interlocked Pipeline

In the execution of instructions, the Blackfin processor architecture implements an interlocked pipeline. When a load instruction executes, the target register of the read operation is marked as busy until the value is returned from the memory system. If a subsequent instruction tries to access this register before the new value is present, the pipeline will stall until the memory operation completes. This stall guarantees that

instructions that require the use of data resulting from the load do not use the previous or invalid data in the register, even though instructions are allowed to start execution before the memory read completes.

This mechanism allows the execution of independent instructions between the load and the instructions that use the read target without requiring the programmer or compiler to know how many cycles are actually needed for the memory-read operation to complete. If the instruction immediately following the load uses the same register, it simply stalls until the value is returned. Consequently, it operates as the programmer expects. However, if four other instructions are placed after the load but before the instruction that uses the same register, all of them execute, and the overall throughput of the processor is improved.

Ordering of Loads and Stores

The relaxation of synchronization between memory access instructions and their surrounding instructions is referred to as weak ordering of loads and stores. Weak ordering implies that the timing of the actual completion of the memory operations—even the order in which these events occur—may not align with how they appear in the sequence of the program source code. All that is guaranteed is:

- Load operations will complete before the returned data is used by a subsequent instruction.
- Load operations using data previously written will use the updated values.
- Store operations will eventually propagate to their ultimate destination.

Because of weak ordering, the memory system is allowed to prioritize reads over writes. In this case, a write that is queued anywhere in the pipeline, but not completed, may be deferred by a subsequent read operation, and the read is allowed to be completed before the write. Reads are

prioritized over writes because the read operation has a dependent operation waiting on its completion, whereas the processor considers the write operation complete, and the write does not stall the pipeline if it takes more cycles to propagate the value out to memory. This behavior could cause a read that occurs in the program source code after a write in the program flow to actually return its value before the write has been completed. This ordering provides significant performance advantages in the operation of most memory instructions. However, it can cause side effects that the programmer must be aware of to avoid improper system operation.

When writing to or reading from nonmemory locations such as I/O device registers, the order of how read and write operations complete is often significant. For example, a read of a status register may depend on a write to a control register. If the address is the same, the read would return a value from the write buffer rather than from the actual I/O device register, and the order of the read and write at the register may be reversed. Both these effects could cause undesirable side effects in the intended operation of the program and peripheral. To ensure that these effects do not occur in code that requires precise (strong) ordering of load and store operations, synchronization instructions (`CSYNC` or `SSYNC`) should be used.

Synchronizing Instructions

When strong ordering of loads and stores is required, as may be the case for sequential writes to an I/O device for setup and control, use the core or system synchronization instructions, `CSYNC` or `SSYNC`, respectively.

The `CSYNC` instruction ensures all pending core operations have completed and the core buffer (between the processor core and the L1 memories) has been flushed before proceeding to the next instruction. Pending core operations may include any pending interrupts, speculative states (such as branch predictions), or exceptions.

Consider the following example code sequence:

```
IF CC JUMP away_from_here
csync;
r0 = [p0];
away_from_here:
```

In the preceding example code, the CSYNC instruction ensures:

- The conditional branch (IF CC JUMP away_from_here) is resolved, forcing stalls into the execution pipeline until the condition is resolved and any entries in the processor store buffer have been flushed.
- All pending interrupts or exceptions have been processed before CSYNC completes.
- The load is not fetched from memory speculatively.

The SSYNC instruction ensures that all side effects of previous operations are propagated out through the interface between the L1 memories and the rest of the chip. In addition to performing the core synchronization functions of CSYNC, the SSYNC instruction flushes any write buffers between the L1 memory and the system domain and generates a sync request to the system that requires acknowledgement before SSYNC completes.

Speculative Load Execution

Load operations from memory do not change the state of the memory value. Consequently, issuing a speculative memory-read operation for a subsequent load instruction usually has no undesirable side effect. In some code sequences, such as a conditional branch instruction followed by a

load, performance may be improved by speculatively issuing the read request to the memory system before the conditional branch is resolved. For example,

```
            IF CC JUMP away_from_here
            R0 = [P2];
            …
away_from_here:
```

If the branch is taken, then the load is flushed from the pipeline, and any results that are in the process of being returned can be ignored. Conversely, if the branch is not taken, the memory will have returned the correct value earlier than if the operation were stalled until the branch condition was resolved.

However, in the case of an I/O device, this could cause an undesirable side effect for a peripheral that returns sequential data from a FIFO or from a register that changes value based on the number of reads that are requested. To avoid this effect, use synchronizing instructions (`CSYNC` or `SSYNC`) to guarantee the correct behavior between read operations.

Store operations never access memory speculatively, because this could cause modification of a memory value before it is determined whether the instruction should have executed.

Conditional Load Behavior

The synchronization instructions force all speculative states to be resolved before a load instruction initiates a memory reference. However, the load instruction itself may generate more than one memory-read operation, because it is interruptible. If an interrupt of sufficient priority occurs between the completion of the synchronization instruction and the completion of the load instruction, the sequencer cancels the load instruction. After execution of the interrupt, the interrupted load is executed again. This approach minimizes interrupt latency. However, it is possible that a memory-read cycle was initiated before the load was canceled, and this

would be followed by a second read operation after the load is executed again. For most memory accesses, multiple reads of the same memory address have no side effects. However, for some memory-mapped devices, such as peripheral data FIFOs, reads are destructive. Each time the device is read, the FIFO advances, and the data cannot be recovered and re-read.

When accessing memory-mapped devices that have state dependencies on the number of read operations on a given address location, disable interrupts before performing the load operation.

Working With Memory

This section contains information about alignment of data in memory and memory operations that support semaphores between tasks. It also contains a brief discussion of MMR registers and a core MMR programming example.

Alignment

Nonaligned memory operations are not directly supported. A nonaligned memory reference generates a Misaligned Access exception event (see "Exceptions" on page 4-41). However, because some datastreams (such as 8-bit video data) can properly be nonaligned in memory, alignment exceptions may be disabled by using the `DISALGNEXCPT` instruction. Moreover, some instructions in the quad 8-bit group automatically disable alignment exceptions.

Cache Coherency

For shared data, software must provide cache coherency support as required. To accomplish this, use the `FLUSH` instruction (see "Data Cache Control Instructions" on page 6-40), and/or explicit line invalidation through the core MMRs (see "Data Test Registers" on page 6-41).

Atomic Operations

The processor provides a single atomic operation: TESTSET. Atomic operations are used to provide noninterruptible memory operations in support of semaphores between tasks. The TESTSET instruction loads an indirectly addressed memory half word, tests whether the low byte is zero, and then sets the most significant bit (MSB) of the low memory byte without affecting any other bits. If the byte is originally zero, the instruction sets the CC bit. If the byte is originally nonzero, the instruction clears the CC bit. The sequence of this memory transaction is atomic—hardware bus locking insures that no other memory operation can occur between the test and set portions of this instruction. The TESTSET instruction can be interrupted by the core. If this happens, the TESTSET instruction is executed again upon return from the interrupt.

The TESTSET instruction can address the entire 4G byte memory space, but should not target on-core memory (L1 or MMR space) since atomic access to this memory is not supported.

The memory architecture always treats atomic operations as cache inhibited accesses even if the CPLB descriptor for the address indicates cache enabled access. However, executing TESTSET operations on cacheable regions of memory is not recommended since the architecture cannot guarantee a cacheable location of memory is coherent when the TESTSET instruction is executed.

Memory-Mapped Registers

The MMR reserved space is located at the top of the memory space (0xFFC0 0000). This region is defined as non-cacheable and is divided between the system MMRs (0xFFC0 0000–0xFFE0 0000) and core MMRs (0xFFE0 0000–0xFFFF FFFF).

If strong ordering is required, place a synchronization instruction after stores to MMRs. For more information, see "Load/Store Operation" on page 6-66.

All MMRs are accessible only in Supervisor mode. Access to MMRs in User mode generates a protection violation exception. Attempts to access MMR space using DAG1 also generates a protection violation exception.

All core MMRs are read and written using 32-bit aligned accesses. However, some MMRs have fewer than 32 bits defined. In this case, the unused bits are reserved. System MMRs may be 16 bits.

Accesses to nonexistent MMRs generate an illegal access exception. The system ignores writes to read-only MMRs.

Appendix A provides a summary of all Core MMRs. Appendix B provides a summary of all System MMRs.

Core MMR Programming Code Example

Core MMRs may be accessed only as aligned 32-bit words. Nonaligned access to MMRs generates an exception event. Listing 6-1 shows the instructions required to manipulate a generic core MMR.

Listing 6-1. Core MMR Programming

```
CLI R0;   /*  stop interrupts and save IMASK */
P0 = MMR_BASE;   /*  32-bit instruction to load base of MMRs */
R1 = [P0 + TIMER_CONTROL_REG];   /*  get value of control reg */
BITSET R1, #N;   /*  set bit N */
[P0 + TIMER_CONTROL_REG] = R1;   /*  restore control reg */
CSYNC;   /*  assures that the control reg is written */
STI R0;   /*  enable interrupts */
```

The `CLI` instruction saves the contents of the `IMASK` register and disables interrupts by clearing `IMASK`. The `STI` instruction restores the contents of the `IMASK` register, thus enabling interrupts. The instructions between `CLI` and `STI` are not interruptible.

Terminology

The following terminology is used to describe memory.

cache block. The smallest unit of memory that is transferred to/from the next level of memory from/to a cache as a result of a cache miss.

cache hit. A memory access that is satisfied by a valid, present entry in the cache.

cache line. Same as cache block. In this chapter, cache line is used for cache block.

cache miss. A memory access that does not match any valid entry in the cache.

direct-mapped. Cache architecture in which each line has only one place in which it can appear in the cache. Also described as 1-Way associative.

dirty or **modified**. A state bit, stored along with the tag, indicating whether the data in the data cache line has been changed since it was copied from the source memory and, therefore, needs to be updated in that source memory.

exclusive, clean. The state of a data cache line, indicating that the line is valid and that the data contained in the line matches that in the source memory. The data in a clean cache line does not need to be written to source memory before it is replaced.

fully associative. Cache architecture in which each line can be placed anywhere in the cache.

index. Address portion that is used to select an array element (for example, a line index).

invalid. Describes the state of a cache line. When a cache line is invalid, a cache line match cannot occur.

least recently used (LRU) algorithm. Replacement algorithm, used by cache, that first replaces lines that have been unused for the longest time.

Level 1 (L1) memory. Memory that is directly accessed by the core with no intervening memory subsystems between it and the core.

little endian. The native data store format of the Blackfin processor. Words and half words are stored in memory (and registers) with the least significant byte at the lowest byte address and the most significant byte in the highest byte address of the data storage location.

replacement policy. The function used by the processor to determine which line to replace on a cache miss. Often, an LRU algorithm is employed.

set. A group of *N*-line storage locations in the Ways of an *N*-Way cache, selected by the `INDEX` field of the address (see Figure 6-7).

set associative. Cache architecture that limits line placement to a number of sets (or Ways).

tag. Upper address bits, stored along with the cached data line, to identify the specific address source in memory that the cached line represents.

valid. A state bit, stored with the tag, indicating that the corresponding tag and data are current and correct and can be used to satisfy memory access requests.

victim. A dirty cache line that must be written to memory before it can be replaced to free space for a cache line allocation.

Way. An array of line storage elements in an *N*-Way cache (see Figure 6-7).

write back. A cache write policy, also known as *copyback*. The write data is written only to the cache line. The modified cache line is written to source memory only when it is replaced. Cache lines are allocated on both reads and writes.

write through. A cache write policy (also known as store through). The write data is written to both the cache line and to the source memory. The modified cache line is *not* written to the source memory when it is replaced. Cache lines must be allocated on reads, and may be allocated on writes (depending on mode).

7 CHIP BUS HIERARCHY

This chapter discusses on-chip buses, how data moves through the system, and factors that determine the system organization. The chapter also describes the system internal chip interfaces and discusses the system interconnects and associated system buses.

Internal Interfaces

Figure 7-1 shows the core processor and system boundaries as well as the interfaces between them.

Internal Clocks

The core processor clock (CCLK) rate is highly programmable with respect to CLKIN. The CCLK rate is divided down from the Phase Locked Loop (PLL) output rate. This divider ratio is set using the CSEL parameter of the PLL Divide register.

The Peripheral Access Bus (PAB), the DMA Access Bus (DAB), the External Access Bus (EAB), the DMA Core Bus (DCB), the DMA External Bus (DEB), the External Port Bus (EPB), and the External Bus Interface Unit (EBIU) run at system clock frequency (SCLK domain). This divider ratio is set using the SSEL parameter of the PLL Divide register and must be set so that these buses run as specified in the processor data sheet, and slower than or equal to the core clock frequency.

These buses can also be cycled at a programmable frequency to reduce power consumption, or to allow the core processor to run at an optimal frequency. Note all synchronous peripherals derive their timing from the SCLK. For example, the UART clock rate is determined by further dividing this clock frequency.

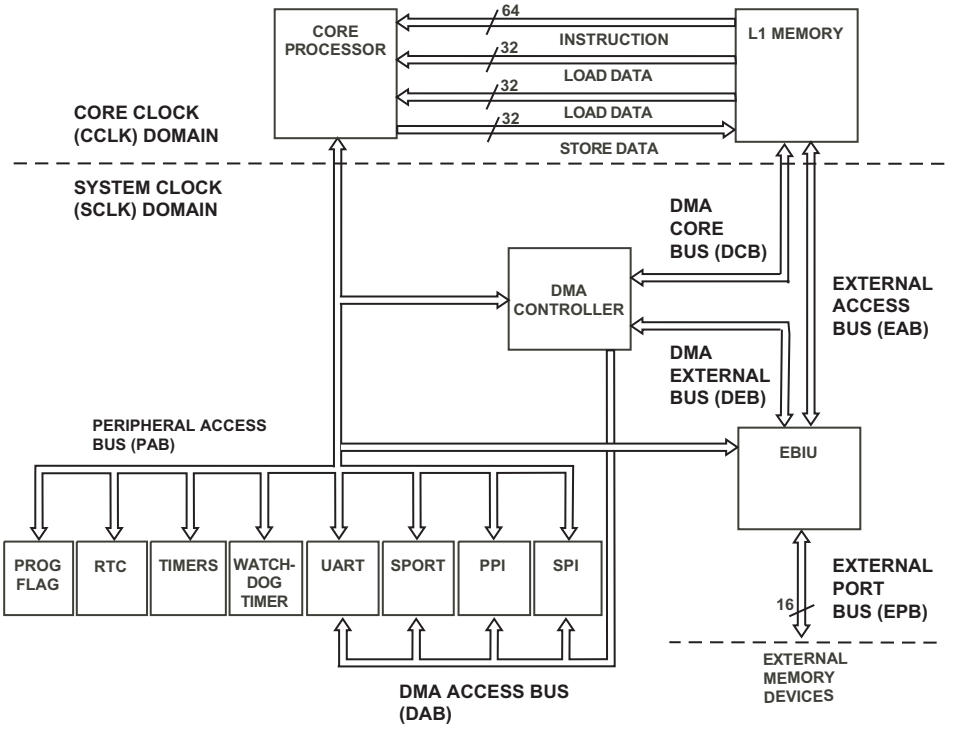


Figure 7-1. Processor Bus Hierarchy

Core Overview

For the purposes of this discussion, Level 1 memories (L1) are included in the description of the core; they have full bandwidth access from the processor core with a 64-bit instruction bus and two 32-bit data buses.

Figure 7-2 shows the core processor and its interfaces to the peripherals and external memory resources.

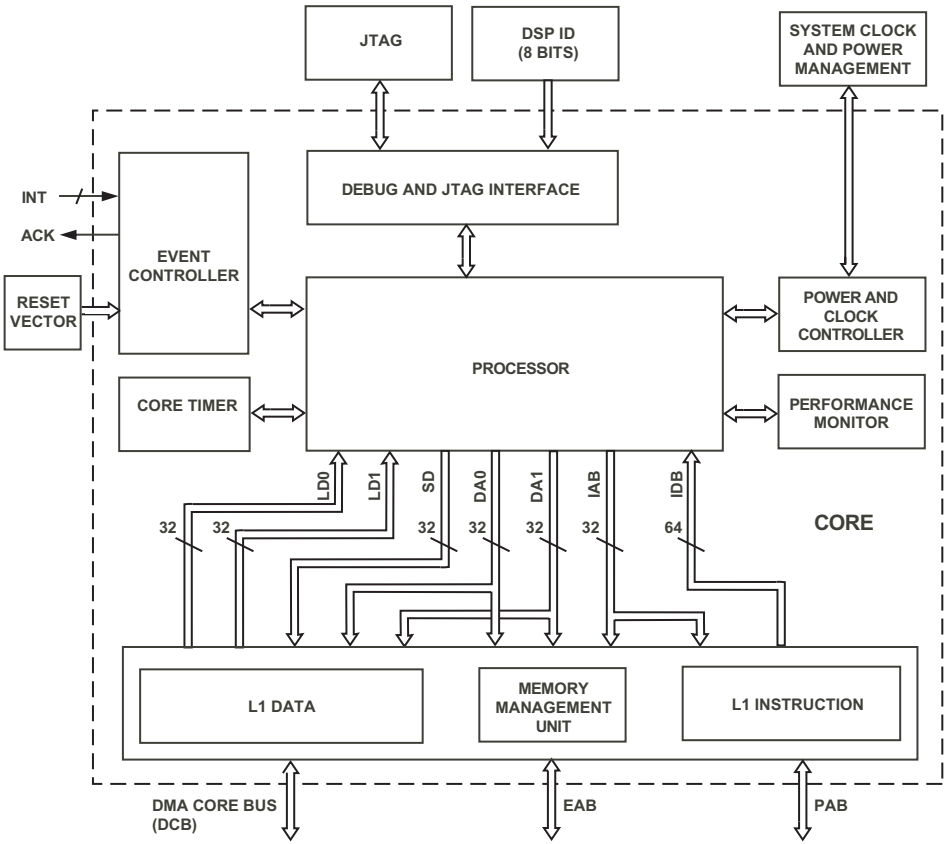


Figure 7-2. Core Block Diagram

The core can generate up to three simultaneous off-core accesses per cycle.

The core bus structure between the processor and L1 memory runs at the full core frequency and has data paths up to 64 bits.

When the instruction request is filled, the 64-bit read can contain a single 64-bit instruction or any combination of 16-, 32-, or 64-bit (partial) instructions.

When cache is enabled, four 64-bit read requests are issued to support 32-byte line fill burst operations. These requests are pipelined so that each transfer after the first is filled in a single, consecutive cycle.

System Overview

The system includes the controllers for system interrupts, test/emulation, and clock and power management. Synchronous clock domain conversion is provided to support clock domain transactions between the core and the system.

System Interfaces

The processor system includes:

- The peripheral set (Timers, Real-Time Clock, programmable flags, UART, SPORTs, PPI, Watchdog timer, and SPI)
- The external memory controller (EBIU)
- The DMA controller
- The interfaces between these, the system, and the optional external (off-chip) resources

See Figure 7-2.

The following sections describe the on-chip interfaces between the system and the peripherals:

- Peripheral Access Bus (PAB)
- DMA Access Bus (DAB)
- DMA Core Bus (DCB)
- DMA External Bus (DEB)
- External Access Bus (EAB)

The External Bus Interface Unit (EBIU) is the primary chip pin bus. The EBIU is discussed in Chapter 17, "External Bus Interface Unit."

Peripheral Access Bus (PAB)

The processor has a dedicated peripheral bus. A low latency peripheral bus keeps core stalls to a minimum and allows for manageable interrupt latencies to time-critical peripherals. All peripheral resources accessed through the PAB are mapped into the system MMR space of the processor memory map. The core can access system MMR space through the PAB bus.

The core processor has byte addressability, but the programming model is restricted to only 32-bit (aligned) access to the system MMRs. Byte access to this region is not supported.

PAB Arbitration

The core is the only master on this bus. No arbitration is necessary.

PAB Performance

For the PAB, the primary performance criteria is latency, not throughput. Transfer latencies for both read and write transfers on the PAB are 2 SCLK cycles.

For example, the core can transfer up to 32 bits per access to the PAB slaves. With the core clock running at 2x the frequency of the system clock, the first and subsequent system MMR read or write accesses take 4 core clocks (CCLK) of latency.

The PAB has a maximum frequency of SCLK.

PAB Agents (Masters, Slaves)

The processor core can master bus operations on the PAB. All peripherals have a peripheral bus slave interface which allows the core to access control and status state. These registers are mapped into the system MMR space of the memory map. Appendix B lists system MMR addresses.

The slaves on the PAB bus are:

- Event Controller
- Clock and Power Management Controller
- Watchdog Timer
- Real-Time Clock (RTC)
- Timer 0, 1, and 2
- SPORT0
- SPORT1
- SPI
- Programmable Flags
- UART
- PPI
- Asynchronous Memory Controller (AMC)

- SDRAM Controller (SDC)
- DMA Controller

DMA Access Bus (DAB), DMA Core Bus (DCB), DMA External Bus (DEB)

The DAB, DCB, and DEB buses provide a means for DMA-capable peripherals to gain access to on-chip and off-chip memory with little or no degradation in core bandwidth to memory.

DAB Arbitration

There are six DMA-capable peripherals in the processor system, including the Memory DMA controller. Twelve DMA channels and bus masters support these devices. The peripheral DMA controllers can transfer data between peripherals and internal or external memory. Both the read and write channels of the Memory DMA controller access their descriptor lists through the DAB.

The DCB has priority over the core processor on arbitration into L1 configured as SRAM. For off-chip memory, the core (by default) has priority over the DEB for accesses to the EPB. The processor has a programmable priority arbitration policy on the DAB. Table 7-1 shows the default arbitration priority. In addition, by setting the `CDPRIO` bit in the `EBIU_AMGCTL` register, all DEB transactions to the EPB have priority over core accesses to external memory. Use of this bit is application-dependent. For example, if you are polling a peripheral mapped to asynchronous memory with long access times, by default the core will “win” over DMA requests. By setting the `CDPRIO` bit, the core would be held off until DMA requests were serviced.

Table 7-1. DAB, DCB, and DEB Arbitration Priority

DAB, DCB, DEB Master	Default Arbitration Priority
PPI	0 - highest
SPORT0 RCV DMA Controller	1
SPORT1 RCV DMA Controller	3
SPORT0 XMT DMA Controller	2
SPORT1 XMT DMA Controller	4
SPI DMA Controller	5
UART RCV Controller	6
UART XMT Controller	7
Memory DMA0 (dest) Controller	8
Memory DMA0 (source) Controller	9
Memory DMA1 (dest) Controller	10
Memory DMA1 (source) Controller	11 - lowest

DAB, DCB, and DEB Performance

The processor DAB supports data transfers of 16 bits or 32 bits. The data bus has a 16-bit width with a maximum frequency as specified in the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet*.

The DAB has a dedicated port into L1 memory. No stalls occur as long as the core access and the DMA access are not to the same memory bank (4K byte size for L1). If there is a conflict, DMA is the highest priority requester, followed by the core.

Note that a locked transfer by the core processor (for example, execution of a `TESTSET` instruction) effectively disables arbitration for the addressed memory bank or resource until the memory lock is deasserted. DMA controllers cannot perform locked transfers.

DMA access to L1 memory can only be stalled by an access already in progress from another DMA channel. Latencies caused by these stalls are in addition to any arbitration latencies.

The core processor and the DAB must arbitrate for access to external memory through the EBIU. This additional arbitration latency added to the latency required to read off-chip memory devices can significantly degrade DAB throughput, potentially causing peripheral data buffers to underflow or overflow. If you use DMA peripherals other than the Memory DMA controller, and you target external memory for DMA accesses, you need to carefully analyze your specific traffic patterns. Make sure that isochronous peripherals targeting internal memory have enough allocated bandwidth and the appropriate maximum arbitration latencies.

DAB Bus Agents (Masters)

All peripherals capable of sourcing a DMA access are masters on this bus, as shown in Table 7-1. A single arbiter supports a programmable priority arbitration policy for access to the DAB.

When two or more DMA master channels are actively requesting the DAB, bus utilization is considerably higher due to the DAB's pipelined design. Bus arbitration cycles are concurrent with the previous DMA access's data cycles.

External Access Bus (EAB)

The EAB provides a way for the processor core to directly access off-chip memory.

Arbitration of the External Bus

Arbitration for use of external port bus interface resources is required because of possible contention between the potential masters of this bus. A fixed-priority arbitration scheme is used. That is, core accesses via the EAB will be of higher priority than those from the DMA External Bus (DEB).

DEB/EAB Performance

The DEB and the EAB support single word accesses of either 8-bit or 16-bit data types. The DEB and the EAB operate at the same frequency as the PAB and the DAB, up to the maximum `SCLK` frequency specified in the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet*.

Memory DMA transfers can result in repeated accesses to the same memory location. Because the memory DMA controller has the potential of simultaneously accessing on-chip and off-chip memory, considerable throughput can be achieved. The throughput rate for an on-chip/off-chip memory access is limited by the slower of the two accesses.

In the case where the transfer is from on-chip to on-chip memory or from off-chip to off-chip memory, the burst accesses cannot occur simultaneously. The transfer rate is then determined by adding each transfer plus an additional cycle between each transfer.

Table 7-2 shows many types of 16-bit memory DMA transfers. In the table, it is assumed that no other DMA activity is conflicting with ongoing operations. The numbers in the table are theoretical values. These values may be higher when they are measured on actual hardware due to a variety of reasons relating to the device that is connected to the EBIU.

For non-DMA accesses (for example, a core access via the EAB), a 32-bit access to SDRAM (of the form `R0 = [P0];` where P0 points to an address in SDRAM) will always be more efficient than executing two 16-bit

accesses (of the form `R0 = W[P0++];` where P0 points to an address in SDRAM). In this example, a 32-bit SDRAM read will take 10 `SCLK` cycles while 2 16-bit reads will take 9 `SCLK` cycles each.

Table 7-2. Performance of DMA Access to External Memory

Source	Destination	Approximate SCLKs For n Words (from start of DMA to interrupt at end)
16-bit SDRAM	L1 Data memory	n + 14
L1 Data memory	16-bit SDRAM	n + 11
16-bit Async memory	L1 Data memory	xn + 12, where x is the number of wait states + setup/hold SCLK cycles (minimum x = 2)
L1 Data memory	16-bit Async memory	xn + 9, where x is the number of wait states + setup/hold SCLK cycles (minimum x = 2)
16-bit SDRAM	16-bit SDRAM	10 + (17n/7)
16-bit Async memory	16-bit Async memory	10 + 2xn, where x is the number of wait states + setup/hold SCLK cycles (minimum x = 2)
L1 Data memory	L1 Data memory	2n + 12

8 DYNAMIC POWER MANAGEMENT

This chapter describes the Dynamic Power Management functionality of the processor. This functionality includes:

- Clocking
- Phase Locked Loop (PLL)
- Dynamic Power Management Controller
- Operating Modes
- Voltage Control

Clocking

The input clock into the processor, CLKIN, provides the necessary clock frequency, duty cycle, and stability to allow accurate internal clock multiplication by means of an on-chip Phase Locked Loop (PLL) module. During normal operation, the user programs the PLL with a multiplication factor for CLKIN. The resulting, multiplied signal is the Voltage Controlled Oscillator (VCO) clock. A user-programmable value then divides the VCO clock signal to generate the core clock (CCLK).

A user-programmable value divides the VCO signal to generate the system clock (SCLK). The SCLK signal clocks the Peripheral Access Bus (PAB), DMA Access Bus (DAB), External Access Bus (EAB), and the External Bus Interface Unit (EBIU).

These buses run at the PLL frequency divided by 1–15 (SCLK domain). Using the SSEL parameter of the PLL Divide register, select a divider value that allows these buses to run at or below the maximum SCLK rate specified in the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet*.

To optimize performance and power dissipation, the processor allows the core and system clock frequencies to be changed dynamically in a "coarse adjustment." For a "fine adjustment," the PLL clock frequency can also be varied.

Phase Locked Loop and Clock Control

To provide the clock generation for the core and system, the processor uses an analog PLL with programmable state machine control.

The PLL design serves a wide range of applications. It emphasizes embedded and portable applications and low cost, general-purpose processors, in which performance, flexibility, and control of power dissipation are key features. This broad range of applications requires a wide range of frequencies for the clock generation circuitry. The input clock may be a crystal, a crystal oscillator, or a buffered, shaped clock derived from an external system clock oscillator.

The PLL interacts with the Dynamic Power Management Controller (DPMC) block to provide power management functions for the processor. For information about the DPMC, see "Dynamic Power Management Controller" on page 8-12.

PLL Overview

Subject to the maximum VCO frequency, the PLL supports a wide range of multiplier ratios and achieves multiplication of the input clock, CLKIN. To achieve this wide multiplication range, the processor uses a combination of programmable dividers in the PLL feedback circuit and output configuration blocks.

Figure 8-1 illustrates a conceptual model of the PLL circuitry, configuration inputs, and resulting outputs. In the figure, the VCO is an intermediate clock from which the core clock (CCLK) and system clock (SCLK) are derived.

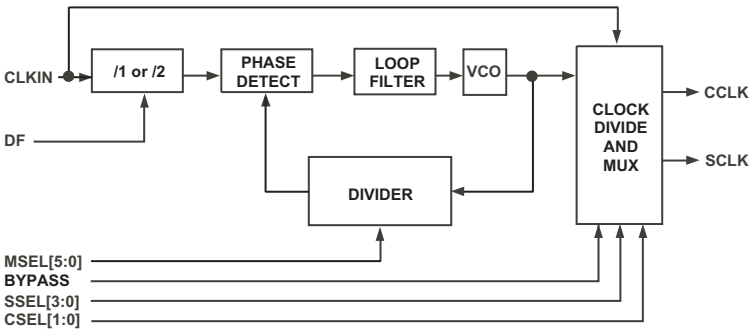


Figure 8-1. PLL Block Diagram

PLL Clock Multiplier Ratios

The PLL Control register (PLL_CTL) governs the operation of the PLL. For details about the PLL_CTL register, see "PLL_CTL Register" on page 8-7.

The Divide Frequency (DF) bit and Multiplier Select (MSEL[5:0]) field configure the various PLL clock dividers:

- DF enables the input divider
- MSEL[5:0] controls the feedback dividers

  The reset value of MSEL is 0xA. This value can be reprogrammed at startup in the boot code.

Table 8-1 illustrates the VCO multiplication factors for the various MSEL and DF settings. In this table, the value x represents the input clock (CLKIN) frequency.

As shown in the table, different combinations of MSEL[5:0] and DF can generate the same VCO frequencies. For a given application, one combination may provide lower power or satisfy the VCO maximum frequency. Under normal conditions, setting DF to 1 typically results in lower power dissipation. See the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* for maximum and minimum frequencies for CLKIN, CCLK, and VCO.

Table 8-1. MSEL Encodings

Signal name MSEL[5:0]	VCO Frequency DF = 0	DF = 1
0	64x	32x
1	1x	0.5x
2	2x	1x
N = 3–62	Nx	0.5Nx
63	63x	31.5x

Core Clock/System Clock Ratio Control

Table 8-2 describes the programmable relationship between the VCO frequency and the core clock. Table 8-3 shows the relationship of the VCO frequency to the system clock. Note the divider ratio must be chosen to limit the SCLK to a frequency specified in the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet*. The SCLK drives all synchronous, system-level logic.

The divider ratio control bits, CSEL and SSEL, are in the PLL Divide register (PLL_DIV). For information about this register, see "PLL_DIV Register" on page 8-7. Appendix B shows the register addresses.

The reset value of CSEL[1:0] is 0x0 (/1), and the reset value of SSEL[3:0] is 0x5. These values can be reprogrammed at startup by the boot code.

By writing the appropriate value to PLL_DIV, you can change the CSEL and SSEL value dynamically. Note the divider ratio of the core clock can never be greater than the divider ratio of the system clock. If the PLL_DIV register is programmed to illegal values, the SCLK divider is automatically increased to be greater than or equal to the core clock divider.

The PLL_DIV register can be programmed at any time to change the CCLK and SCLK divide values without entering the Idle state.

Table 8-2. Core Clock Ratio

Signal Name CSEL[1:0]	Divider Ratio VCO/CCLK	Example Frequency Ratios (MHz) VCO	CCLK
00	1	300	300
01	2	600	300
10	4	600	150
11	8	400	50

As long as the MSEL and DF control bits in the PLL Control register (PLL_CTL) remain constant, the PLL is locked.

Table 8-3. System Clock Ratio

Signal Name SSEL[3:0]	Divider Ratio VCO/SCLK	Example Frequency Ratios (MHz) VCO	SCLK
0000	Reserved	N/A	N/A
0001	1:1	100	100
0010	2:1	200	100
0011	3:1	400	133
0100	4:1	500	125
0101	5:1	600	120
0110	6:1	600	100
N = 7–15	N:1	600	600/N

If changing the clock ratio via writing a new SSEL value into PLL_DIV, take care that the enabled peripherals do not suffer data loss due to SCLK frequency changes.

PLL Registers

The user interface to the PLL is through four memory-mapped registers (MMRs):

- The PLL Divide register (PLL_DIV)
- The PLL Control register (PLL_CTL)
- The PLL Status register (PLL_STAT)
- The PLL Lock Count register (PLL_LOCKCNT)

All four registers are 16-bit MMRs and must be accessed with aligned 16-bit reads/writes.

PLL_DIV Register

The PLL Divide register (PLL_DIV) divides the PLL output clock to create the processor Core Clock (CCLK) and the System Clock (SCLK). These values can be independently changed during processing to reduce power dissipation without changing the PLL state. The only restrictions are the resulting CCLK frequency must be greater than or equal to the SCLK frequency, and SCLK must fall within the allowed range specified in the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet*. If the CCLK and SCLK divide values are programmed otherwise, the SCLK value is automatically adjusted to be slower than or equal to the core clock. Figure 8-2 shows the bits in the PLL_DIV register.

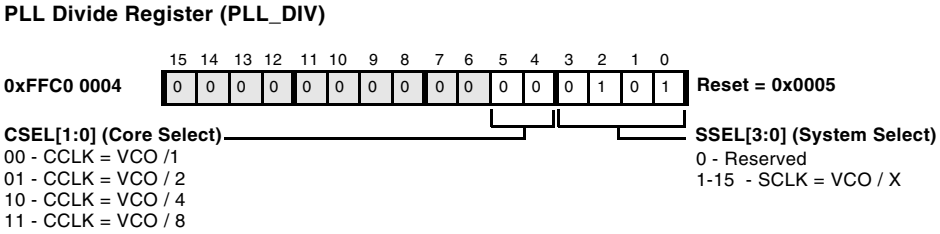


Figure 8-2. PLL Divide Register

PLL_CTL Register

The PLL Control register (PLL_CTL) controls operation of the PLL (see Figure 8-3). Note changes to the PLL_CTL register do not take effect immediately. In general, the PLL_CTL register is first programmed with new values, and then a specific PLL programming sequence must be executed to implement the changes. See "PLL Programming Sequence" on page 8-20.

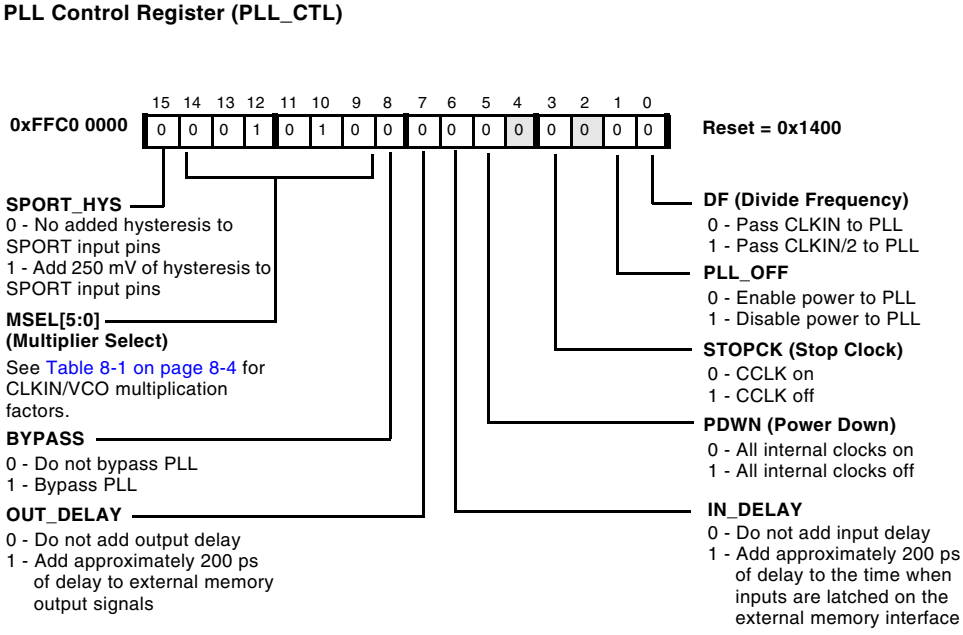


Figure 8-3. The PLL Control Register

The following fields of the PLL_CTL register are used to control the PLL:

- SPORT_HYS] – This bit is used to add 250 mV of hysteresis to the SPORT input pins to provide better immunity to system noise on SPORT clock and frame sync signals configured as inputs.
- MSEL[5:0] – The Multiplier Select (MSEL) field defines the input clock to VCO clock (CLKIN to VCO) multiplier.
- BYPASS – This bit is used to bypass the PLL. When BYPASS is set, CLKIN is passed directly to the core and peripheral clocks.

- OUT_DELAY – This bit is used to add approximately 200ps of delay to external memory output signals. See the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* to determine if this bit should be set.
- IN_DELAY – This bit is used to add approximately 200ps of delay to the time when inputs are latched on the external memory interface. See the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* to determine if this bit should be set.
- PDWN – The Power Down (PDWN) bit is used to place the processor in the Deep Sleep operating mode.

  For information about operating modes, see "Operating Modes" on page 8-13.
- STOPCK – The Stop Clock (STOPCK) bit is used to enable/disable the core clock, CCLK.
- PLL_OFF – This bit is used to enable/disable power to the PLL.
- DF – The Divide Frequency (DF) bit determines whether CLKIN is passed directly to the PLL or CLKIN/2 is passed.

PLL_STAT Register

The PLL Status register (PLL_STAT) indicates the operating mode of the PLL and processor (see Figure 8-4). For more information about operating modes, see "Operating Modes" on page 8-13.

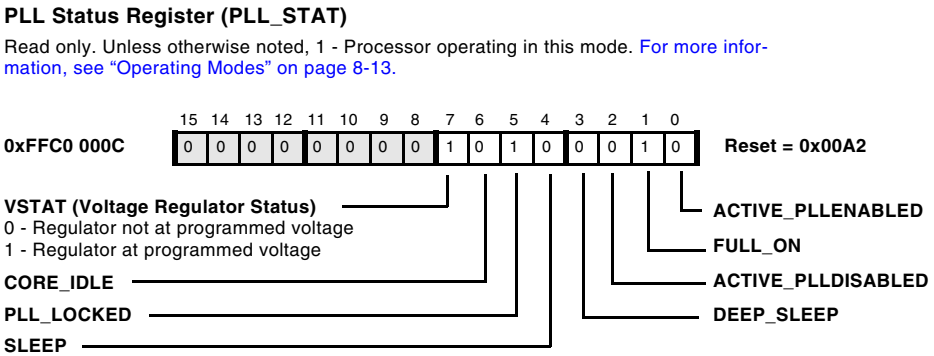


Figure 8-4. PLL Status Register

The following fields are used in the PLL_STAT register:

- VSTAT (Voltage Regulator Status) – This bit indicates whether the voltage regulator has reached the programmed voltage.

  When changing voltage levels, the core must be put into an Idle operating state to allow the PLL to lock with the new voltage level. See "PLL Programming Sequence" on page 8-20.

- CORE_IDLE – This bit is set to 1 when the Blackfin processor core is idled; that is, an IDLE instruction has executed, and the core is awaiting a wakeup signal.

- PLL_LOCKED – This field is set to 1 when the internal PLL lock counter has incremented to the value set in the PLL Lock Count register (PLL_LOCKCNT). For more information, see "PLL_LOCKCNT Register" on page 8-11.

- SLEEP – This field is set to 1 when the processor is in Sleep operating mode.
- DEEP_SLEEP – This field is set to 1 when the processor is in Deep Sleep operating mode.
- ACTIVE_PLLDISABLED – This field is set to 1 when the processor is in Active operating mode with the PLL powered down.
- FULL_ON – This field is set to 1 when the processor is in Full On operating mode.
- ACTIVE_PLLENABLED – This field is set to 1 when the processor is in Active operating mode with the PLL powered up.

PLL_LOCKCNT Register

When changing clock frequencies in the PLL, the PLL requires time to stabilize and lock to the new frequency.

The PLL Lock Count register (PLL_LOCKCNT) defines the number of CLKIN cycles that occur before the processor sets the PLL_LOCKED bit in the PLL_STAT register. When executing the PLL programming sequence, the internal PLL lock counter begins incrementing upon execution of the IDLE instruction. The lock counter increments by 1 each CLKIN cycle. When the lock counter has incremented to the value defined in the PLL_LOCKCNT register, the PLL_LOCKED bit is set.

See the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* for more information about PLL stabilization time and programmed values for this register. For more information about operating

modes, see “Operating Modes” on page 8-13. For further information about the PLL programming sequence, see “PLL Programming Sequence” on page 8-20.

PLL Lock Count Register (PLL_LOCKCNT)

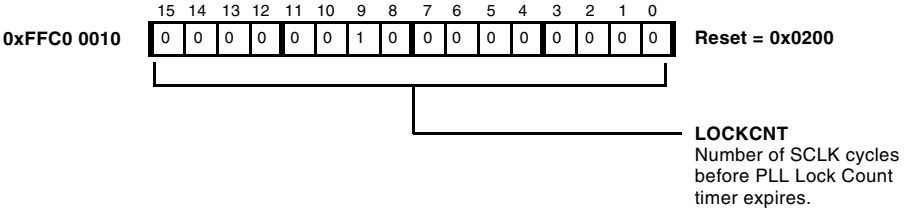


Figure 8-5. PLL Lock Count Register

Dynamic Power Management Controller

The Dynamic Power Management Controller (DPMC) works in conjunction with the PLL, allowing the user to control the processor’s performance characteristics and power dissipation dynamically. The DPMC provides these features that allow the user to control performance and power:

- Multiple operating modes – The processor works in four operating modes, each with different performance characteristics and power dissipation profiles. See “Operating Modes” on page 8-13.
- Peripheral clocks – Clocks to each peripheral are disabled automatically when the peripheral is disabled.

- Voltage control – The processor provides an on-chip switching regulator controller which, with some external components, can generate internal voltage levels from the external Vdd (V_{DDEXT}) supply.

  Depending on the needs of the system, the voltage level can be reduced to save power. See "VR_CTL Register" on page 8-26.

Operating Modes

The processor works in four operating modes, each with unique performance and power saving benefits. Table 8-4 summarizes the operational characteristics of each mode.

Table 8-4. Operational Characteristics

Operating Mode	Power Savings	PLL Status	PLL Bypassed	CCLK	SCLK	Allowed DMA Access
Full On	None	Enabled	No	Enabled	Enabled	L1
Active	Medium	Enabled [1]	Yes	Enabled	Enabled	L1
Sleep	High	Enabled	No	Disabled	Enabled	–
Deep Sleep	Maximum	Disabled	–	Disabled	Disabled	–

1 PLL can also be disabled in this mode.

Dynamic Power Management Controller States

Power management states are synonymous with the PLL control state. The state of the DPMC/PLL can be determined by reading the PLL Status register (see "PLL_STAT Register" on page 8-10). In all modes except Sleep and Deep Sleep, the core can either execute instructions or be in Idle core state. If the core is in the Idle state, it can be awakened.

In all modes except Active, the SCLK frequency is determined by the SSEL-specified ratio to VCO. In Sleep mode, although the core clock is disabled, SCLK continues to run at the specified SSEL ratio.

The following sections describe the DPMC/PLL states in more detail, as they relate to the power management controller functions.

Full On Mode

Full On mode is the maximum performance mode. In this mode, the PLL is enabled and not bypassed. Full On mode is the normal execution state of the processor, with the processor and all enabled peripherals running at full speed. DMA access is available to L1 memories. From Full On mode, the processor can transition directly to Active, Sleep, or Deep Sleep modes, as shown in Figure 8-6.

Active Mode

In Active mode, the PLL is enabled but bypassed. Because the PLL is bypassed, the processor's core clock (CCLK) and system clock (SCLK) run at the input clock (CLKIN) frequency. DMA access is available to appropriately configured L1 memories.

In Active mode, it is possible not only to bypass, but also to disable the PLL. If disabled, the PLL must be re-enabled before transitioning to Full On or Sleep modes.

From Active mode, the processor can transition directly to Full On, Sleep, or Deep Sleep modes.

Sleep Mode

Sleep mode significantly reduces power dissipation by idling the core processor. The CCLK is disabled in this mode; however, SCLK continues to run at the speed configured by MSEL and SSEL bit settings. As CCLK is disabled,

DMA access is available only to external memory in Sleep mode. From Sleep mode, a wakeup event causes the processor to transition to one of these modes:

- Active mode if the BYPASS bit in the PLL_CTL register is set
- Full On mode if the BYPASS bit is cleared

Deep Sleep Mode

Deep Sleep mode maximizes power savings by disabling the PLL, CCLK, and SCLK. In this mode, the processor core and all peripherals except the Real-Time Clock (RTC) are disabled. DMA is not supported in this mode.

Deep Sleep mode can be exited only by an RTC interrupt or hardware reset event. An RTC interrupt causes the processor to transition to Active mode; a hardware reset begins the hardware reset sequence. For more information about hardware reset, see "Hardware Reset" on page 3-13.

Note an RTC interrupt in Deep Sleep mode automatically resets some fields of the PLL Control register (PLL_CTL). See Table 8-5.

When in Deep Sleep operating mode, clocking to the SDRAM is turned off. Before entering Deep Sleep mode, software should ensure either that important information in SDRAM is saved to a non-volatile memory, or that SDRAM is placed in Self-Refresh mode.

Table 8-5. Control Register Values after RTC Wakeup Interrupt

Field	Value
PLL_OFF	0
STOPCK	0

Table 8-5. Control Register Values after RTC Wakeup Interrupt (Cont'd)

Field	Value
PDWN	0
BYPASS	1

Hibernate State

For lowest possible power dissipation, this state allows the internal supply (V_{DDINT}) to be powered down, while keeping the I/O supply (V_{DDEXT}) running. Although not strictly an operating mode like the four modes detailed above, it is illustrative to view it as such in the diagram of Figure 8-6. Since this feature is coupled to the on-chip switching regulator controller, it is discussed in detail in "Powering Down the Core (Hibernate State)" on page 8-29.

Operating Mode Transitions

Figure 8-6 graphically illustrates the operating modes and transitions. In the diagram, ellipses represent operating modes. Arrows between the ellipses show the allowed transitions into and out of each mode.

The text next to each transition arrow shows the fields in the PLL Control register (`PLL_CTL`) that must be changed for the transition to occur. For example, the transition from Full On mode to Sleep mode indicates that the `STOPCK` bit must be set to 1 and the `PDWN` bit must be set to 0. For information about how to effect mode transitions, see "Programming Operating Mode Transitions" on page 8-19.

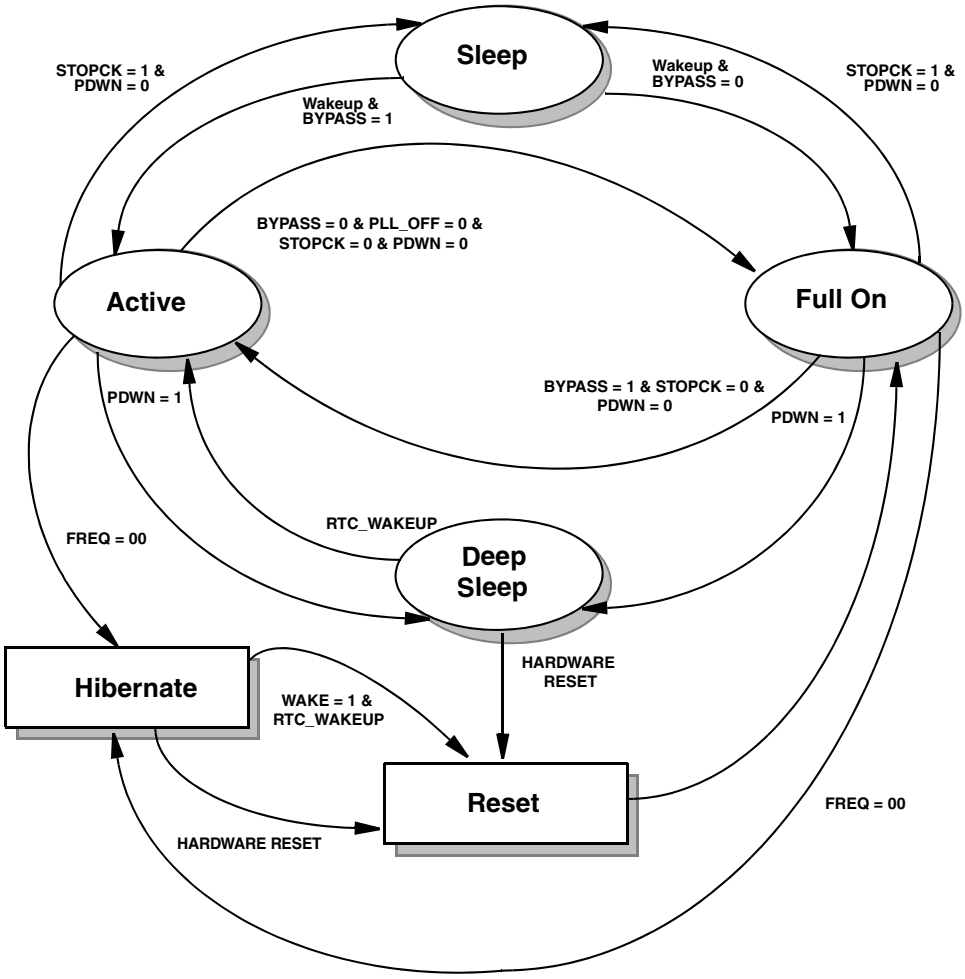


Figure 8-6. Operating Mode Transitions

In addition to the mode transitions shown in Figure 8-6, the PLL can be modified while in Active operating mode. Power to the PLL can be applied and removed, and new clock-in to VCO clock (CLKIN to VCO) multiplier ratios can be programmed. Described in detail below, these changes to the PLL do not take effect immediately. As with operating mode transitions, the PLL programming sequence must be executed for these changes to take effect (see "PLL Programming Sequence" on page 8-20).

- PLL Disabled: In addition to being bypassed in the Active mode, power to the PLL can be removed.

  When power is removed from the PLL, additional power savings are achieved although they are relatively small. To remove power to the PLL, set the PLL_OFF bit in the PLL_CTL register, and then execute the PLL programming sequence.

- PLL Enabled: When the PLL is powered down, power can be reapplied later when additional performance is required.

  Power to the PLL must be reapplied before transitioning to Full On or Sleep operating modes. To apply power to the PLL, clear the PLL_OFF bit in the PLL_CTL register, and then execute the PLL programming sequence.

- New Multiplier Ratio in Active Mode: New clock-in to VCO clock (CLKIN to VCO) multiplier ratios can be programmed while in Active mode.

  Although the CLKIN to VCO multiplier changes are not realized in Active mode, forcing the PLL to lock to the new ratio in Active mode before transitioning to Full On mode reduces the transition time, because the PLL is already locked to the new ratio. Note the PLL must be powered up to lock to the new ratio. To program a

new CLKIN to VCO multiplier, write the new MSEL[5:0] and/or DF values to the PLL_CTL register; then execute the PLL programming sequence.

- New Multiplier Ratio in Full On Mode: The multiplier ratio can also be changed while in Full On mode.

  In this case, the PLL state automatically transitions to Active mode while the PLL is locking. After locking, the PLL returns to Full On state. To program a new CLKIN to VCO multiplier, write the new MSEL[5:0] and/or DF values to the PLL_CTL register; then execute the PLL programming sequence (see page 8-20).

Table 8-6 summarizes the allowed operating mode transitions.

Attempting to cause mode transitions other than those shown in Table 8-6 causes unpredictable behavior.

Table 8-6. Allowed Operating Mode Transitions

New Mode	Current Mode			
	Full On	Active	Sleep	Deep Sleep
Full On	–	Allowed	Allowed	–
Active	Allowed	–	Allowed	Allowed
Sleep	Allowed	Allowed	–	–
Deep Sleep	Allowed	Allowed	–	–

Programming Operating Mode Transitions

The operating mode is defined by the state of the PLL_OFF, BYPASS, STOPCK, and PDWN bits of the PLL Control register (PLL_CTL). Merely modifying the bits of the PLL_CTL register does not change the operating mode or the behavior of the PLL. Changes to the PLL_CTL register are realized only after executing a specific code sequence, which is shown in

Listing 8-1. This code sequence first brings the processor to a known, idled state. Once in this idled state, the PLL recognizes and implements the changes made to the PLL_CTL register. After the changes take effect, the processor operates with the new settings, including the new operating mode, if one is programmed.

PLL Programming Sequence

If new values are assigned to MSEL or DF in the PLL Control register (PLL_CTL), the instruction sequence shown in Listing 8-1 puts those changes into effect. The PLL programming sequence is also executed when transitioning between operating states.

Changes to the divider-ratio bits, CSEL and SSEL, can be made dynamically; they do not require execution of the PLL programming sequence.

Listing 8-1. PLL Programming Sequence

```
CLI R0 ;   /* disable interrupts */
IDLE ;   /* drain pipeline and send core into IDLE state */
STI R0 ;   /* re-enable interrupts after wakeup */
```

The first two instructions in the sequence take the core to an idled state with interrupts disabled; the interrupt mask (IMASK) is saved to the R0 register, and the instruction pipeline is halted. The PLL state machine then loads the PLL_CTL register changes into the PLL.

If the PLL_CTL register changes include a new CLKIN to VCO multiplier or the changes reapply power to the PLL, the PLL needs to relock. To relock, the PLL lock counter is first cleared, and then it begins incrementing, once per SCLK cycle. After the PLL lock counter reaches the value programmed into the PLL Lock Count register (PLL_LOCKCNT), the PLL sets the PLL_LOCKED bit in the PLL Status register (PLL_STAT), and the PLL asserts the PLL wakeup interrupt.

Depending on how the PLL_CTL register is programmed, the processor proceeds in one of the following four ways:

- If the PLL_CTL register is programmed to enter either Active or Full On operating mode, the PLL generates a wakeup signal, and then the processor continues with the STI instruction in the sequence, as described in "PLL Programming Sequence Continues" on page 8-22.

  When the state change enters Full On mode from Active mode or Active from Full On, the PLL itself generates a wakeup signal that can be used to exit the idled core state. The wakeup signal is generated by the PLL itself or another peripheral, watchdog or other timer, RTC, or other source. For more information about events that cause the processor to wakeup from being idled, see "SIC_IWR Register" on page 4-26.

- If the PLL_CTL register is programmed to enter the Sleep operating mode, the processor immediately transitions to the Sleep mode and waits for a wakeup signal before continuing.

  When the wakeup signal has been asserted, the instruction sequence continues with the STI instruction, as described in the section, "PLL Programming Sequence Continues" on page 8-22, causing the processor to transition to:

  —Active mode if BYPASS in the PLL_CTL register is set

  —Full On mode if the BYPASS bit is cleared

- If the PLL_CTL register is programmed to enter Deep Sleep operating mode, the processor immediately transitions to Deep Sleep mode and waits for an RTC interrupt or hardware reset signal:

  —An RTC interrupt causes the processor to enter Active operating mode and continue with the STI instruction in the sequence, as described below.

  —A hardware reset causes the processor to execute the reset sequence, as described in "Hardware Reset" on page 3-13.

- If no operating mode transition is programmed, the PLL generates a wakeup signal, and the processor continues with the STI instruction in the sequence, as described in the following section.

PLL Programming Sequence Continues

The instruction sequence shown in Listing 8-1 then continues with the STI instruction. Interrupts are re-enabled, IMASK is restored, and normal program flow resumes.

To prevent spurious activity, DMA should be suspended while executing this instruction sequence.

Examples

The following code examples illustrate how to effect various operating mode transitions. Some setup code has been removed for clarity, and the following assumptions are made:

- P0 points to the PLL Control register (PLL_CTL). P1 points to the PLL Divide register.
- The PLL wakeup interrupt is enabled as a wakeup signal.
- MSEL[5:0] and DF in PLL_CTL are set to (b#011111) and (b#0) respectively, signifying a CLKIN to VCO multiplier of 31x.

Active Mode to Full On Mode

Listing 8-2 provides code for transitioning from Active operating mode to Full On mode.

Listing 8-2. Transitioning From Active Mode to Full On Mode

```
CLI R2;   /* disable interrupts, copy IMASK to R2 */
R1.L = 0x3E00;   /* clear BYPASS bit */
W[P0] = R1;   /* and write to PLL_CTL */

IDLE;   /* drain pipeline, enter idled state, wait for PLL wakeup
*/
STI R2;   /* after PLL wakeup occurs, restore interrupts and
IMASK */
...   /* processor is now in Full On mode */
```

Full On Mode to Active Mode

Listing 8-3 provides code for transitioning from Full On operating mode to Active mode.

Listing 8-3. Transitioning From Full On Mode to Active Mode

```
CLI R2;   /* disable interrupts, copy IMASK to R2 */
R1.L = 0x3F00;   /* set BYPASS bit */
W[P0] = R1;   /* and write to PLL_CTL */

IDLE;   /* drain pipeline, enter idled state, wait for PLL wakeup
*/
STI R2;   /* after PLL wakeup occurs, restore interrupts and
IMASK */
...   /* processor is now in Active mode */
```

In the Full On Mode, Change CLKIN to VCO Multiplier From 31x to 2x

Listing 8-4 provides code for changing CLKIN to VCO multiplier from 31x to 2x in Full On operating mode.

Listing 8-4. Changing CLKIN to VCO Multiplier

```
CLI R2;   /* disable interrupts, copy IMASK to R2 */
R1.L = 0x0400;   /* change VCO multiplier to 2x */
W[P0] = R1;   /* by writing to PLL_CTL */

IDLE;   /* drain pipeline, enter idled state, wait for PLL wakeup
*/
STI R2;   /* after PLL wakeup occurs, restore interrupts and
IMASK */
...   /* processor is now in Full On mode, with the CLKIN to VCO
multiplier set to 2x */
```

Dynamic Supply Voltage Control

In addition to clock frequency control, the processor provides the capability to run the core processor at different voltage levels. As power dissipation is proportional to the voltage squared, significant power reductions can be accomplished when lower voltages are used.

The processor uses three power domains. These power domains are shown in Table 8-7. Each power domain has a separate V_{DD} supply. Note the internal logic of the processor and much of the processor I/O can be run

over a range of voltages. See the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* for details on the allowed voltage ranges for each power domain and power dissipation data.

Table 8-7. Power Domains

Power Domain	V_{DD} Range
All internal logic except RTC	Variable
Real-Time Clock I/O and internal logic	Variable
All other I/O	Variable

Power Supply Management

The processor provides an on-chip switching regulator controller which, with some external hardware, can generate internal voltage levels from the external V_{DDEXT} supply with an external power transistor as shown in Figure 8-7. This voltage level can be reduced to save power, depending upon the needs of the system.

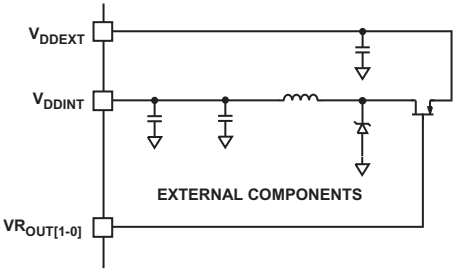


Figure 8-7. Processor Voltage Regulator

When increasing the V_{DDINT} voltage, the external FET will switch on for a longer period. The V_{DDEXT} supply should have appropriate capacitive bypassing to enable it to provide sufficient current without drooping the supply voltage.

VR_CTL Register

The on-chip core voltage regulator controller manages the internal logic voltage levels for the V_{DDINT} supply. The Voltage Regulator Control register (VR_CTL) controls the regulator (see Figure 8-8). Writing to VR_CTL initiates a PLL relock sequence.

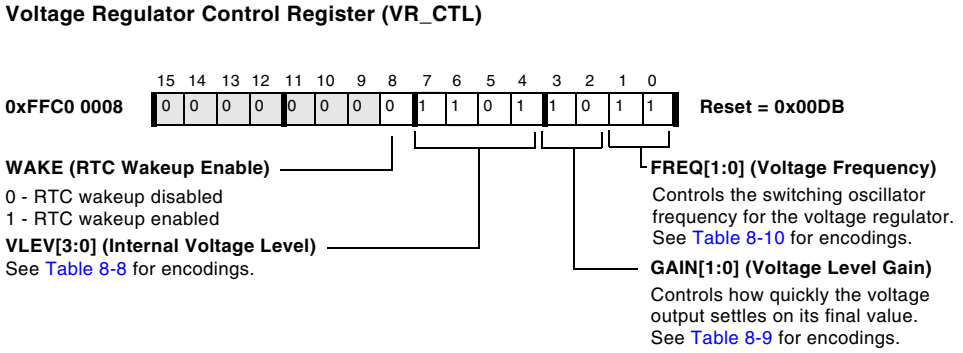


Figure 8-8. Voltage Regulator Control Register

The following fields of the VR_CTL register are used to control internal logic voltage levels:

- WAKE – The Wakeup-enable (WAKE) control bit allows the voltage regulator to be awakened from powerdown (FREQ=00) upon an interrupt from the RTC.

- VLEV[3:0] – The Voltage Level (VLEV) field identifies the nominal internal voltage level. Please refer to the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* for the applicable VLEV voltage range and associated voltage tolerances.
- FREQ[1:0] – The Frequency (FREQ) field controls the switching oscillator frequency for the voltage regulator. A higher frequency setting allows for smaller switching capacitor and inductor values, while potentially generating more EMI (electromagnetic interference).

To bypass onboard regulation, program a value of b#00 in the FREQ field and leave the VROUT pins floating.

- GAIN[1:0] – The Gain (GAIN) field controls the internal loop gain of the switching regulator loop; this bit controls how quickly the voltage output settles on its final value. In general, higher gain allows for quicker settling times but causes more overshoot in the process.

Table 8-8 lists the voltage level values for VLEV[3:0].

Table 8-8. VLEV Encodings

VLEV	Voltage
0000–0101	Reserved
0110	.85 volts
0111	.90 volts
1000	.95 volts
1001	1.00 volts
1010	1.05 volts
1011	1.10 volts
1100	1.15 volts

Table 8-8. VLEV Encodings (Cont'd)

VLEV	Voltage
1101	1.20 volts
1110	1.25 volts
1111	1.30 volts

For legal VLEV values with respect to voltage tolerance, consult the appropriate processor-specific data sheet.

Table 8-9 lists the switching frequency values configured by `FREQ[1:0]`.

Table 8-9. FREQ Encodings

FREQ	Value
00	Powerdown/Bypass onboard regulation
01	333 kHz
10	667 kHz
11	1MHz

Table 8-10 lists the gain levels configured by `GAIN[1:0]`.

Table 8-10. GAIN Encodings

GAIN	Value
00	5
01	10
10	20
11	50

Changing Voltage

Minor changes in operating voltage can be accommodated without requiring special consideration or action by the application program. See the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* for more information about voltage tolerances and allowed rates of change.

Reducing the processor's operating voltage to greatly conserve power or raising the operating voltage to greatly increase performance will probably require significant changes to the operating voltage level. To ensure predictable behavior when varying the operating voltage, the processor should be brought to a known and stable state before the operating voltage is modified.

The recommended procedure is to follow the PLL programming sequence when varying the voltage. After changing the voltage level in the VR_CTL register, the PLL will automatically enter the Active mode when the processor enters the Idle state. At that point the voltage level will change and the PLL will relock with the new voltage. After the PLL_LOCKCNT has expired, the part will return to the Full On state. When changing voltages, a larger PLL_LOCKCNT value may be necessary than when changing just the PLL frequency. See the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* for details.

After the voltage has been changed to the new level, the processor can safely return to any operational mode so long as the operating parameters, such as core clock frequency (CCLK), are within the limits specified in the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* for the new operating voltage level.

Powering Down the Core (Hibernate State)

The internal supply regulator for the processor can be shut off by writing b#00 to the FREQ bits of the VR_CTL register. This disables both CCLK and SCLK. Furthermore, it sets the internal power supply voltage (V_{DDINT}) to

0 V, eliminating any leakage currents from the processor. The internal supply regulator can be woken up either by a Real-Time Clock wakeup or by asserting the $\overline{\texttt{RESET}}$ pin.

If the on-chip supply controller is bypassed, so that V_{DDINT} is sourced externally, the only way to power down the core is to remove the external V_{DDINT} voltage source.

When the core is powered down, V_{DDINT} is set to 0 V, and thus the internal state of the processor is not maintained. Therefore, any critical information stored internally (memory contents, register contents, and so on) must be written to a non-volatile storage device prior to removing power.

Powering down V_{DDINT} does not affect V_{DDEXT}. While V_{DDEXT} is still applied to the processor, external pins are maintained at a tri-state level, unless otherwise specified.

To power down the internal supply:

1. Write 0 to the `SIC_IWR` register to prevent peripheral resources from interrupting the Hibernate process.
2. Write to `VR_CTL`, setting the `FREQ` bits to `b#00`. If the Real-Time Clock is being used to wake up from Hibernate, also set the `WAKE` bit to 1.
3. Execute this code sequence:

```
CLI R0 ;
IDLE ;
```

4. When the Idle state is reached, V_{DDINT} will transition to 0 V.

5. When the processor is woken up, whether by RTC or by a reset interrupt, the PLL relocks and the boot sequence defined by the `BMODE[1:0]` pin settings takes effect.

Failure to allow V_{DDINT} to complete the transition to 0 V before waking up the processor can cause undesired results.

9 DIRECT MEMORY ACCESS

The processor uses Direct Memory Access (DMA) to transfer data within memory spaces or between a memory space and a peripheral. The processor can specify data transfer operations and return to normal processing while the fully integrated DMA controller carries out the data transfers independent of processor activity.

The DMA controller can perform several types of data transfers:

- Between memory and memory (MDMA)
  ("Memory DMA" on page 9-48)
- Between memory and the Serial Peripheral Interface (SPI)
  (Chapter 10, "SPI Compatible Port Controllers")
- Between memory and a Serial Port (SPORT)
  (Chapter 12, "Serial Port Controllers")
- Between memory and the UART Port
  (Chapter 13, "UART Port Controller")
- Between memory and the Parallel Peripheral Interface (PPI)
  (Chapter 11, "Parallel Peripheral Interface")

The system includes six DMA-capable peripherals, including the Memory DMA controller (MDMA). The following twelve DMA channels support these devices:

- PPI Receive/Transmit DMA Controller
- SPORT0 Receive DMA Controller

- SPORT0 Transmit DMA Controller
- SPORT1 Receive DMA Controller
- SPORT1 Transmit DMA Controller
- SPI Receive/Transmit DMA Controller
- UART Receive DMA Controller
- UART Transmit DMA Controller
- MDMA Stream 1 Transmit (Destination)
- MDMA Stream 1 Receive (Source)
- MDMA Stream 0 Transmit (Destination)
- MDMA Stream 0 Receive (Source)

This chapter describes the features common to all the DMA channels, as well as how DMA operations are set up. For specific peripheral features, see the appropriate peripheral chapter for additional information. Performance and bus arbitration for DMA operations can be found in "DAB, DCB, and DEB Performance" on page 7-8.

DMA transfers on the processor can be descriptor-based or register-based. Descriptor-based DMA transfers require a set of parameters stored within memory to initiate a DMA sequence. This sort of transfer allows the chaining together of multiple DMA sequences. In descriptor-based DMA operations, a DMA channel can be programmed to automatically set up and start another DMA transfer after the current sequence completes. Register-based DMA allows the processor to directly program DMA control registers to initiate a DMA transfer. On completion, the control registers may be automatically updated with their original setup values for continuous transfer, if needed.

DMA and Memory DMA Registers

For convenience, discussions in this chapter use generic (non-peripheral specific) DMA and Memory DMA register names.

- Generic DMA register names are listed in Table 9-1.
- Generic Memory DMA register names are listed in Table 9-3.

DMA registers fall into three categories:

- Parameter registers, such as `DMAx_CONFIG` and `DMAx_X_COUNT`

  Only Parameter registers can be loaded directly from descriptor elements; descriptor elements are listed in Table 9-2.

The letter *x* in DMA*x* represents a specific DMA-capable peripheral. For example, for DMA with default channel mapping, `DMA6_CONFIG` represents the `DMA_CONFIG` register for the UART *RX* peripheral. For default DMA channel mappings, see Table 9-16.

- Current registers, such as `DMAx_CURR_ADDR` and `DMAx_CURR_X_COUNT`
- Control/Status registers, such as `DMAx_IRQ_STATUS` and `DMAx_PERIPHERAL_MAP`

Table 9-1 lists the generic names of the DMA registers. For each register, the table also shows the MMR offset, a brief description of the register, the register category, and reset value.

Table 9-1. Generic Names of the DMA Memory-Mapped Registers

MMR Offset	Generic MMR Name	MMR Description	Register Category
0x00	NEXT_DESC_PTR	Link pointer to next descriptor	Parameter
0x04	START_ADDR	Start address of current buffer	Parameter
0x08	DMA_CONFIG	DMA Configuration register, including enable bit	Parameter
0x0C	Reserved	Reserved	
0x10	X_COUNT	Inner loop count	Parameter
0x14	X_MODIFY	Inner loop address increment, in bytes	Parameter
0x18	Y_COUNT	Outer loop count (2D only)	Parameter
0x1C	Y_MODIFY	Outer loop address increment, in bytes	Parameter
0x20	CURR_DESC_PTR	Current Descriptor Pointer	Current
0x24	CURR_ADDR	Current DMA Address	Current
0x28	IRQ_STATUS	Interrupt Status register: Contains Completion and DMA Error Interrupt status and channel state (Run/Fetch/Paused)	Control/ Status
0x2C	PERIPHERAL_MAP	Peripheral to DMA Channel Mapping: Contains a 4-bit value specifying the peripheral to associate with this DMA channel (Read-only for MDMA channels)	Control/ Status
0x30	CURR_X_COUNT	Current count (1D) or intra-row X count (2D); counts down from X_COUNT	Current
0x34	Reserved	Reserved	
0x38	CURR_Y_COUNT	Current row count (2D only); counts down from Y_COUNT	Current
0x3C	Reserved	Reserved	

All DMA registers can be accessed as 16-bit entities. However, the following registers may also be accessed as 32-bit registers:

- NEXT_DESC_PTR
- START_ADDR
- CURR_DESC_PTR
- CURR_ADDR

When these four registers are accessed as 16-bit entities, only the lower 16 bits can be accessed.

Naming Conventions for DMA MMRs

Because confusion might arise between descriptor element names and generic DMA register names, this chapter uses the naming conventions in Table 9-2, where:

- The left column lists the generic name of the MMR, which is used when discussing the general operation of the DMA engine.

Note the generic names in the left column are not actually mapped to resources in the processor.

- The middle column lists the specific MMR name. Only specific MMR names are mapped to processor resources.

  In DMA*x*, the letter *x* represents the number of the DMA channel. For instance, DMA3_IRQ_STATUS is the IRQ_STATUS MMR for DMA Channel #3.

The channel number can be assigned by default or can be programmed. For the DMA channel numbers and the default peripheral mapping, see Table 9-16.

- The last column lists the macro assigned to each descriptor element in memory.

  The macro name in the last column serves only to clarify the discussion of how the DMA engine operates.

Table 9-2. Naming Conventions: DMA MMRs and Descriptor Elements

Generic MMR Name	Specific MMR Name (x = DMA Channel Number)	Name of Corresponding Descriptor Element in Memory
DMA_CONFIG	DMAx_CONFIG	DMACFG
NEXT_DESC_PTR	DMAx_NEXT_DESC_PTR	NDPH (upper 16 bits), NDPL (lower 16 bits)
START_ADDR	DMAx_START_ADDR	SAH (upper 16 bits), SAL (lower 16 bits)
X_COUNT	DMAx_X_COUNT	XCNT
Y_COUNT	DMAx_Y_COUNT	YCNT
X_MODIFY	DMAx_X_MODIFY	XMOD
Y_MODIFY	DMAx_Y_MODIFY	YMOD
CURR_DESC_PTR	DMAx_CURR_DESC_PTR	N/A
CURR_ADDR	DMAx_CURR_ADDR	N/A
CURR_X_COUNT	DMAx_CURR_X_COUNT	N/A
CURR_Y_COUNT	DMAx_CURR_Y_COUNT	N/A
IRQ_STATUS	DMAx_IRQ_STATUS	N/A
PERIPHERAL_MAP	DMAx_PERIPHERAL_MAP	N/A

Naming Conventions for Memory DMA Registers

The names of Memory DMA registers differ somewhat from the names of other DMA registers. Memory DMA streams cannot be reassigned to different channels, whereas the peripherals associated with DMA can be mapped to any DMA channel between 0 and 7.

Table 9-3 shows the naming conventions for Memory DMA registers. In each name, the letters *yy* have four possible values:

- S0, Memory DMA Source Stream 0
- D0, Memory DMA Destination Stream 0
- S1, Memory DMA Source Stream 1
- D1, Memory DMA Destination Stream 1

Table 9-3. Naming Conventions for Memory DMA Registers

Generic MMR Name	Memory DMA MMR Name (yy = S0, S1, D0, or D1)	Name of Corresponding Descriptor Element in Memory
DMA_CONFIG	MDMA_yy_CONFIG	DMACFG
NEXT_DESC_PTR	MDMA_yy_NEXT_DESC_PTR	NDPH (upper 16 bits), NDPL (lower 16 bits)
START_ADDR	MDMA_yy_START_ADDR	SAH (upper 16 bits), SAL (lower 16 bits)
X_COUNT	MDMA_yy_X_COUNT	XCNT
Y_COUNT	MDMA_yy_Y_COUNT	YCNT
X_MODIFY	MDMA_yy_X_MODIFY	XMOD
Y_MODIFY	MDMA_yy_Y_MODIFY	YMOD
CURR_DESC_PTR	MDMA_yy_CURR_DESC_PTR	N/A
CURR_ADDR	MDMA_yy_CURR_ADDR	N/A

Table 9-3. Naming Conventions for Memory DMA Registers (Cont'd)

Generic MMR Name	Memory DMA MMR Name (yy = S0, S1, D0, or D1)	Name of Corresponding Descriptor Element in Memory
CURR_X_COUNT	MDMA_yy_CURR_X_COUNT	N/A
CURR_Y_COUNT	MDMA_yy_CURR_Y_COUNT	N/A
IRQ_STATUS	MDMA_yy_IRQ_STATUS	N/A
PERIPHERAL_MAP	MDMA_yy_PERIPHERAL_MAP	N/A

DMAx_NEXT_DESC_PTR/MDMA_yy_NEXT_DESC_PTR Register

The Next Descriptor Pointer register (DMAx_NEXT_DESC_PTR/MDMA_yy_NEXT_DESC_PTR) specifies where to look for the start of the next descriptor block when the DMA activity specified by the current descriptor block finishes. This register is used in small and large descriptor list modes. At the start of a descriptor fetch in either of these modes, the 32-bit NEXT_DESC_PTR register is copied into the CURR_DESC_PTR register. Then, during the descriptor fetch, the CURR_DESC_PTR register increments after each element of the descriptor is read in.

In small and large descriptor list modes, the NEXT_DESC_PTR register, and not the CURR_DESC_PTR register, must be programmed directly via MMR access before starting DMA operation.

In Descriptor Array mode, the Next Descriptor Pointer register is disregarded, and fetching is controlled only by the CURR_DESC_PTR register.

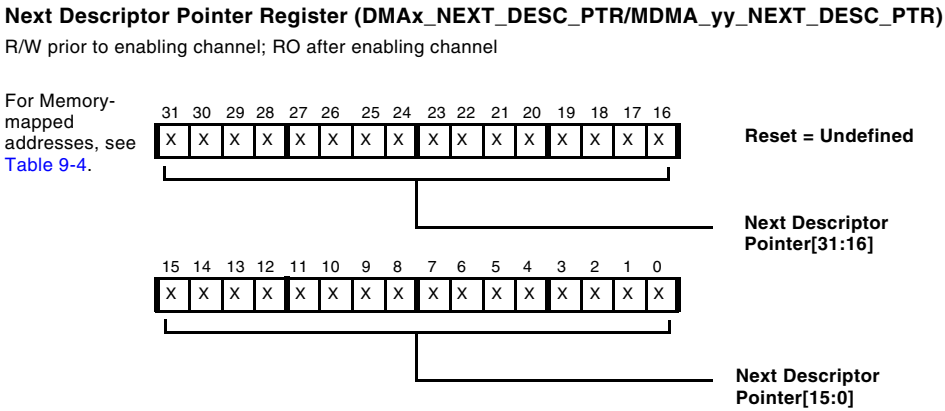


Figure 9-1. Next Descriptor Pointer Register

Table 9-4. Next Descriptor Pointer Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_NEXT_DESC_PTR	0xFFC0 0C00
DMA1_NEXT_DESC_PTR	0xFFC0 0C40
DMA2_NEXT_DESC_PTR	0xFFC0 0C80
DMA3_NEXT_DESC_PTR	0xFFC0 0CC0
DMA4_NEXT_DESC_PTR	0xFFC0 0D00
DMA5_NEXT_DESC_PTR	0xFFC0 0D40
DMA6_NEXT_DESC_PTR	0xFFC0 0D80
DMA7_NEXT_DESC_PTR	0xFFC0 0DC0
MDMA_D0_NEXT_DESC_PTR	0xFFC0 0E00
MDMA_S0_NEXT_DESC_PTR	0xFFC0 0E40
MDMA_D1_NEXT_DESC_PTR	0xFFC0 0E80
MDMA_S1_NEXT_DESC_PTR	0xFFC0 0EC0

DMAx_START_ADDR/MDMA_yy_START_ADDR Register

The Start Address register (DMAx_START_ADDR/MDMA_yy_START_ADDR), shown in Figure 9-2, contains the start address of the data buffer currently targeted for DMA.

Start Address Register (DMAx_START_ADDR/ MDMA_yy_START_ADDR)

R/W prior to enabling channel; RO after enabling channel

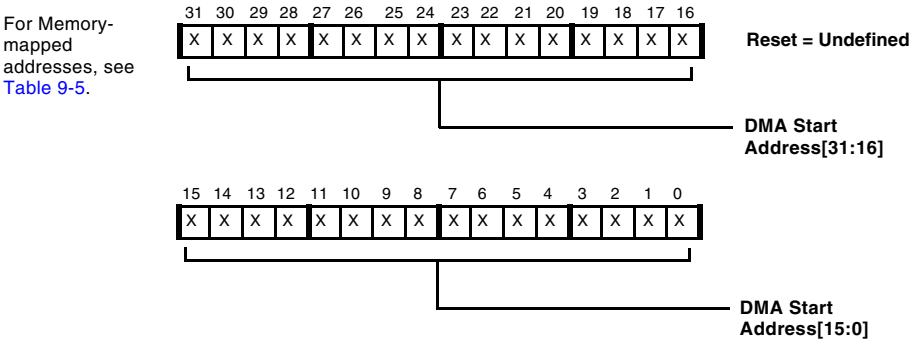


Figure 9-2. Start Address Register

Table 9-5. Start Address Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_START_ADDR	0xFFC0 0C04
DMA1_START_ADDR	0xFFC0 0C44
DMA2_START_ADDR	0xFFC0 0C84
DMA3_START_ADDR	0xFFC0 0CC4
DMA4_START_ADDR	0xFFC0 0D04
DMA5_START_ADDR	0xFFC0 0D44
DMA6_START_ADDR	0xFFC0 0D84
DMA7_START_ADDR	0xFFC0 0DC4
MDMA_D0_START_ADDR	0xFFC0 0E04
MDMA_S0_START_ADDR	0xFFC0 0E44
MDMA_D1_START_ADDR	0xFFC0 0E84
MDMA_S1_START_ADDR	0xFFC0 0EC4

DMAx_CONFIG/MDMA_yy_CONFIG Register

The DMA Configuration register (DMAx_CONFIG/MDMA_yy_CONFIG), shown in Figure 9-3, is used to set up DMA parameters and operating modes. Note that writing the DMA_CONFIG register while DMA is already running will cause a DMA error unless writing with the DMAEN bit set to 0.

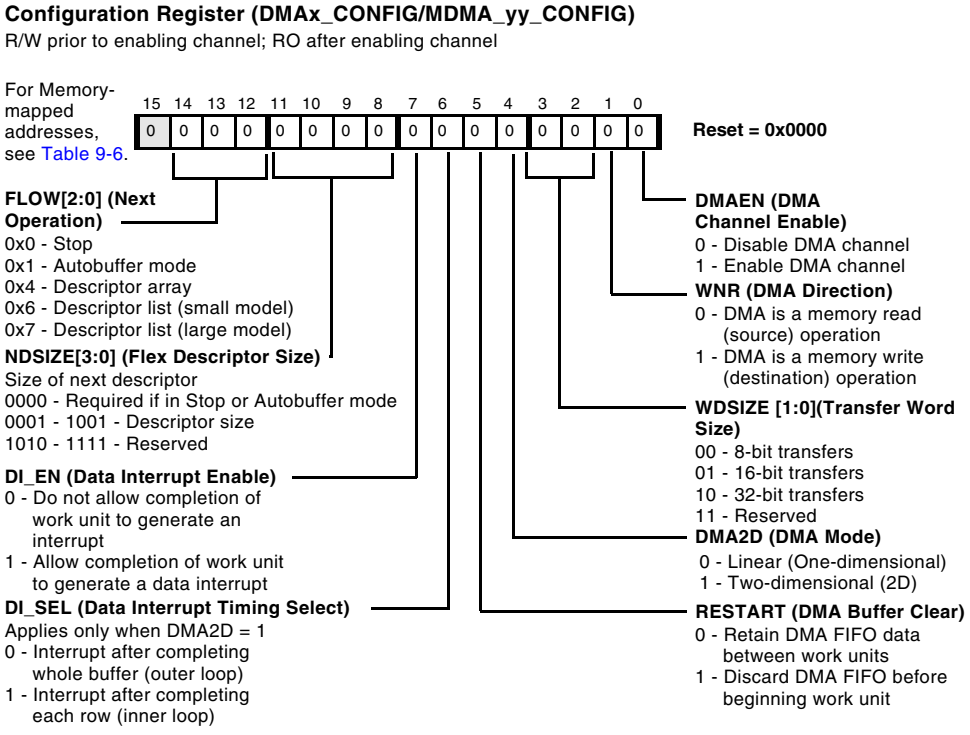


Figure 9-3. Configuration Register

Table 9-6. Configuration Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CONFIG	0xFFC0 0C08
DMA1_CONFIG	0xFFC0 0C48
DMA2_CONFIG	0xFFC0 0C88
DMA3_CONFIG	0xFFC0 0CC8
DMA4_CONFIG	0xFFC0 0D08
DMA5_CONFIG	0xFFC0 0D48
DMA6_CONFIG	0xFFC0 0D88
DMA7_CONFIG	0xFFC0 0DC8
MDMA_D0_CONFIG	0xFFC0 0E08
MDMA_S0_CONFIG	0xFFC0 0E48
MDMA_D1_CONFIG	0xFFC0 0E88
MDMA_S1_CONFIG	0xFFC0 0EC8

The fields of the DMAx_CONFIG register are used to set up DMA parameters and operating modes.

- FLOW[2:0] (Next Operation). This field specifies the type of DMA transfer to follow the present one. The flow options are:

  0x0 - Stop. When the current work unit completes, the DMA channel stops automatically, after signaling an interrupt (if selected). The DMA_RUN status bit in the DMAx_IRQ_STATUS register changes from 1 to 0, while the DMAEN bit in the DMAx_CONFIG register is unchanged. In this state, the channel is paused. Peripheral interrupts are still filtered out by the DMA unit. The channel may be restarted simply by another write to the DMAx_CONFIG register specifying the next work unit, in which the DMAEN bit is set to 1.

0x1 - Autobuffer Mode. In this mode, no descriptors in memory are used. Instead, DMA is performed in a continuous circular buffer fashion based on user-programmed DMAx MMR settings. Upon completion of the work unit, the Parameter registers are reloaded into the Current registers, and DMA resumes immediately with zero overhead. Autobuffer mode is stopped by a user write of 0 to the DMAEN bit in the DMAx_CONFIG register.

0x4 - Descriptor Array Mode. This mode fetches a descriptor from memory that does not include the NDPH or NDPL elements. Because the descriptor does not contain a Next Descriptor Pointer entry, the DMA engine defaults to using the CURR_DESC_PTR register to step through descriptors, thus allowing a group of descriptors to follow one another in memory like an array.

0x6 - Descriptor List (Small Model) Mode. This mode fetches a descriptor from memory that includes NDPL, but not NDPH. Therefore, the high 16 bits of the Next Descriptor Pointer field are taken from the upper 16 bits of the NEXT_DESC_PTR register, thus confining all descriptors to a specific 64K page in memory.

0x7 - Descriptor List (Large Model) Mode. This mode fetches a descriptor from memory that includes NDPH and NDPL, thus allowing maximum flexibility in locating descriptors in memory.

- NDSIZE[3:0] (Flex Descriptor Size). This field specifies the number of descriptor elements in memory to load. This field must be 0 if in Stop or Autobuffer mode. If NDSIZE and FLOW specify a descriptor that extends beyond YMOD, a DMA error results.
- DI_EN (Data Interrupt Enable). This bit specifies whether to allow completion of a work unit to generate a data interrupt.

- DI_SEL (Data Interrupt Timing Select). This bit specifies the timing of a data interrupt—after completing the whole buffer or after completing each row of the inner loop. This bit is used only in 2D DMA operation.

- RESTART (DMA Buffer Clear). This bit specifies whether receive data held in the channel's data FIFO should be preserved (RESTART = 0) or discarded (RESTART = 1) before beginning the next work unit. Receive data is automatically discarded when the DMAEN bit changes from 0 to 1, typically when a channel is first enabled. Received FIFO data should usually be retained between work units if the work units make up a continuous datastream. If, however, a new work unit starts a new datastream, the RESTART bit should be set to 1 to clear out any previously received data.

The RESTART bit applies only to memory write DMA channels. It is reserved in the cases of memory read DMA channels and MDMA channels, and must be 0 in those cases.

In memory write DMA channels, the RESTART bit only affects the first work unit initiated by a write to the DMAx_CONFIG register. The RESTART bit has no effect if it is set in DMACFG elements of DMA descriptors.

- DMA2D (DMA Mode). This bit specifies whether DMA mode involves only X_COUNT and X_MODIFY (one-dimensional DMA) or also involves Y_COUNT and Y_MODIFY (two-dimensional DMA).

- WDSIZE[1:0] (Transfer Word Size). The DMA engine supports transfers of 8-, 16-, or 32-bit items. Each request/grant results in a single memory access (although two cycles are required to transfer 32-bit data through a 16-bit memory port or through the 16-bit

DMA Access bus). The DMA Address Pointer registers' increment sizes (strides) must be a multiple of the transfer unit size—1 for 8-bit, 2 for 16-bit, 4 for 32-bit.

- WNR (DMA Direction). This bit specifies DMA direction—memory read (0) or memory write (1).
- DMAEN (DMA Channel Enable). This bit specifies whether to enable a given DMA channel.

When a peripheral DMA channel is enabled, interrupts from the peripheral denote DMA requests. When a channel is disabled, the DMA unit ignores the peripheral interrupt and passes it directly to the interrupt controller. To avoid unexpected results, take care to enable the DMA channel before enabling the peripheral, and to disable the peripheral before disabling the DMA channel.

DMAx_X_COUNT/MDMA_yy_X_COUNT Register

For 2D DMA, the Inner Loop Count register (DMAx_X_COUNT/MDMA_yy_X_COUNT), shown in Figure 9-4, contains the inner loop count. For 1D DMA, it specifies the number of elements to read in. For details, see "Two-Dimensional DMA" on page 9-45. A value of 0 in X_COUNT corresponds to 65,536 elements.

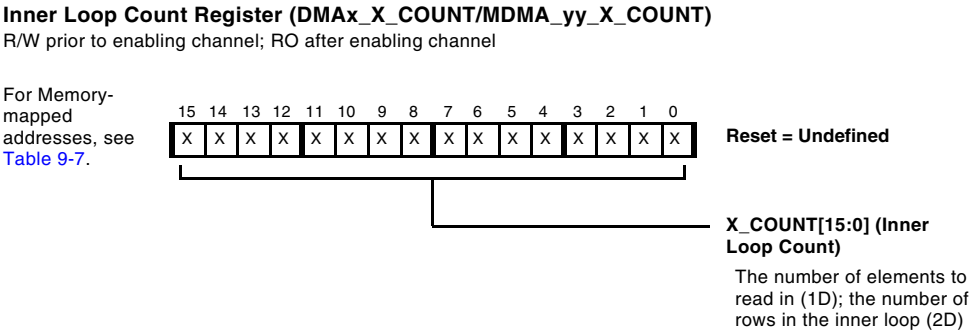


Figure 9-4. Inner Loop Count Register

Table 9-7. Inner Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_X_COUNT	0xFFC0 0C10
DMA1_X_COUNT	0xFFC0 0C50
DMA2_X_COUNT	0xFFC0 0C90
DMA3_X_COUNT	0xFFC0 0CD0
DMA4_X_COUNT	0xFFC0 0D10
DMA5_X_COUNT	0xFFC0 0D50
DMA6_X_COUNT	0xFFC0 0D90
DMA7_X_COUNT	0xFFC0 0DD0
MDMA_D0_X_COUNT	0xFFC0 0E10
MDMA_S0_X_COUNT	0xFFC0 0E50
MDMA_D1_X_COUNT	0xFFC0 0E90
MDMA_S1_X_COUNT	0xFFC0 0ED0

DMAx_X_MODIFY/MDMA_yy_X_MODIFY Register

The Inner Loop Address Increment register (`DMAx_X_MODIFY/MDMA_yy_X_MODIFY`) contains a signed, two's-complement byte-address increment. In 1D DMA, this increment is the stride that is applied after transferring each element.

Note `X_MODIFY` is specified in bytes, regardless of the DMA transfer size.

In 2D DMA, this increment is applied after transferring each element in the inner loop, up to but not including the last element in each inner loop. After the last element in each inner loop, the `Y_MODIFY` register is applied instead, except on the very last transfer of each work unit. The `X_MODIFY` register is always applied on the last transfer of a work unit.

The X_MODIFY field may be set to 0. In this case, DMA is performed repeatedly to or from the same address. This is useful, for example, in transferring data between a data register and an external memory-mapped peripheral.

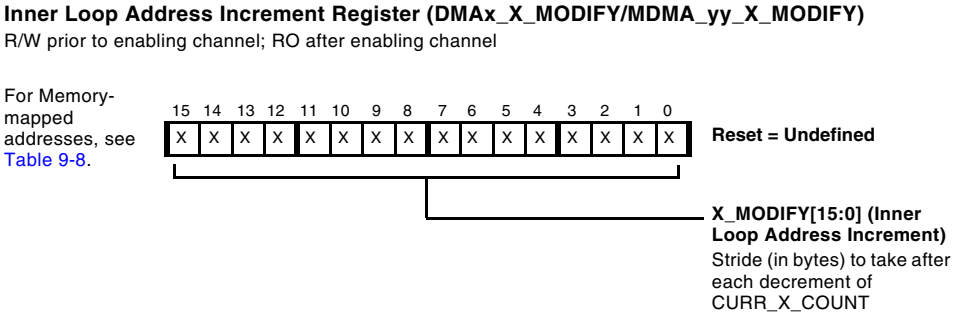


Figure 9-5. Inner Loop Address Increment Register

Table 9-8. Inner Loop Address Increment Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_X_MODIFY	0xFFC0 0C14
DMA1_X_MODIFY	0xFFC0 0C54
DMA2_X_MODIFY	0xFFC0 0C94
DMA3_X_MODIFY	0xFFC0 0CD4
DMA4_X_MODIFY	0xFFC0 0D14
DMA5_X_MODIFY	0xFFC0 0D54
DMA6_X_MODIFY	0xFFC0 0D94
DMA7_X_MODIFY	0xFFC0 0DD4
MDMA_D0_X_MODIFY	0xFFC0 0E14
MDMA_S0_X_MODIFY	0xFFC0 0E54

Table 9-8. Inner Loop Address Increment Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
MDMA_D1_X_MODIFY	0xFFC0 0E94
MDMA_S1_X_MODIFY	0xFFC0 0ED4

DMAx_Y_COUNT/MDMA_yy_Y_COUNT Register

For 2D DMA, the Outer Loop Count register (DMAx_Y_COUNT/MDMA_yy_Y_COUNT) contains the outer loop count. It is not used in 1D DMA mode. This register contains the number of rows in the outer loop of a 2D DMA sequence. For details, see "Two-Dimensional DMA" on page 9-45.

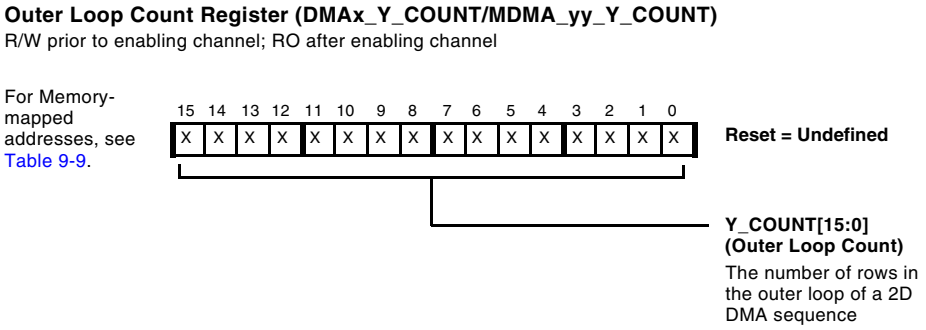


Figure 9-6. Outer Loop Count Register

Table 9-9. Outer Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_Y_COUNT	0xFFC0 0C18
DMA1_Y_COUNT	0xFFC0 0C58
DMA2_Y_COUNT	0xFFC0 0C98

Table 9-9. Outer Loop Count Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DMA3_Y_COUNT	0xFFC0 0CD8
DMA4_Y_COUNT	0xFFC0 0D18
DMA5_Y_COUNT	0xFFC0 0D58
DMA6_Y_COUNT	0xFFC0 0D98
DMA7_Y_COUNT	0xFFC0 0DD8
MDMA_D0_Y_COUNT	0xFFC0 0E18
MDMA_S0_Y_COUNT	0xFFC0 0E58
MDMA_D1_Y_COUNT	0xFFC0 0E98
MDMA_S1_Y_COUNT	0xFFC0 0ED8

DMAx_Y_MODIFY/MDMA_yy_Y_MODIFY Register

The Outer Loop Address Increment register (`DMAx_Y_MODIFY/MDMA_yy_Y_MODIFY`) contains a signed, two's-complement value. This byte-address increment is applied after each decrement of the `CURR_Y_COUNT` register except for the last item in the 2D array where the `CURR_Y_COUNT` also expires. The value is the offset between the last word of one "row" and the first word of the next "row." For details, see "Two-Dimensional DMA" on page 9-45.

Note `Y_MODIFY` is specified in bytes, regardless of the DMA transfer size.

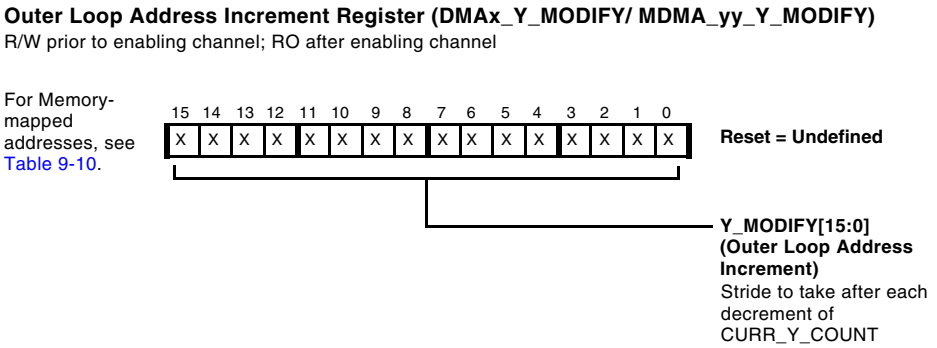


Figure 9-7. Outer Loop Address Increment Register

Table 9-10. Outer Loop Address Increment Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_Y_MODIFY	0xFFC0 0C1C
DMA1_Y_MODIFY	0xFFC0 0C5C
DMA2_Y_MODIFY	0xFFC0 0C9C
DMA3_Y_MODIFY	0xFFC0 0CDC
DMA4_Y_MODIFY	0xFFC0 0D1C
DMA5_Y_MODIFY	0xFFC0 0D5C
DMA6_Y_MODIFY	0xFFC0 0D9C
DMA7_Y_MODIFY	0xFFC0 0DDC
MDMA_D0_Y_MODIFY	0xFFC0 0E1C
MDMA_S0_Y_MODIFY	0xFFC0 0E5C
MDMA_D1_Y_MODIFY	0xFFC0 0E9C
MDMA_S1_Y_MODIFY	0xFFC0 0EDC

DMAx_CURR_DESC_PTR/MDMA_yy_CURR_DESC_PTR Register

The Current Descriptor Pointer register (DMAx_CURR_DESC_PTR/MDMA_yy_CURR_DESC_PTR) contains the memory address for the next descriptor element to be loaded. For FLOW mode settings that involve descriptors (FLOW = 4, 6, or 7), this register is used to read descriptor elements into appropriate MMRs before a DMA work block begins. For Descriptor List modes (FLOW = 6 or 7), this register is initialized from the NEXT_DESC_PTR register before loading each descriptor. Then, the address in the CURR_DESC_PTR register increments as each descriptor element is read in.

When the entire descriptor has been read, the CURR_DESC_PTR register contains this value:

```
Descriptor Start Address + Descriptor Size (# of elements)
```

For Descriptor Array mode (FLOW = 4), this register, and not the NEXT_DESC_PTR register, must be programmed by MMR access before starting DMA operation.

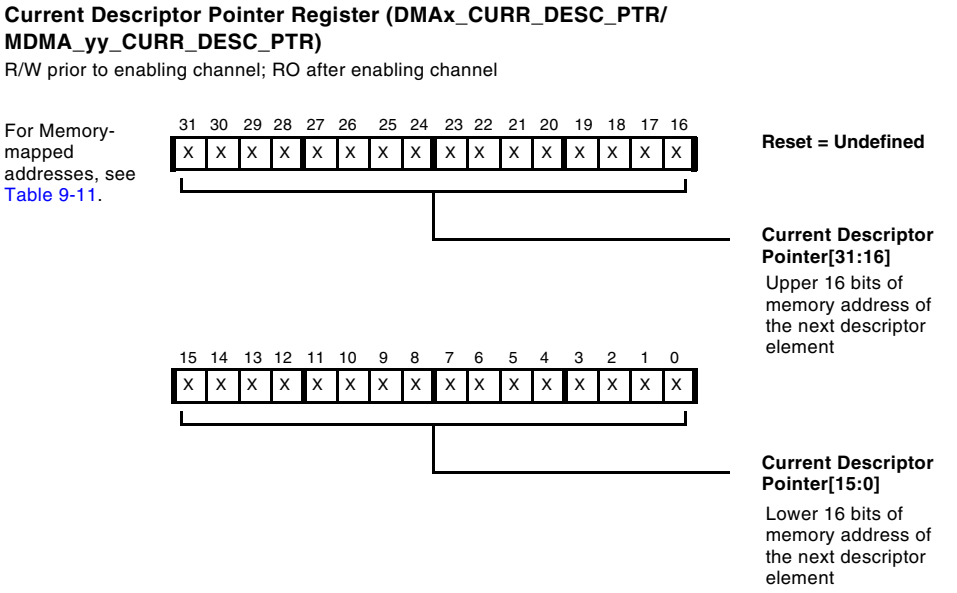


Figure 9-8. Current Descriptor Pointer Register

Table 9-11. Current Descriptor Pointer Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CURR_DESC_PTR	0xFFC0 0C20
DMA1_CURR_DESC_PTR	0xFFC0 0C60
DMA2_CURR_DESC_PTR	0xFFC0 0CA0
DMA3_CURR_DESC_PTR	0xFFC0 0CE0
DMA4_CURR_DESC_PTR	0xFFC0 0D20
DMA5_CURR_DESC_PTR	0xFFC0 0D60
DMA6_CURR_DESC_PTR	0xFFC0 0DA0
DMA7_CURR_DESC_PTR	0xFFC0 0DE0
MDMA_D0_CURR_DESC_PTR	0xFFC0 0E20

Table 9-11. Current Descriptor Pointer Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
MDMA_S0_CURR_DESC_PTR	0xFFC0 0E60
MDMA_D1_CURR_DESC_PTR	0xFFC0 0EA0
MDMA_S1_CURR_DESC_PTR	0xFFC0 0EE0

DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR Register

The Current Address register (DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR), shown in Figure 9-9, contains the present DMA transfer address for a given DMA session. At the start of a DMA session, the CURR_ADDR register is loaded from the START_ADDR register, and it is incremented as each transfer occurs. The Current Address register contains 32 bits.

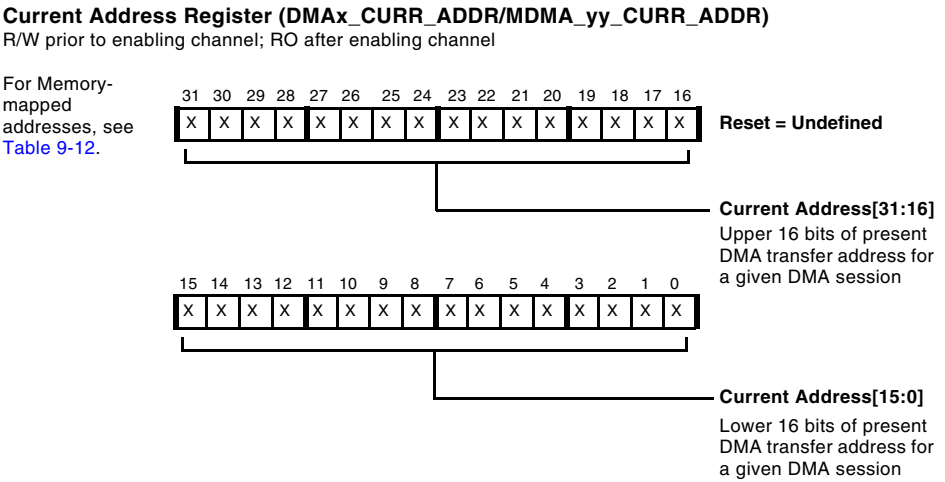


Figure 9-9. Current Address Register

Table 9-12. Current Address Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CURR_ADDR	0xFFC0 0C24
DMA1_CURR_ADDR	0xFFC0 0C64
DMA2_CURR_ADDR	0xFFC0 0CA4
DMA3_CURR_ADDR	0xFFC0 0CE4
DMA4_CURR_ADDR	0xFFC0 0D24
DMA5_CURR_ADDR	0xFFC0 0D64
DMA6_CURR_ADDR	0xFFC0 0DA4
DMA7_CURR_ADDR	0xFFC0 0DE4
MDMA_D0_CURR_ADDR	0xFFC0 0E24
MDMA_S0_CURR_ADDR	0xFFC0 0E64
MDMA_D1_CURR_ADDR	0xFFC0 0EA4
MDMA_S1_CURR_ADDR	0xFFC0 0EE4

DMAx_CURR_X_COUNT/MDMA_yy_CURR_X_COUNT Register

The Current Inner Loop Count register (DMAx_CURR_X_COUNT/MDMA_yy_CURR_X_COUNT) is loaded by the X_COUNT register at the beginning of each DMA session (for 1D DMA) and also after the end of DMA for each row (for 2D DMA). Otherwise it is decremented each time an element is transferred. Expiration of the count in this register signifies that DMA is complete. In 2D DMA, the CURR_X_COUNT register value is 0 only when the entire transfer is complete. Between rows it is equal to the value of the X_COUNT register.

Current Inner Loop Count Register (DMAx_CURR_X_COUNT/ MDMA_yy_CURR_X_COUNT)

R/W prior to enabling channel; RO after enabling channel

For Memory-mapped addresses, see Table 9-13.

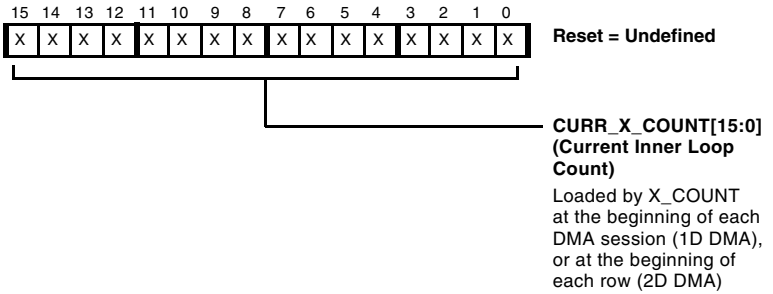


Figure 9-10. Current Inner Loop Count Register

Table 9-13. Current Inner Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CURR_X_COUNT	0xFFC0 0C30
DMA1_CURR_X_COUNT	0xFFC0 0C70
DMA2_CURR_X_COUNT	0xFFC0 0CB0
DMA3_CURR_X_COUNT	0xFFC0 0CF0
DMA4_CURR_X_COUNT	0xFFC0 0D30
DMA5_CURR_X_COUNT	0xFFC0 0D70
DMA6_CURR_X_COUNT	0xFFC0 0DB0
DMA7_CURR_X_COUNT	0xFFC0 0DF0
MDMA_D0_CURR_X_COUNT	0xFFC0 0E30
MDMA_S0_CURR_X_COUNT	0xFFC0 0E70
MDMA_D1_CURR_X_COUNT	0xFFC0 0EB0
MDMA_S1_CURR_X_COUNT	0xFFC0 0EF0

DMAx_CURR_Y_COUNT/MDMA_yy_CURR_Y_COUNT Register

The Current Outer Loop Count register (DMAx_CURR_Y_COUNT/MDMA_yy_CURR_Y_COUNT) is loaded by the Y_COUNT register at the beginning of each 2D DMA session. It is not used for 1D DMA. This register is decremented each time the CURR_X_COUNT register expires during 2D DMA operation (1 to X_COUNT or 1 to 0 transition), signifying completion of an entire row transfer. After a 2D DMA session is complete, CURR_Y_COUNT = 1 and CURR_X_COUNT = 0.

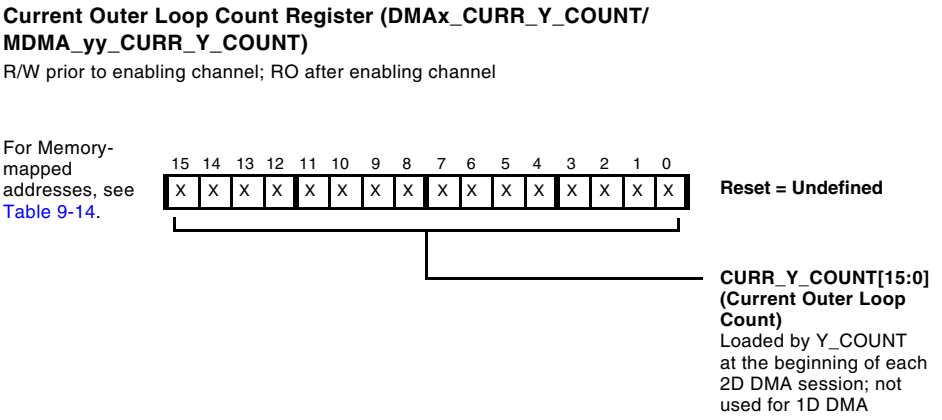


Figure 9-11. Current Outer Loop Count Register

Table 9-14. Current Outer Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CURR_Y_COUNT	0xFFC0 0C38
DMA1_CURR_Y_COUNT	0xFFC0 0C78
DMA2_CURR_Y_COUNT	0xFFC0 0CB8
DMA3_CURR_Y_COUNT	0xFFC0 0CF8
DMA4_CURR_Y_COUNT	0xFFC0 0D38
DMA5_CURR_Y_COUNT	0xFFC0 0D78
DMA6_CURR_Y_COUNT	0xFFC0 0DB8
DMA7_CURR_Y_COUNT	0xFFC0 0DF8
MDMA_D0_CURR_Y_COUNT	0xFFC0 0E38
MDMA_S0_CURR_Y_COUNT	0xFFC0 0E78
MDMA_D1_CURR_Y_COUNT	0xFFC0 0EB8
MDMA_S1_CURR_Y_COUNT	0xFFC0 0EF8

DMAx_PERIPHERAL_MAP/MDMA_yy_PERIPHERAL_MAP Register

Each DMA channel's Peripheral Map register (`DMAx_PERIPHERAL_MAP/MDMA_yy_PERIPHERAL_MAP`) contains bits that:

- Map the channel to a specific peripheral.
- Identify whether the channel is a Peripheral DMA channel or a Memory DMA channel.

Note a 1:1 mapping should exist between DMA channels and peripherals. The user is responsible for ensuring that multiple DMA channels are not mapped to the same peripheral and that multiple peripherals are not mapped to the same DMA port. If

multiple channels are mapped to the same peripheral, only one channel is connected (the lowest priority channel). If a nonexistent peripheral (for example, 0xF in the PMAP field) is mapped to a channel, that channel is disabled—DMA requests are ignored, and no DMA grants are issued. The DMA requests are also not forwarded from the peripheral to the interrupt controller.

Follow these steps to swap the DMA channel priorities of two channels. Assume that channels 6 and 7 are involved.

1. Make sure DMA is disabled on channels 6 and 7.
2. Write DMA6_PERIPHERAL_MAP with 0x7000 and DMA7_PERIPHERAL_MAP with 0x6000.
3. Enable DMA on channels 6 and/or 7.

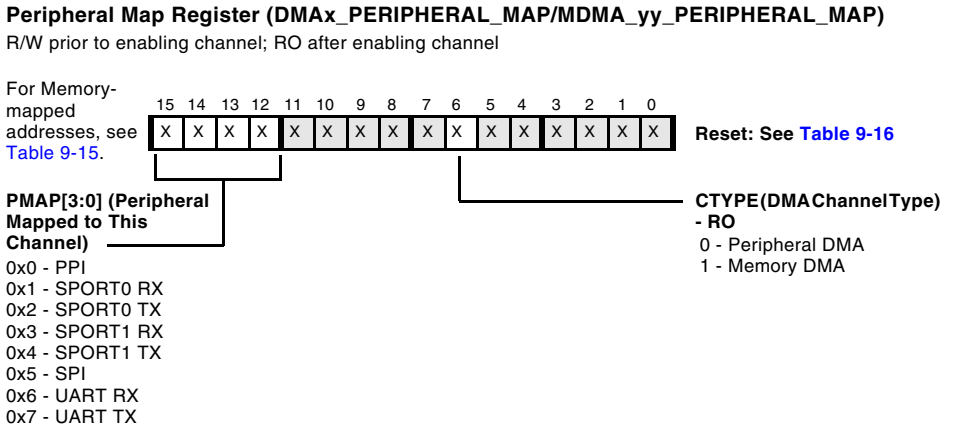


Figure 9-12. Peripheral Map Register

Table 9-15. Peripheral Map Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_PERIPHERAL_MAP	0xFFC0 0C2C
DMA1_PERIPHERAL_MAP	0xFFC0 0C6C
DMA2_PERIPHERAL_MAP	0xFFC0 0CAC
DMA3_PERIPHERAL_MAP	0xFFC0 0CEC
DMA4_PERIPHERAL_MAP	0xFFC0 0D2C
DMA5_PERIPHERAL_MAP	0xFFC0 0D6C
DMA6_PERIPHERAL_MAP	0xFFC0 0DAC
DMA7_PERIPHERAL_MAP	0xFFC0 0DEC
MDMA_D0_PERIPHERAL_MAP	0xFFC0 0E2C
MDMA_S0_PERIPHERAL_MAP	0xFFC0 0E6C
MDMA_D1_PERIPHERAL_MAP	0xFFC0 0EAC
MDMA_S1_PERIPHERAL_MAP	0xFFC0 0EEC

Table 9-16 lists the binary peripheral map settings for each DMA-capable peripheral.

Table 9-16. Peripheral Mapping

DMA Channel	Default Peripheral Mapping	Default PERIPHERAL_MAP Setting (Binary)	Comments
0 (highest priority)	PPI	b#0000 0000 0000 0000	
1	SPORT0 RX	b#0001 0000 0000 0000	
2	SPORT0 TX	b#0010 0000 0000 0000	
3	SPORT1 RX	b#0011 0000 0000 0000	
4	SPORT1 TX	b#0100 0000 0000 0000	

Table 9-16. Peripheral Mapping (Cont'd)

DMA Channel	Default Peripheral Mapping	Default PERIPHERAL_MAP Setting (Binary)	Comments
5	SPI	b#0101 0000 0000 0000	
6	UART RX	b#0110 0000 0000 0000	
7	UART TX	b#0111 0000 0000 0000	
8	Mem DMA Stream 0 Destination	b#0000 0000 0100 0000	Not reassignable
9	Mem DMA Stream 0 Source	b#0000 0000 0100 0000	Not reassignable
10	Mem DMA Stream 1 Destination	b#0000 0000 0100 0000	Not reassignable
11 (lowest priority)	Mem DMA Stream 1 Source	b#0000 0000 0100 0000	Not reassignable

DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS Register

The Interrupt Status register (`DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS`), shown in Figure 9-13, contains bits that record whether the DMA channel:

- Is enabled and operating, enabled but stopped, or disabled.
- Is fetching data or a DMA descriptor.
- Has detected that a global DMA interrupt or a channel interrupt is being asserted.
- Has logged occurrence of a DMA error.

Note the `DMA_DONE` interrupt is asserted when the last memory access (read or write) has completed.

For a memory transfer to a peripheral, there may be up to four data words in the channel's DMA FIFO when the interrupt occurs. At

this point, it is normal to immediately start the next work unit. If, however, the application needs to know when the final data item is actually transferred to the peripheral, the application can test or poll the DMA_RUN bit. As long as there is undelivered transmit data in the FIFO, the DMA_RUN bit is 1.

For a memory write DMA channel, the state of the DMA_RUN bit has no meaning after the last DMA_DONE event has been signaled. It does not indicate the status of the DMA FIFO.

For MemDMA transfers where it is not desired to use an interrupt to notify when the DMA operation has ended, software should poll the DMA_DONE bit, and not the DMA_RUN bit, to determine when the transaction has completed.

Table 9-17. Interrupt Status Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_IRQ_STATUS	0xFFC0 0C28
DMA1_IRQ_STATUS	0xFFC0 0C68
DMA2_IRQ_STATUS	0xFFC0 0CA8
DMA3_IRQ_STATUS	0xFFC0 0CE8
DMA4_IRQ_STATUS	0xFFC0 0D28
DMA5_IRQ_STATUS	0xFFC0 0D68
DMA6_IRQ_STATUS	0xFFC0 0DA8
DMA7_IRQ_STATUS	0xFFC0 0DE8
MDMA_D0_IRQ_STATUS	0xFFC0 0E28
MDMA_S0_IRQ_STATUS	0xFFC0 0E68
MDMA_D1_IRQ_STATUS	0xFFC0 0EA8
MDMA_S1_IRQ_STATUS	0xFFC0 0EE8

Interrupt Status Register (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS)

For Memory-mapped addresses, see Table 9-17.

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

Reset = 0x0000

DMA_RUN (DMA Channel Running) - RO
This bit is set to 1 automatically when the DMA_CONFIG register is written
0 - This DMA channel is disabled, or it is enabled but paused (FLOW mode 0)
1 - This DMA channel is enabled and operating, either transferring data or fetching a DMA descriptor

DFETCH (DMA Descriptor Fetch) - RO
This bit is set to 1 automatically when the DMA_CONFIG register is written with FLOW modes 4–7
0 - This DMA channel is disabled, or it is enabled but stopped (FLOW mode 0)
1 - This DMA channel is enabled and presently fetching a DMA descriptor

DMA_DONE (DMA Completion Interrupt Status) - W1C
0 - No interrupt is being asserted for this channel
1 - DMA work unit has completed, and this DMA channel's interrupt is being asserted

DMA_ERR (DMA Error Interrupt Status) - W1C
0 - No DMA error has occurred
1 - A DMA error has occurred, and the global DMA Error interrupt is being asserted. After this error occurs, the contents of the DMA Current registers are unspecified. Control/Status and Parameter registers are unchanged.

Figure 9-13. Interrupt Status Register

The processor supports a flexible interrupt control structure with three interrupt sources.

- Data driven interrupts (see Table 9-18)
- Peripheral Error interrupts
- DMA Error interrupts (for example, Bad Descriptor or Bus Error)

Separate Interrupt Request (IRQ) levels are allocated for Data and Peripheral Error interrupts, and DMA Error interrupts.

Table 9-18. Data Driven Interrupts

Interrupt Name	Description
No Interrupt	Interrupts can be disabled for a given work unit.
Peripheral Interrupt	These are peripheral (non-DMA) interrupts.
Row Completion	DMA Interrupts can occur on the completion of a row (CURR_X_COUNT expiration).
Buffer Completion	DMA Interrupts can occur on the completion of an entire buffer (when CURR_X_COUNT and CURR_Y_COUNT expire).

All DMA channels are OR'ed together into one system-level DMA Error interrupt. The individual `IRQ_STATUS` words of each channel can be read to identify the channel that caused the DMA Error interrupt.

Note the `DMA_DONE` and `DMA_ERR` interrupt indicators are write-one-to-clear (W1C).

When switching a peripheral from DMA to non-DMA mode, the peripheral's interrupts should be disabled during the mode switch (via the appropriate peripheral registers or `SIC_IMASK`) so that no unintended interrupt is generated on the shared DMA/interrupt request line.

Flex Descriptor Structure

DMA flex descriptors are variable sized data structures whose contents are loaded into DMA Parameter registers. The sequence of registers in the descriptor is essentially fixed (among three similar variations), but the length of the descriptor is completely programmable. The DMA channel registers are ordered so that the registers that are most commonly reloaded per work unit are at the lowest MMR addresses. The user may choose whether or not to use descriptors. If not using descriptors, the user can write the DMA MMRs directly to start DMA, and use either Autobuffer mode for continuous operation or Stop mode for single-buffer operation.

To use descriptors, the user programs the NDSIZE field of the DMAx_CONFIG register with the number of DMA registers to load from the descriptor, starting with the lowest MMR address. The user may select a descriptor size from one entry (the lower 16 bits of START_ADDR) to nine entries (all the DMA parameters).

The three variations of the descriptor value sequences depend on whether a Next Descriptor Pointer is included and, if so, what kind.

- None included (Descriptor Array mode)
- The lower 16 bits of the Next Descriptor Pointer (Descriptor List, Small Model)
- All 32 bits of the Next Descriptor Pointer (Descriptor List, Large Model)

All the other registers not loaded from the descriptor retain their prior values, although the CURR_ADDR, CURR_X_COUNT, and CURR_Y_COUNT registers are reloaded between the descriptor fetch and the start of DMA operation.

There are certain DMA settings that are not allowed to change from one descriptor to the next in a chain (Small or Large List and Array modes). These are DMA Direction, Word Size, and Memory Space (that is, switching between internal and external memory).

A single descriptor chain cannot control the transfer of a sequence of data buffers which reside in different memory spaces. Instead, group the data buffers into chains of buffers in the same space, but do not link the chains together. Transfer the first chain, wait for its final interrupt, and then start the next chain with an MMR write to the `DMA_CONFIG` register.

Note that while the user must locate each chain's data buffers in the same memory space, the descriptor structures themselves may be placed in any memory space, and they may link from a descriptor in one space to a descriptor in another space without restriction.

Table 9-19 shows the offsets for descriptor elements in the three modes described above. Note the names in the table list the descriptor elements in memory, not the actual MMRs into which they are eventually loaded.

Table 9-19. Parameter Registers and Descriptor Offsets

Descriptor Offset	Descriptor Array Mode	Small Descriptor List Mode	Large Descriptor List Mode
0x0	SAL	NDPL	NDPL
0x2	SAH	SAL	NDPH
0x4	DMACFG	SAH	SAL
0x6	XCNT	DMACFG	SAH
0x8	XMOD	XCNT	DMACFG
0xA	YCNT	XMOD	XCNT
0xC	YMOD	YCNT	XMOD
0xE		YMOD	YCNT
0x10			YMOD

DMA Operation Flow

Figure 9-14 and Figure 9-15 describe the DMA Flow.

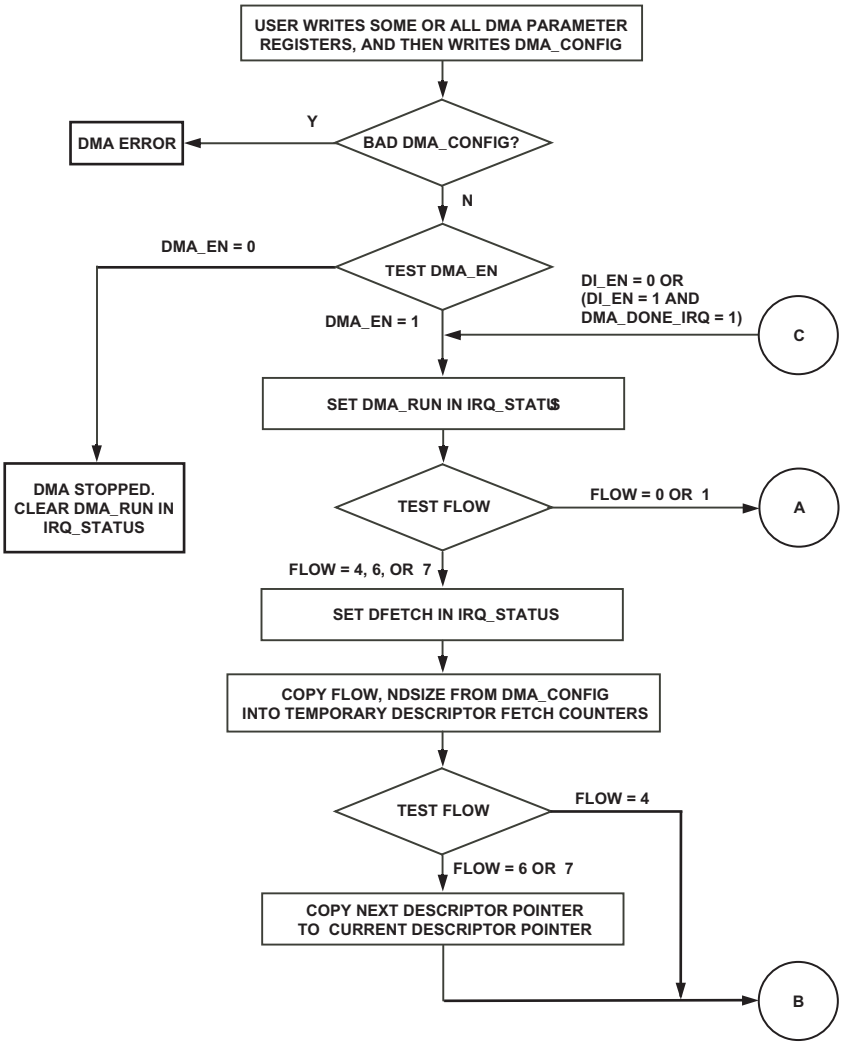


Figure 9-14. DMA Flow, From DMA Controller's Point of View (1 of 2)

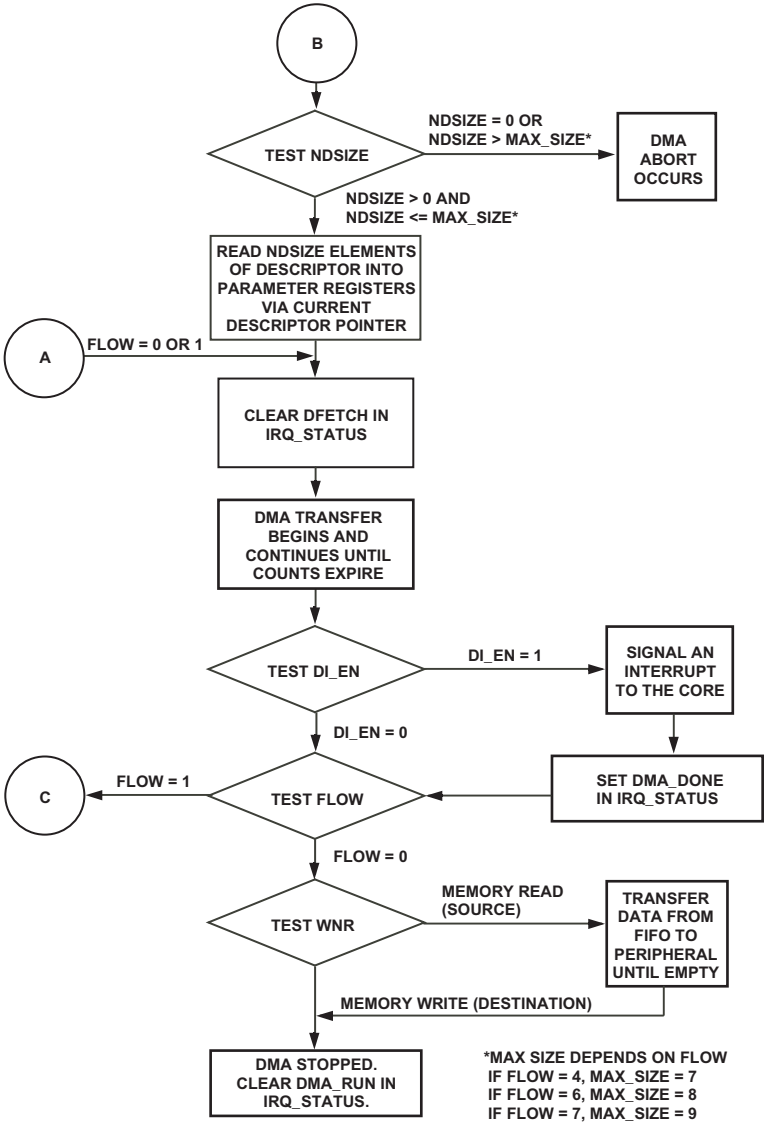


Figure 9-15. DMA Flow, From DMA Controller's Point of View (2 of 2)

DMA Startup

This section discusses starting DMA "from scratch." This is similar to starting it after it has been paused by FLOW = 0 mode.

Before initiating DMA for the first time on a given channel, be sure to initialize all Parameter registers. Be especially careful to initialize the upper 16 bits of the NEXT_DESC_PTR and START_ADDR registers, because they might not otherwise be accessed, depending on the chosen FLOW mode of operation.

To start DMA operation on a given channel, some or all of the DMA Parameter registers must first be written directly. At a minimum, the NEXT_DESC_PTR register (or CURR_DESC_PTR register in FLOW = 4 mode) must be written at this stage, but the user may wish to write other DMA registers that might be static throughout the course of DMA activity (for example, X_MODIFY, Y_MODIFY). The contents of NDSIZE and FLOW in DMA_CONFIG indicate which registers, if any, are fetched from descriptor elements in memory. After the descriptor fetch, if any, is completed, DMA operation begins, initiated by writing DMA_CONFIG with DMAEN = 1.

When DMA_CONFIG is written directly, the DMA controller recognizes this as the special startup condition that occurs when starting DMA for the first time on this channel or after the engine has been stopped (FLOW = 0).

When the descriptor fetch is complete and DMAEN = 1, the DMACFG descriptor element that was read into DMA_CONFIG assumes control. Before this point, the direct write to DMA_CONFIG had control. In other words, the WDSIZE, DI_EN, DI_SEL, RESTART, and DMA2D fields will be taken from the DMACFG value in the descriptor read from memory, while these field values initially written to the DMA_CONFIG register are ignored.

As Figure 9-14 and Figure 9-15 show, at startup the FLOW and NDSIZE bits in DMA_CONFIG determine the course of the DMA setup process. The FLOW value determines whether to load more Current registers from descriptor

elements in memory, while the NDSIZE bits detail how many descriptor elements to fetch before starting DMA. DMA registers not included in the descriptor are not modified from their prior values.

If the FLOW value specifies Small or Large Descriptor List modes, the NEXT_DESC_PTR is copied into CURR_DESC_PTR. Then, fetches of new descriptor elements from memory are performed, indexed by CURR_DESC_PTR, which is incremented after each fetch. If NDPL and/or NDPH is part of the descriptor, then these values are loaded into NEXT_DESC_PTR, but the fetch of the current descriptor continues using CURR_DESC_PTR. After completion of the descriptor fetch, CURR_DESC_PTR points to the next 16-bit word in memory past the end of the descriptor.

If neither NDPH nor NDPL are part of the descriptor (that is, in Descriptor Array mode, FLOW = 4), then the transfer from NDPH/NDPL into CURR_DESC_PTR does not occur. Instead, descriptor fetch indexing begins with the value in CURR_DESC_PTR.

If DMACFG is not part of the descriptor, the previous DMA_CONFIG settings (as written by MMR access at startup) control the work unit operation. If DMACFG is part of the descriptor, then the DMA_CONFIG value programmed by the MMR access controls only the loading of the first descriptor from memory. The subsequent DMA work operation is controlled by the low byte of the descriptor's DMACFG and by the Parameter registers loaded from the descriptor. The bits DI_EN, DI_SEL, DMA2D, WDSIZE, and WNR in the value programmed by the MMR access are disregarded.

The DMA_RUN and DFETCH status bits in the IRQ_STATUS register indicate the state of the DMA channel. After a write to DMA_CONFIG, the DMA_RUN and DFETCH bits can be automatically set to 1. No data interrupts are signaled as a result of loading the first descriptor from memory.

After the above steps, the Current registers are loaded automatically from the appropriate descriptor elements, overwriting their previous contents, as follows.

- START_ADDR is copied to CURR_ADDR
- X_COUNT is copied to CURR_X_COUNT
- Y_COUNT is copied to CURR_Y_COUNT

Then DMA data transfer operation begins, as shown in Figure 9-15.

DMA Refresh

On completion of a work unit, the DMA controller:

- Completes the transfer of all data between memory and the DMA unit.
- If enabled by DI_EN, signals an interrupt to the core and sets the DMA_DONE bit in the channel's IRQ_STATUS register.
- If FLOW = 0 (Stop) only:

  Stops operation by clearing the DMA_RUN bit in IRQ_STATUS after any data in the channel's DMA FIFO has been transferred to the peripheral.
- During the fetch in FLOW modes 4, 6, and 7, the DMA controller sets the DFETCH bit in IRQ_STATUS to 1. At this point, the DMA operation depends on whether FLOW = 4, 6, or 7, as follows:

  If FLOW = 4 (Descriptor Array):

  Loads a new descriptor from memory into DMA registers via the contents of CURR_DESC_PTR, while incrementing CURR_DESC_PTR. The descriptor size comes from the NDSIZE field of the DMA_CONFIG

value prior to the beginning of the fetch.

If FLOW = 6 (Descriptor List Small):

Copies the 32-bit NEXT_DESC_PTR into CURR_DESC_PTR. Next, fetches a descriptor from memory into DMA registers via the new contents of CURR_DESC_PTR, while incrementing CURR_DESC_PTR. The first descriptor element loaded is a new 16-bit value for the lower 16 bits of NEXT_DESC_PTR, followed by the rest of the descriptor elements. The high 16 bits of NEXT_DESC_PTR will retain their former value. This supports a shorter, more efficient descriptor than the Descriptor List Large model, suitable whenever the application can place the channel's descriptors in the same 64K byte range of memory.

If FLOW = 7 (Descriptor List Large):

Copies the 32-bit NEXT_DESC_PTR into CURR_DESC_PTR. Next, fetches a descriptor from memory into DMA registers via the new contents of CURR_DESC_PTR, while incrementing CURR_DESC_PTR. The first descriptor element loaded is a new 32-bit value for the full NEXT_DESC_PTR, followed by the rest of the descriptor elements. The high 16 bits of NEXT_DESC_PTR may differ from their former value. This supports a fully flexible descriptor list which can be located anywhere in internal memory or external memory.

Note if it is necessary to link from a descriptor chain whose descriptors are in one 64K byte area to another chain whose descriptors are outside that area, only one descriptor needs to use FLOW = 7—just the descriptor which contains the link leaving the 64K byte range. All the other descriptors located together in the same 64K byte areas may use FLOW = 6.

- If FLOW = 1, 4, 6, or 7 (Autobuffer, Descriptor Array, Descriptor List Small, or Descriptor List Large, respectively):

  (Re)loads the Current registers:
  CURR_ADDR loaded from START_ADDR
  CURR_X_COUNT loaded from X_COUNT
  CURR_Y_COUNT loaded from Y_COUNT

  The DFETCH bit in IRQ_STATUS is then cleared, after which the DMA transfer begins again, as shown in Figure 9-15.

To Stop DMA Transfers

In FLOW = 0 mode, DMA stops automatically after the work unit is complete.

If a list or array of descriptors is used to control DMA, and if every descriptor contains a DMACFG element, then the final DMACFG element should have a FLOW = 0 setting to gracefully stop the channel.

In Autobuffer (FLOW = 1) mode, or if a list or array of descriptors without DMACFG elements is used, then the DMA transfer process must be terminated by an MMR write to the DMAx_CONFIG register with a value whose DMAEN bit is 0. A write of 0 to the entire register will always terminate DMA gracefully (without DMA Abort).

Before enabling the channel again, make sure that any slow memory read operations that may have started are completed (for example, reads from slow external memory). Do not enable the channel again until any such reads are complete.

To Trigger DMA Transfers

If a DMA has been stopped in FLOW = 0 mode, the DMA_RUN bit in the DMA Interrupt Status register remains set until the content of the internal DMA FIFOs has been completely processed. Once the DMA_RUN bit clears, it is safe to restart the DMA by simply writing again to the DMA Configuration register. The DMA sequence is repeated with the previous settings.

Similarly, a descriptor-based DMA sequence that has been stopped temporarily with a FLOW = 0 descriptor can be continued with a new write to the Configuration register. When the DMA controller detects the FLOW = 0 condition by loading the DMACFG field from memory, it has already updated the Next Descriptor pointer, regardless of whether operating in Descriptor Array mode or Descriptor List mode.

The Next Descriptor pointer remains valid, if the DMA halts and is restarted. As soon as the DMA_RUN bit clears, software can restart the DMA and force the DMA controller to fetch the next descriptor. To accomplish this, the software writes a value with the DMAEN bit set and with proper values in the FLOW and NDSIZE fields into the Configuration register. The next descriptor is fetched if FLOW equals 0x4, 0x6, or 0x7. In this mode of operation, the NDSIZE field should at least span up to the DMACFG field to overwrite the Configuration register immediately.

If all DMACFG fields in a descriptor chain have the FLOW and NDSIZE fields set to zero, the individual DMA sequences do not start until triggered by software. This is useful when the DMAs need to be synchronized with other events in the system, and it is typically performed by interrupt service routines. A single MMR write is required to trigger the next DMA sequence.

Especially when applied to MemDMA channels, such scenarios play an important role. Usually, the timing of MemDMAs cannot be controlled. By halting descriptor chains or rings this way, the whole DMA transaction can be broken into pieces that are individually triggered by software.

Source and destination channels of a MemDMA may differ in descriptor structure. However, the total work count must match when the DMA stops. Whenever a MemDMA is stopped, destination and source channels should both provide the same FLOW = 0 mode after exactly the same number of words. Accordingly, both channels need to be started afterward.

Two-Dimensional DMA

Two-dimensional (2D) DMA supports arbitrary row and column sizes up to 64 K x 64 K elements, as well as arbitrary X_MODIFY and Y_MODIFY values up to ±32 K bytes. Furthermore, Y_MODIFY can be negative, allowing implementation of interleaved datastreams. The X_COUNT and Y_COUNT values specify the row and column sizes, where X_COUNT must be 2 or greater.

The start address and modify values are in bytes, and they must be aligned to a multiple of the DMA transfer word size (WDSIZE[1:0] in DMA_CONFIG). Misalignment causes a DMA error.

The X_MODIFY value is the byte-address increment that is applied after each transfer that decrements the CURR_X_COUNT register. The X_MODIFY value is not applied when the inner loop count is ended by decrementing CURR_X_COUNT from 1 to 0, except that it is applied on the final transfer when CURR_Y_COUNT is 1 and CURR_X_COUNT decrements from 1 to 0.

The Y_MODIFY value is the byte-address increment that is applied after each decrement of CURR_Y_COUNT. However, the Y_MODIFY value is not applied to the last item in the array on which the outer loop count (CURR_Y_COUNT) also expires by decrementing from 1 to 0.

After the last transfer completes, CURR_Y_COUNT = 1, CURR_X_COUNT = 0, and CURR_ADDR is equal to the last item's address plus X_MODIFY. Note if the DMA channel is programmed to refresh automatically (Autobuffer mode), then these registers will be loaded from X_COUNT, Y_COUNT, and START_ADDR upon the first data transfer.

Examples

Example 1: Retrieve a 16 × 8 block of bytes from a video frame buffer of size (N × M) pixels:

```
X_MODIFY = 1
X_COUNT = 16
Y_MODIFY = N-15 (offset from the end of one row to the start of
another)
Y_COUNT = 8
```

This produces the following address offsets from the start address:

```
0,1,2,...15,
N,N + 1, ... N + 15,
2N, 2N + 1,... 2N + 15, ...
7N, 7N + 1,... 7N + 15,
```

Example 2: Receive a video datastream of bytes,
(R,G,B pixels) × (N × M image size):

```
X_MODIFY = (N * M)
X_COUNT = 3
Y_MODIFY = 1 - 2(N * M) (negative)
Y_COUNT = (N * M)
```

This produces the following address offsets from the start address:

```
0, (N * M), 2(N * M),
1, (N * M) + 1, 2(N * M) + 1,
2, (N * M) + 2, 2(N * M) + 2,
    ...
(N * M) - 1, 2(N * M) - 1, 3(N * M) - 1,
```

More 2D DMA Examples

Examples of DMA styles supported by flex descriptors include:

- A single linear buffer that stops on completion (`FLOW` = Stop mode)
- A linear buffer with stride greater than 1 (`X_MODIFY > 1`)
- A circular, auto-refreshing buffer that interrupts on each full buffer
- A similar buffer that interrupts on fractional buffers (for example, 1/2, 1/4) (2D DMA)
- 1D DMA, using a set of identical ping-pong buffers defined by a linked ring of 3-word descriptors, each containing { link pointer, 32-bit address }
- 1D DMA, using a linked list of 5-word descriptors containing { link pointer, 32-bit address, length, config } (ADSP-2191 style)
- 2D DMA, using an array of 1-word descriptors, specifying only the base DMA address within a common data page
- 2D DMA, using a linked list of 9-word descriptors, specifying everything

Memory DMA

This section describes the Memory DMA (MDMA) controller, which provides memory-to-memory DMA transfers among the various memory spaces. These include L1 memory and external synchronous/ asynchronous memories.

Each MDMA controller contains a DMA FIFO, an 8-word by 16-bit FIFO block used to transfer data to and from either L1 or the External Access Bus (EAB). Typically, it is used to transfer data between external memory and internal memory. It will also support DMA from Boot ROM on the EAB bus. The FIFO can be used to hold DMA data transferred between two L1 memory locations or between two external memory locations.

The processor provides four MDMA channels:

- Two source channels (for reading from memory)
- Two destination channels (for writing to memory)

Each source/destination channel forms a “stream,” and these two streams are hardwired for DMA priorities 8 through 11.

- Priority 8: Memory DMA Destination Stream D0
- Priority 9: Memory DMA Source Stream D0
- Priority 10: Memory DMA Destination Stream D1
- Priority 11: Memory DMA Source Stream D1

Memory DMA Stream 0 takes precedence over Memory DMA Stream 1, unless round robin scheduling is used. Note it is illegal to program a source stream for memory write or a destination stream for memory read.

The channels support 8-, 16-, and 32-bit Memory DMA transfers, but both ends of the MDMA transfer must be programmed to the same word size. In other words, the MDMA transfer does not perform packing or unpacking of data; each read results in one write. Both ends of the MDMA FIFO for a given stream are granted priority at the same time. Each pair shares an 8-word-deep 16-bit FIFO. The source DMA engine fills the FIFO, while the destination DMA engine empties it. The FIFO depth allows the burst transfers of the External Access Bus (EAB) and DMA Access Bus (DAB) to overlap, significantly improving throughput on block transfers between internal and external memory. Two separate descriptor blocks are required to supply the operating parameters for each MDMA pair, one for the source channel and one for the destination channel.

Because the source and destination DMA engines share a single FIFO buffer, the descriptor blocks must be configured to have the same data size. It is possible to have a different mix of descriptors on both ends as long as the total count is the same.

To start an MDMA transfer operation, the MMRs for the source and destination streams are written, each in a manner similar to peripheral DMA.

Note the DMA_CONFIG register for the source stream must be written before the DMA_CONFIG register for the destination stream.

When the destination DMA_CONFIG register is written, MDMA operation starts, after a latency of 3 SCLK cycles.

First, if either MDMA stream has been selected to use descriptors, the descriptors are fetched from memory. The destination stream descriptors are fetched first. Then, after a latency of 4 SCLK cycles after the last descriptor word is returned from memory (or typically 8 SCLK cycles after the fetch of the last descriptor word, due to memory pipelining), the source MDMA stream begins fetching data from the source buffer. The

resulting data is deposited in the MDMA stream's 8-location FIFO, and then after a latency of 2 SCLK cycles, the destination MDMA stream begins writing data to the destination memory buffer.

MDMA Bandwidth

If source and destination are in different memory spaces (one internal and one external), the internal and external memory transfers are typically simultaneous and continuous, maintaining 100% bus utilization of the internal and external memory interfaces. This performance is affected by core-to-system clock frequency ratios. At ratios below about 2.5:1, synchronization and pipeline latencies result in lower bus utilization in the system clock domain. At a clock ratio of 2:1, for example, DMA typically runs at 2/3 of the system clock rate. At higher clock ratios, full bandwidth is maintained.

If source and destination are in the same memory space (both internal or both external), the MDMA stream typically prefetches a burst of source data into the FIFO, and then automatically turns around and delivers all available data from the FIFO to the destination buffer. The burst length is dependent on traffic, and is equal to 3 plus the memory latency at the DMA in SCLKs (which is typically 7 for internal transfers and 6 for external transfers).

DMA Performance Optimization

The DMA system is designed to provide maximum throughput per channel and maximum utilization of the internal buses, while accommodating the inherent latencies of memory accesses.

A key feature of the DMA architecture is the separation of the activity on the peripheral DMA bus (the DMA Access Bus (DAB)) from the activity on the buses between the DMA and memory (the DMA Core Bus (DCB) and the DMA External Bus (DEB)). Each peripheral DMA channel has its

own data FIFO which lies between the DAB bus and the memory buses. These FIFOs automatically prefetch data from memory for transmission and buffer received data for later memory writes. This allows the peripheral to be granted a DMA transfer with very low latency compared to the total latency of a pipelined memory access, permitting the repeat rate (bandwidth) of each DMA channel to be as fast as possible.

Peripheral DMA channels have a maximum transfer rate of one 16-bit word per two system clocks, per channel, in either direction.

MDMA channels have a maximum transfer rate of one 16-bit word per one system clock (SCLK), per channel.

When all DMA channels' traffic is taken in the aggregate:

- Transfers between the peripherals and the DMA unit have a maximum rate of one 16-bit transfer per system clock.
- Transfers between the DMA unit and internal memory (L1) have a maximum rate of one 16-bit transfer per system clock.
- Transfers between the DMA unit and external memory have a maximum rate of one 16-bit transfer per system clock.

Some considerations which limit the actual performance include:

- Accesses to internal or external memory which conflict with core accesses to the same memory. This can cause delays, for example, for accessing the same L1 bank, for opening/closing SDRAM pages, or while filling cache lines.
- Each direction change from RX to TX on the DAB bus imposes a one SCLK cycle delay.
- Direction changes on the DCB bus (for example, write followed by read) to the same bank of internal memory can impose delays.

- Direction changes (for example, read followed by write) on the DEB bus to external memory can each impose a several-cycle delay.
- MMR accesses to DMA registers other than DMAx_CONFIG, DMAx_IRQSTAT, or DMAx_PERIPHERAL_MAP will stall all DMA activity for one cycle per 16-bit word transferred. In contrast, MMR accesses to the Control/Status registers do not cause stalls or wait states.
- Reads from DMA registers other than Control/Status registers use one PAB bus wait state, delaying the core for several core clocks.
- Descriptor fetches consume one DMA memory cycle per 16-bit word read from memory, but do not delay transfers on the DAB bus.
- Initialization of a DMA channel stalls DMA activity for one cycle. This occurs when DMAEN changes from 0 to 1 or when the RESTART bit is set to 1 in the DMAx_CONFIG register.

Several of these factors may be minimized by proper design of the application software. It is often possible to structure the software to avoid internal and external memory conflicts by careful allocation of data buffers within banks and pages, and by planning for low cache activity during critical DMA operations. Furthermore, unnecessary MMR accesses can be minimized, especially by using descriptors or autobuffering.

Efficiency loss caused by excessive direction changes (thrashing) can be minimized by the processor's traffic control features, described in the next section.

Prioritization and Traffic Control

DMA channels are ordinarily granted service strictly according to their priority. The priority of a channel is simply its channel number, where lower priority numbers are granted first. Thus, peripherals with high data

rates or low latency requirements should be assigned to lower numbered (higher priority) channels using the DMAx_PERIPHERAL_MAP registers. The Memory DMA streams are always lower priority than the peripherals, but as they request service continuously, they ensure that any time slots unused by peripheral DMA are applied to MDMA transfers. By default, when more than one MDMA stream is enabled and ready, only the highest priority MDMA stream is granted. If it is desirable for the MDMA streams to share the available bandwidth, however, the MDMA_ROUND_ROBIN_PERIOD may be programmed to select each stream in turn for a fixed number of transfers.

In the processor DMA, there are two completely separate but simultaneous prioritization processes—the DAB bus prioritization and the memory bus (DCB and DEB) prioritization. Peripherals that are requesting DMA via the DAB bus, and whose data FIFOs are ready to handle the transfer, compete with each other for DAB bus cycles. Similarly but separately, channels whose FIFOs need memory service (prefetch or post-write) compete together for access to the memory buses. MDMA streams compete for memory access as a unit, and source and destination may be granted together if their memory transfers do not conflict. In this way, internal-to-external or external-to-internal memory transfers may occur at the full system clock rate (SCLK). Examples of memory conflict include simultaneous access to the same memory space and simultaneous attempts to fetch descriptors. Special processing may occur if a peripheral is requesting DMA but its FIFO is not ready (for example, an empty transmit FIFO or full receive FIFO). For more information, see “Urgent DMA Transfers” on page 9-59.

Traffic control is an important consideration in optimizing use of DMA resources. Traffic control is a way to influence how often the transfer direction on the data buses may change, by automatically grouping same direction transfers together. The DMA block provides a traffic control mechanism controlled by the DMA_TC_PER and DMA_TC_CNT registers. This mechanism performs the optimization without real-time processor intervention, and without the need to program transfer bursts into the DMA

work unit streams. Traffic can be independently controlled for each of the three buses (DAB, DCB, and DEB) with simple counters. In addition, alternation of transfers among MDMA streams can be controlled with the MDMA_ROUND_ROBIN_COUNT field of the DMA_TC_CNT register. See "MDMA Priority and Scheduling" on page 9-57.

Using the traffic control features, the DMA system preferentially grants data transfers on the DAB or memory buses which are going in the same read/write direction as the previous transfer, until either the traffic control counter times out, or until traffic stops or changes direction on its own. When the traffic counter reaches zero, the preference is changed to the opposite flow direction. These directional preferences work as if the priority of the opposite direction channels were decreased by 16.

For example, if channels 3 and 5 were requesting DAB access, but lower priority channel 5 is going "with traffic" and higher priority channel 3 is going "against traffic," then channel 3's effective priority becomes 19, and channel 5 would be granted instead. If, on the next cycle, only channels 3 and 6 were requesting DAB transfers, and these transfer requests were both "against traffic," then their effective priorities would become 19 and 22, respectively. One of the channels (channel 3) is granted, even though its direction is opposite to the current flow. No bus cycles are wasted, other than any necessary delay required by the bus turnaround.

This type of traffic control represents a trade-off of latency to improve utilization (efficiency). Higher traffic timeouts might increase the length of time each request waits for its grant, but it often dramatically improves the maximum attainable bandwidth in congested systems, often to above 90%.

To disable preferential DMA prioritization, program the DMA_TC_PER register to 0x0000.

DMA_TC_PER and DMA_TC_CNT Registers

The DMA Traffic Control Counter Period register (DMA_TC_PER) and the DMA Traffic Control Counter register (DMA_TC_CNT) work with other DMA registers to define traffic control.

The MDMA_ROUND_ROBIN_COUNT field shows the current transfer count remaining in the MDMA round robin period. It initializes to MDMA_ROUND_ROBIN_PERIOD whenever DMA_TC_PER is written, whenever a different MDMA stream is granted, or whenever every MDMA stream is idle. It then counts down to 0 with each MDMA transfer. When this count decrements from 1 to 0, the next available MDMA stream is selected.

The DAB_TRAFFIC_COUNT field shows the current cycle count remaining in the DAB traffic period. It initializes to DAB_TRAFFIC_PERIOD whenever DMA_TC_PER is written, or whenever the DAB bus changes direction or becomes idle. It then counts down from DAB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DAB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DAB access is treated preferentially, which may result in a direction change. When this count is 0 and a DAB bus access occurs, the count is reloaded from DAB_TRAFFIC_PERIOD to begin a new burst.

The DEB_TRAFFIC_COUNT field shows the current cycle count remaining in the DEB traffic period. It initializes to DEB_TRAFFIC_PERIOD whenever DMA_TC_PER is written, or whenever the DEB bus changes direction or becomes idle. It then counts down from DEB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DEB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DEB access is treated preferentially, which may result in a direction change. When this count is 0 and a DEB bus access occurs, the count is reloaded from DEB_TRAFFIC_PERIOD to begin a new burst.

The DCB_TRAFFIC_COUNT field shows the current cycle count remaining in the DCB traffic period. It initializes to DCB_TRAFFIC_PERIOD whenever DMA_TC_PER is written, or whenever the DCB bus changes direction or becomes idle. It then counts down from DCB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DCB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DCB access is treated preferentially, which may result in a direction change. When this count is 0 and a DCB bus access occurs, the count is reloaded from DCB_TRAFFIC_PERIOD to begin a new burst.

DMA Traffic Control Counter Period Register (DMA_TC_PER)

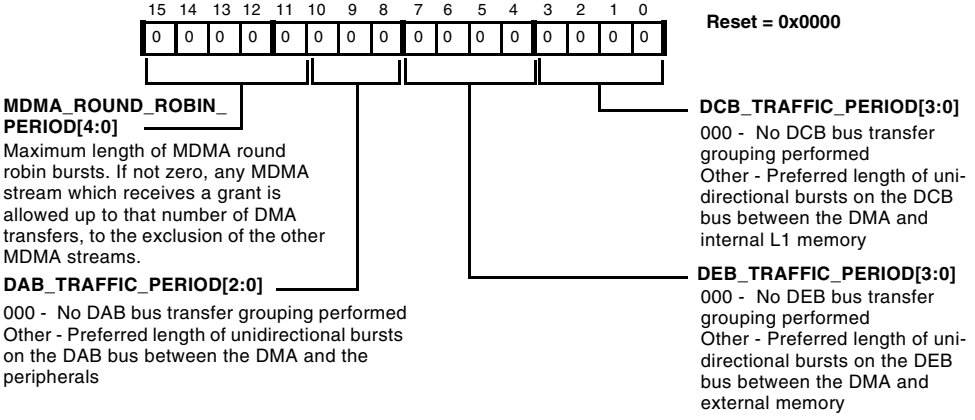


Figure 9-16. DMA Traffic Control Counter Period Register

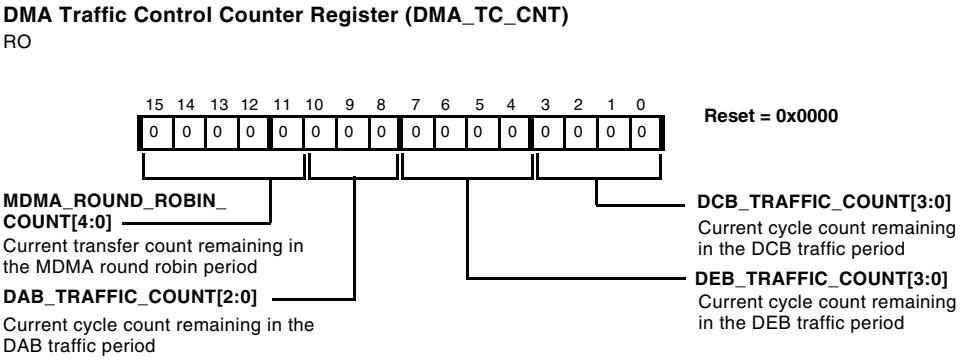


Figure 9-17. DMA Traffic Control Counter Register

MDMA Priority and Scheduling

All MDMA operations have lower precedence than any peripheral DMA operations. MDMA thus makes effective use of any memory bandwidth unused by peripheral DMA traffic.

If two MDMA streams are used (S0-D0 and S1-D1), the user may choose to allocate bandwidth either by fixed stream priority or by a round robin scheme. This is selected by programming the `MDMA_ROUND_ROBIN_PERIOD` field in the `DMA_TC_PER` register (see "Prioritization and Traffic Control" on page 9-52).

If this field is set to 0, then MDMA is scheduled by fixed priority. MDMA Stream 0 takes precedence over MDMA Stream 1 whenever Stream 0 is ready to perform transfers. Since an MDMA Stream is typically capable of transferring data on every available cycle, this could cause MDMA Stream 1 traffic to be delayed for an indefinite time until any and all MDMA Stream 0 operations are complete. This scheme could be appropriate in systems where low duration but latency sensitive data buffers need to be moved immediately, interrupting long duration, low priority background transfers.

If the MDMA_ROUND_ROBIN_PERIOD field is set to some nonzero value in the range 1 <= P <= 31, then a round robin scheduling method is used. The two MDMA streams are granted bus access in alternation in bursts of up to P data transfers. This could be used in systems where two transfer processes need to coexist, each with a guaranteed fraction of the available bandwidth. For example, one stream might be programmed for internal-to-external moves while the other is programmed for external-to-internal moves, and each would be allocated approximately equal data bandwidth.

In round robin operation, the MDMA stream selection at any time is either "free" or "locked." Initially, the selection is free. On any free cycle available to MDMA (when no peripheral DMA accesses take precedence), if either or both MDMA streams request access, the higher precedence stream will be granted (Stream 0 in case of conflict), and that stream's selection is then "locked." The MDMA_ROUND_ROBIN_COUNT counter field in the DMA_TC_CNT register is loaded with the period P from MDMA_ROUND_ROBIN_PERIOD, and MDMA transfers begin. The counter is decremented on every data transfer (as each data word is written to memory). After the transfer corresponding to a count of 1, the MDMA stream selection is passed automatically to the other stream with zero overhead, and the MDMA_ROUND_ROBIN_COUNT counter is reloaded with the period value P from MDMA_ROUND_ROBIN_PERIOD. In this cycle, if the other MDMA stream is ready to perform a transfer, the stream selection is locked on the new MDMA stream. If the other MDMA stream is not ready to perform a transfer, then no transfer is performed, and on the next cycle the stream selection unlocks and becomes free again.

If round robin operation is used when only one MDMA stream is active, one idle cycle will occur for each P MDMA data cycles, slightly lowering bandwidth by a factor of 1/(P+1). If both MDMA streams are used, however, memory DMA can operate continuously with zero additional overhead for alternation of streams (other than overhead cycles normally associated with reversal of read/write direction to memory, for example).

By selection of various round robin period values P which limit how often the MDMA streams alternate, maximal transfer efficiency can be maintained.

Urgent DMA Transfers

Typically, DMA transfers for a given peripheral occur at regular intervals. Generally, the shorter the interval, the higher the priority that should be assigned to the peripheral. If the average bandwidth of all the peripherals is not too large a fraction of the total, then all peripherals' requests should be granted as required.

Occasionally, instantaneous DMA traffic might exceed the available bandwidth, causing congestion. This may occur if L1 or external memory is temporarily stalled, perhaps for an SDRAM page swap or a cache line fill. Congestion might also occur if one or more DMA channels initiates a flurry of requests, perhaps for descriptor fetches or to fill a FIFO in the DMA or in the peripheral.

If congestion persists, lower priority DMA peripherals may become starved for data. Even though the peripheral's priority is low, if the necessary data transfer does not take place before the end of the peripheral's regular interval, system failure may result. To minimize this possibility, the DMA unit detects peripherals whose need for data has become urgent, and preferentially grants them service at the highest priority.

A DMA channel's request for memory service is defined as "urgent" if both:

- The channel's FIFO is not ready for a DAB bus transfer (that is, a transmit FIFO is empty or a receive FIFO is full), and
- The peripheral is asserting its DMA request line.

Descriptor fetches may be urgent, if they are necessary to initiate or continue a DMA work unit chain for a starving peripheral. DMA requests from an MDMA channel are never urgent.

When one or more DMA channels express an urgent memory request, two events occur:

- All non-urgent memory requests are decreased in priority by 32, guaranteeing that only an urgent request will be granted. The urgent requests compete with each other, if there is more than one, and directional preference among urgent requests is observed.
- The resulting memory transfer is marked for expedited processing in the targeted memory system (L1 or external), and so are all prior incomplete memory transfers ahead of it in that memory system. This may cause a series of external memory core accesses to be delayed for a few cycles so that a peripheral's urgent request may be accommodated.

The preferential handling of urgent DMA transfers is completely automatic. No user controls are required for this function to operate.

Software Management of DMA

Several synchronization and control methods are available for use in development of software tasks which manage DMA and MDMA (see also "Memory DMA" on page 9-48). Such software needs to be able to accept requests for new DMA transfers from other software tasks, integrate these transfers into existing transfer queues, and reliably notify other tasks when the transfers are complete.

In the processor, it is possible for each DMA peripheral and MDMA stream to be managed by a separate task or to be managed together with any other stream. Each DMA channel has independent, orthogonal control registers, resources, and interrupts, so that the selection of the control

scheme for one channel does not affect the choice of control scheme on other channels. For example, one peripheral can use a linked-descriptor-list, interrupt-driven scheme while another peripheral can simultaneously use a demand-driven, buffer-at-a-time scheme synchronized by polling of the IRQ_STATUS register.

Synchronization of Software and DMA

A critical element of software DMA management is synchronization of DMA buffer completion with the software. This can best be done using interrupts, polling of IRQ_STATUS, or a combination of both. Polling for address or count can only provide synchronization within loose tolerances comparable to pipeline lengths.

Interrupt-based synchronization methods must avoid interrupt overrun, or the failure to invoke a DMA channel's interrupt handler for every interrupt event due to excessive latency in processing of interrupts. Generally, the system design must either ensure that only one interrupt per channel is scheduled (for example, at the end of a descriptor list), or that interrupts are spaced sufficiently far apart in time that system processing budgets can guarantee every interrupt is serviced. Note since every interrupt channel has its own distinct interrupt, interaction among the interrupts of different peripherals is much simpler to manage.

Polling of the CURR_ADDR, CURR_DESC_PTR, or CURR_X/Y_COUNT registers is not recommended as a method of precisely synchronizing DMA with data processing, due to DMA FIFOs and DMA/memory pipelining. The Current Address, Pointer, and Count registers change several cycles in advance of the completion of the corresponding memory operation, as measured by the time at which the results of the operation would first be visible to the core by memory read or write instructions. For example, in a DMA memory write operation to external memory, assume a DMA write by channel A is initiated that causes the SDRAM to perform a page open operation which will take many system clock cycles. The DMA engine may then move on to another DMA operation by channel B which does

not in itself incur latency, but will be stalled behind the slow operation by channel A. Software monitoring channel B could not safely conclude whether the memory location pointed to by channel B's CURR_ADDR has or has not been written, based on examination of the CURR_ADDR register contents.

Polling of the Current Address, Pointer, and Count registers can permit loose synchronization of DMA with software, however, if allowances are made for the lengths of the DMA/memory pipeline. The length of the DMA FIFO for a peripheral DMA channel is four locations (either four 8- or 16-bit data elements, or two 32-bit data elements) and for an MDMA FIFO is eight locations (four 32-bit data elements). The DMA will not advance Current Address/Pointer/Count registers if these FIFOs are filled with incomplete work (including reads that have been started but not yet finished).

Additionally, the length of the combined DMA and L1 pipelines to internal memory is approximately six 8- or 16-bit data elements. The length of the DMA and External Bus Interface Unit (EBIU) pipelines is approximately three data elements, when measured from the point where a DMA register update is visible to an MMR read to the point where DMA and core accesses to memory become strictly ordered. If the DMA FIFO length and the DMA/memory pipeline length are added, an estimate can be made of the maximum number of incomplete memory operations in progress at one time. (Note this is a maximum, as the DMA/memory pipeline may include traffic from other DMA channels.)

For example, assume a peripheral DMA channel is transferring a work unit of 100 data elements into internal memory and its CURR_X_COUNT register reads a value of 60 remaining elements, so that processing of the first 40 elements has at least been started. The total pipeline length is no greater than the sum of 4 (for the PDMA FIFO) plus 6 (for the DMA/memory pipeline), or 10 data elements, so it is safe to conclude that the DMA transfer of the first 40-10 = 30 data elements is complete.

For precise synchronization, software should either wait for an interrupt or consult the channel's IRQ_STATUS register to confirm completion of DMA, rather than polling Current Address/Pointer/Count registers. When the DMA system issues an interrupt or changes an IRQ_STATUS bit, it guarantees that the last memory operation of the work unit has been completed and will definitely be visible to DSP code. For memory read DMA, the final memory read data will have been safely received in the DMA's FIFO; for memory write DMA, the DMA unit will have received an acknowledge from L1 memory or the EBIU that the data has been written.

The following examples show methods of synchronizing software with several different styles of DMA.

Single-Buffer DMA Transfers

Synchronization is simple if a peripheral's DMA activity consists of isolated transfers of single buffers. DMA activity is initiated by software writes to the channel's Control registers. The user may choose to use a single descriptor in memory, in which case the software only needs to write the DMA_CONFIG and the NEXT_DESC_PTR registers. Alternatively, the user may choose to write all the MMR registers directly from software, ending with the write to the DMA_CONFIG register.

The simplest way to signal completion of DMA is by an interrupt. This is selected by the DI_EN bit in the DMA_CONFIG register, and by the necessary setup of the System Interrupt Controller. If it is desirable not to use an interrupt, the software can poll for completion by reading the IRQ_STATUS register and testing the DMA_RUN bit. If this bit is zero, the buffer transfer has completed.

Continuous Transfers Using Autobuffering

If a peripheral's DMA data consists of a steady, periodic stream of signal data, DMA autobuffering (`FLOW = 1`) may be an effective option. Here, DMA is transferred from or to a memory buffer with a circular addressing scheme, using either one- or two-dimensional indexing with zero processor and DMA overhead for looping. Synchronization options include:

- 1D, interrupt-driven—software is interrupted at the conclusion of each buffer. The critical design consideration is that the software must deal with the first items in the buffer before the next DMA transfer, which might overwrite or re-read the first buffer location before it is processed by software. This scheme may be workable if the system design guarantees that the data repeat period is longer than the interrupt latency under all circumstances.
- 2D, interrupt-driven (double buffering)—the DMA buffer is partitioned into two or more sub-buffers, and interrupts are selected (set `DI_SEL = 1` in `DMA_CONFIG`) to be signaled at the completion of each DMA inner loop. In this way, a traditional double buffer or "ping-pong" scheme could be implemented.

  For example, two 512-word sub-buffers inside a 1K word buffer could be used to receive 16-bit peripheral data with these settings:

  `START_ADDR` = buffer base address
  `DMA_CONFIG = 0x10D7` (`FLOW = 1`, `DI_EN = 1`, `DI_SEL = 1`, `DMA2D = 1`, `WDSIZE = 01`, `WNR = 1`, `DMAEN = 1`)
  `X_COUNT = 512`
  `X_MODIFY = 2` for 16-bit data
  `Y_COUNT = 2` for two sub-buffers
  `Y_MODIFY = 2`, same as `X_MODIFY` for contiguous sub-buffers

- 2D, polled—if interrupt overhead is unacceptable but the loose synchronization of address/count register polling is acceptable, a 2D multibuffer synchronization scheme may be used. For example, assume receive data needs to be processed in packets of sixteen 32-bit elements. A four-part 2D DMA buffer can be allocated where each of the four sub-buffers can hold one packet with these settings:

  `START_ADDR =` buffer base address
  `DMA_CONFIG = 0x101B` (`FLOW = 1`, `DI_EN = 0`, `DMA2D = 1`, `WDSIZE = 10`, `WNR = 1`, `DMAEN = 1`)
  `X_COUNT = 16`
  `X_MODIFY = 4` for 32-bit data
  `Y_COUNT = 4` for four sub-buffers
  `Y_MODIFY = 4`, same as `X_MODIFY` for contiguous sub-buffers

  The synchronization core might read `Y_COUNT` to determine which sub-buffer is currently being transferred, and then allow one full sub-buffer to account for pipelining. For example, if a read of `Y_COUNT` shows a value of 3, then the software should assume that sub-buffer 3 is being transferred, but some portion of sub-buffer 2 may not yet be received. The software could, however, safely proceed with processing sub-buffers 1 or 0.

- 1D unsynchronized FIFO—if a system's design guarantees that the processing of a peripheral's data and the DMA rate of the data will remain correlated in the steady state, but that short-term latency variations must be tolerated, it may be appropriate to build a simple FIFO. Here, the DMA channel may be programmed using 1D Autobuffer mode addressing without any interrupts or polling.

Descriptor Structures

DMA descriptors may be used to transfer data to or from memory data structures that are not simple 1D or 2D arrays. For example, if a packet of data is to be transmitted from several different locations in memory

(a header from one location, a payload from a list of several blocks of memory managed by a memory pool allocator, and a small trailer containing a checksum), a separate DMA descriptor can be prepared for each memory area, and the descriptors can be grouped in either an array or list as desired by selecting the appropriate FLOW setting in DMA_CONFIG.

The software can synchronize with the progress of the structure's transfer by selecting interrupt notification for one or more of the descriptors. For example, the software might select interrupt notification for the header's descriptor and for the trailer's descriptor, but not for the payload blocks' descriptors.

It is important to remember the meaning of the various fields in the DMA_CONFIG descriptor elements when building a list or array of DMA descriptors. In particular:

- The lower byte of DMA_CONFIG specifies the DMA transfer to be performed by the *current* descriptor (for example, interrupt-enable, 2D mode)
- The upper byte of DMA_CONFIG specifies the format of the *next* descriptor in the chain. The NDSIZE and FLOW fields in a given descriptor do not correspond to the format of the descriptor itself; they specify the link to the next descriptor, if any.

On the other hand, when the DMA unit is being restarted, both bytes of the DMA_CONFIG value written to the DMA channel's DMA_CONFIG register should correspond to the current descriptor. At a minimum, the FLOW, NDSIZE, WNR, and DMAEN fields must all agree with the current descriptor; the WDSIZE, DI_EN, DI_SEL, RESTART, and DMA2D fields will be taken from the DMA_CONFIG value in the descriptor read from memory (and the field values initially written to the register are ignored).

Descriptor Queue Management

A system designer might want to write a DMA Manager facility which accepts DMA requests from other software. The DMA Manager software does not know in advance when new work requests will be received or what these requests might contain. The software could manage these transfers using a circular linked list of DMA descriptors, where each descriptor's `NDPH and NDPL` members point to the next descriptor, and the last descriptor points to the first.

The code that writes into this descriptor list could use the processor's circular addressing modes (`I`, `L`, `M`, and `B` registers), so that it does not need to use comparison and conditional instructions to manage the circular structure. In this case, the `NDPH` and `NDPL` members of each descriptor could even be written once at startup, and skipped over as each descriptor's new contents are written.

The recommended method for synchronization of a descriptor queue is through the use of an interrupt. The descriptor queue is structured so that at least the final valid descriptor is always programmed to generate an interrupt.

There are two general methods for managing a descriptor queue using interrupts:

- Interrupt on every descriptor
- Interrupt minimally - only on the last descriptor

Descriptor Queue Using Interrupts on Every Descriptor

In this system, the DMA Manager software synchronizes with the DMA unit by enabling an interrupt on every descriptor. This method should only be used if system design can guarantee that each interrupt event will be serviced separately (no interrupt overrun).

To maintain synchronization of the descriptor queue, the non-interrupt software maintains a count of descriptors added to the queue, while the interrupt handler maintains a count of completed descriptors removed from the queue. The counts are equal only when the DMA channel is paused after having processed all the descriptors.

When each new work request is received, the DMA Manager software initializes a new descriptor, taking care to write a DMA_CONFIG value with a FLOW value of 0. Next, the software compares the descriptor counts to determine if the DMA channel is running or not. If the DMA channel is paused (counts equal), the software increments its count and then starts the DMA unit by writing the new descriptor's DMA_CONFIG value to the DMA channel's DMA_CONFIG register.

If the counts are unequal, the software instead modifies the next-to-last descriptor's DMA_CONFIG value so that its upper half (FLOW and NDSIZE) now describes the newly queued descriptor. This operation does not disrupt the DMA channel, provided the rest of the descriptor data structure is initialized in advance. It is necessary, however, to synchronize the software to the DMA to correctly determine whether the new or the old DMA_CONFIG value was read by the DMA channel.

This synchronization operation should be performed in the interrupt handler. First, upon interrupt, the handler should read the channel's IRQ_STATUS register. If the DMA_RUN status bit is set, then the channel has moved on to processing another descriptor, and the interrupt handler may increment its count and exit. If the DMA_RUN status bit is not set, however, then the channel has paused, either because there are no more descriptors to process, or because the last descriptor was queued too late (that is, the modification of the next-to-last descriptor's DMA_CONFIG element occurred after that element was read into the DMA unit.) In this case, the interrupt handler should write the DMA_CONFIG value appropriate for the last descriptor to the DMA channel's DMA_CONFIG register, increment the completed descriptor count, and exit.

Again, this system can fail if the system's interrupt latencies are large enough to cause any of the channel's DMA interrupts to be dropped. An interrupt handler capable of safely synchronizing multiple descriptors' interrupts would need to be complex, performing several MMR accesses to ensure robust operation. In such a system environment, a minimal interrupt synchronization method is preferred.

Descriptor Queue Using Minimal Interrupts

In this system, only one DMA interrupt event is possible in the queue at any time. The DMA interrupt handler for this system can also be extremely short. Here, the descriptor queue is organized into an "active" and a "waiting" portion, where interrupts are enabled only on the last descriptor in each portion.

When each new DMA request is processed, the software's non-interrupt code fills in a new descriptor's contents and adds it to the waiting portion of the queue. The descriptor's `DMA_CONFIG` word should have a `FLOW` value of zero. If more than one request is received before the DMA queue completion interrupt occurs, the non-interrupt code should queue later descriptors, forming a waiting portion of the queue that is disconnected from the active portion of the queue being processed by the DMA unit. In other words, all but the last active descriptors contain `FLOW` values >= 4 and have no interrupt enable set, while the last active descriptor contains a `FLOW` of 0 and an interrupt enable bit `DI_EN` set to 1. Also, all but the last waiting descriptors contain `FLOW` values >= 4 and no interrupt enables set, while the last waiting descriptor contains a `FLOW` of 0 and an interrupt enable bit set to 1. This ensures that the DMA unit can automatically process the whole active queue and then issue one interrupt. Also, this arrangement makes it easy to start the waiting queue within the interrupt handler by a single `DMA_CONFIG` register write.

After queuing a new waiting descriptor, the non-interrupt software should leave a message for its interrupt handler in a memory mailbox location containing the desired DMA_CONFIG value to use to start the first waiting descriptor in the waiting queue (or 0 to indicate no descriptors are waiting.)

It is critical that the software not modify the contents of the active descriptor queue directly, once its processing by the DMA unit has been started, unless careful synchronization measures are taken. In the most straightforward implementation of a descriptor queue, the DMA Manager software would never modify descriptors on the active queue; instead, the DMA Manager waits until the DMA queue completion interrupt indicates the processing of the entire active queue is complete.

When a DMA queue completion interrupt is received, the interrupt handler reads the mailbox from the non-interrupt software and writes the value in it to the DMA channel's DMA_CONFIG register. This single register write restarts the queue, effectively transforming the waiting queue to an active queue. The interrupt handler should then pass a message back to the non-interrupt software indicating the location of the last descriptor accepted into the active queue. If, on the other hand, the interrupt handler reads its mailbox and finds a DMA_CONFIG value of zero, indicating there is no more work to perform, then it should pass an appropriate message (for example, zero) back to the non-interrupt software indicating that the queue has stopped. This simple handler should be able to be coded in a very small number of instructions.

The non-interrupt software which accepts new DMA work requests needs to synchronize the activation of new work with the interrupt handler. If the queue has stopped (that is, if the mailbox from the interrupt software is zero), the non-interrupt software is responsible for starting the queue (writing the first descriptor's DMA_CONFIG value to the channel's DMA_CONFIG register). If the queue is not stopped, however, the non-interrupt software must not write the DMA_CONFIG register

(which would cause a DMA error), but instead it should queue the descriptor onto the waiting queue and update its mailbox directed to the interrupt handler.

DMA Errors (Aborts)

The DMA controller flags conditions that cause the DMA process to end abnormally (that is, abort). This functionality is provided as a tool for system development and debug, as a way to detect DMA-related programming errors. DMA errors (aborts) are detected by the DMA channel module in the cases listed below. When a DMA error occurs, the channel is immediately stopped (DMA_RUN goes to 0) and any prefetched data is discarded. In addition, a DMA_ERROR interrupt is asserted.

There is only one DMA_ERROR interrupt for the whole DMA controller, which is asserted whenever any of the channels has detected an error condition.

The DMA_ERROR interrupt handler must do these things for each channel:

- Read each channel's IRQ_STATUS register to look for a channel with the DMA_ERR bit set (bit 1).
- Clear the problem with that channel (for example, fix register values).
- Clear the DMA_ERR bit (write IRQ_STATUS with bit 1 = 1).

The following error conditions are detected by the DMA hardware and result in a DMA Abort interrupt.

- The Configuration register contains invalid values:

  - Incorrect WDSIZE value (WDSIZE = b#11)
  - Bit 15 not set to 0
  - Incorrect FLOW value (FLOW = 2, 3, or 5)
  - NDSIZE value does not agree with FLOW. See Table 9-20.

- A disallowed register write occurred while the channel was running. Only the DMA_CONFIG and IRQ_STATUS registers can be written when DMA_RUN = 1.

- An address alignment error occurred during any memory access. For example, DMA_CONFIG register WDSIZE = 1 (16 bit) but the least significant bit (LSB) of the address is not equal to 0, or WDSIZE = 2 (32 bit) but the two LSBs of the address are not equal to 00.

- A memory space transition was attempted (internal-to-external or vice versa).

- A memory access error occurred. Either an access attempt was made to an internal address not populated or defined as cache, or an external access caused an error (signaled by the external memory interface).

Some prohibited situations are not detected by the DMA hardware. No DMA abort is signaled for these situations:

- DMA_CONFIG Direction bit (WNR) does not agree with the direction of the mapped peripheral.

- DMA_CONFIG Direction bit does not agree with the direction of the MDMA channel.

- DMA_CONFIG Word Size (WDSIZE) is not supported by the mapped peripheral.

- DMA_CONFIG Word Size in source and destination of the MDMA stream are not equal.
- Descriptor chain indicates data buffers that are not in the same internal/external memory space.
- In 2D DMA, X_COUNT = 1.

Table 9-20. Legal NDSIZE Values

FLOW	NDSIZE	Note
0	0	
1	0	
4	0 < NDSIZE <= 7	Descriptor array, no descriptor pointer fetched
6	0 < NDSIZE <= 8	Descriptor list, small descriptor pointer fetched
7	0 < NDSIZE <= 9	Descriptor list, large descriptor pointer fetched

10 SPI COMPATIBLE PORT CONTROLLERS

The processor has a Serial Peripheral Interface (SPI) port that provides an I/O interface to a wide variety of SPI compatible peripheral devices.

With a range of configurable options, the SPI port provides a glueless hardware interface with other SPI compatible devices. SPI is a four-wire interface consisting of two data pins, a device select pin, and a clock pin. SPI is a full-duplex synchronous serial interface, supporting master modes, slave modes, and multimaster environments. The SPI compatible peripheral implementation also supports programmable baud rate and clock phase/polarities. The SPI features the use of open drain drivers to support the multimaster scenario and to avoid data contention.

Typical SPI compatible peripheral devices that can be used to interface to the SPI compatible interface include:

- Other CPUs or microcontrollers
- Codecs
- A/D converters
- D/A converters
- Sample rate converters
- SP/DIF or AES/EBU digital audio transmitters and receivers
- LCD displays

- Shift registers
- FPGAs with SPI emulation

The SPI is an industry-standard synchronous serial link that supports communication with multiple SPI compatible devices. The SPI peripheral is a synchronous, four-wire interface consisting of two data pins (MOSI and MISO), one device select pin ($\overline{\text{SPISS}}$), and a gated clock pin (SCK). With the two data pins, it allows for full-duplex operation to other SPI compatible devices. The SPI also includes programmable baud rates, clock phase, and clock polarity.

The SPI can operate in a multimaster environment by interfacing with several other devices, acting as either a master device or a slave device. In a multimaster environment, the SPI interface uses open drain outputs to avoid data bus contention.

Figure 10-1 provides a block diagram of the SPI. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the SCK rate, to and from other SPI devices. SPI data is transmitted and received at the same time through the use of a shift register. When an SPI transfer occurs, data is simultaneously transmitted (shifted serially out of the shift register) as new data is received (shifted serially into the other end of the same shift register). The SCK synchronizes the shifting and sampling of the data on the two serial data pins.

During SPI data transfers, one SPI device acts as the SPI link master, where it controls the data flow by generating the SPI serial clock and asserting the SPI device select signal ($\overline{\text{SPISS}}$). The other SPI device acts as the slave and accepts new data from the master into its shift register, while it transmits requested data out of the shift register through its SPI transmit data pin. Multiple processors can take turns being the master device, as can other microcontrollers or microprocessors. One master device can also simultaneously shift data into multiple slaves (known as Broadcast mode). However, only one slave may drive its output to write data back to

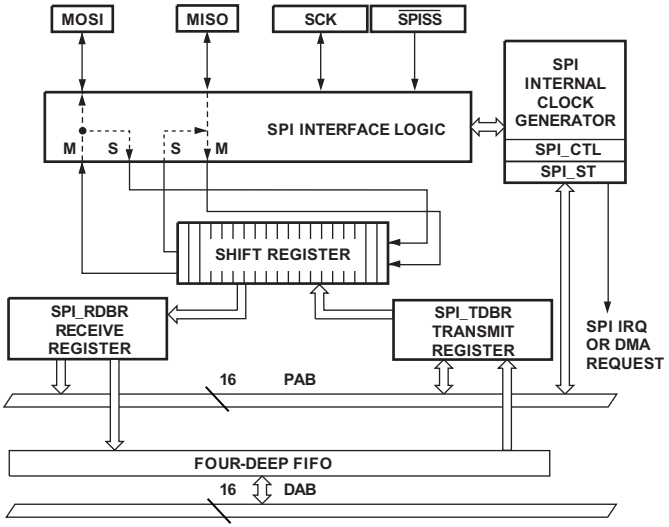


Figure 10-1. SPI Block Diagram

the master at any given time. This must be enforced in Broadcast mode, where several slaves can be selected to receive data from the master, but only one slave at a time can be enabled to send data back to the master.

In a multimaster or multidevice environment where multiple processors are connected via their SPI ports, all MOSI pins are connected together, all MISO pins are connected together, and all SCK pins are connected together.

For a multislave environment, the processor can make use of seven programmable flags, PF1–PF7, that are dedicated SPI slave select signals for the SPI slave devices.

At reset, the SPI is disabled and configured as a slave.

Interface Signals

The following section discusses the SPI signals.

Serial Peripheral Interface Clock Signal (SCK)

The SCK signal is the SPI clock signal. This control signal is driven by the master and controls the rate at which data is transferred. The master may transmit data at a variety of baud rates. The SCK signal cycles once for each bit transmitted. It is an output signal if the device is configured as a master, and an input signal if the device is configured as a slave.

The SCK is a gated clock that is active during data transfers only for the length of the transferred word. The number of active clock edges is equal to the number of bits driven on the data lines. Slave devices ignore the serial clock if the Serial Peripheral Slave Select Input ($\overline{\text{SPISS}}$) is driven inactive (high).

The SCK is used to shift out and shift in the data driven on the MISO and MOSI lines. Clock polarity and clock phase relative to data are programmable in the SPI Control register (SPI_CTL) and define the transfer format (see "SPI Transfer Formats" on page 10-20).

Serial Peripheral Interface Slave Select Input Signal

The $\overline{\text{SPISS}}$ signal is the SPI Serial Peripheral Slave Select Input signal. This is an active-low signal used to enable a processor when it is configured as a slave device. This input-only pin behaves like a chip select and is provided by the master device for the slave devices. For a master device, it can act as an error signal input in case of the multimaster environment. In multimaster mode, if the $\overline{\text{SPISS}}$ input signal of a master is asserted

(driven low), and the PSSE bit in the SPI_CTL register is enabled, an error has occurred. This means that another device is also trying to be the master device.

The $\overline{\text{SPISS}}$ signal is the same pin as the PF0 pin.

Master Out Slave In (MOSI)

The MOSI pin is the Master Out Slave In pin, one of the bidirectional I/O data pins. If the processor is configured as a master, the MOSI pin becomes a data transmit (output) pin, transmitting output data. If the processor is configured as a slave, the MOSI pin becomes a data receive (input) pin, receiving input data. In an SPI interconnection, the data is shifted out from the MOSI output pin of the master and shifted into the MOSI input(s) of the slave(s).

Master In Slave Out (MISO)

The MISO pin is the Master In Slave Out pin, one of the bidirectional I/O data pins. If the processor is configured as a master, the MISO pin becomes a data receive (input) pin, receiving input data. If the processor is configured as a slave, the MISO pin becomes a data transmit (output) pin, transmitting output data. In an SPI interconnection, the data is shifted out from the MISO output pin of the slave and shifted into the MISO input pin of the master.

Only one slave is allowed to transmit data at any given time.

The SPI configuration example in Figure 10-2 illustrates how the processor can be used as the slave SPI device. The 8-bit host microcontroller is the SPI master.

The processor can be booted via its SPI interface to allow user application code and data to be downloaded before runtime.

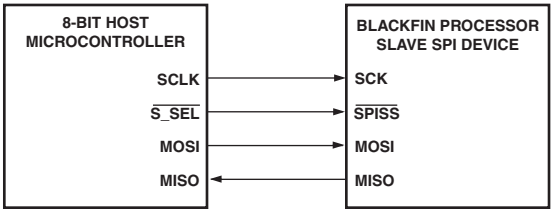


Figure 10-2. ADSP-BF533 as Slave SPI Device

Interrupt Output

The SPI has two interrupt output signals: a data interrupt and an error interrupt.

The behavior of the SPI data interrupt signal depends on the Transfer Initiation mode bit field (TIMOD) in the SPI Control register. In DMA mode (TIMOD = 1X), the data interrupt acts as a DMA request and is generated when the DMA FIFO is ready to be written to (TIMOD = 11) or read from (TIMOD = 10). In non-DMA mode (TIMOD = 0X), a data interrupt is generated when the SPI_TDBR is ready to be written to (TIMOD = 01) or when the SPI_RDBR is ready to be read from (TIMOD = 00).

An SPI Error interrupt is generated in a master when a Mode Fault Error occurs, in both DMA and non-DMA modes. An error interrupt can also be generated in DMA mode when there is an underflow (TXE when TIMOD = 11) or an overflow (RBSY when TIMOD = 10) error condition. In non-DMA mode, the underflow and overflow conditions set the TXE and RBSY bits in the SPI_STAT register, respectively, but do not generate an error interrupt.

For more information about this interrupt output, see the discussion of the TIMOD bits in "SPI_CTL Register" on page 10-8.

SPI Registers

The SPI peripheral includes a number of user-accessible registers. Some of these registers are also accessible through the DMA bus. Four registers contain control and status information: SPI_BAUD, SPI_CTL, SPI_FLG, and SPI_STAT. Two registers are used for buffering receive and transmit data: SPI_RDBR and SPI_TDBR. For information about DMA-related registers, see Chapter 9, "Direct Memory Access." The shift register, SFDR, is internal to the SPI module and is not directly accessible.

See "Error Signals and Flags" on page 10-28 for more information about how the bits in these registers are used to signal errors and other conditions. See "Register Functions" on page 10-19 for more information about SPI register and bit functions.

SPI_BAUD Register

The SPI Baud Rate register (SPI_BAUD) is used to set the bit transfer rate for a master device. When configured as a slave, the value written to this register is ignored. The serial clock frequency is determined by this formula:

```
SCK Frequency = (Peripheral clock frequency SCLK)/(2 x SPI_BAUD)
```

Writing a value of 0 or 1 to the register disables the serial clock. Therefore, the maximum serial clock rate is one-fourth the system clock rate.

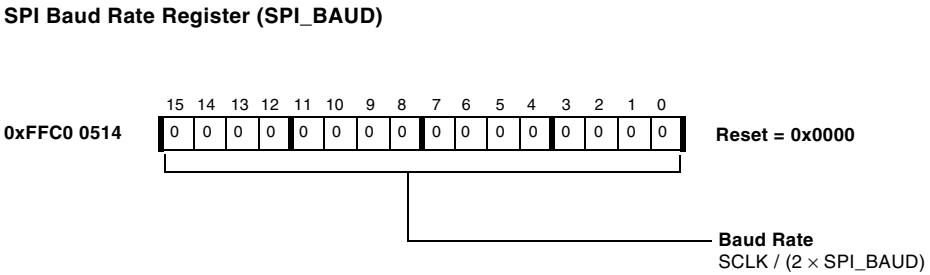


Figure 10-3. SPI Baud Rate Register

Table 10-1 lists several possible baud rate values for SPI_BAUD.

Table 10-1. SPI Master Baud Rate Example

SPI_BAUD Decimal Value	SPI Clock (SCK) Divide Factor	Baud Rate for SCLK at 100 MHz
0	N/A	N/A
1	N/A	N/A
2	4	25 MHz
3	6	16.7 MHz
4	8	12.5 MHz
65,535 (0xFFFF)	131,070	763 Hz

SPI_CTL Register

The SPI Control register (SPI_CTL) is used to configure and enable the SPI system. This register is used to enable the SPI interface, select the device as a master or slave, and determine the data transfer format and word size.

The term “word” refers to a single data transfer of either 8 bits or 16 bits, depending on the word length (SIZE) bit in SPI_CTL. There are two special bits which can also be modified by the hardware: SPE and MSTR.

The TIMOD field is used to specify the action that initiates transfers to/from the receive/transmit buffers. When set to 00, a SPI port transaction is begun when the receive buffer is read. Data from the first read will need to be discarded since the read is needed to initiate the first SPI port transaction. When set to 01, the transaction is initiated when the transmit buffer is written. A value of 10 selects DMA Receive mode and the first transaction is initiated by enabling the SPI for DMA Receive mode. Subsequent individual transactions are initiated by a DMA read of the SPI_RDBR. A value of 11 selects DMA Transmit mode and the transaction is initiated by a DMA write of the SPI_TDBR.

The PSSE bit is used to enable the $\overline{\text{SPISS}}$ input for master. When not used, $\overline{\text{SPISS}}$ can be disabled, freeing up a chip pin as general-purpose I/O.

The EMISO bit enables the MISO pin as an output. This is needed in an environment where the master wishes to transmit to various slaves at one time (broadcast). Only one slave is allowed to transmit data back to the master. Except for the slave from whom the master wishes to receive, all other slaves should have this bit cleared.

The SPE and MSTR bits can be modified by hardware when the MODF bit of the Status register is set. See “Mode Fault Error” on page 10-28.

Figure 10-4 provides the bit descriptions for SPI_CTL.

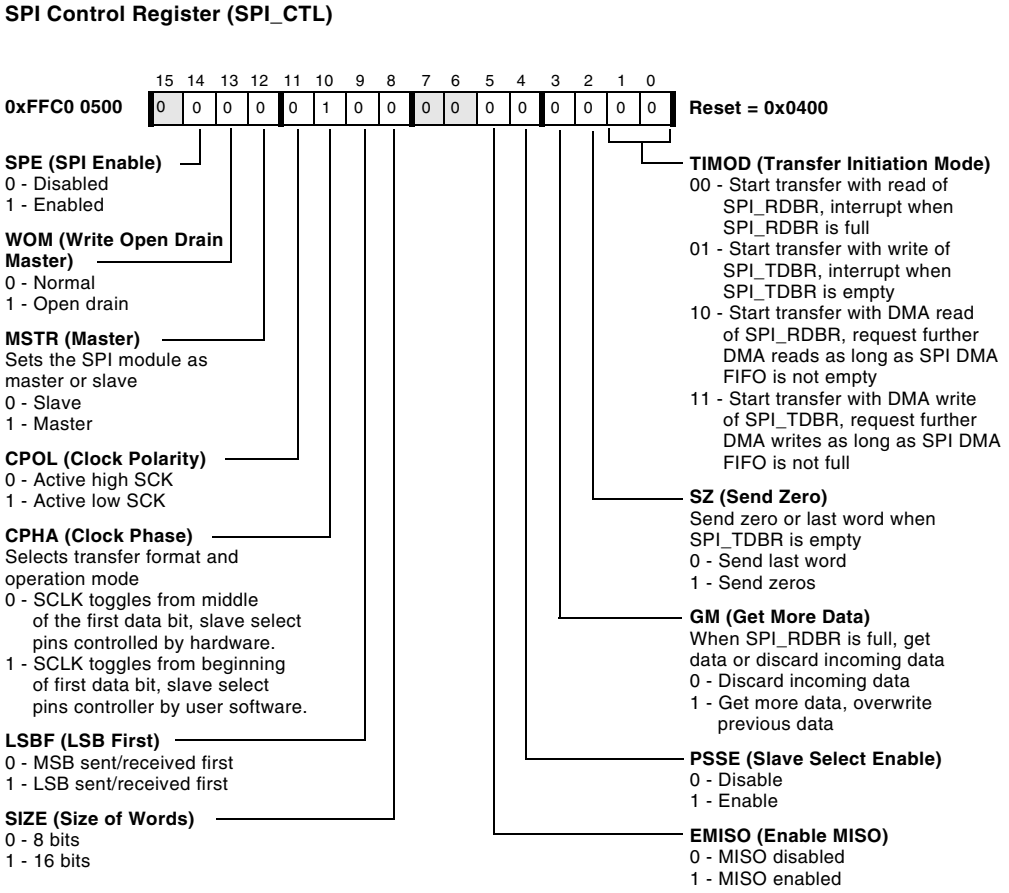


Figure 10-4. SPI Control Register

SPI_FLG Register

If the SPI is enabled as a master, the SPI uses the SPI Flag register (SPI_FLG) to enable up to seven general-purpose programmable flag pins to be used as individual slave select lines. In Slave mode, the SPI_FLG bits have no effect, and each SPI uses the $\overline{\text{SPISS}}$ input as a slave select. Figure 10-5 shows the SPI_FLG register diagram.

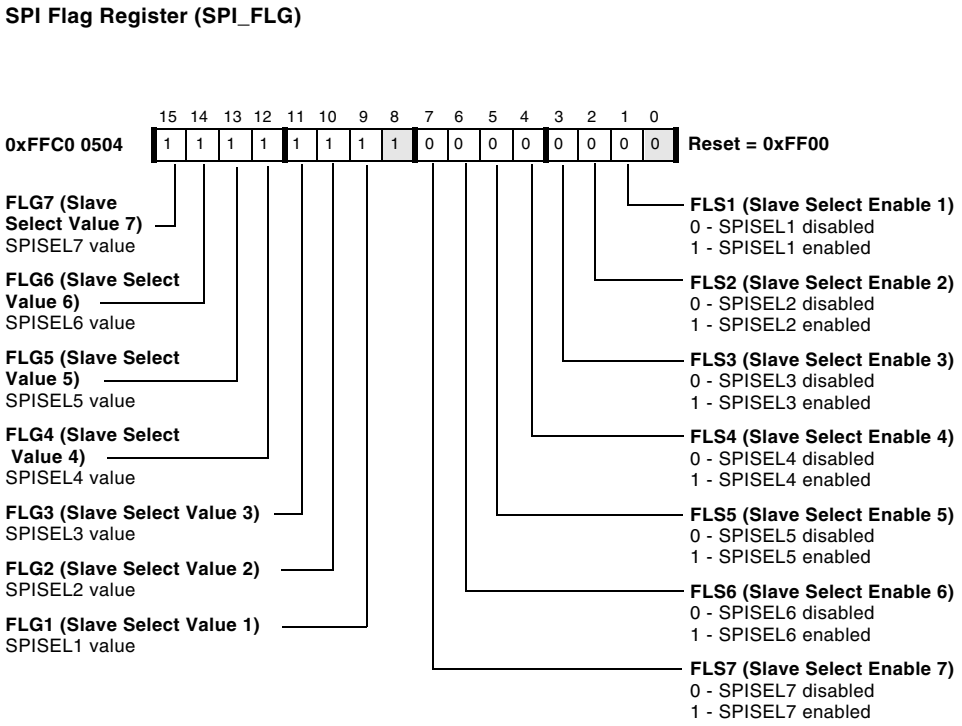


Figure 10-5. SPI Flag Register

The SPI_FLG register consists of two sets of bits that function as follows.

- Slave Select Enable (FLSx) bits

  Each FLSx bit corresponds to a Programmable Flag (PFx) pin. When a FLSx bit is set, the corresponding PFx pin is driven as a slave select. For example, if FLS1 is set in SPI_FLG, PF1 is driven as a slave select (SPISEL1). Table 10-2 shows the association of the FLSx bits and the corresponding PFx pins.

  If the FLSx bit is not set, the general-purpose programmable flag registers (FIO_DIR and others) configure and control the corresponding PFx pin.

- Slave Select Value (FLGx) bits

  When a PFx pin is configured as a slave select output, the FLGx bits can determine the value driven onto the output. If the CPHA bit in SPI_CTL is set, the output value is set by software control of the FLGx bits. The SPI protocol permits the slave select line to either remain asserted (low) or be deasserted between transferred words. The user must set or clear the appropriate FLGx bits. For example, to drive PF3 as a slave select, FLS3 in SPI_FLG must be set. Clearing FLG3 in SPI_FLG drives PF3 low; setting FLG3 drives PF3 high. The PF3 pin can be cycled high and low between transfers by setting and clearing FLG3. Otherwise, PF3 remains active (low) between transfers.

  If CPHA = 0, the SPI hardware sets the output value and the FLGx bits are ignored. The SPI protocol requires that the slave select be deasserted between transferred words. In this case, the SPI hardware controls the pins. For example, to use PF3 as a slave select pin, it is only necessary to set the FLS3 bit in SPI_FLG. It is not necessary to write to the FLG3 bit, because the SPI hardware automatically drives the PF3 pin.

Table 10-2. SPI_FLG Bit Mapping to PFx Pins

Bit	Name	Function	PFx Pin	Default
0		Reserved		0
1	FLS1	SPISEL1 Enable	PF1	0
2	FLS2	SPISEL2 Enable	PF2	0
3	FLS3	SPISEL3 Enable	PF3	0
4	FLS4	SPISEL4 Enable	PF4	0
5	FLS5	SPISEL5 Enable	PF5	0
6	FLS6	SPISEL6 Enable	PF6	0
7	FLS7	SPISEL7 Enable	PF7	0
8		Reserved		1
9	FLG1	SPISEL1 Value	PF1	1
10	FLG2	SPISEL2 Value	PF2	1
11	FLG3	SPISEL3 Value	PF3	1
12	FLG4	SPISEL4 Value	PF4	1
13	FLG5	SPISEL5 Value	PF5	1
14	FLG6	SPISEL6 Value	PF6	1
15	FLG7	SPISEL7 Value	PF7	1

Slave Select Inputs

If the SPI is in Slave mode, $\overline{\texttt{SPISS}}$ acts as the slave select input. When enabled as a master, $\overline{\texttt{SPISS}}$ can serve as an error detection input for the SPI in a multimaster environment. The PSSE bit in SPI_CTL enables this feature. When PSSE = 1, the $\overline{\texttt{SPISS}}$ input is the master mode error input. Otherwise, $\overline{\texttt{SPISS}}$ is ignored.

Use of FLS Bits in SPI_FLG for Multiple Slave SPI Systems

The FLSx bits in the SPI_FLG register are used in a multiple slave SPI environment. For example, if there are eight SPI devices in the system including a processor master, the master processor can support the SPI mode transactions across the other seven devices. This configuration requires only one master processor in this multislave environment. For example, assume that the SPI is the master. The seven flag pins (PF1-PF7) on the processor master can be connected to each of the slave SPI device's $\overline{\text{SPISS}}$ pins. In this configuration, the FLSx bits in SPI_FLG can be used in three cases.

In cases 1 and 2, the processor is the master and the seven microcontrollers/peripherals with SPI interfaces are slaves. The processor can:

1. Transmit to all seven SPI devices at the same time in a broadcast mode. Here, all FLSx bits are set.
2. Receive and transmit from one SPI device by enabling only one slave SPI device at a time.

   In case 3, all eight devices connected via SPI ports can be other processors.
3. If all the slaves are also processors, then the requester can receive data from only one processor (enabled by clearing the EMISO bit in the six other slave processors) at a time and transmit broadcast data to all seven at the same time. This EMISO feature may be available in some other microcontrollers. Therefore, it is possible to use the EMISO feature with any other SPI device that includes this functionality.

Figure 10-6 shows one processor as a master with three processors (or other SPI compatible devices) as slaves.

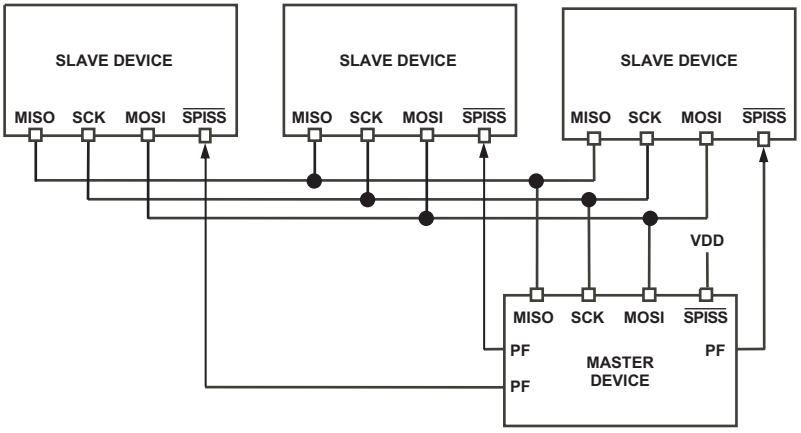


Figure 10-6. Single-Master, Multiple-Slave Configuration

SPI_STAT Register

The SPI Status register (SPI_STAT) is used to detect when an SPI transfer is complete or if transmission/reception errors occur. The SPI_STAT register can be read at any time.

Some of the bits in SPI_STAT are read-only and other bits are sticky. Bits that provide information only about the SPI are read-only. These bits are set and cleared by the hardware. Sticky bits are set when an error condition occurs. These bits are set by hardware and must be cleared by software. To clear a sticky bit, the user must write a 1 to the desired bit position of SPI_STAT. For example, if the TXE bit is set, the user must write a 1 to bit 2 of SPI_STAT to clear the TXE error condition. This allows the user to read SPI_STAT without changing its value.

Sticky bits are cleared on a reset, but are not cleared on an SPI disable.

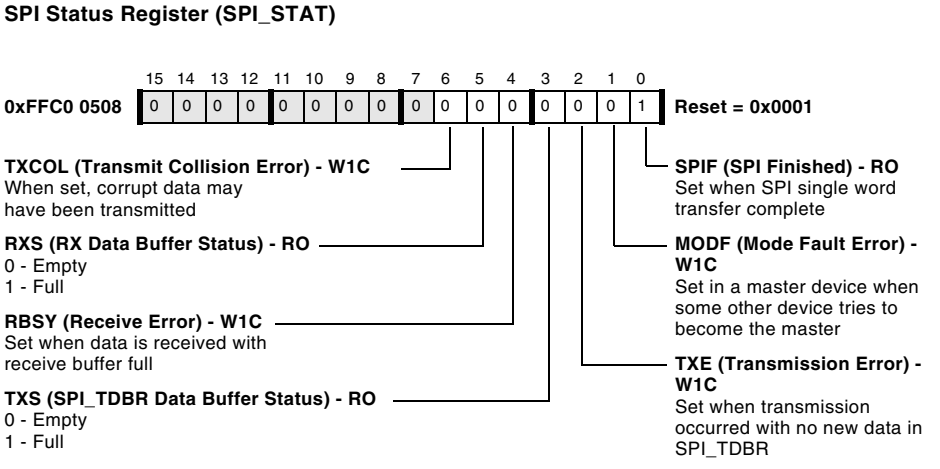


Figure 10-7. SPI Status Register

The transmit buffer becomes full after it is written to. It becomes empty when a transfer begins and the transmit value is loaded into the shift register. The receive buffer becomes full at the end of a transfer when the shift register value is loaded into the receive buffer. It becomes empty when the receive buffer is read.

The SPIF bit is set when the SPI port is disabled.

Upon entering DMA mode, the transmit buffer and the receive buffer become empty. That is, the TXS bit and the RXS bit are initially cleared upon entering DMA mode.

When using DMA for SPI transmit, the DMA_DONE interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll TXS in the SPI_STAT register until it goes low for 2 successive reads, at which point the SPI DMA FIFO will be empty. When the SPIF bit subsequently goes low, the last word has been transferred and the SPI can be disabled or enabled for another mode.

SPI_TDBR Register

The SPI Transmit Data Buffer register (SPI_TDBR) is a 16-bit read-write register. Data is loaded into this register before being transmitted. Just prior to the beginning of a data transfer, the data in SPI_TDBR is loaded into the Shift Data register (SFDR). A read of SPI_TDBR can occur at any time and does not interfere with or initiate SPI transfers.

When the DMA is enabled for transmit operation, the DMA engine loads data into this register for transmission just prior to the beginning of a data transfer. A write to SPI_TDBR should not occur in this mode because this data will overwrite the DMA data to be transmitted.

When the DMA is enabled for receive operation, the contents of SPI_TDBR are repeatedly transmitted. A write to SPI_TDBR is permitted in this mode, and this data is transmitted.

If the Send Zeros control bit (SZ in the SPI_CTL register) is set, SPI_TDBR may be reset to 0 under certain circumstances.

If multiple writes to SPI_TDBR occur while a transfer is already in progress, only the last data written is transmitted. None of the intermediate values written to SPI_TDBR are transmitted. Multiple writes to SPI_TDBR are possible, but not recommended.

SPI Transmit Data Buffer Register (SPI_TDBR)

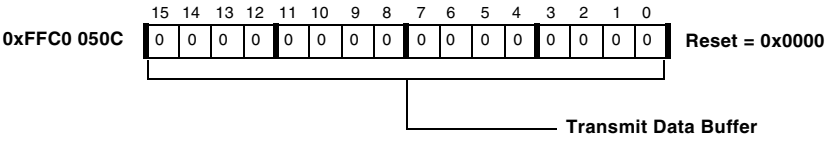


Figure 10-8. SPI Transmit Data Buffer Register

SPI_RDBR Register

The SPI Receive Data Buffer register (SPI_RDBR) is a 16-bit read-only register. At the end of a data transfer, the data in the shift register is loaded into SPI_RDBR. During a DMA receive operation, the data in SPI_RDBR is automatically read by the DMA. When SPI_RDBR is read via software, the RXS bit is cleared and an SPI transfer may be initiated (if TIMOD = 00).

SPI Receive Data Buffer Register (SPI_RDBR)
RO

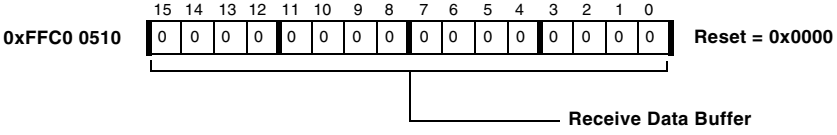


Figure 10-9. SPI Receive Data Buffer Register

SPI_SHADOW Register

The SPI RDBR Shadow register (SPI_SHADOW), has been provided for use in debugging software. This register is at a different address than the receive data buffer, SPI_RDBR, but its contents are identical to that of SPI_RDBR. When a software read of SPI_RDBR occurs, the RXS bit in

SPI_STAT is cleared and an SPI transfer may be initiated (if TIMOD = 00 in SPI_CTL). No such hardware action occurs when the SPI_SHADOW register is read. The SPI_SHADOW register is read-only.

SPI RDBR Shadow Register (SPI_SHADOW)
RO

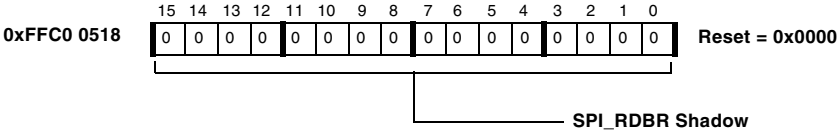


Figure 10-10. SPI RDBR Shadow Register

Register Functions

Table 10-3 summarizes the functions of the SPI registers.

Table 10-3. SPI Register Mapping

Register Name	Function	Notes
SPI_CTL	SPI port control	SPE and MSTR bits can also be modified by hardware (when MODF is set)
SPI_FLG	SPI port flag	Bits 0 and 8 are reserved
SPI_STAT	SPI port status	SPIF bit can be set by clearing SPE in SPI_CTL
SPI_TDBR	SPI port transmit data buffer	Register contents can also be modified by hardware (by DMA and/or when SZ = 1 in SPI_CTL)
SPI_RDBR	SPI port receive data buffer	When register is read, hardware events are triggered
SPI_BAUD	SPI port baud control	Value of 0 or 1 disables the serial clock
SPI_SHADOW	SPI port data	Register has the same contents as SPI_RDBR, but no action is taken when it is read

SPI Transfer Formats

The SPI supports four different combinations of serial clock phase and polarity (SPI modes 0-3). These combinations are selected using the CPOL and CPHA bits in SPI_CTL, as shown in Figure 10-11.

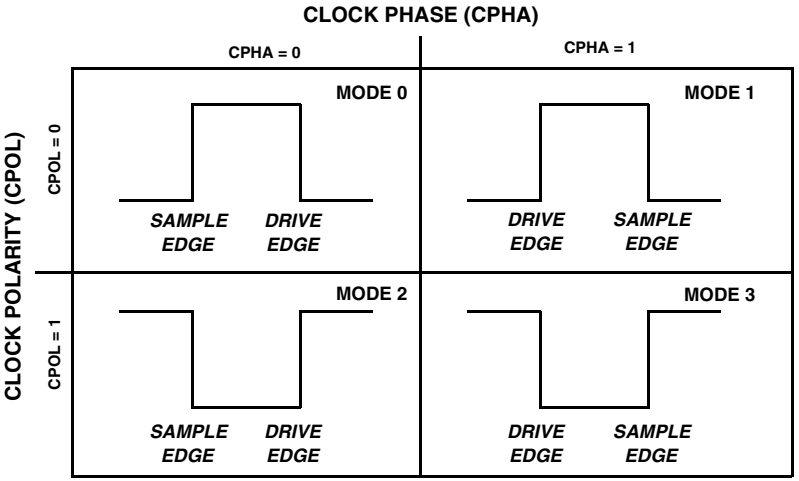


Figure 10-11. SPI Modes of Operation

Figure 10-12 and Figure 10-13 demonstrate the two basic transfer formats as defined by the CPHA bit. Two waveforms are shown for SCK—one for CPOL = 0 and the other for CPOL = 1. The diagrams may be interpreted as master or slave timing diagrams since the SCK, MISO, and MOSI pins are directly connected between the master and the slave. The MISO signal is the output from the slave (slave transmission), and the MOSI signal is the output from the master (master transmission). The SCK signal is generated by the master, and the $\overline{\text{SPISS}}$ signal is the slave device select input to the slave from the master. The diagrams represent an 8-bit transfer (SIZE = 0) with

the Most Significant Bit (MSB) first (LSBF = 0). Any combination of the SIZE and LSBF bits of SPI_CTL is allowed. For example, a 16-bit transfer with the Least Significant Bit (LSB) first is another possible configuration.

The clock polarity and the clock phase should be identical for the master device and the slave device involved in the communication link. The transfer format from the master may be changed between transfers to adjust to various requirements of a slave device.

When CPHA = 0, the slave select line, $\overline{\text{SPISS}}$, must be inactive (high) between each serial transfer. This is controlled automatically by the SPI hardware logic. When CPHA = 1, $\overline{\text{SPISS}}$ may either remain active (low) between successive transfers or be inactive (high). This must be controlled by the software via manipulation of SPI_FLG.

Figure 10-12 shows the SPI transfer protocol for CPHA = 0. Note SCK starts toggling in the middle of the data transfer, SIZE = 0, and LSBF = 0.

Figure 10-13 shows the SPI transfer protocol for CPHA = 1. Note SCK starts toggling at the beginning of the data transfer, SIZE = 0, and LSBF = 0.

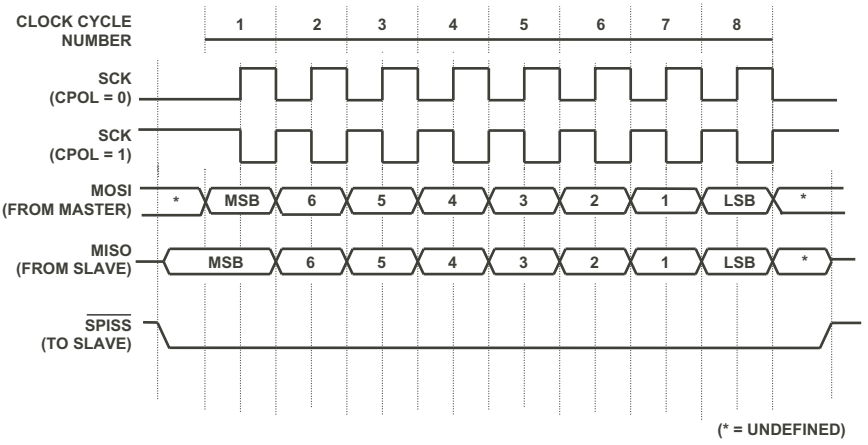


Figure 10-12. SPI Transfer Protocol for CPHA = 0

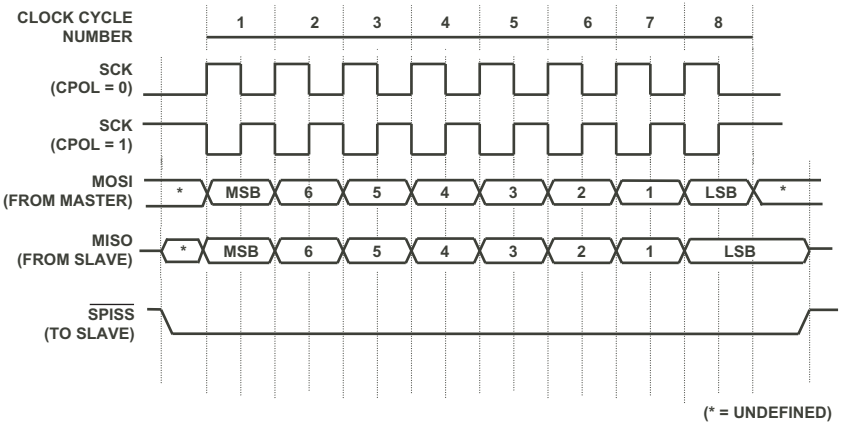


Figure 10-13. SPI Transfer Protocol for CPHA = 1

SPI General Operation

The SPI can be used in a single master as well as multimaster environment. The MOSI, MISO, and the SCK signals are all tied together in both configurations. SPI transmission and reception are always enabled simultaneously, unless the Broadcast mode has been selected. In Broadcast mode, several slaves can be enabled to receive, but only one of the slaves must be in Transmit mode driving the MISO line. If the transmit or receive is not needed, it can simply be ignored. This section describes the clock signals, SPI operation as a master and as a slave, and error generation.

Precautions must be taken to avoid data corruption when changing the SPI module configuration. The configuration must not be changed during a data transfer. The clock polarity should only be changed when no slaves are selected. An exception to this is when an SPI communication link consists of a single master and a single slave, CPHA = 1, and the slave select

input of the slave is always tied low. In this case, the slave is always selected and data corruption can be avoided by enabling the slave only after both the master and slave devices are configured.

In a multimaster or multislave SPI system, the data output pins (MOSI and MISO) can be configured to behave as open drain outputs, which prevents contention and possible damage to pin drivers. An external pull-up resistor is required on both the MOSI and MISO pins when this option is selected.

The WOM bit controls this option. When WOM is set and the SPI is configured as a master, the MOSI pin is three-stated when the data driven out on MOSI is a logic high. The MOSI pin is not three-stated when the driven data is a logic low. Similarly, when WOM is set and the SPI is configured as a slave, the MISO pin is three-stated if the data driven out on MISO is a logic high.

Clock Signals

The SCK signal is a gated clock that is only active during data transfers for the duration of the transferred word. The number of active edges is equal to the number of bits driven on the data lines. The clock rate can be as high as one-fourth of the SCLK rate. For master devices, the clock rate is determined by the 16-bit value of SPI_BAUD. For slave devices, the value in SPI_BAUD is ignored. When the SPI device is a master, SCK is an output signal. When the SPI is a slave, SCK is an input signal. Slave devices ignore the serial clock if the slave select input is driven inactive (high).

The SCK signal is used to shift out and shift in the data driven onto the MISO and MOSI lines. The data is always shifted out on one edge of the clock and sampled on the opposite edge of the clock. Clock polarity and clock phase relative to data are programmable into SPI_CTL and define the transfer format.

Master Mode Operation

When the SPI is configured as a master (and DMA mode is not selected), the interface operates in the following manner.

1. The core writes to SPI_FLG, setting one or more of the SPI Flag Select bits (FLSx). This ensures that the desired slaves are properly deselected while the master is configured.
2. The core writes to the SPI_BAUD and SPI_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and other necessary information.
3. If CPHA = 1, the core activates the desired slaves by clearing one or more of the SPI flag bits (FLGx) of SPI_FLG.
4. The TIMOD bits in SPI_CTL determine the SPI Transfer Initiate mode. The transfer on the SPI link begins upon either a data write by the core to the transmit data buffer (SPI_TDBR) or a data read of the receive data buffer (SPI_RDBR).
5. The SPI then generates the programmed clock pulses on SCK and simultaneously shifts data out of MOSI and shifts data in from MISO. Before a shift, the shift register is loaded with the contents of the SPI_TDBR register. At the end of the transfer, the contents of the shift register are loaded into SPI_RDBR.
6. With each new Transfer Initiate command, the SPI continues to send and receive words, according to the SPI Transfer Initiate mode.

If the transmit buffer remains empty or the receive buffer remains full, the device operates according to the states of the SZ and GM bits in SPI_CTL. If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits 0s on the MOSI pin. One word is transmitted for each new Transfer Initiate command. If SZ = 0 and the transmit buffer is empty, the device

repeatedly transmits the last word it transmitted before the transmit buffer became empty. If GM = 1 and the receive buffer is full, the device continues to receive new data from the MISO pin, overwriting the older data in the SPI_RDBR buffer. If GM = 0 and the receive buffer is full, the incoming data is discarded, and SPI_RDBR is not updated.

Transfer Initiation From Master (Transfer Modes)

When a device is enabled as a master, the initiation of a transfer is defined by the two TIMOD bits of SPI_CTL. Based on those two bits and the status of the interface, a new transfer is started upon either a read of SPI_RDBR or a write to SPI_TDBR. This is summarized in Table 10-4.

If the SPI port is enabled with TIMOD = 01 or TIMOD = 11, the hardware immediately issues a first interrupt or DMA request.

Table 10-4. Transfer Initiation

TIMOD	Function	Transfer Initiated Upon	Action, Interrupt
00	Transmit and Receive	Initiate new single word transfer upon read of SPI_RDBR and previous transfer completed.	Interrupt active when receive buffer is full. Read of SPI_RDBR clears interrupt.
01	Transmit and Receive	Initiate new single word transfer upon write to SPI_TDBR and previous transfer completed.	Interrupt active when transmit buffer is empty. Writing to SPI_TDBR clears interrupt.

Table 10-4. Transfer Initiation (Cont'd)

TIMOD	Function	Transfer Initiated Upon	Action, Interrupt
10	Receive with DMA	Initiate new multiword transfer upon enabling SPI for DMA mode. Individual word transfers begin with a DMA read of SPI_RDBR, and last transfer completed.	Request DMA reads as long as SPI DMA FIFO is not empty.
11	Transmit with DMA	Initiate new multiword transfer upon enabling SPI for DMA mode. Individual word transfers begin with a DMA write to SPI_TDBR, and last transfer completed.	Request DMA writes as long as SPI DMA FIFO is not full.

Slave Mode Operation

When a device is enabled as a slave (and DMA mode is not selected), the start of a transfer is triggered by a transition of the $\overline{\text{SPISS}}$ select signal to the active state (low), or by the first active edge of the clock (SCK), depending on the state of CPHA.

These steps illustrate SPI operation in the Slave mode:

1. The core writes to SPI_CTL to define the mode of the serial link to be the same as the mode setup in the SPI master.
2. To prepare for the data transfer, the core writes data to be transmitted into SPI_TDBR.
3. Once the $\overline{\text{SPISS}}$ falling edge is detected, the slave starts shifting data out on MISO and in from MOSI on SCK edges, depending on the states of CPHA aand CPOL.

4. Reception/transmission continues until SPISS is released or until the slave has received the proper number of clock cycles.
5. The slave device continues to receive/transmit with each new falling edge transition on SPISS and/or SCK clock edge.

If the transmit buffer remains empty or the receive buffer remains full, the device operates according to the states of the SZ and GM bits in SPI_CTL. If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits 0s on the MISO pin. If SZ = 0 and the transmit buffer is empty, it repeatedly transmits the last word it transmitted before the transmit buffer became empty. If GM = 1 and the receive buffer is full, the device continues to receive new data from the MOSI pin, overwriting the older data in SPI_RDBR. If GM = 0 and the receive buffer is full, the incoming data is discarded, and SPI_RDBR is not updated.

Slave Ready for a Transfer

When a device is enabled as a slave, the actions shown in Table 10-5 are necessary to prepare the device for a new transfer.

Table 10-5. Transfer Preparation

TIMOD	Function	Action, Interrupt
00	Transmit and Receive	Interrupt active when receive buffer is full. Read of SPI_RDBR clears interrupt.
01	Transmit and Receive	Interrupt active when transmit buffer is empty. Writing to SPI_TDBR clears interrupt.
10	Receive with DMA	Request DMA reads as long as SPI DMA FIFO is not empty.
11	Transmit with DMA	Request DMA writes as long as SPI DMA FIFO is not full.

Error Signals and Flags

The status of a device is indicated by the SPI_STAT register. See "SPI_STAT Register" on page 10-15 for more information.

Mode Fault Error (MODF)

The MODF bit is set in SPI_STAT when the $\overline{\text{SPISS}}$ input pin of a device enabled as a master is driven low by some other device in the system. This occurs in multimaster systems when another device is also trying to be the master. To enable this feature, the PSSE bit in SPI_CTL must be set. This contention between two drivers can potentially damage the driving pins. As soon as this error is detected, these actions occur:

- The MSTR control bit in SPI_CTL is cleared, configuring the SPI interface as a slave
- The SPE control bit in SPI_CTL is cleared, disabling the SPI system
- The MODF status bit in SPI_STAT is set
- An SPI Error interrupt is generated

These four conditions persist until the MODF bit is cleared by software. Until the MODF bit is cleared, the SPI cannot be re-enabled, even as a slave. Hardware prevents the user from setting either SPE or MSTR while MODF is set.

When MODF is cleared, the interrupt is deactivated. Before attempting to re-enable the SPI as a master, the state of the $\overline{\text{SPISS}}$ input pin should be checked to make sure the pin is high. Otherwise, once SPE and MSTR are set, another Mode Fault Error condition immediately occurs.

When SPE and MSTR are cleared, the SPI data and clock pin drivers (MOSI, MISO, and SCK) are disabled. However, the slave select output pins revert to being controlled by the programmable flag registers. This could lead to

contention on the slave select lines if these lines are still driven by the processor. To ensure that the slave select output drivers are disabled once an MODF error occurs, the program must configure the programmable flag registers appropriately.

When enabling the MODF feature, the program must configure as inputs all of the PFx pins that will be used as slave selects. Programs can do this by configuring the direction of the PFx pins prior to configuring the SPI. This ensures that, once the MODF error occurs and the slave selects are automatically reconfigured as PFx pins, the slave select output drivers are disabled.

Transmission Error (TXE)

The TXE bit is set in SPI_STAT when all the conditions of transmission are met, and there is no new data in SPI_TDBR (SPI_TDBR is empty). In this case, the contents of the transmission depend on the state of the SZ bit in SPI_CTL. The TXE bit is sticky (W1C).

Reception Error (RBSY)

The RBSY flag is set in the SPI_STAT register when a new transfer is completed, but before the previous data can be read from SPI_RDBR. The state of the GM bit in the SPI_CTL register determines whether SPI_RDBR is updated with the newly received data. The RBSY bit is sticky (W1C).

Transmit Collision Error (TXCOL)

The TXCOL flag is set in SPI_STAT when a write to SPI_TDBR coincides with the load of the shift register. The write to SPI_TDBR can be via software or the DMA. The TXCOL bit indicates that corrupt data may have been loaded into the shift register and transmitted. In this case, the data in SPI_TDBR may not match what was transmitted. This error can easily be avoided by proper software control. The TXCOL bit is sticky (W1C).

Beginning and Ending an SPI Transfer

The start and finish of an SPI transfer depend on whether the device is configured as a master or a slave, whether the CPHA mode is selected, and whether the Transfer Initiation mode (TIMOD) is selected. For a master SPI with CPHA = 0, a transfer starts when either SPI_TDBR is written to or SPI_RDBR is read, depending on TIMOD. At the start of the transfer, the enabled slave select outputs are driven active (low). However, the SCK signal remains inactive for the first half of the first cycle of SCK. For a slave with CPHA = 0, the transfer starts as soon as the $\overline{\text{SPISS}}$ input goes low.

For CPHA = 1, a transfer starts with the first active edge of SCK for both slave and master devices. For a master device, a transfer is considered finished after it sends the last data and simultaneously receives the last data bit. A transfer for a slave device ends after the last sampling edge of SCK.

The RXS bit defines when the receive buffer can be read. The TXS bit defines when the transmit buffer can be filled. The end of a single word transfer occurs when the RXS bit is set, indicating that a new word has just been received and latched into the receive buffer, SPI_RDBR. For a master SPI, RXS is set shortly after the last sampling edge of SCK. For a slave SPI, RXS is set shortly after the last SCK edge, regardless of CPHA or CPOL. The latency is typically a few SCLK cycles and is independent of TIMOD and the baud rate. If configured to generate an interrupt when SPI_RDBR is full (TIMOD = 00), the interrupt goes active one SCLK cycle after RXS is set. When not relying on this interrupt, the end of a transfer can be detected by polling the RXS bit.

To maintain software compatibility with other SPI devices, the SPIF bit is also available for polling. This bit may have a slightly different behavior from that of other commercially available devices. For a slave device, SPIF is cleared shortly after the start of a transfer ($\overline{\text{SPISS}}$ going low for CPHA = 0, first active edge of SCK on CPHA = 1), and is set at the same time as RXS. For a master device, SPIF is cleared shortly after the start of a

transfer (either by writing the SPI_TDBR or reading the SPI_RDBR, depending on TIMOD), and is set one-half SCK period after the last SCK edge, regardless of CPHA or CPOL.

The time at which SPIF is set depends on the baud rate. In general, SPIF is set after RXS, but at the lowest baud rate settings (SPI_BAUD < 4). The SPIF bit is set before RXS is set, and consequently before new data is latched into SPI_RDBR, because of the latency. Therefore, for SPI_BAUD = 2 or SPI_BAUD = 3, RXS must be set before SPIF to read SPI_RDBR. For larger SPI_BAUD settings, RXS is guaranteed to be set before SPIF is set.

If the SPI port is used to transmit and receive at the same time, or to switch between receive and transmit operation frequently, then the TIMOD = 00 mode may be the best operation option. In this mode, software performs a dummy read from the SPI_RDBR register to initiate the first transfer. If the first transfer is used for data transmission, software should write the value to be transmitted into the SPI_TDBR register before performing the dummy read. If the transmitted value is arbitrary, it is good practice to set the SZ bit to ensure zero data is transmitted rather than random values. When receiving the last word of an SPI stream, software should ensure that the read from the SPI_RDBR register does not initiate another transfer. It is recommended to disable the SPI port before the final SPI_RDBR read access. Reading the SPI_SHADOW register is not sufficient as it does not clear the interrupt request.

In master mode with the CPHA bit set, software should manually assert the required slave select signal before starting the transaction. After all data has been transferred, software typically releases the slave select again. If the SPI slave device requires the slave select line to be asserted for the complete transfer, this can be done in the SPI interrupt service routine only when operating in TIMOD = 00 or TIMOD = 10 mode. With TIMOD = 01 or TIMOD = 11, the interrupt is requested while the transfer is still in progress.

DMA

The SPI port also can use Direct Memory Access (DMA). For more information on DMA, see "DMA and Memory DMA Registers" on page 9-3.

DMA Functionality

The SPI has a single DMA engine which can be configured to support either an SPI transmit channel or a receive channel, but not both simultaneously. Therefore, when configured as a transmit channel, the received data will essentially be ignored.

When configured as a receive channel, what is transmitted is irrelevant. A 16-bit by four-word FIFO (without burst capability) is included to improve throughput on the DMA Access Bus (DAB).

When using DMA for SPI transmit, the `DMA_DONE` interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll `TXS` in the `SPI_STAT` register until it goes low for 2 successive reads, at which point the SPI DMA FIFO will be empty. When the `SPIF` bit subsequently gets set, the last word has been transferred and the SPI can be disabled or enabled for another mode.

The four-word FIFO is cleared when the SPI port is disabled.

Master Mode DMA Operation

When enabled as a master with the DMA engine configured to transmit or receive data, the SPI interface operates as follows.

1. The processor core writes to the appropriate DMA registers to enable the SPI DMA Channel and to configure the necessary work units, access direction, word count, and so on. For more information, see "DMA and Memory DMA Registers" on page 9-3.

2. The processor core writes to the SPI_FLG register, setting one or more of the SPI flag select bits (FLSx).

3. The processor core writes to the SPI_BAUD and SPI_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and so on. The TIMOD field should be configured to select either "Receive with DMA" (TIMOD = 10) or "Transmit with DMA" (TIMOD = 11) mode.

4. If configured for receive, a receive transfer is initiated upon enabling of the SPI. Subsequent transfers are initiated as the SPI reads data from the SPI_RDBR register and writes to the SPI DMA FIFO. The SPI then requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

   If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. As the SPI writes data from the SPI DMA FIFO into the SPI_TDBR register, it initiates a transfer on the SPI link.

5. The SPI then generates the programmed clock pulses on SCK and simultaneously shifts data out of MOSI and shifts data in from MISO. For receive transfers, the value in the shift register is loaded into

the SPI_RDBR register at the end of the transfer. For transmit transfers, the value in the SPI_TDBR register is loaded into the shift register at the start of the transfer.

6. In Receive mode, as long as there is data in the SPI DMA FIFO (the FIFO is not empty), the SPI continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA Word Count register transitions from 1 to 0. The SPI continues receiving words until SPI DMA mode is disabled.

   In Transmit mode, as long as there is room in the SPI DMA FIFO (the FIFO is not full), the SPI continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI DMA FIFO until the SPI DMA Word Count register transitions from 1 to 0. The SPI continues transmitting words until the SPI DMA FIFO is empty.

For receive DMA operations, if the DMA engine is unable to keep up with the receive datastream, the receive buffer operates according to the state of the GM bit. If GM = 1 and the DMA FIFO is full, the device continues to receive new data from the MISO pin, overwriting the older data in the SPI_RDBR register. If GM = 0, and the DMA FIFO is full, the incoming data is discarded, and the SPI_RDBR register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty (and TXE is set). If SZ = 1, the device repeatedly transmits 0s on the MOSI pin. If SZ = 0, it repeatedly transmits the contents of the SPI_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, the master SPI initiates a word transfer only when there is data in the DMA FIFO. If the DMA FIFO is empty, the SPI waits for the DMA engine to write to the DMA FIFO before starting the transfer. All aspects of SPI receive operation should be ignored when configured in Transmit DMA mode, including the data in the SPI_RDBR register, and the status of the RXS and RBSY bits. The RBSY

overrun conditions cannot generate an error interrupt in this mode. The TXE underrun condition cannot happen in this mode (master DMA TX mode), because the master SPI will not initiate a transfer if there is no data in the DMA FIFO.

Writes to the SPI_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the SPI_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPI_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = 10), or when the DMA FIFO is not full (when TIMOD = 11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = 10).

A master SPI DMA sequence may involve back-to-back transmission and/or reception of multiple DMA work units. The SPI controller supports such a sequence with minimal core interaction.

Slave Mode DMA Operation

When enabled as a slave with the DMA engine configured to transmit or receive data, the start of a transfer is triggered by a transition of the $\overline{\text{SPISS}}$ signal to the active-low state or by the first active edge of SCK, depending on the state of CPHA.

The following steps illustrate the SPI receive or transmit DMA sequence in an SPI slave (in response to a master command).

1. The processor core writes to the appropriate DMA registers to enable the SPI DMA Channel and configure the necessary work units, access direction, word count, and so on. For more information, see "DMA and Memory DMA Registers" on page 9-3.

2. The processor core writes to the SPI_CTL register to define the mode of the serial link to be the same as the mode setup in the SPI master. The TIMOD field will be configured to select either "Receive with DMA" (TIMOD = 10) or "Transmit with DMA" (TIMOD = 11) mode.

3. If configured for receive, once the slave select input is active, the slave starts receiving and transmitting data on SCK edges. The value in the shift register is loaded into the SPI_RDBR register at the end of the transfer. As the SPI reads data from the SPI_RDBR register and writes to the SPI DMA FIFO, it requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

   If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. The SPI then reads data from the SPI DMA FIFO and writes to the SPI_TDBR register, awaiting the start of the next transfer. Once the slave select input is active, the slave starts receiving and transmitting data on SCK edges. The value in the SPI_TDBR register is loaded into the shift register at the start of the transfer.

4. In Receive mode, as long as there is data in the SPI DMA FIFO (FIFO not empty), the SPI slave continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA Word Count register transitions from 1 to 0. The SPI slave continues receiving words on SCK edges as long as the slave select input is active.

   In Transmit mode, as long as there is room in the SPI DMA FIFO (FIFO not full), the SPI slave continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI DMA FIFO until the SPI DMA

Word Count register transitions from 1 to 0. The SPI slave continues transmitting words on SCK edges as long as the slave select input is active.

For receive DMA operations, if the DMA engine is unable to keep up with the receive datastream, the receive buffer operates according to the state of the GM bit. If GM = 1 and the DMA FIFO is full, the device continues to receive new data from the MOSI pin, overwriting the older data in the SPI_RDBR register. If GM = 0 and the DMA FIFO is full, the incoming data is discarded, and the SPI_RDBR register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty and TXE is set. If SZ = 1, the device repeatedly transmits 0s on the MISO pin. If SZ = 0, it repeatedly transmits the contents of the SPI_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, if the DMA engine is unable to keep up with the transmit stream, the transmit port operates according to the state of the SZ bit. If SZ = 1 and the DMA FIFO is empty, the device repeatedly transmits 0s on the MISO pin. If SZ = 0 and the DMA FIFO is empty, it repeatedly transmits the last word it transmitted before the DMA buffer became empty. All aspects of SPI receive operation should be ignored when configured in Transmit DMA mode, including the data in the SPI_RDBR register, and the status of the RXS and RBSY bits. The RBSY overrun conditions cannot generate an error interrupt in this mode.

Writes to the SPI_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the SPI_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPI_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = 10), or when the DMA FIFO is not full (when TIMOD = 11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = 10), or when there is a TXE underflow error condition (when TIMOD = 11).

Timing

The enable lead time (T1), the enable lag time (T2), and the sequential transfer delay time (T3) each must always be greater than or equal to one-half the SCK period. See Figure 10-14. The minimum time between successive word transfers (T4) is two SCK periods. This is measured from the last active edge of SCK of one word to the first active edge of SCK of the next word. This is independent of the configuration of the SPI (CPHA, MSTR, and so on).

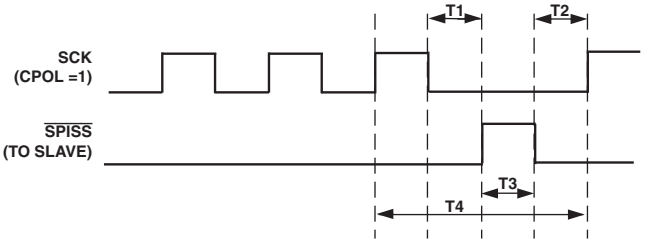


Figure 10-14. SPI Timing

For a master device with CPHA = 0, the slave select output will be inactive (high) for at least one-half the SCK period. In this case, T1 and T2 will each always be equal to one-half the SCK period.

11 PARALLEL PERIPHERAL INTERFACE

The Parallel Peripheral Interface (PPI) is a half-duplex, bidirectional port accommodating up to 16 bits of data. It has a dedicated clock pin, three multiplexed frame sync pins, and four dedicated data pins. Up to 12 additional data pins are available by reconfiguring the PF pins. The highest system throughput is achieved with 8-bit data, since two 8-bit data samples can be packed as a single 16-bit word. In such a case, the earlier sample is placed in the 8 least significant bits (LSBs).

The PPI_CLK pin can accept an external clock input up to SCLK/2. It cannot source a clock internally. Table 11-1 shows the pin interface for the PPI.

If a programmable flag pin is configured for PPI use, its bit position in programmable flag MMRs will read back as 0.

Table 11-1. PPI Pins

Signal Name	Function	Direction	Alternate Function
PPI15	Data	Bidirectional	PF4, SPI Enable Output
PPI14	Data	Bidirectional	PF5, SPI Enable Output
PPI13	Data	Bidirectional	PF6, SPI Enable Output
PPI12	Data	Bidirectional	PF7, SPI Enable Output
PPI11	Data	Bidirectional	PF8
PPI10	Data	Bidirectional	PF9
PPI9	Data	Bidirectional	PF10
PPI8	Data	Bidirectional	PF11

Table 11-1. PPI Pins (Cont'd)

Signal Name	Function	Direction	Alternate Function
PPI7	Data	Bidirectional	PF12
PPI6	Data	Bidirectional	PF13
PPI5	Data	Bidirectional	PF14
PPI4	Data	Bidirectional	PF15
PPI3	Data	Bidirectional	N/A
PPI2	Data	Bidirectional	N/A
PPI1	Data	Bidirectional	N/A
PPI0	Data	Bidirectional	N/A
PPI_FS3	Frame Sync3/Field	Bidirectional	PF3, SPI Enable Output
PPI_FS2	Frame Sync2/VSYNC	Bidirectional	Timer 2
PPI_FS1	Frame Sync1/HSYNC	Bidirectional	Timer 1
PPI_CLK	Up to SCLK/2	Input Clock	N/A

PPI Registers

The PPI has five memory-mapped registers (MMRs) that regulate its operation. These registers are the PPI Control register (PPI_CONTROL), the PPI Status register (PPI_STATUS), the Delay Count register (PPI_DELAY), the Transfer Count register (PPI_COUNT), and the Lines Per Frame register (PPI_FRAME).

Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

PPI_CONTROL Register

The PPI Control register (PPI_CONTROL) configures the PPI for operating mode, control signal polarities, and data width of the port. See Figure 11-1 for a bit diagram of this MMR.

The POLC and POLS bits allow for selective signal inversion of the PPI_CLK and PPI_FS1/PPI_FS2 signals, respectively. This provides a mechanism to connect to data sources and receivers with a wide array of control signal polarities. Often, the remote data source/receiver also offers configurable signal polarities, so the POLC and POLS bits simply add increased flexibility.

The DLEN[2:0] field is programmed to specify the width of the PPI port in any mode. Note any width from 8 to 16 bits is supported, with the exception of a 9-bit port width. Any PF pins that are unused by the PPI as a result of the DLEN setting are free to be used in their normal PF capacity.

In ITU-R 656 modes, the DLEN field should not be configured for anything greater than a 10-bit port width. If it is, the PPI will reserve extra pins, making them unusable by other peripherals.

The SKIP_EN bit, when set, enables the selective skipping of data elements being read in through the PPI. By ignoring data elements, the PPI is able to conserve DMA bandwidth.

When the SKIP_EN bit is set, the SKIP_EO bit allows the PPI to ignore either the odd or the even elements in an input datastream. This is useful, for instance, when reading in a color video signal in YCbCr format (Cb, Y, Cr, Y, Cb, Y, Cr, Y...). Skipping every other element allows the PPI to only read in the luma (Y) or chroma (Cr or Cb) values. This could also be useful when synchronizing two processors to the same incoming video stream. One processor could handle luma processing and the other (whose SKIP_EO bit is set differently from the first processor's) could handle chroma processing. This skipping feature is valid in ITU-R 656 modes and RX modes with external frame syncs.

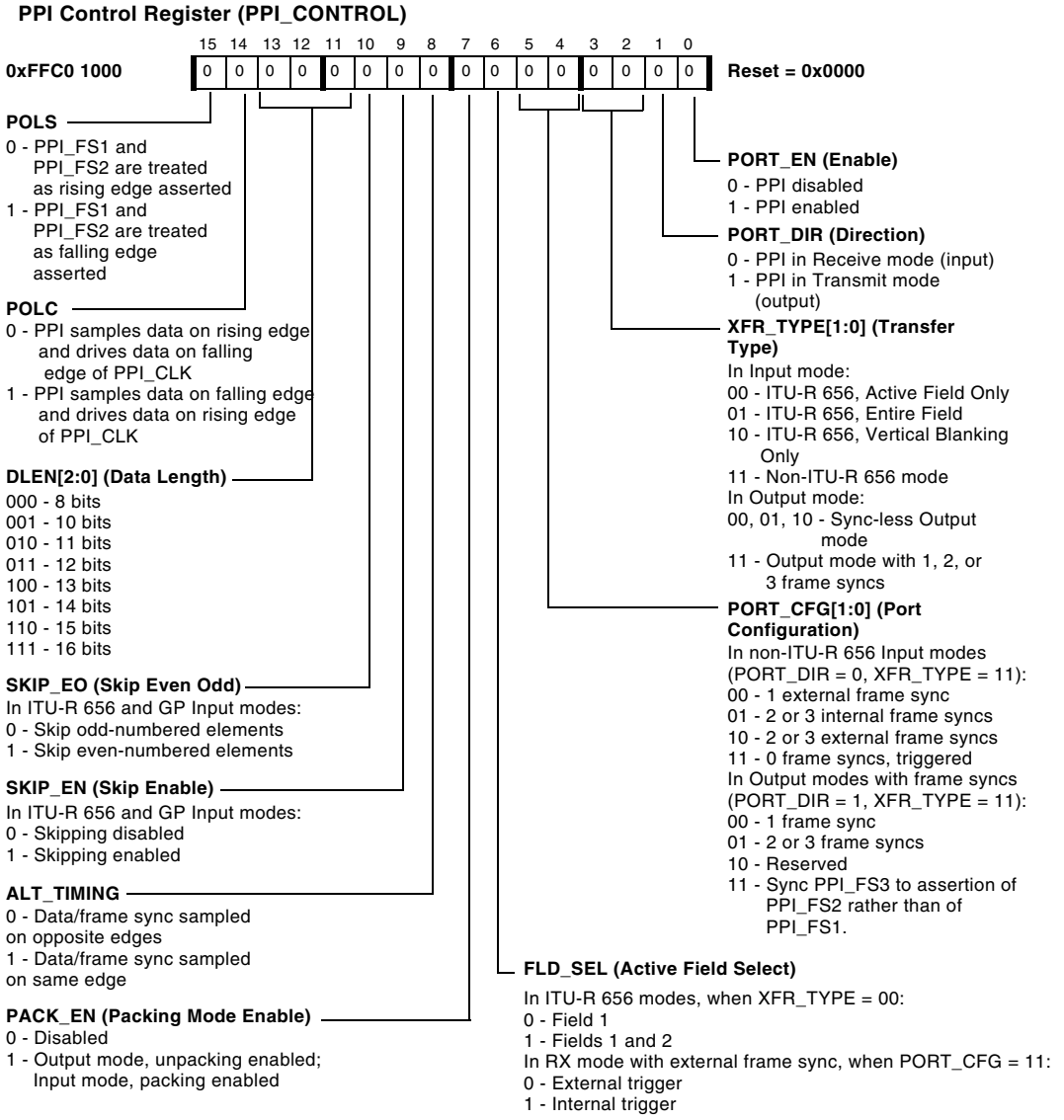


Figure 11-1. PPI Control Register

The ALT_TIMING bit provides the capability to have the frame sync and data pins sampled on the same PPI clock edge, rather than on opposite PPI clock edges, which is the default behavior. The PACK_EN bit only has meaning when the PPI port width (selected by DLEN[2:0]) is 8 bits. Every PPI_CLK-initiated event on the DMA bus (that is, an input or output operation) handles 16-bit entities. In other words, an input port width of 10 bits still results in a 16-bit input word for every PPI_CLK; the upper 6 bits are 0s. Likewise, a port width of 8 bits also results in a 16-bit input word, with the upper 8 bits all 0s. In the case of 8-bit data, it is usually more efficient to pack this information so that there are two bytes of data for every 16-bit word. This is the function of the PACK_EN bit. When set, it enables packing for all RX modes.

Consider this data transported into the PPI via DMA:

0xCE, 0xFA, 0xFE, 0xCA....

- With PACK_EN set:

  This is read into the PPI, configured for an 8-bit port width:

  ```
  0xCE, 0xFA, 0xFE, 0xCA...
  ```

  This is transferred onto the DMA bus:

  ```
  0xFACE, 0xCAFE, ...
  ```

- With PACK_EN cleared:

  This is read into the PPI:

  ```
  0xCE, 0xFA, 0xFE, 0xCA, ...
  ```

  This is transferred onto the DMA bus:

  ```
  0x00CE, 0x00FA, 0x00FE, 0x00CA, ...
  ```

For TX modes, setting PACK_EN enables unpacking of bytes. Consider this data in memory, to be transported out through the PPI via DMA:

0xFACE CAFE.... (0xFA and 0xCA are the two Most Significant Bits (MSBs) of their respective 16-bit words)

- With PACK_EN set:

  This is DMAed to the PPI:

  ```
  0xFACE, 0xCAFE, ...
  ```

  This is transferred out through the PPI, configured for an 8-bit port width (note LSBs are transferred first):

  ```
  0xCE, 0xFA, 0xFE, 0xCA, ...
  ```

- With PACK_EN cleared:

  This is DMAed to the PPI:

  ```
  0xFACE, 0xCAFE, ...
  ```

  This is transferred out through the PPI, configured for an 8-bit port width:

  ```
  0xCE, 0xFE, ...
  ```

The FLD_SEL bit is used primarily in the Active Field Only ITU-R 656 mode. The FLD_SEL bit determines whether to transfer in only Field 1 of each video frame, or both Fields 1 and 2. Thus, it allows a savings in DMA bandwidth by transferring only every other field of active video.

The PORT_CFG[1:0] field is used to configure the operating mode of the PPI. It operates in conjunction with the PORT_DIR bit, which sets the direction of data transfer for the port. The XFR_TYPE[1:0] field is also used to configure operating mode and is discussed below. See Table 11-2 for the possible operating modes for the PPI.

Table 11-2. PPI Possible Operating Modes

PPI Mode	# of Syncs	PORT_DIR	PORT_CFG	XFR_TYPE	POLC	POLS	FLD_SEL
RX mode, 0 frame syncs, external trigger	0	0	11	11	0 or 1	0 or 1	0
RX mode, 0 frame syncs, internal trigger	0	0	11	11	0 or 1	0 or 1	1
RX mode, 1 external frame sync	1	0	00	11	0 or 1	0 or 1	X
RX mode, 2 or 3 external frame syncs	3	0	10	11	0 or 1	0 or 1	X
RX mode, 2 or 3 internal frame syncs	3	0	01	11	0 or 1	0 or 1	X
RX mode, ITU-R 656, Active Field Only	embed-ded	0	XX	00	0 or 1	0	0 or 1
RX mode, ITU-R 656, Vertical Blanking Only	embed-ded	0	XX	10	0 or 1	0	X
RX mode, ITU-R 656, Entire Field	embed-ded	0	XX	01	0 or 1	0	X
TX mode, 0 frame syncs	0	1	XX	00, 01, 10	0 or 1	0 or 1	X
TX mode, 1 internal or external frame sync	1	1	00	11	0 or 1	0 or 1	X
TX mode, 2 external frame syncs	2	1	01	11	0 or 1	0 or 1	X
TX mode, 2 or 3 internal frame syncs, FS3 sync'd to FS1 assertion	3	1	01	11	0 or 1	0 or 1	X
TX mode, 2 or 3 internal frame syncs, FS3 sync'd to FS2 assertion	3	1	11	11	0 or 1	0 or 1	X

The XFR_TYPE[1:0] field configures the PPI for various modes of operation. Refer to Table 11-2 to see how XFR_TYPE[1:0] interacts with other bits in PPI_CONTROL to determine the PPI operating mode.

The PORT_EN bit, when set, enables the PPI for operation.

Note that, when configured as an input port, the PPI does not start data transfer after being enabled until the appropriate synchronization signals are received. If configured as an output port, transfer (including the appropriate synchronization signals) begins as soon as the frame syncs (Timer units) are enabled, so all frame syncs must be configured before this happens. Refer to the section "Frame Synchronization in GP Modes" on page 11-28 for more information.

PPI_STATUS Register

The PPI Status register (PPI_STATUS) contains bits that provide information about the current operating state of the PPI.

The ERR_DET bit is a sticky bit that denotes whether or not an error was detected in the ITU-R 656 control word preamble. The bit is valid only in ITU-R 656 modes. If ERR_DET = 1, an error was detected in the preamble. If ERR_DET = 0, no error was detected in the preamble.

The ERR_NCOR bit is sticky and is relevant only in ITU-R 656 modes. If ERR_NCOR = 0 and ERR_DET = 1, all preamble errors that have occurred have been corrected. If ERR_NCOR = 1, an error in the preamble was detected but not corrected. This situation generates a PPI Error interrupt, unless this condition is masked off in the SIC_IMASK register.

The FT_ERR bit is sticky and indicates, when set, that a Frame Track Error has occurred. It is valid for RX modes only. In this condition, the programmed number of lines per frame in PPI_FRAME does not match up with

the “frame start detect” condition (see the information note on page 11-12). A Frame Track Error generates a PPI Error interrupt, unless this condition is masked off in the SIC_IMASK register.

The FLD bit is set or cleared at the same time as the change in state of F (in ITU-R 656 modes) or PPI_FS3 (in other RX modes). It is valid for Input modes only. The state of FLD reflects the current state of the F or PPI_FS3 signals. In other words, the FLD bit always reflects the current video field being processed by the PPI.

The OVR bit is sticky and indicates, when set, that the PPI FIFO has overflowed and can accept no more data. A FIFO Overflow Error generates a PPI Error interrupt, unless this condition is masked off in the SIC_IMASK register.


The PPI FIFO is 16 bits wide and has 16 entries.

The UNDR bit is sticky and indicates, when set, that the PPI FIFO has underrun and is data-starved. A FIFO Underrun Error generates a PPI Error interrupt, unless this condition is masked off in the SIC_IMASK register.

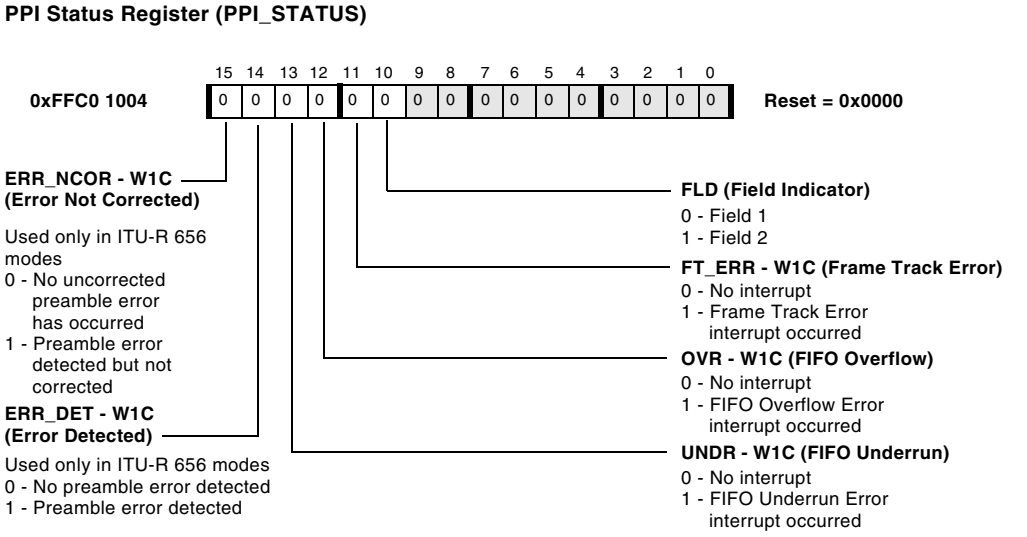


Figure 11-2. PPI Status Register

PPI_DELAY Register

The Delay Count register (PPI_DELAY) can be used in all configurations except ITU-R 656 modes and GP modes with 0 frame syncs. It contains a count of how many PPI_CLK cycles to delay after assertion of PPI_FS1 before starting to read in or write out data.

Note in TX modes using at least one frame sync, there is a one-cycle delay beyond what is specified in the PPI_DELAY register.

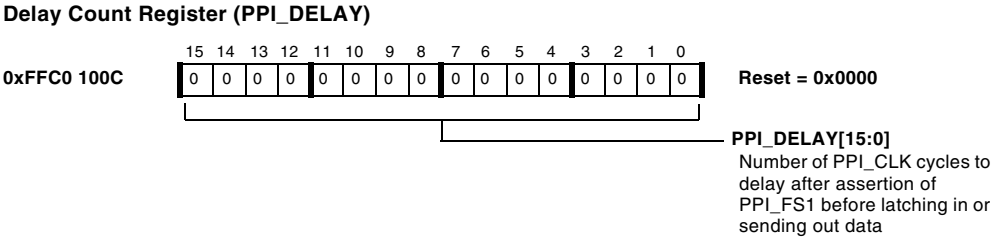


Figure 11-3. Delay Count Register

PPI_COUNT Register

The Transfer Count register (PPI_COUNT) is used only in cases where recurring hardware frame syncs (either externally or internally generated) are involved. It is not needed in ITU-R 656 modes or modes with 0 frame syncs. For RX modes, this register holds the number of samples to read into the PPI per line, minus one. For TX modes, it holds the number of samples to write out through the PPI per line, minus one. The register itself does not actually decrement with each transfer. Thus, at the beginning of a new line of data, there is no need to rewrite the value of this register. For example, to receive or transmit 100 samples through the PPI, set PPI_COUNT to 99.

Take care to ensure that the number of samples programmed into PPI_COUNT is in keeping with the number of samples expected during the "horizontal" interval specified by PPI_FS1.

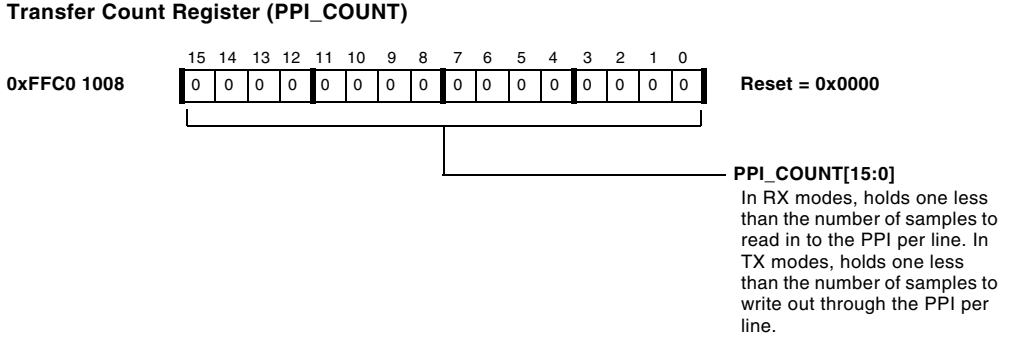


Figure 11-4. Transfer Count Register

PPI_FRAME Register

The Lines Per Frame (PPI_FRAME) register is used in all TX and RX modes with external frame syncs. For ITU-R 656 modes, this register holds the number of lines expected per frame of data, where a frame is defined as Field 1 and Field 2 combined, designated by the F indicator in the ITU-R stream. Here, a line is defined as a complete ITU-R 656 SAV-EAV cycle.

For non-ITU-R 656 modes with external frame syncs, a frame is defined as the data bounded between PPI_FS2 assertions, regardless of the state of PPI_FS3. A line is defined as a complete PPI_FS1 cycle. In these modes, PPI_FS3 is used only to determine the original "frame start" each time the PPI is enabled. It is ignored on every subsequent field and frame, and its state (high or low) is not important except during the original frame start.

If the start of a new frame (or field, for ITU-R 656 mode) is detected before the number of lines specified by PPI_FRAME have been transferred, a Frame Track Error results, and the FT_ERR bit in PPI_STATUS is set. However, the PPI still automatically reinitializes to count to the value programmed in PPI_FRAME, and data transfer continues.

In ITU-R 656 modes, a frame start detect happens on the falling edge of F, the Field indicator. This occurs at the start of Field 1.

In RX mode with 3 external frame syncs, a frame start detect refers to a condition where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This occurs at the start of Field 1.

Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

When using RX mode with 3 external frame syncs, and only 2 syncs are needed, configure the PPI for three-frame-sync operation and provide an external pull-down to GND for the PPI_FS3 pin.

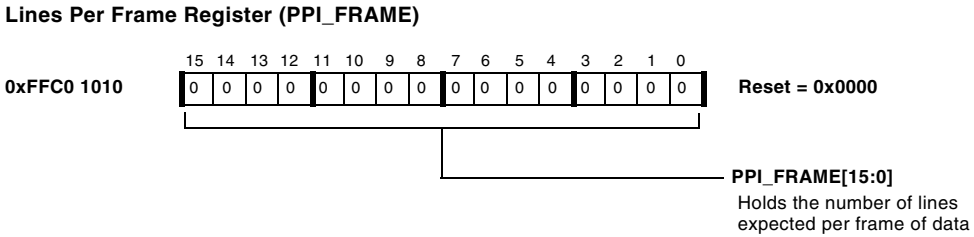


Figure 11-5. Lines Per Frame Register

ITU-R 656 Modes

The PPI supports three input modes for ITU-R 656-framed data. These modes are described in this section. Although the PPI does not explicitly support an ITU-R 656 output mode, recommendations for using the PPI for this situation are provided as well.

ITU-R 656 Background

According to the ITU-R 656 recommendation (formerly known as CCIR-656), a digital video stream has the characteristics shown in Figure 11-6 and Figure 11-7 for 525/60 (NTSC) and 625/50 (PAL) systems. The processor supports only the Bit-parallel mode of ITU-R 656. Both 8- and 10-bit video element widths are supported.

In this mode, the Horizontal (H), Vertical (V), and Field (F) signals are sent as an embedded part of the video datastream in a series of bytes that form a control word. The Start of Active Video (SAV) and End of Active Video (EAV) signals indicate the beginning and end of data elements to read in on each line. SAV occurs on a 1-to-0 transition of H, and EAV begins on a 0-to-1 transition of H. An entire field of video is comprised of Active Video + Horizontal Blanking (the space between an EAV and SAV code) and Vertical Blanking (the space where V = 1). A field of video

commences on a transition of the F bit. The “odd field” is denoted by a value of F = 0, whereas F = 1 denotes an even field. Progressive video makes no distinction between Field 1 and Field 2, whereas interlaced video requires each field to be handled uniquely, because alternate rows of each field combine to create the actual video image.

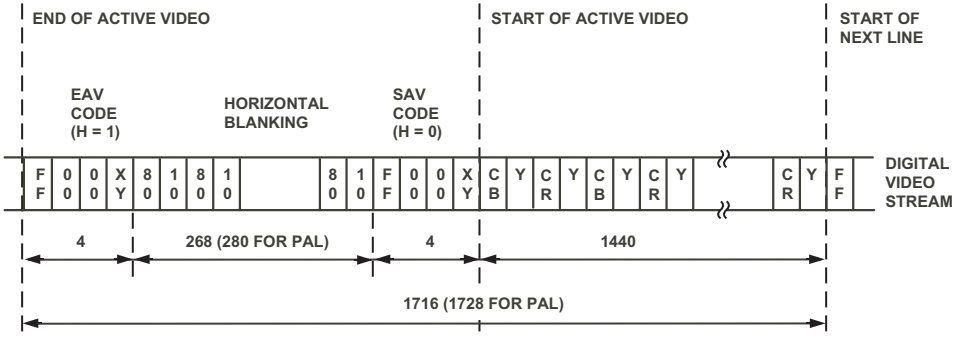
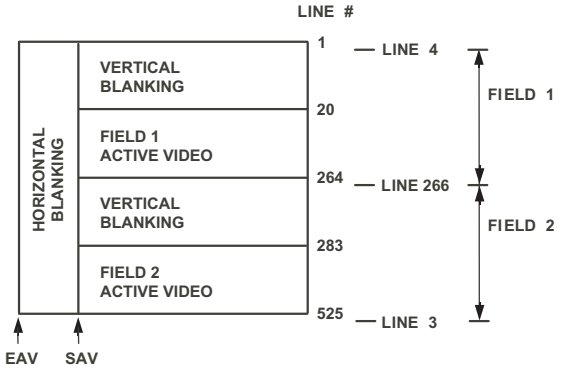
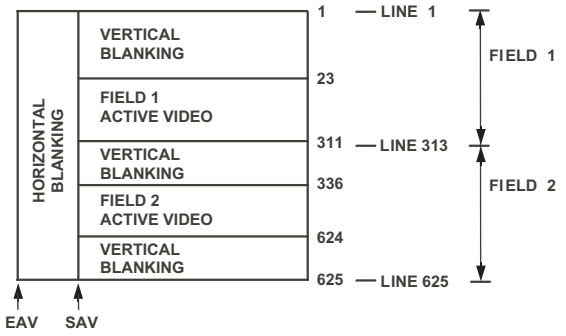


Figure 11-6. ITU-R 656 8-Bit Parallel Data Stream for NTSC (PAL) Systems



LINE NUMBER	F	V	H (EAV)	H (SAV)
1-3, 266-282	1	1	1	0
4-19, 264-265	0	1	1	0
20-263	0	0	1	0
283-525	1	0	1	0



LINE NUMBER	F	V	H (EAV)	H (SAV)
1-22, 311-312	0	1	1	0
23-310	0	0	1	0
313-335, 624-625	1	1	1	0
336-623	1	0	1	0

Figure 11-7. Typical Video Frame Partitioning for NTSC/PAL Systems for ITU-R BT.656-4

The SAV and EAV codes are shown in more detail in Table 11-3. Note there is a defined preamble of three bytes (0xFF, 0x00, 0x00), followed by the XY Status word, which, aside from the `F` (Field), `V` (Vertical Blanking) and `H` (Horizontal Blanking) bits, contains four protection bits for single-bit error detection and correction. Note `F` and `V` are only allowed to change as part of EAV sequences (that is, transition from `H = 0` to `H = 1`). The bit definitions are as follows:

- `F = 0` for Field 1
- `F = 1` for Field 2

- V = 1 during Vertical Blanking
- V = 0 when not in Vertical Blanking
- H = 0 at SAV
- H = 1 at EAV
- P3 = V XOR H
- P2 = F XOR H
- P1 = F XOR V
- P0 = F XOR V XOR H

In many applications, video streams other than the standard NTSC/PAL formats (for example, CIF, QCIF) can be employed. Because of this, the processor interface is flexible enough to accommodate different row and field lengths. In general, as long as the incoming video has the proper EAV/SAV codes, the PPI can read it in. In other words, a CIF image could be formatted to be "656-compliant," where EAV and SAV values define the range of the image for each line, and the V and F codes can be used to delimit fields and frames.

Table 11-3. Control Byte Sequences for 8-Bit and 10-Bit ITU-R 656 Video

	8-bit Data								10-bit Data	
	D9 (MSB)	D8	D7	D6	D5	D4	D3	D2	D1	D0
Preamble	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
Control Byte	1	F	V	H	P3	P2	P1	P0	0	0

ITU-R 656 Input Modes

Figure 11-8 shows a general illustration of data movement in the ITU-R 656 input modes. In the figure, the clock `CLK` is either provided by the video source or supplied externally by the system.

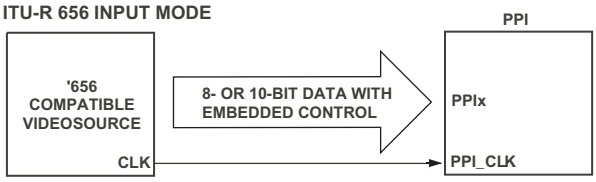


Figure 11-8. ITU-R 656 Input Modes

There are three submodes supported for ITU-R 656 inputs: Entire Field, Active Video Only, and Vertical Blanking Interval Only. Figure 11-9 shows these three submodes.

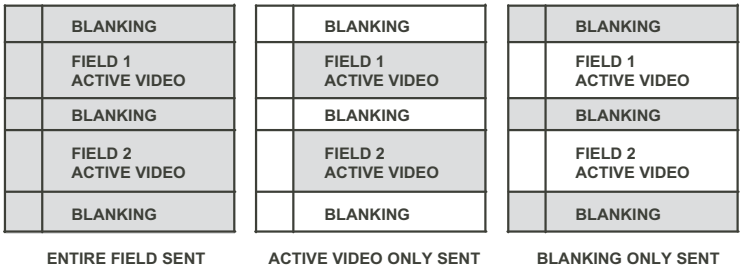


Figure 11-9. ITU-R 656 Input Submodes

Entire Field

In this mode, the entire incoming bitstream is read in through the PPI. This includes Active Video as well as control byte sequences and ancillary data that may be embedded in Horizontal and Vertical Blanking Intervals. Data transfer starts immediately after synchronization to Field 1 occurs, but does not include the first EAV code that contains the F = 0 assignment.

Note the first line transferred in after enabling the PPI will be missing its first 4-byte preamble. However, subsequent lines and frames should have all control codes intact.

One side benefit of this mode is that it enables a "loopback" feature through which a frame or two of data can be read in through the PPI and subsequently output to a compatible video display device. Of course, this requires multiplexing on the PPI pins, but it enables a convenient way to verify that 656 data can be read into and written out from the PPI.

Active Video Only

This mode is used when only the active video portion of a field is of interest, and not any of the blanking intervals. The PPI ignores (does not read in) all data between EAV and SAV, as well as all data present when V = 1. In this mode, the control byte sequences are not stored to memory; they are filtered out by the PPI. After synchronizing to the start of Field 1, the PPI ignores incoming samples until it sees an SAV.

In this mode, the user specifies the number of total (active plus vertical blanking) lines per frame in the `PPI_FRAME` MMR.

Vertical Blanking Interval (VBI) Only

In this mode, data transfer is only active while V = 1 is in the control byte sequence. This indicates that the video source is in the midst of the Vertical Blanking Interval (VBI), which is sometimes used for ancillary data

transmission. The ITU-R 656 recommendation specifies the format for these ancillary data packets, but the PPI is not equipped to decode the packets themselves. This task must be handled in software. Horizontal blanking data is logged where it coincides with the rows of the VBI. Control byte sequence information is always logged. The user specifies the number of total lines (Active plus Vertical Blanking) per frame in the `PPI_FRAME` MMR.

Note the VBI is split into two regions within each field. From the PPI's standpoint, it considers these two separate regions as one contiguous space. However, keep in mind that frame synchronization begins at the start of Field 1, which doesn't necessarily correspond to the start of Vertical Blanking. For instance, in 525/60 systems, the start of Field 1 (F = 0) corresponds to Line 4 of the VBI.

ITU-R 656 Output Mode

The PPI does not explicitly provide functionality for framing an ITU-R 656 output stream with proper preambles and blanking intervals. However, with the TX mode with 0 frame syncs, this process can be supported manually. Essentially, this mode provides a streaming operation from memory out through the PPI. Data and control codes can be set up in memory prior to sending out the video stream. With the 2D DMA engine, this could be performed in a number of ways. For instance, one line of blanking (H + V) could be stored in a buffer and sent out N times by the DMA controller when appropriate, before proceeding to DMA active video. Alternatively, one entire field (with control codes and blanking) can be set up statically in a buffer while the DMA engine transfers only the active video region into the buffer, on a frame-by-frame basis.

Frame Synchronization in ITU-R 656 Modes

Synchronization in ITU-R 656 modes always occurs at the falling edge of F, the field indicator. This corresponds to the start of Field 1. Consequently, up to two fields might be ignored (for example, if Field 1 just started before the PPI-to-camera channel was established) before data is received into the PPI.

Because all H and V signalling is embedded in the datastream in ITU-R 656 modes, the PPI_COUNT register is not necessary. However, the PPI_FRAME register is used in order to check for synchronization errors. The user programs this MMR for the number of lines expected in each frame of video, and the PPI keeps track of the number of EAV-to-SAV transitions that occur from the start of a frame until it decodes the end-of-frame condition (transition from F = 1 to F = 0). At this time, the actual number of lines processed is compared against the value in PPI_FRAME. If there is a mismatch, the FT_ERR bit in the PPI_STATUS register is asserted. For instance, if an SAV transition is missed, the current field will only have NUM_ROWS - 1 rows, but resynchronization will reoccur at the start of the next frame.

Upon completing reception of an entire field, the Field Status bit is toggled in the PPI_STATUS register. This way, an interrupt service routine (ISR) can discern which field was just read in.

General-Purpose PPI Modes

The General-Purpose (GP) PPI modes are intended to suit a wide variety of data capture and transmission applications. Table 11-4 summarizes these modes. If a particular mode shows a given PPI_FSx frame sync not being used, this implies that the pin is available for its alternate, multiplexed processor function (that is, as a timer or flag pin). The exception to

this is that when the PPI is configured for a 2-frame-sync mode, PPI_FS3 cannot be used as a general-purpose flag, even though it is not used by the PPI.

Table 11-4. General-Purpose PPI Modes

GP PPI Mode	PPI_FS1 Direction	PPI_FS2 Direction	PPI_FS3 Direction	Data Direction
RX mode, 0 frame syncs, external trigger	Input	Not used	Not used	Input
RX mode, 0 frame syncs, internal trigger	Not used	Not used	Not used	Input
RX mode, 1 external frame sync	Input	Not used	Not used	Input
RX mode, 2 or 3 external frame syncs	Input	Input	Input	Input
RX mode, 2 or 3 internal frame syncs	Output	Output	Output	Input
TX mode, 0 frame syncs	Not used	Not used	Not used	Output
TX mode, 1 external frame sync	Input	Not used	Not used	Output
TX mode, 2 external frame syncs	Input	Input	Output	Output
TX mode, 1 internal frame sync	Output	Not used	Not used	Output
TX mode, 2 or 3 internal frame syncs	Output	Output	Output	Output

Figure 11-10 illustrates the general flow of the GP modes. The top of the diagram shows an example of RX mode with 1 external frame sync. After the PPI receives the hardware frame sync pulse (PPI_FS1), it delays for the duration of the PPI_CLK cycles programmed into PPI_DELAY. The DMA controller then transfers in the number of samples specified by PPI_COUNT. Every sample that arrives after this, but before the next PPI_FS1 frame sync arrives, is ignored and not transferred onto the DMA bus.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been read in, the sample counter reinitializes to 0 and starts to count up to PPI_COUNT again. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

The bottom of Figure 11-10 shows an example of TX mode, 1 internal frame sync. After PPI_FS1 is asserted, there is a latency of 1 PPI_CLK cycle, and then there is a delay for the number of PPI_CLK cycles programmed into PPI_DELAY. Next, the DMA controller transfers out the number of samples specified by PPI_COUNT. No further DMA takes place until the next PPI_FS1 sync and programmed delay occur.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been transferred out, the sync has priority and starts a new line transfer sequence. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

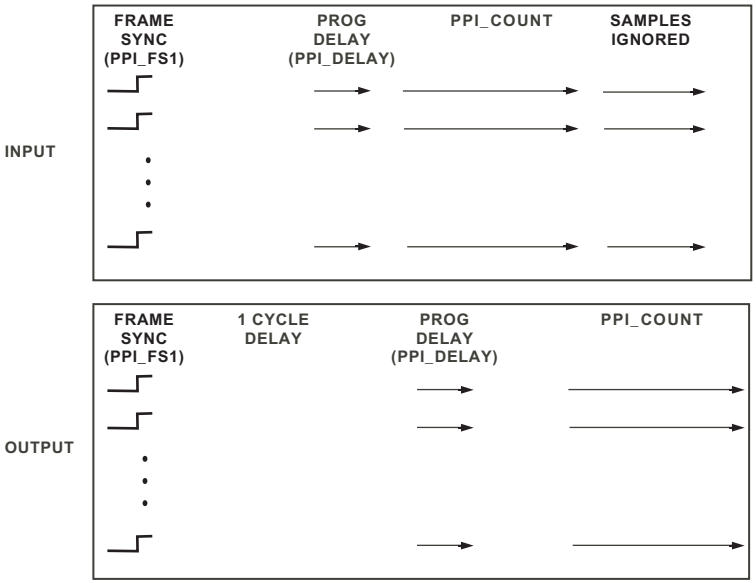


Figure 11-10. General Flow for GP Modes (Assumes Positive Assertion of PPI_FS1)

Data Input (RX) Modes

The PPI supports several modes for data input. These modes differ chiefly by the way the data is framed. Refer to Table 11-2 for information on how to configure the PPI for each mode.

No Frame Syncs

These modes cover the set of applications where periodic frame syncs are not generated to frame the incoming data. There are two options for starting the data transfer, both configured by the PPI_CONTROL register.

- External trigger: An external source sends a single frame sync (tied to PPI_FS1) at the start of the transaction, when FLD_SEL = 0 and PORT_CFG = b#11.
- Internal trigger: Software initiates the process by setting PORT_EN = 1 with FLD_SEL = 1 and PORT_CFG = b#11.

All subsequent data manipulation is handled via DMA. For example, an arrangement could be set up between alternating 1K memory buffers. When one fills up, DMA continues with the second buffer, at the same time that another DMA operation is clearing the first memory buffer for reuse.

Due to clock domain synchronization in RX modes with no frame syncs, there may be a delay of at least 2 PPI_CLK cycles between when the mode is enabled and when valid data is received. Therefore, detection of the start of valid data should be managed by software.

1, 2, or 3 External Frame Syncs

The 1-sync mode is intended for analog-to-digital converter (ADC) applications. The top part of Figure 11-11 shows a typical illustration of the system setup for this mode.

The 3-sync mode shown at the bottom of Figure 11-11 supports video applications that use hardware signalling (HSYNC, VSYNC, FIELD) in accordance with the ITU-R 601 recommendation. The mapping for the frame syncs in this mode is PPI_FS1 = HSYNC, PPI_FS2 = VSYNC, PPI_FS3 = FIELD. Please refer to "Frame Synchronization in GP Modes" on page 11-28 for more information about frame syncs in this mode.

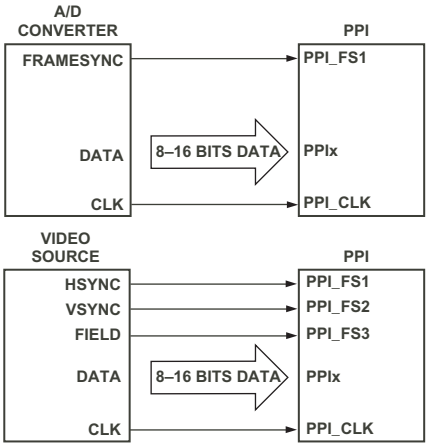


Figure 11-11. RX Mode, External Frame Syncs

A 2-sync mode is implicitly supported by pulling PPI_FS3 to GND by an external resistor when configured in 3-sync mode.

2 or 3 Internal Frame Syncs

This mode can be useful for interfacing to video sources that can be slaved to a master processor. In other words, the processor controls when to read from the video source by asserting PPI_FS1 and PPI_FS2, and then reading data into the PPI. The PPI_FS3 frame sync provides an indication of which field is currently being transferred, but since it is an output, it can simply be left floating if not used. Figure 11-12 shows a sample application for this mode.

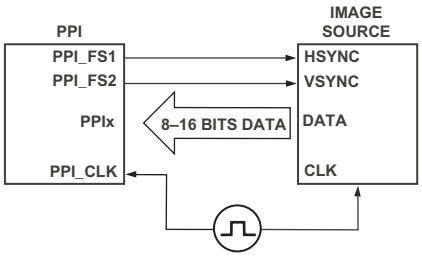


Figure 11-12. RX Mode, Internal Frame Syncs

Data Output (TX) Modes

The PPI supports several modes for data output. These modes differ chiefly by the way the data is framed. Refer to Table 11-2 for information on how to configure the PPI for each mode.

No Frame Syncs

In this mode, data blocks specified by the DMA controller are sent out through the PPI with no framing. That is, once the DMA channel is configured and enabled, and the PPI is configured and enabled, data transfers will take place immediately, synchronized to PPI_CLK. See Figure 11-13 for an illustration of this mode.

In this mode, there is a delay of up to 16 SCLK cycles (for > 8-bit data) or 32 SCLK cycles (for 8-bit data) between enabling the PPI and transmission of valid data. Furthermore, DMA must be configured to transmit at least 16 samples (for > 8-bit data) or 32 samples (for 8-bit data).

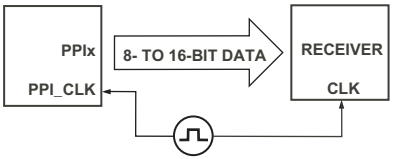


Figure 11-13. TX Mode, 0 Frame Syncs

1 or 2 External Frame Syncs

In these modes, an external receiver can frame data sent from the PPI. Both 1-sync and 2-sync modes are supported. The top diagram in Figure 11-14 shows the 1-sync case, while the bottom diagram illustrates the 2-sync mode.

There is a mandatory delay of 1.5 `PPI_CLK` cycles, plus the value programmed in `PPI_DELAY`, between assertion of the external frame sync(s) and the transfer of valid data out through the PPI.

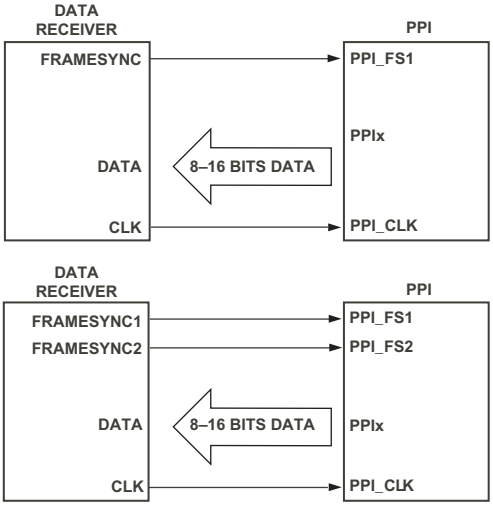


Figure 11-14. TX Mode, 1 or 2 External Frame Syncs

1, 2, or 3 Internal Frame Syncs

The 1-sync mode is intended for interfacing to digital-to-analog converters (DACs) with a single frame sync. The top part of Figure 11-15 shows an example of this type of connection.

The 3-sync mode is useful for connecting to video and graphics displays, as shown in the bottom part of Figure 11-15. A 2-sync mode is implicitly supported by leaving `PPI_FS3` unconnected in this case.

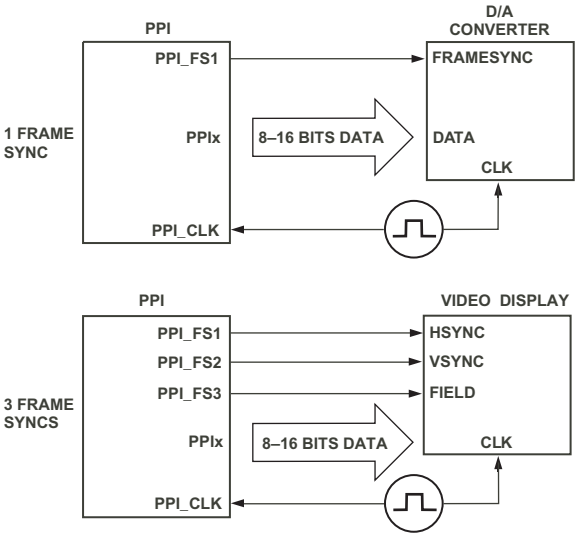


Figure 11-15. PPI GP Output

Frame Synchronization in GP Modes

Frame synchronization in GP modes operates differently in modes with internal frame syncs than in modes with external frame syncs.

Modes with Internal Frame Syncs

In modes with internal frame syncs, PPI_FS1 and PPI_FS2 link directly to the Pulsewidth Modulation (PWM) circuits of Timer 1 and Timer 2, respectively. This allows for arbitrary pulse widths and periods to be programmed for these signals using the existing TIMERx registers. This capability accommodates a wide range of timing needs. Note these PWM circuits are clocked by PPI_CLK, not by SCLK or PF1 (as during conventional Timer PWM operation). If PPI_FS2 is not used in the configured PPI mode, Timer 2 operates as it normally would, unrestricted in functionality. The state of PPI_FS3 depends completely on the state of PPI_FS1 and/or PPI_FS2, so PPI_FS3 has no inherent programmability.

To program PPI_FS1 and/or PPI_FS2 for operation in an internal frame sync mode:

1. Configure and enable DMA for the PPI. See "DMA Operation" on page 11-31.
2. Configure the width and period for each frame sync signal via TIMER1_WIDTH and TIMER1_PERIOD (for PPI_FS1), or TIMER2_WIDTH and TIMER2_PERIOD (for PPI_FS2).
3. Set up TIMER1_CONFIG for PWM_OUT mode (for PPI_FS1). If used, configure TIMER2_CONFIG for PWM_OUT mode (for PPI_FS2). This includes setting CLK_SEL = 1 and TIN_SEL = 1 for each timer.
4. Write to PPI_CONTROL to configure and enable the PPI.
5. Write to TIMER_ENABLE to enable Timer 1 and/or Timer 2.

It is important to guarantee proper frame sync polarity between the PPI and Timer peripherals. To do this, make sure that if PPI_CONTROL[15:14] = b#10 or b#11, the PULSE_HI bit is cleared in TIMER1_CONFIG and TIMER2_CONFIG. Likewise, if PPI_CONTROL[15:14] = b#00 or b#01, the PULSE_HI bit should be set in TIMER1_CONFIG and TIMER2_CONFIG.

To switch to another PPI mode not involving internal frame syncs:

1. Disable the PPI (using PPI_CONTROL).
2. Disable the timers (using TIMER_DISABLE).

Modes with External Frame Syncs

In RX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins become edge-sensitive inputs. In such a mode, Timers 1 and 2 can be used for a purpose not involving the TMR1 and TMR2 pins. However, timer access to a TMRx pin is disabled when the PPI is using that pin for a PPI_FSx frame sync input function. For modes that do not require PPI_FS2, Timer 2 is not restricted in functionality and can be operated as if the PPI were not being used (that is, the TMR2 pin becomes available for timer use as well). For more information on configuring and using the timers, please refer to Chapter 15, "Timers."

In RX Mode with 3 external frame syncs, the start of frame detection occurs where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This happens at the start of Field 1.

Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

In TX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins are treated as edge-sensitive inputs. In this mode, it is not necessary to configure the timer(s) associated with the frame sync(s) as input(s), or to enable them via the TIMER_ENABLE register. Additionally, the actual timers themselves are available for use, even though the timer pin(s) are taken over by the PPI. In this case, there is no requirement that the timebase (configured by TIN_SEL in TIMERx_CONFIG) be PPI_CLK.

However, if using a timer whose pin is connected to an external frame sync, be sure to disable the pin via the OUT_DIS bit in TIMERx_CONFIG. Then the timer itself can be configured and enabled for non-PPI use without affecting PPI operation in this mode. For more information, see Chapter 15, "Timers."

DMA Operation

The PPI must be used with the processor's DMA engine. This section discusses how the two interact. For additional information about the DMA engine, including explanations of DMA registers and DMA operations, please refer to Chapter 9, "Direct Memory Access."

The PPI DMA channel can be configured for either transmit or receive operation, and it has a maximum throughput of (PPI_CLK) x (16 bits/transfer). In modes where data lengths are greater than 8 bits, only one element can be clocked in per PPI_CLK cycle, and this results in reduced bandwidth (since no packing is possible). The highest throughput is achieved with 8-bit data and PACK_EN = 1 (packing mode enabled). Note for 16-bit packing mode, there must be an even number of data elements.

Configuring the PPI's DMA channel is a necessary step toward using the PPI interface. It is the DMA engine that generates interrupts upon completion of a row, frame, or partial-frame transfer. It is also the DMA engine that coordinates the origination or destination point for the data that is transferred through the PPI.

The processor's 2D DMA capability allows the processor to be interrupted at the end of a line or after a frame of video has been transferred, as well as if a DMA Error occurs. In fact, the specification of the DMAx_XCOUNT and DMAx_YCOUNT MMRs allows for flexible data interrupt points. For example,

assume the DMA registers XMODIFY = YMODIFY = 1. Then, if a data frame contains 320 x 240 bytes (240 rows of 320 bytes each), these conditions hold:

- Setting XCOUNT = 320, YCOUNT = 240, and DI_SEL = 1 (the DI_SEL bit is located in DMAx_CONFIG) will interrupt on every row transferred, for the entire frame.
- Setting XCOUNT = 320, YCOUNT = 240, and DI_SEL = 0 will interrupt only on the completion of the frame (when 240 rows of 320 bytes have been transferred).
- Setting XCOUNT = 38,400 (320 x 120), YCOUNT = 2, and DI_SEL = 1 will cause an interrupt when half of the frame has been transferred, and again when the whole frame has been transferred.

Following is the general procedure for setting up DMA operation with the PPI. Please refer to "DMA and Memory DMA Registers" on page 9-3 for details regarding configuration of DMA.

1. Configure DMA registers as appropriate for desired DMA operating mode.
2. Enable the DMA channel for operation.
3. Configure appropriate PPI registers.
4. Enable the PPI by writing a 1 to bit 0 in PPI_CONTROL.

Data Transfer Scenarios

Figure 11-16 shows two possible ways to use the PPI to transfer in video. These diagrams are very generalized, and bandwidth calculations must be made only after factoring in the exact PPI mode and settings (for example, transfer Field 1 only, transfer odd and even elements).

The top part of the diagram shows a situation appropriate for, as an example, JPEG compression. The first N rows of video are DMAed into L1 memory via the PPI. Once in L1, the compression algorithm operates on the data and sends the compressed result out from the processor via the SPORT. Note that no SDRAM access was necessary in this approach.

The bottom part of the diagram takes into account a more formidable compression algorithm, such as MPEG-2 or MPEG-4. Here, the raw video is transferred directly into SDRAM. Independently, a Memory DMA channel transfers data blocks between SDRAM and L1 memory for intermediate processing stages. Finally, the compressed video exits the processor via the SPORT.

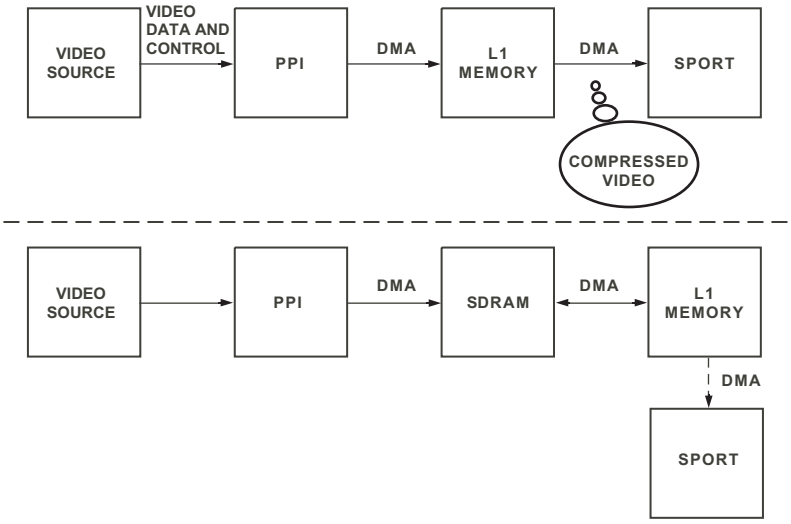


Figure 11-16. PPI Possible Data Transfer Scenarios

12 SERIAL PORT CONTROLLERS

The processor has two identical synchronous serial ports, or SPORTs. These support a variety of serial data communications protocols and can provide a direct interconnection between processors in a multiprocessor system.

The serial ports (SPORT0 and SPORT1) provide an I/O interface to a wide variety of peripheral serial devices. SPORTs provide synchronous serial data transfer only; the processor provides asynchronous RS-232 data transfer via the UART. Each SPORT has one group of pins (primary data, secondary data, clock, and frame sync) for transmit and a second set of pins for receive. The receive and transmit functions are programmed separately. Each SPORT is a full duplex device, capable of simultaneous data transfer in both directions. The SPORTs can be programmed for bit rate, frame sync, and number of bits per word by writing to memory-mapped registers.

In this text, the naming conventions for registers and pins use a lower case `x` to represent a digit. In this chapter, for example, the name `RFSx` pins indicates `RFS0` and `RFS1` (corresponding to SPORT0 and SPORT1, respectively). In this chapter, LSB refers to least significant bit, and MSB refers to most significant bit.

Both SPORTs have the same capabilities and are programmed in the same way. Each SPORT has its own set of control registers and data buffers.

The SPORTs use frame sync pulses to indicate the beginning of each word or packet, and the bit clock marks the beginning of each data bit. External bit clock and frame sync are available for the TX and RX buffers.

With a range of clock and frame synchronization options, the SPORTs allow a variety of serial communication protocols, including H.100, and provide a glueless hardware interface to many industry-standard data converters and codecs.

The SPORTs can operate at up to an SCLK/2 clock rate with an externally generated clock, or 1/2 the system clock rate for an internally generated serial port clock. The SPORT external clock must always be less than the SCLK frequency. Independent transmit and receive clocks provide greater flexibility for serial communications.

SPORT clocks and frame syncs can be internally generated by the system or received from an external source. The SPORTs can operate with a transmission format of LSB first or MSB first, with word lengths selectable from 3 to 32 bits. They offer selectable transmit modes and optional μ-law or A-law companding in hardware. SPORT data can be automatically transferred between on-chip and off-chip memories using DMA block transfers. Additionally, each of the SPORTs offers a TDM (Time-Division-Multiplexed) Multichannel mode.

Each of the SPORTs offers these features and capabilities:

- Provides independent transmit and receive functions.
- Transfers serial data words from 3 to 32 bits in length, either MSB first or LSB first.
- Provides alternate framing and control for interfacing to I^2S serial devices, as well as other audio formats (for example, left-justified stereo serial data).
- Has FIFO plus double buffered data (both receive and transmit functions have a data buffer register and a Shift register), providing additional time to service the SPORT.

- Provides two synchronous transmit and two synchronous receive data pins and buffers in each SPORT to double the total supported datastreams.
- Performs A-law and μ-law hardware companding on transmitted and received words. (See "Companding" on page 12-35 for more information.)
- Internally generates serial clock and frame sync signals in a wide range of frequencies or accepts clock and frame sync input from an external source.
- Operates with or without frame synchronization signals for each data word, with internally generated or externally generated frame signals, with active high or active low frame signals, and with either of two configurable pulse widths and frame signal timing.
- Performs interrupt-driven, single word transfers to and from on-chip memory under processor control.
- Provides Direct Memory Access transfer to and from memory under DMA Master control. DMA can be autobuffer-based (a repeated, identical range of transfers) or descriptor-based (individual or repeated ranges of transfers with differing DMA parameters).
- Executes DMA transfers to and from on-chip memory. Each SPORT can automatically receive and transmit an entire block of data.
- Permits chaining of DMA operations for multiple data blocks.
- Has a multichannel mode for TDM interfaces. Each SPORT can receive and transmit data selectively from a Time-Division-Multiplexed serial bitstream on 128 contiguous channels from a stream of up to 1024 total channels. This mode can be useful as a network communication scheme for multiple processors. The 128 channels available to the processor can be selected to start at any channel

location from 0 to 895 = (1023 – 128). Note the Multichannel Select registers and the WSIZE register control which subset of the 128 channels within the active region can be accessed.

Table 12-1 shows the pins for each SPORT.

Table 12-1. Serial Port (SPORT) Pins

Pin[1]	Description
DTxPRI	Transmit Data Primary
DTxSEC	Transmit Data Secondary
TSCLKx	Transmit Clock
TFSx	Transmit Frame Sync
DRxPRI	Receive Data Primary
DRxSEC	Receive Data Secondary
RSCLKx	Receive Clock
RFSx	Receive Frame Sync

1 A lowercase x within a pin name represents a possible value of 0 or 1 (corresponding to SPORT0 or SPORT1).

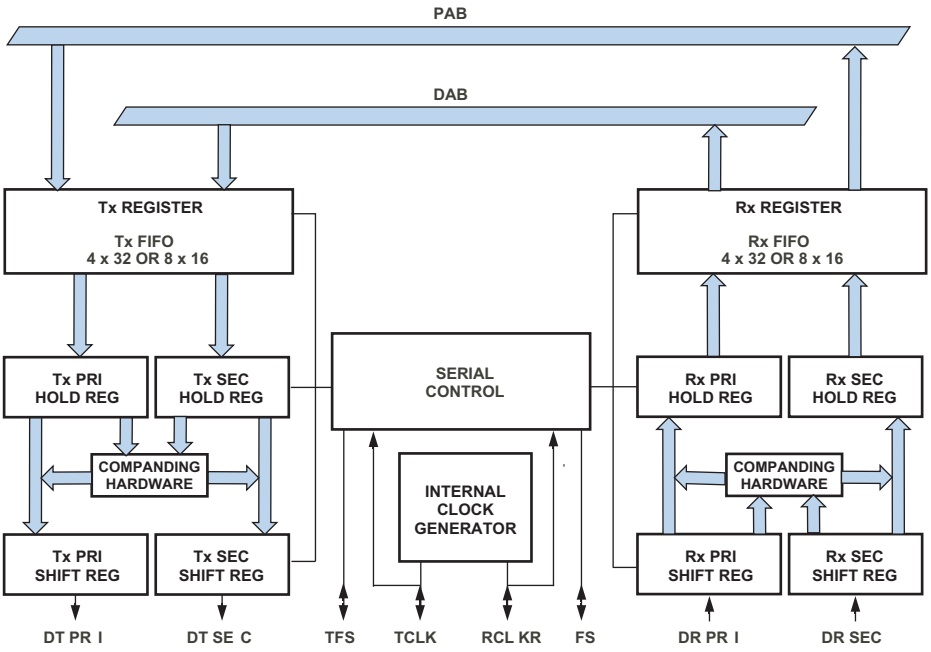
A SPORT receives serial data on its DRxPRI and DRxSEC inputs and transmits serial data on its DTxPRI and DTxSEC outputs. It can receive and transmit simultaneously for full-duplex operation. For transmit, the data bits (DTxPRI and DTxSEC) are synchronous to the transmit clock (TSCLKx). For receive, the data bits (DRxPRI and DRxSEC) are synchronous to the receive clock (RSCLKx). The serial clock is an output if the processor generates it, or an input if the clock is externally generated. Frame synchronization signals RFSx and TFSx are used to indicate the start of a serial data word or stream of serial words.

The primary and secondary data pins provide a method to increase the data throughput of the serial port. They do not behave as totally separate SPORTs; rather, they operate in a synchronous manner (sharing clock and

frame sync) but on separate data. The data received on the primary and secondary pins is interleaved in main memory and can be retrieved by setting a stride in the Data Address Generators (DAG) unit. For more information about DAGs, see Chapter 5, "Data Address Generators." Similarly, for TX, data should be written to the TX register in an alternating manner—first primary, then secondary, then primary, then secondary, and so on. This is easily accomplished with the processor's powerful DAGs.

In addition to the serial clock signal, data must be signalled by a frame synchronization signal. The framing signal can occur either at the beginning of an individual word or at the beginning of a block of words.

The following figure shows a simplified block diagram of a single SPORT. Data to be transmitted is written from an internal processor register to the SPORT's SPORTx_TX register via the peripheral bus. This data is optionally compressed by the hardware and automatically transferred to the TX Shift register. The bits in the Shift register are shifted out on the SPORT's DTxPRI/DTxSEC pin, MSB first or LSB first, synchronous to the serial clock on the TSCLKx pin. The receive portion of the SPORT accepts data from the DRxPRI/DRxSEC pin synchronous to the serial clock on the RSCLKx pin. When an entire word is received, the data is optionally expanded, then automatically transferred to the SPORT's SPORTx_RX register, and then into the RX FIFO where it is available to the processor.



NOTE 1: ALL WIDE ARROW DATA PATHS ARE 16 OR 32 BITS WIDE, DEPENDING ON SLEN. FOR SLEN = 2 TO 15, A 16-BIT DATA PATH WITH 8-DEEP FIFO IS USED. FOR SLEN = 16 TO 31, A 32-BIT DATA PATH WITH 4-DEEP FIFO IS USED.
NOTE 2: Tx REGISTER IS THE BOTTOM OF THE Tx FIFO, Rx REGISTER IS THE TOP OF THE Rx FIFO.

Figure 12-1. SPORT Block Diagram

Figure 12-2 shows a possible port connection for the SPORTs. Note serial devices A and B must be synchronous, as they share common frame syncs and clocks. The same is true for serial devices C and D.

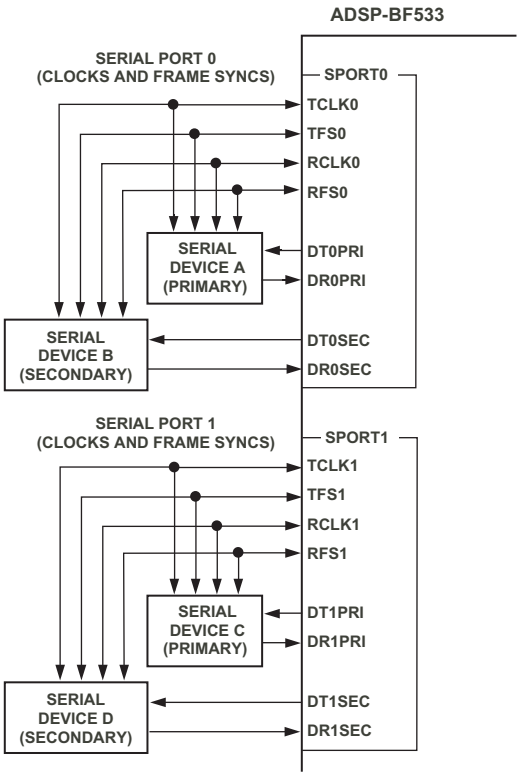


Figure 12-2. SPORT Connections

Figure 12-3 shows an example of a stereo serial device with three transmit and two receive channels connected to the processor.

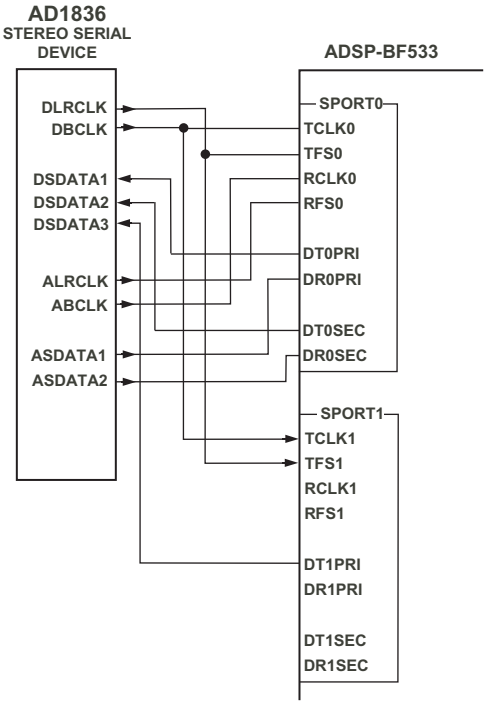


Figure 12-3. Stereo Serial Connection

SPORT Operation

This section describes general SPORT operation, illustrating the most common use of a SPORT. Since the SPORT functionality is configurable, this description represents just one of many possible configurations.

Writing to a SPORT's SPORTx_TX register readies the SPORT for transmission. The TFS signal initiates the transmission of serial data. Once transmission has begun, each value written to the SPORTx_TX register is transferred through the FIFO to the internal Transmit Shift register. The bits are then sent, beginning with either the MSB or the LSB as specified in the SPORTx_TCR1 register. Each bit is shifted out on the driving edge of TSCLKx. The driving edge of TSCLKx can be configured to be rising or falling. The SPORT generates the transmit interrupt or requests a DMA transfer as long as there is space in the TX FIFO.

As a SPORT receives bits, they accumulate in an internal receive register. When a complete word has been received, it is written to the SPORT FIFO register and the receive interrupt for that SPORT is generated or a DMA transfer is initiated. Interrupts are generated differently if DMA block transfers are performed. For information about DMA, see Chapter 9, "Direct Memory Access."

SPORT Disable

The SPORTs are automatically disabled by a processor hardware or software reset. A SPORT can also be disabled directly by clearing the SPORT's transmit or receive enable bits (TSPEN in the SPORTx_TCR1 register and RSPEN in the SPORTx_RCR1 register, respectively). Each method has a different effect on the SPORT.

A processor reset disables the SPORTs by clearing the SPORTx_TCR1, SPORTx_TCR2, SPORTx_RCR1, and SPORTx_RCR2 registers (including the TSPEN and RSPEN enable bits) and the SPORTx_TCLKDIV, SPORTX_RCLKDIV, SPORTx_TFSDIVx, and SPORTx_RFSDIVx Clock and Frame Sync Divisor registers. Any ongoing operations are aborted.

Clearing the TSPEN and RSPEN enable bits disables the SPORTs and aborts any ongoing operations. Status bits are also cleared. Configuration bits remain unaffected and can be read by the software in order to be altered or overwritten. To disable the SPORT output clock, set the SPORT to be disabled.

Note that disabling a SPORT via TSPEN/RSPEN may shorten any currently active pulses on the TFSx/RFSx and TSCLKx/RSCLKx pins, if these signals are configured to be generated internally.

The SPORTs are ready to start transmitting or receiving data no later than three serial clock cycles after they are enabled in the SPORTx_TCR1 or SPORTx_RCR1 register. No serial clock cycles are lost from this point on. The first internal frame sync will occur one frame sync delay after the SPORTs are ready. External frame syncs can occur as soon as the SPORT is ready.

When disabling the SPORT from multichannel operation, first disable TXEN and then disable RXEN. Note both TXEN and RXEN must be disabled before reenabling. Disabling only TX or RX is not allowed.

Setting SPORT Modes

SPORT configuration is accomplished by setting bit and field values in configuration registers. Each SPORT must be configured prior to being enabled. Once the SPORT is enabled, further writes to the SPORT Configuration registers are disabled (except for SPORTx_RCLKDIV, SPORTx_TCLKDIV, and Multichannel Mode Channel Select registers). To change values in all other SPORT Configuration registers, disable the SPORT by clearing TSPEN in SPORTx_TCR1 and/or RSPEN in SPORTx_RCR1.

Each SPORT has its own set of control registers and data buffers. These registers are described in detail in the following sections. All control and status bits in the SPORT registers are active high unless otherwise noted.

Register Writes and Effective Latency

When the SPORT is disabled (TSPEN and RSPEN cleared), SPORT register writes are internally completed at the end of the SCLK cycle in which they occurred, and the register reads back the newly-written value on the next cycle.

When the SPORT is enabled to transmit (TSPEN set) or receive (RSPEN set), corresponding SPORT Configuration register writes are disabled (except for SPORTx_RCLKDIV, SPORTx_TCLKDIV, and Multichannel Mode Channel Select registers). The SPORTx_TX register writes are always enabled; SPORTx_RX, SPORTx_CHNL, and SPORTx_STAT are read-only registers.

After a write to a SPORT register, while the SPORT is disabled, any changes to the control and mode bits generally take effect when the SPORT is re-enabled.

Most configuration registers can only be changed while the SPORT is disabled (TSPEN/RSPEN = 0). Changes take effect after the SPORT is re-enabled. The only exceptions to this rule are the TCLKDIV/RCLKDIV registers and Multichannel Select registers.

SPORTx_TCR1 and SPORTx_TCR2 Registers

The main control registers for the transmit portion of each SPORT are the Transmit Configuration registers, SPORTx_TCR1 and SPORTx_TCR2.

A SPORT is enabled for transmit if Bit 0 (TSPEN) of the Transmit Configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT transmission.

When the SPORT is enabled to transmit (TSPEN set), corresponding SPORT Configuration register writes are not allowed except for SPORTx_TCLKDIV and Multichannel Mode Channel Select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORTx_TCR1 is not written except for bit 0 (TSPEN). For example:

```
write (SPORTx_TCR1, 0x0001) ;   /* SPORT TX Enabled */
write (SPORTx_TCR1, 0xFF01) ;   /* ignored, no effect */
write(SPORTx_TCR1,0xFFF0);   /*SPORTdisabled,SPORTx_TCR1
still equal to 0x0000 */
```

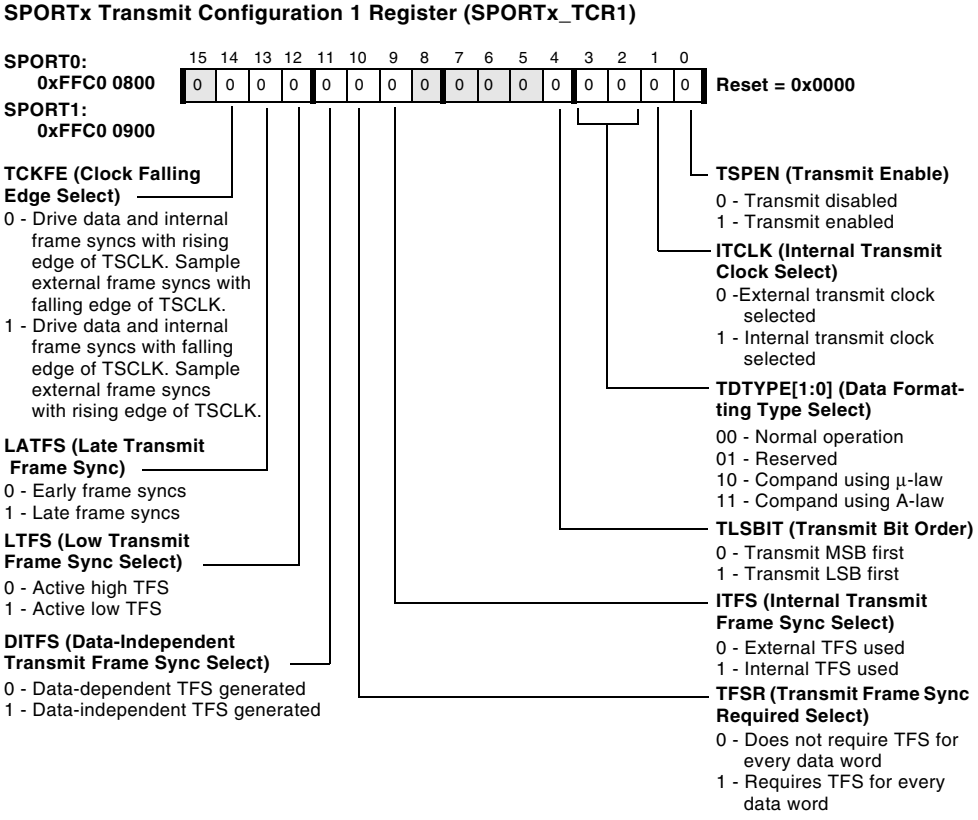


Figure 12-4. SPORTx Transmit Configuration 1 Register

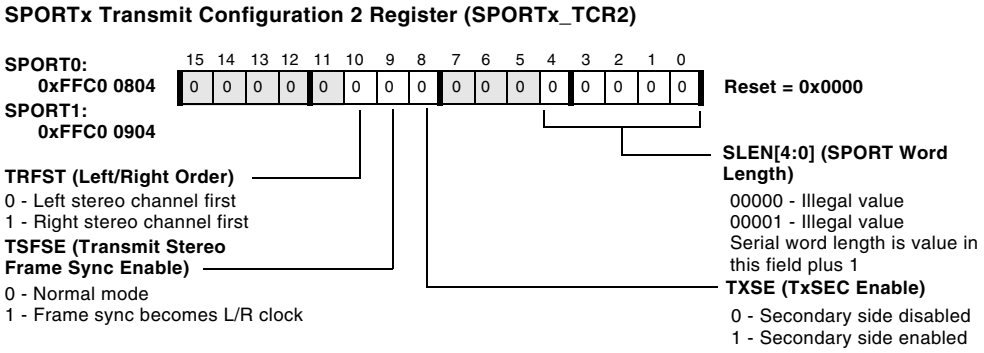


Figure 12-5. SPORTx Transmit Configuration 2 Register

Additional information for the SPORTx_TCR1 and SPORTx_TCR2 Transmit Configuration register bits includes:

- **Transmit Enable** (TSPEN). This bit selects whether the SPORT is enabled to transmit (if set) or disabled (if cleared).

  Setting TSPEN causes an immediate assertion of a SPORT TX interrupt, indicating that the TX data register is empty and needs to be filled. This is normally desirable because it allows centralization of the transmit data write code in the TX interrupt service routine (ISR). For this reason, the code should initialize the ISR and be ready to service TX interrupts before setting TSPEN.

  Similarly, if DMA transfers will be used, DMA control should be configured correctly before setting TSPEN. Set all DMA control registers before setting TSPEN.

  Clearing TSPEN causes the SPORT to stop driving data, TSCLK, and frame sync pins; it also shuts down the internal SPORT circuitry. In low power applications, battery life can be extended by clearing TSPEN whenever the SPORT is not in use.

All SPORT control registers should be programmed before TSPEN is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write SPORTx_TCR1 with all of the necessary bits, including TSPEN.

- **Internal Transmit Clock Select**. (ITCLK). This bit selects the internal transmit clock (if set) or the external transmit clock on the TSCLK pin (if cleared). The TCLKDIV MMR value is not used when an external clock is selected.
- **Data Formatting Type Select**. The two TDTYPE bits specify data formats used for single and multichannel operation.
- **Bit Order Select**. (TLSBIT). The TLSBIT bit selects the bit order of the data words transmitted over the SPORT.
- **Serial Word Length Select**. (SLEN). The serial word length (the number of bits in each word transmitted over the SPORTs) is calculated by adding 1 to the value of the SLEN field:

```
Serial Word Length = SLEN + 1;
```

  The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field. Three common settings for the SLEN field are 15, to transmit a full 16-bit word; 7, to transmit an 8-bit byte; and 23, to transmit a 24-bit word. The processor can load 16- or 32-bit values into the transmit buffer via DMA or an MMR write instruction; the SLEN field tells the SPORT how many of those bits to shift out of the register over the serial link. The serial port transfers bits [SLEN:0] from the transmit buffer.

The frame sync signal is controlled by the SPORTx_TFSDIV and SPORTx_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync

divider must be programmed into the Frame Sync Divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal Transmit Frame Sync Select**. (ITFS). This bit selects whether the SPORT uses an internal TFS (if set) or an external TFS (if cleared).
- **Transmit Frame Sync Required Select**. (TFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a Transmit Frame Sync for every data word.

The TFSR bit is normally set during SPORT configuration. A frame sync pulse is used to mark the beginning of each word or data packet, and most systems need a frame sync to function properly.

- **Data-Independent Transmit Frame Sync Select**. (DITFS). This bit selects whether the SPORT generates a data-independent TFS (sync at selected interval) or a data-dependent TFS (sync when data is present in SPORTx_TX) for the case of internal frame sync select (ITFS = 1). The DITFS bit is ignored when external frame syncs are selected.

  The frame sync pulse marks the beginning of the data word. If DITFS is set, the frame sync pulse is issued on time, whether the SPORTx_TX register has been loaded or not; if DITFS is cleared, the frame sync pulse is only generated if the SPORTx_TX data register has been loaded. If the receiver demands regular frame sync pulses, DITFS should be set, and the processor should keep loading the SPORTx_TX register on time. If the receiver can tolerate occasional late frame sync pulses, DITFS should be cleared to prevent the SPORT from transmitting old data twice or transmitting garbled data if the processor is late in loading the SPORTx_TX register.
- **Low Transmit Frame Sync Select**. (LTFS). This bit selects an active low TFS (if set) or active high TFS (if cleared).

- **Late Transmit Frame Sync**. (LATFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
- **Clock Drive/Sample Edge Select**. (TCKFE). This bit selects which edge of the TCLKx signal the SPORT uses for driving data, for driving internally generated frame syncs, and for sampling externally generated frame syncs. If set, data and internally generated frame syncs are driven on the falling edge, and externally generated frame syncs are sampled on the rising edge. If cleared, data and internally generated frame syncs are driven on the rising edge, and externally generated frame syncs are sampled on the falling edge.
- **TxSec Enable**. (TXSE). This bit enables the transmit secondary side of the serial port (if set).
- **Stereo Serial Enable**. (TSFSE). This bit enables the Stereo Serial operating mode of the serial port (if set). By default this bit is cleared, enabling normal clocking and frame sync.
- **Left/Right Order**. (TRFST). If this bit is set, the right channel is transmitted first in Stereo Serial operating mode. By default this bit is cleared, and the left channel is transmitted first.

SPORTx_RCR1 and SPORTx_RCR2 Registers

The main control registers for the receive portion of each SPORT are the Receive Configuration registers, SPORTx_RCR1 and SPORTx_RCR2.

A SPORT is enabled for receive if Bit 0 (RSPEN) of the Receive Configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT reception.

When the SPORT is enabled to receive (RSPEN set), corresponding SPORT Configuration register writes are not allowed except for SPORTx_RCLKDIV and Multichannel Mode Channel Select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORTx_RCR1 is not written except for bit 0 (RSPEN). For example:

```
write (SPORTx_RCR1, 0x0001) ;  /* SPORT RX Enabled */
write (SPORTx_RCR1, 0xFF01) ;  /* ignored, no effect */
write (SPORTx_RCR1, 0xFFF0) ;  /* SPORT disabled, SPORTx_RCR1
still equal to 0x0000 */
```

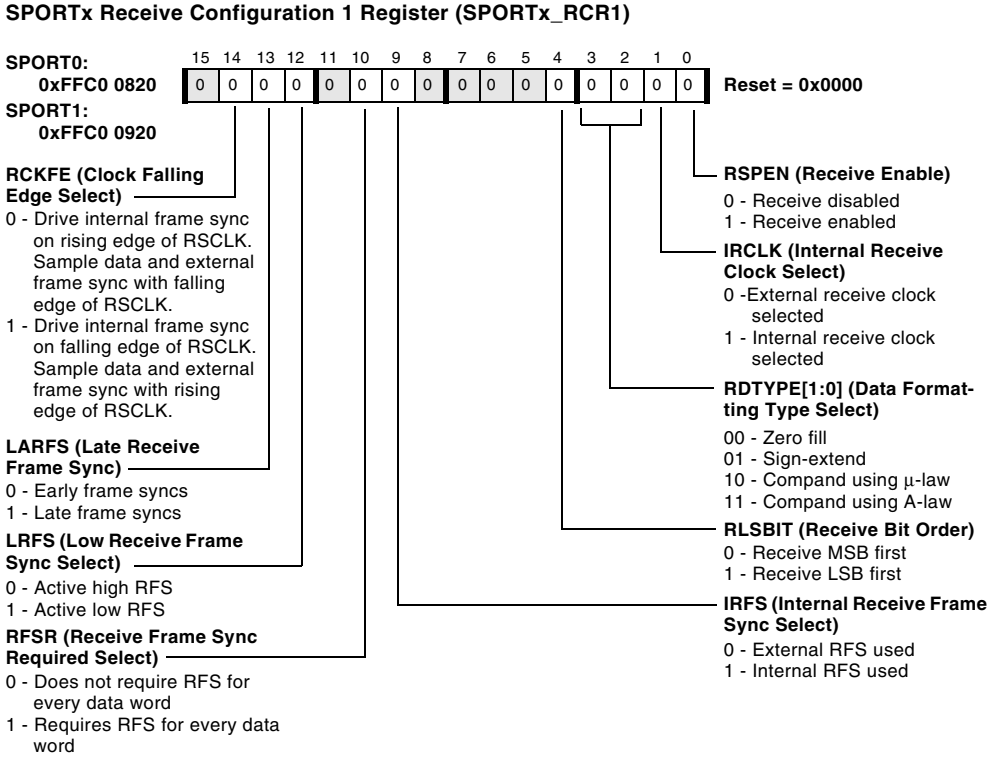


Figure 12-6. SPORTx Receive Configuration 1 Register

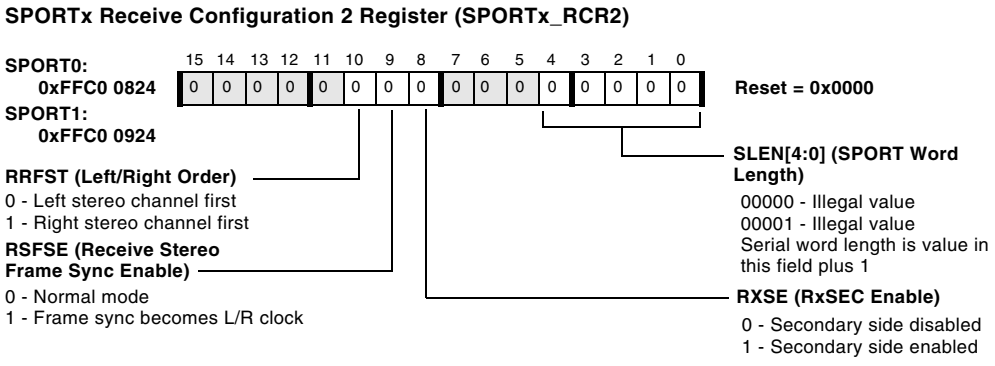


Figure 12-7. SPORTx Receive Configuration 2 Register

Additional information for the SPORTx_RCR1 and SPORTxRCR2 Receive Configuration register bits:

- **Receive Enable**. (RSPEN). This bit selects whether the SPORT is enabled to receive (if set) or disabled (if cleared). Setting the RSPEN bit turns on the SPORT and causes it to sample data from the data receive pins as well as the receive bit clock and Receive Frame Sync pins if so programmed.

  Setting RSPEN enables the SPORTx receiver, which can generate a SPORTx RX interrupt. For this reason, the code should initialize the ISR and the DMA control registers, and should be ready to service RX interrupts before setting RSPEN. Setting RSPEN also generates DMA requests if DMA is enabled and data is received. Set all DMA control registers before setting RSPEN.

  Clearing RSPEN causes the SPORT to stop receiving data; it also shuts down the internal SPORT receive circuitry. In low power applications, battery life can be extended by clearing RSPEN whenever the SPORT is not in use.

All SPORT control registers should be programmed before RSPEN is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write SPORTx_RCR1 with all of the necessary bits, including RSPEN.

- **Internal Receive Clock Select**. (IRCLK). This bit selects the internal receive clock (if set) or external receive clock (if cleared). The RCLKDIV MMR value is not used when an external clock is selected.
- **Data Formatting Type Select**. (RDTYPE). The two RDTYPE bits specify one of four data formats used for single and multichannel operation.
- **Bit Order Select**. (RLSBIT). The RLSBIT bit selects the bit order of the data words received over the SPORTs.
- **Serial Word Length Select**. (SLEN). The serial word length (the number of bits in each word received over the SPORTs) is calculated by adding 1 to the value of the SLEN field. The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field.

The frame sync signal is controlled by the SPORTx_TFSDIV and SPORTx_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync divider must be programmed into the Frame Sync Divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal Receive Frame Sync Select**. (IRFS). This bit selects whether the SPORT uses an internal RFS (if set) or an external RFS (if cleared).
- **Receive Frame Sync Required Select**. (RFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a Receive Frame Sync for every data word.

- **Low Receive Frame Sync Select**. (LRFS). This bit selects an active low RFS (if set) or active high RFS (if cleared).
- **Late Receive Frame Sync**. (LARFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
- **Clock Drive/Sample Edge Select**. (RCKFE). This bit selects which edge of the RSCLK clock signal the SPORT uses for sampling data, for sampling externally generated frame syncs, and for driving internally generated frame syncs. If set, internally generated frame syncs are driven on the falling edge, and data and externally generated frame syncs are sampled on the rising edge. If cleared, internally generated frame syncs are driven on the rising edge, and data and externally generated frame syncs are sampled on the falling edge.
- **RxSec Enable**. (RXSE). This bit enables the receive secondary side of the serial port (if set).
- **Stereo Serial Enable**. (RSFSE). This bit enables the Stereo Serial operating mode of the serial port (if set). By default this bit is cleared, enabling normal clocking and frame sync.
- **Left/Right Order**. (RRFST). If this bit is set, the right channel is received first in Stereo Serial operating mode. By default this bit is cleared, and the left channel is received first.

Data Word Formats

The format of the data words transferred over the SPORTs is configured by the combination of transmit SLEN and receive SLEN; RDTYPE; TDTYPE; RLSBIT; and TLSBIT bits of the SPORTx_TCR1, SPORTx_TCR2, SPORTx_RCR1, and SPORTx_RCR2 registers.

SPORTx_TX Register

The SPORTx Transmit Data register (SPORTx_TX) is a write-only register. Reads produce a Peripheral Access Bus (PAB) error. Writes to this register cause writes into the transmitter FIFO. The 16-bit wide FIFO is 8 deep for word length <= 16 and 4 deep for word length > 16. The FIFO is common to both primary and secondary data and stores data for both. Data ordering in the FIFO is shown in the Figure 12-8.

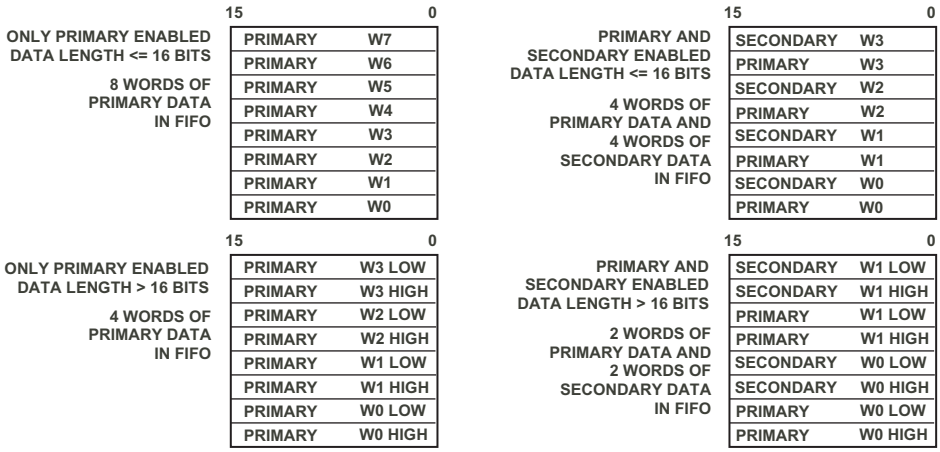


Figure 12-8. SPORT Transmit FIFO Data Ordering

It is important to keep the interleaving of primary and secondary data in the FIFO as shown. This means that PAB/DMA writes to the FIFO must follow an order of primary first, and then secondary, if secondary is enabled. DAB/PAB writes must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit write. Use a 32-bit write for word length greater than 16 bits.

When transmit is enabled, data from the FIFO is assembled in the TX Hold register based on TXSE and SLEN, and then shifted into the primary and secondary shift registers. From here, the data is shifted out serially on the DTPRI and DTSEC pins.

The SPORT TX interrupt is asserted when TSPEN = 1 and the TX FIFO has room for additional words. This interrupt does not occur if SPORT DMA is enabled. For DMA operation, see Chapter 9, "Direct Memory Access."

The Transmit Underflow Status bit (TUVF) is set in the SPORT Status register when a Transmit Frame Sync occurs and no new data has been loaded into the serial shift register. In Multichannel Mode (MCM), TUVF is set whenever the serial shift register is not loaded, and transmission begins on the current enabled channel. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the serial port (writing TXEN = 0).

If software causes the core processor to attempt a write to a full TX FIFO with a SPORTx_TX write, the new data is lost and no overwrites occur to data in the FIFO. The TOVF status bit is set and a SPORT error interrupt is asserted. The TOVF bit is a sticky bit; it is only cleared by disabling the SPORT TX. To find out whether the core processor can access the SPORTx_TX register without causing this type of error, read the register's status first. The TXF bit in the SPORT Status register is 0 if space is available for another word in the FIFO.

The TXF and TOVF status bits in the SPORTx Status register are updated upon writes from the core processor, even when the SPORT is disabled.

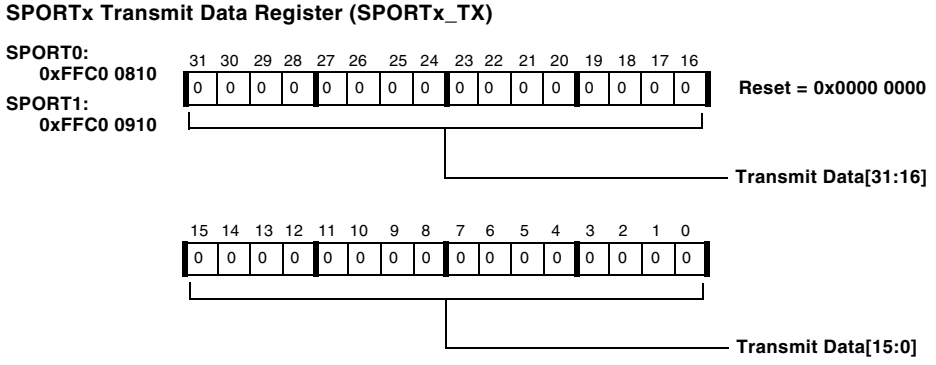


Figure 12-9. SPORTx Transmit Data Register

SPORTx_RX Register

The SPORTx Receive Data register (SPORTx_RX) is a read-only register. Writes produce a PAB error. The same location is read for both primary and secondary data. Reading from this register space causes reading of the receive FIFO. This 16-bit FIFO is 8 deep for receive word length <= 16 and 4 deep for length > 16 bits. The FIFO is shared by both primary and secondary receive data. The order for reading using PAB/DMA reads is important since data is stored in differently depending on the setting of the SLEN and RXSE configuration bits.

Data storage and data ordering in the FIFO is shown in Figure 12-10.

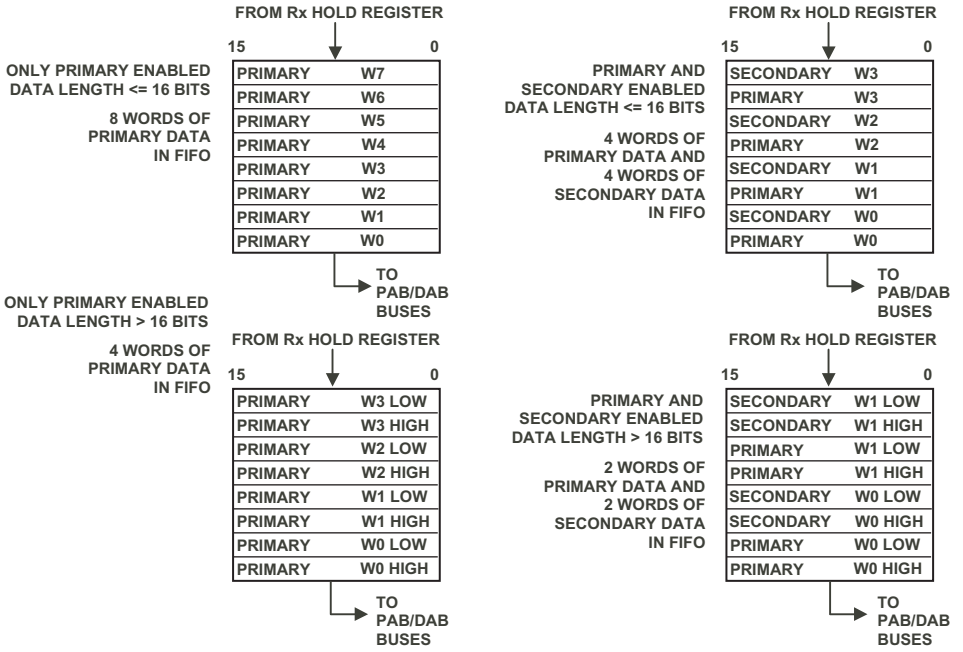


Figure 12-10. SPORT Receive FIFO Data Ordering

When reading from the FIFO for both primary and secondary data, read primary first, followed by secondary. DAB/PAB reads must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit read. Use a 32-bit read for word length greater than 16 bits.

When receiving is enabled, data from the DRPRI pin is loaded into the RX primary shift register, while data from the DRSEC pin is loaded into the RX secondary shift register. At transfer completion of a word, data is shifted into the RX Hold registers for primary and secondary data, respectively. Data from the Hold registers is moved into the FIFO based on RXSE and SLEN.

The SPORT RX interrupt is generated when RSPEN = 1 and the RX FIFO has received words in it. When the core processor has read all the words in the FIFO, the RX interrupt is cleared. The SPORT RX interrupt is set only if SPORT RX DMA is disabled; otherwise, the FIFO is read by DMA reads.

If the program causes the core processor to attempt a read from an empty RX FIFO, old data is read, the RUVF flag is set in the SPORTx_STAT register, and the SPORT error interrupt is asserted. The RUVF bit is a sticky bit and is cleared only when the SPORT is disabled. To determine if the core can access the RX registers without causing this error, first read the RX FIFO status (RXNE in the SPORTx Status register). The RUVF status bit is updated even when the SPORT is disabled.

The ROVF status bit is set in the SPORTx_STAT register when a new word is assembled in the RX Shift register and the RX Hold register has not moved the data to the FIFO. The previously written word in the Hold register is overwritten. The ROVF bit is a sticky bit; it is only cleared by disabling the SPORT RX.

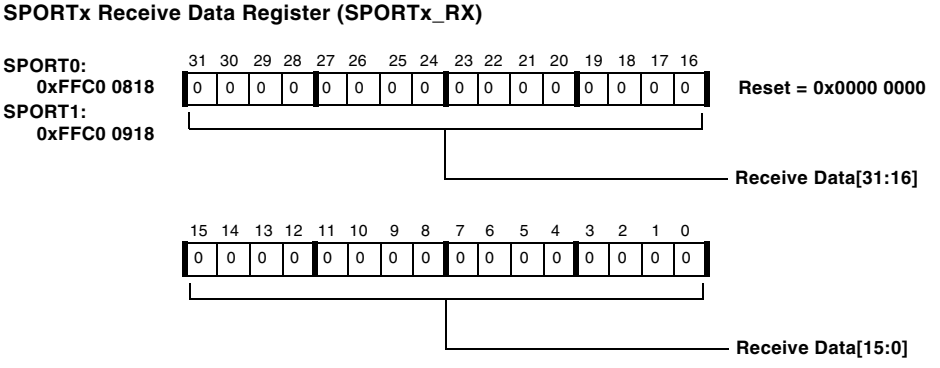


Figure 12-11. SPORTx Receive Data Register

SPORTx_STAT Register

The SPORT Status register (SPORTx_STAT) is used to determine if the access to a SPORT RX or TX FIFO can be made by determining their full or empty status.

The TXF bit in the SPORT Status register indicates whether there is room in the TX FIFO. The RXNE status bit indicates whether there are words in the RX FIFO. The TXHRE bit indicates if the TX Hold register is empty.

The Transmit Underflow Status bit (TUVF) is set whenever the TFS signal occurs (from either an external or internal source) while the TX Shift register is empty. The internally generated TFS may be suppressed whenever SPORTx_TX is empty by clearing the DITFS control bit in the SPORT Configuration register. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the serial port (writing TXEN = 0).

For continuous transmission (TFSR = 0), TUVF is set at the end of a transmitted word if no new word is available in the TX Hold register.

The TOVF bit is set when a word is written to the TX FIFO when it is full. It is a sticky W1C bit and is also cleared by writing TXEN = 0. Both TXF and TOVF are updated even when the SPORT is disabled.

When the SPORT RX Hold register is full, and a new receive word is received in the Shift register, the Receive Overflow Status bit (ROVF) is set in the SPORT Status register. It is a sticky W1C bit and is also cleared by disabling the serial port (writing RXEN = 0).

The RUVF bit is set when a read is attempted from the RX FIFO and it is empty. It is a sticky W1C bit and is also cleared by writing RXEN = 0. The RUVF bit is updated even when the SPORT is disabled.

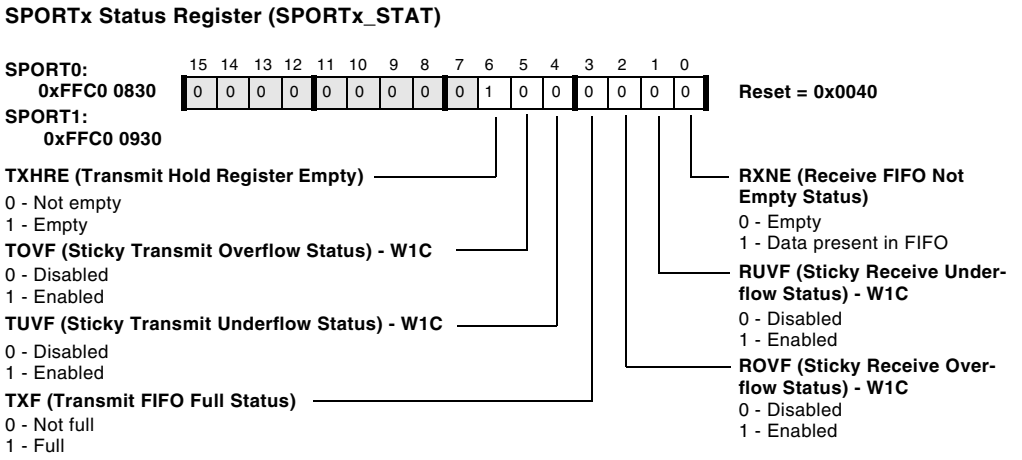


Figure 12-12. SPORTx Status Register

SPORT RX, TX, and Error Interrupts

The SPORT RX interrupt is asserted when RSPEN is enabled and any words are present in the RX FIFO. If RX DMA is enabled, the SPORT RX interrupt is turned off and DMA services the RX FIFO.

The SPORT TX interrupt is asserted when TSPEN is enabled and the TX FIFO has room for words. If TX DMA is enabled, the SPORT TX interrupt is turned off and DMA services the TX FIFO.

The SPORT Error interrupt is asserted when any of the sticky status bits (ROVF, RUVF, TOVF, TUVF) are set. The ROVF and RUVF bits are cleared by writing 0 to RSPEN. The TOVF and TUVF bits are cleared by writing 0 to TSPEN.

PAB Errors

The SPORT generates a PAB error for illegal register read or write operations. Examples include:

- Reading a write-only register (for example, SPORT_TX)
- Writing a read-only register (for example, SPORT_RX)
- Writing or reading a register with the wrong size (for example, 32-bit read of a 16-bit register)
- Accessing reserved register locations

SPORTx_TCLKDIV and SPORTx_RCLKDIV Registers

The frequency of an internally generated clock is a function of the system clock frequency (as seen at the SCLK pin) and the value of the 16-bit serial clock divide modulus registers (the SPORTx Transmit Serial Clock Divider register, SPORTx_TCLKDIV, and the SPORTx Receive Serial Clock Divider register, SPORTx_RCLKDIV).

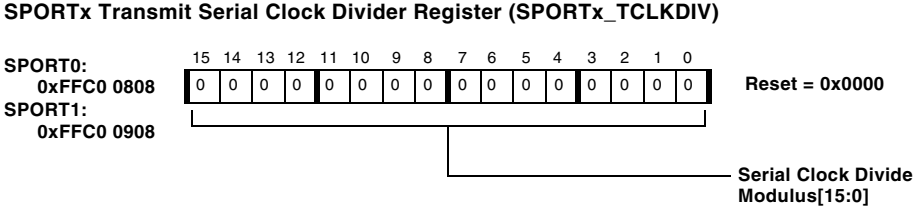


Figure 12-13. SPORTx Transmit Serial Clock Divider Register

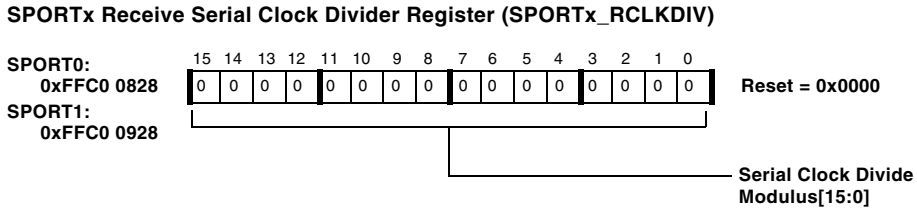


Figure 12-14. SPORTx Receive Serial Clock Divider Register

SPORTx_TFSDIV and SPORTx_RFSDIV Register

The 16-bit SPORTx Transmit Frame Sync Divider register (SPORTx_TFSDIV) and the SPORTx Receive Frame Sync Divider register (SPORTx_RFSDIV) specify how many transmit or receive clock cycles are counted before generating a TFS or RFS pulse when the frame sync is internally generated. In this way, a frame sync can be used to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks.

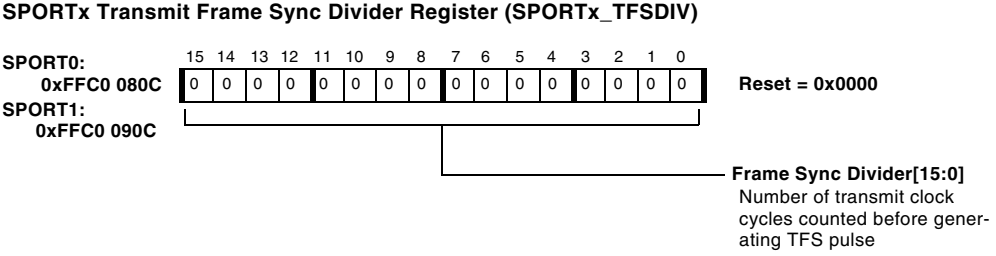


Figure 12-15. SPORTx Transmit Frame Sync Divider Register

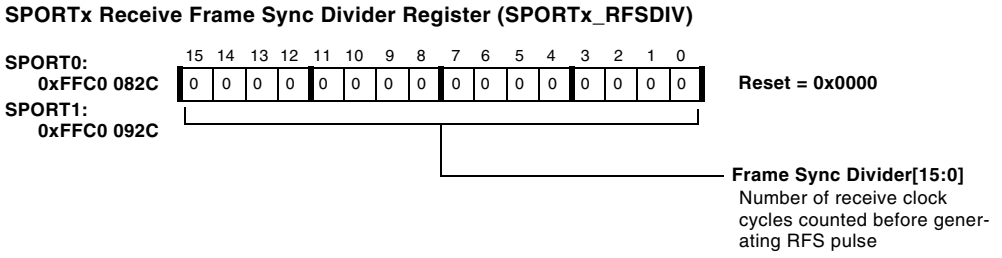


Figure 12-16. SPORTx Receive Frame Sync Divider Register

Clock and Frame Sync Frequencies

The maximum serial clock frequency (for either an internal source or an external source) is SCLK/2. The frequency of an internally generated clock is a function of the system clock frequency (SCLK) and the value of the 16-bit serial clock divide modulus registers, SPORTx_TCLKDIV and SPORTx_RCLKDIV.

```
SPORTx_TCLK frequency =
(SCLK frequency)/(2 x (SPORTx_TCLKDIV + 1))

SPORTx_RCLK frequency =
(SCLK frequency)/(2 x (SPORTx_RCLKDIV + 1))
```

If the value of SPORTx_TCLKDIV or SPORTx_RCLKDIV is changed while the internal serial clock is enabled, the change in TSCLK or RSCLK frequency takes effect at the start of the drive edge of TSCLK or RSCLK that follows the next leading edge of TFS or RFS.

When an internal frame sync is selected (ITFS = 1 in the SPORTx_TCR1 register or IRFS = 1 in the SPORTx_RCR1 register) and frame syncs are not required, the first frame sync does not update the clock divider if the value in SPORTx_TCLKDIV or SPORTx_RCLKDIV has changed. The second frame sync will cause the update.

The SPORTx_TFSDIV and SPORTx_RFSDIV registers specify the number of transmit or receive clock cycles that are counted before generating a TFS or RFS pulse (when the frame sync is internally generated). This enables a frame sync to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks.

The formula for the number of cycles between frame sync pulses is:

```
# of transmit serial clocks between frame sync assertions =
TFSDIV + 1
# of receive serial clocks between frame sync assertions =
RFSDIV + 1
```

Use the following equations to determine the correct value of TFSDIV or RFSDIV, given the serial clock frequency and desired frame sync frequency:

```
SPORTxTFS frequency = (TSCLKx frequency)/(SPORTx_TFSDIV + 1)

SPORTxRFS frequency = (RSCLKx frequency)/(SPORTx_RFSDIV + 1)
```

The frame sync would thus be continuously active (for transmit if TFSDIV = 0 or for receive if RFSDIV = 0). However, the value of TFSDIV (or RFSDIV) should not be less than the serial word length minus 1 (the value of the SLEN field in SPORTx_TCR2 or SPORTx_RCR2). A smaller value could cause an external device to abort the current operation or have other unpredictable results. If a SPORT is not being used, the TFSDIV (or RFSDIV) divisor can be used as a counter for dividing an external clock or for generating a periodic pulse or periodic interrupt. The SPORT must be enabled for this mode of operation to work.

Maximum Clock Rate Restrictions

Externally generated late Transmit Frame Syncs also experience a delay from arrival to data output, and this can limit the maximum serial clock speed. See the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* for exact timing specifications.

Frame Sync and Clock Example

The following code fragment is a brief example of setting up the clocks and frame sync.

```
r0 = 0x00FF;
p0.l = SPORT0_RFSDIV & 0xFFFF;
p0.h = (SPORT0_RFSDIV >> 16) & 0xFFFF;
w[p0] = r0.l; ssync;
p0.l = SPORT0_TFSDIV & 0xFFFF;
w[p0] = r0.l; ssync;
```

Word Length

Each SPORT channel (transmit and receive) independently handles word lengths of 3 to 32 bits. The data is right-justified in the SPORT data registers if it is fewer than 32 bits long, residing in the LSB positions. The value of the serial word length (SLEN) field in the SPORTx_TCR2 and SPORTx_RCR2 registers of each SPORT determines the word length according to this formula:

```
Serial Word Length = SLEN + 1
```

The SLEN value should not be set to 0 or 1; values from 2 to 31 are allowed. Continuous operation (when the last bit of the current word is immediately followed by the first bit of the next word) is restricted to word sizes of 4 or longer (so SLEN ≥ 3).

Bit Order

Bit order determines whether the serial word is transmitted MSB first or LSB first. Bit order is selected by the RLSBIT and TLSBIT bits in the SPORTx_RCR1 and SPORTx_TCR1 registers. When RLSBIT (or TLSBIT) = 0, serial words are received (or transmitted) MSB first. When RLSBIT (or TLSBIT) = 1, serial words are received (or transmitted) LSB first.

Data Type

The TDTYPE field of the SPORTx_TCR1 register and the RDTYPE field of the SPORTx_RCR1 register specify one of four data formats for both single and multichannel operation. See Table 12-2.

Table 12-2. TDTYPE, RDTYPE, and Data Formatting

TDTYPE or RDTYPE	SPORTx_TCR1 Data Formatting	SPORTx_RCR1 Data Formatting
00	Normal operation	Zero fill
01	Reserved	Sign extend
10	Compand using µ-law	Compand using µ-law
11	Compand using A-law	Compand using A-law

These formats are applied to serial data words loaded into the SPORTx_RX and SPORTx_TX buffers. SPORTx_TX data words are not actually zero filled or sign extended, because only the significant bits are transmitted.

Companding

Companding (a contraction of COMpressing and exPANDing) is the process of logarithmically encoding and decoding data to minimize the number of bits that must be sent. The SPORTs support the two most widely used companding algorithms, μ-law and A-law. The processor compands data according to the CCITT G.711 specification. The type of companding can be selected independently for each SPORT.

When companding is enabled, valid data in the SPORTx_RX register is the right-justified, expanded value of the eight LSBs received and sign extended to 16 bits. A write to SPORTx_TX causes the 16-bit value to be compressed to eight LSBs (sign extended to the width of the transmit word) and written to the internal transmit register. Although the companding standards support only 13-bit (A-law) or 14-bit (μ-law) maximum word lengths, up to 16-bit word lengths can be used. If the magnitude of the word value is greater than the maximum allowed, the value is automatically compressed to the maximum positive or negative value.

Lengths greater than 16 bits are not supported for companding operation.

Clock Signal Options

Each SPORT has a transmit clock signal (TSCLK) and a receive clock signal (RSCLK). The clock signals are configured by the TCKFE and RCKFE bits of the SPORTx_TCR1 and SPORTx_RCR1 registers. Serial clock frequency is configured in the SPORTx_TCLKDIV and SPORTx_RCLKDIV registers.

The receive clock pin may be tied to the transmit clock if a single clock is desired for both receive and transmit.

Both transmit and receive clocks can be independently generated internally or input from an external source. The ITCLK bit of the SPORTx_TCR1 Configuration register and the IRCLK bit in the SPORTx_RCR1 Configuration register determines the clock source.

When IRCLK or ITCLK = 1, the clock signal is generated internally by the core, and the TSCLK or RSCLK pin is an output. The clock frequency is determined by the value of the serial clock divisor in the SPORTx_RCLKDIV register.

When IRCLK or ITCLK = 0, the clock signal is accepted as an input on the TSCLK or RSCLK pins, and the serial clock divisors in the SPORTx_TCLKDIV/SPORTx_RCLKDIV registers are ignored. The externally generated serial clocks do not need to be synchronous with the core system clock or with each other. The core system clock must have a higher frequency than RSCLK and TSCLK.

Frame Sync Options

Framing signals indicate the beginning of each serial word transfer. The framing signals for each SPORT are TFS (Transmit Frame Sync) and RFS (Receive Frame Sync). A variety of framing options are available; these options are configured in the SPORT Configuration registers (SPORTx_TCR1, SPORTx_TCR2, SPORTx_RCR1 and SPORTx_RCR2). The TFS and RFS signals of a SPORT are independent and are separately configured in the control registers.

Framed Versus Unframed

The use of multiple frame sync signals is optional in SPORT communications. The TFSR (Transmit Frame Sync Required Select) and RFSR (Receive Frame Sync Required Select) control bits determine whether frame sync signals are required. These bits are located in the SPORTx_TCR1 and SPORTx_RCR1 registers.

When TFSR = 1 or RFSR = 1, a frame sync signal is required for every data word. To allow continuous transmitting by the SPORT, each new data word must be loaded into the SPORTx_TX Hold register before the previous word is shifted out and transmitted.

When TFSR = 0 or RFSR = 0, the corresponding frame sync signal is not required. A single frame sync is needed to initiate communications but is ignored after the first bit is transferred. Data words are then transferred continuously, unframed.

With frame syncs not required, interrupt or DMA requests may not be serviced frequently enough to guarantee continuous unframed data flow. Monitor status bits or check for a SPORT Error interrupt to detect underflow or overflow of data.

Figure 12-17 illustrates framed serial transfers, which have these characteristics:

- TFSR and RFSR bits in the SPORTx_TCR1 and SPORTx_RCR1 registers determine framed or unframed mode.
- Framed mode requires a framing signal for every word. Unframed mode ignores a framing signal after the first word.
- Unframed mode is appropriate for continuous reception.
- Active low or active high frame syncs are selected with the LTFS and LRFS bits of the SPORTx_TCR1 and SPORTx_RCR1 registers.

See "Timing Examples" on page 12-66 for more timing examples.

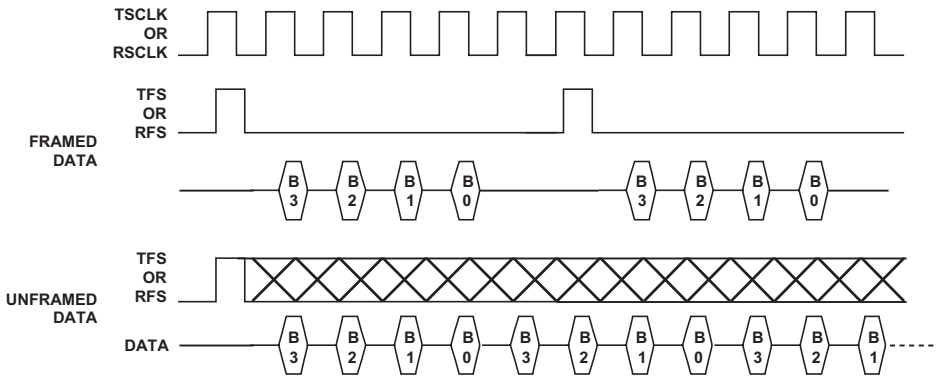


Figure 12-17. Framed Versus Unframed Data

Internal Versus External Frame Syncs

Both Transmit and Receive Frame Syncs can be independently generated internally or can be input from an external source. The ITFS and IRFS bits of the SPORTx_TCR1 and SPORTx_RCR1 registers determine the frame sync source.

When ITFS = 1 or IRFS = 1, the corresponding frame sync signal is generated internally by the SPORT, and the TFS pin or RFS pin is an output. The frequency of the frame sync signal is determined by the value of the frame sync divisor in the SPORTx_TFSDIV or SPORTx_RFSDIV register.

When ITFS = 0 or IRFS = 0, the corresponding frame sync signal is accepted as an input on the TFS pin or RFS pin, and the frame sync divisors in the SPORTx_TFSDIV/SPORTx_RFSDIV registers are ignored.

All of the frame sync options are available whether the signal is generated internally or externally.

Active Low Versus Active High Frame Syncs

Frame sync signals may be either active high or active low (in other words, inverted). The LTFS and LRFS bits of the SPORTx_TCR1 and SPORTx_RCR1 registers determine frame sync logic levels:

- When LTFS = 0 or LRFS = 0, the corresponding frame sync signal is active high.
- When LTFS = 1 or LRFS = 1, the corresponding frame sync signal is active low.

Active high frame syncs are the default. The LTFS and LRFS bits are initialized to 0 after a processor reset.

Sampling Edge for Data and Frame Syncs

Data and frame syncs can be sampled on either the rising or falling edges of the SPORT clock signals. The TCKFE and RCKFE bits of the SPORTx_TCR1 and SPORTx_RCR1 registers select the driving and sampling edges of the serial data and frame syncs.

For the SPORT transmitter, setting TCKFE = 1 in the SPORTx_TCR1 register selects the falling edge of TSCLKx to drive data and internally generated frame syncs and selects the rising edge of TSCLKx to sample externally generated frame syncs. Setting TCKFE = 0 selects the rising edge of TSCLKx to drive data and internally generated frame syncs and selects the falling edge of TSCLKx to sample externally generated frame syncs.

For the SPORT receiver, setting RCKFE = 1 in the SPORTx_RCR1 register selects the falling edge of RSCLKx to drive internally generated frame syncs and selects the rising edge of RSCLKx to sample data and externally generated frame syncs. Setting RCKFE = 0 selects the rising edge of RSCLKx to drive internally generated frame syncs and selects the falling edge of RSCLKx to sample data and externally generated frame syncs.

Note externally generated data and frame sync signals should change state on the opposite edge than that selected for sampling. For example, for an externally generated frame sync to be sampled on the rising edge of the clock (TCKFE = 1 in the SPORTx_TCR1 register), the frame sync must be driven on the falling edge of the clock.

The transmit and receive functions of two SPORTs connected together should always select the same value for TCKFE in the transmitter and RCKFE in the receiver, so that the transmitter drives the data on one edge and the receiver samples the data on the opposite edge.

In Figure 12-18, TCKFE = RCKFE = 0 and transmit and receive are connected together to share the same clock and frame syncs.

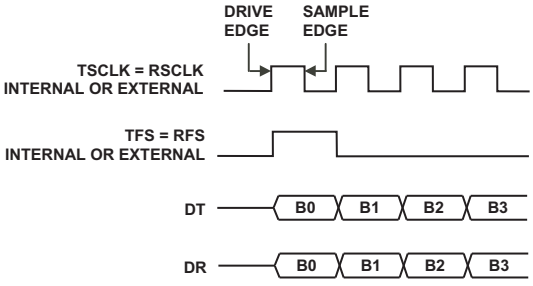


Figure 12-18. Example of TCKFE = RCKFE = 0, Transmit and Receive Connected

In Figure 12-19, TCKFE = RCKFE = 1 and transmit and receive are connected together to share the same clock and frame syncs.

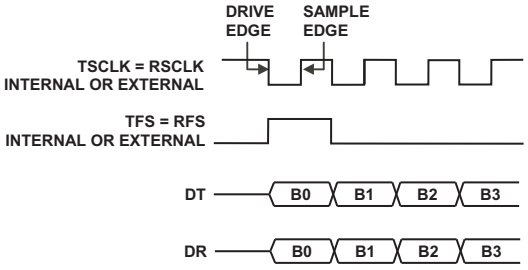


Figure 12-19. Example of TCKFE = RCKFE = 1, Transmit and Receive Connected

Early Versus Late Frame Syncs (Normal Versus Alternate Timing)

Frame sync signals can occur during the first bit of each data word (late) or during the serial clock cycle immediately preceding the first bit (early). The LATFS and LARFS bits of the SPORTx_TCR1 and SPORTx_RCR1 registers configure this option.

When LATFS = 0 or LARFS = 0, early frame syncs are configured; this is the normal mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the serial clock cycle after the frame sync is asserted, and the frame sync is not checked again until the entire word has been transmitted or received. In multichannel operation, this corresponds to the case when multichannel frame delay is 1.

If data transmission is continuous in early framing mode (in other words, the last bit of each word is immediately followed by the first bit of the next word), then the frame sync signal occurs during the last bit of each word.

Internally generated frame syncs are asserted for one clock cycle in early framing mode. Continuous operation is restricted to word sizes of 4 or longer (SLEN ≥ 3).

When LATFS = 1 or LARFS = 1, late frame syncs are configured; this is the alternate mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted. In multichannel operation, this is the case when frame delay is 0. Receive data bits are sampled by serial clock edges, but the frame sync signal is only checked during the first bit of each word. Internally generated frame syncs remain asserted for the entire length of the data word in late framing mode. Externally generated frame syncs are only checked during the first bit.

Figure 12-20 illustrates the two modes of frame signal timing. In summary:

- For the LATFS or LARFS bits of the SPORTx_TCR1 or SPORTx_RCR1 registers: LATFS = 0 or LARFS = 0 for early frame syncs, LATFS = 1 or LARFS = 1 for late frame syncs.
- For early framing, the frame sync precedes data by one cycle. For late framing, the frame sync is checked on the first bit only.
- Data is transmitted MSB first (TLSBIT = 0 or RLSBIT = 0) or LSB first (TLSBIT = 1 or RLSBIT = 1).
- Frame sync and clock are generated internally or externally.

See “Timing Examples” on page 12-66 for more examples.

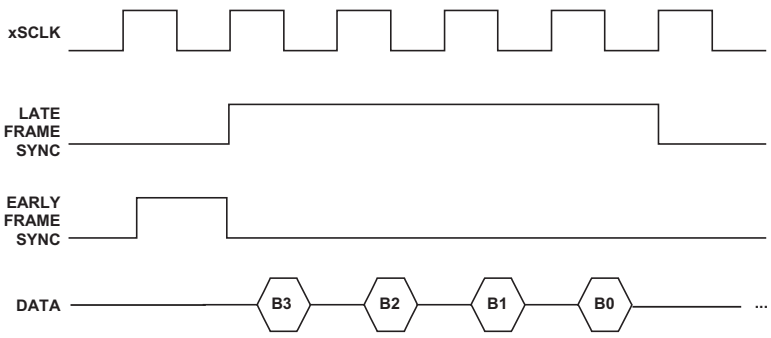


Figure 12-20. Normal Versus Alternate Framing

Data Independent Transmit Frame Sync

Normally the internally generated Transmit Frame Sync signal (TFS) is output only when the SPORTx_TX buffer has data ready to transmit. The Data-Independent Transmit Frame Sync Select bit (DITFS) allows the continuous generation of the TFS signal, with or without new data. The DITFS bit of the SPORTx_TCR1 register configures this option.

When DITFS = 0, the internally generated TFS is only output when a new data word has been loaded into the SPORTx_TX buffer. The next TFS is generated once data is loaded into SPORTx_TX. This mode of operation allows data to be transmitted only when it is available.

When DITFS = 1, the internally generated TFS is output at its programmed interval regardless of whether new data is available in the SPORTx_TX buffer. Whatever data is present in SPORTx_TX is transmitted again with each assertion of TFS. The TUVF (Transmit Underflow Status) bit in the SPORTx_STAT register is set when this occurs and old data is retransmitted. The TUVF status bit is also set if the SPORTx_TX buffer does not have new data when an externally generated TFS occurs. Note that in this mode of operation, data is transmitted only at specified times.

If the internally generated TFS is used, a single write to the SPORTx_TX data register is required to start the transfer.

Moving Data Between SPORTs and Memory

Transmit and receive data can be transferred between the SPORTs and on-chip memory in one of two ways: with single word transfers or with DMA block transfers.

If no SPORT DMA channel is enabled, the SPORT generates an interrupt every time it has received a data word or needs a data word to transmit. SPORT DMA provides a mechanism for receiving or transmitting an entire block or multiple blocks of serial data before the interrupt is generated. The SPORT's DMA controller handles the DMA transfer, allowing the processor core to continue running until the entire block of data is transmitted or received. Interrupt service routines (ISRs) can then operate on the block of data rather than on single words, significantly reducing overhead.

For information about DMA, see Chapter 9, "Direct Memory Access."

Stereo Serial Operation

Several Stereo Serial modes can be supported by the SPORT, including the popular I^2S format. To use these modes, set bits in the SPORT_RCR2 or SPORT_TCR2 registers. Setting RSFSE or TSFSE in SPORT_RCR2 or SPORT_TCR2 changes the operation of the frame sync pin to a left/right clock as required for I^2S and left-justified stereo serial data. Setting this bit enables the SPORT to generate or accept the special LRCLK-style frame sync. All other SPORT control bits remain in effect and should be set appropriately. Figure 12-21 and Figure 12-22 show timing diagrams for Stereo Serial mode operation.

Table 12-3 shows several modes that can be configured using bits in SPORTx_TCR1 and SPORTx_RCR1. The table shows bits for the receive side of the SPORT, but corresponding bits are available for configuring the transmit portion of the SPORT. A control field which may be either set or cleared depending on the user's needs, without changing the standard, is indicated by an "X."

Table 12-3. Stereo Serial Settings

Bit Field	Stereo Audio Serial Scheme		
	I^2S	Left-Justified	DSP Mode
RSFSE	1	1	0
RRFST	0	0	0
LARFS	0	1	0
LRFS	0	1	0
RFSR	1	1	1
RCKFE	1	0	0
SLEN	2 – 31	2 – 31	2 – 31
RLSBIT	0	0	0
RFSDIV (If internal FS is selected.)	2 – Max	2 – Max	2 – Max
RXSE (Secondary Enable is available for RX and TX.)	X	X	X

Note most bits shown as a 0 or 1 may be changed depending on the user's preference, creating many other "almost standard" modes of stereo serial operation. These modes may be of use in interfacing to codecs with slightly non-standard interfaces. The settings shown in Table 12-3 provide glueless interfaces to many popular codecs.

Note RFSDIV or TFSDIV must still be greater than or equal to SLEN. For I^2S operation, RFSDIV or TFSDIV is usually 1/64 of the serial clock rate. With RSFSE set, the formulas to calculate frame sync period and frequency (discussed in "Clock and Frame Sync Frequencies" on page 12-31) still apply, but now refer to one half the period and twice the frequency. For instance, setting RFSDIV or TFSDIV = 31 produces an LRCLK that transitions every 32 serial clock cycles and has a period of 64 serial clock cycles.

The LRFS bit determines the polarity of the RFS or TFS frame sync pin for the channel that is considered a "right" channel. Thus, setting LRFS = 0 (meaning that it is an active high signal) indicates that the frame sync is high for the "right" channel, thus implying that it is low for the "left" channel. This is the default setting.

The RRFST and TRFST bits determine whether the first word received or transmitted is a left or a right channel. If the bit is set, the first word received or transmitted is a right channel. The default is to receive or transmit the left channel word first.

The secondary DRxSEC and DTxSEC pins are useful extensions of the serial port which pair well with Stereo Serial mode. Multiple I^2S streams of data can be transmitted or received using a single SPORT. Note the primary and secondary pins are synchronous, as they share clock and LRCLK (Frame Sync) pins. The transmit and receive sides of the SPORT need not be synchronous, but may share a single clock in some designs. See Figure 12-3, which shows multiple stereo serial connections being made between the processor and an AD1836 codec.

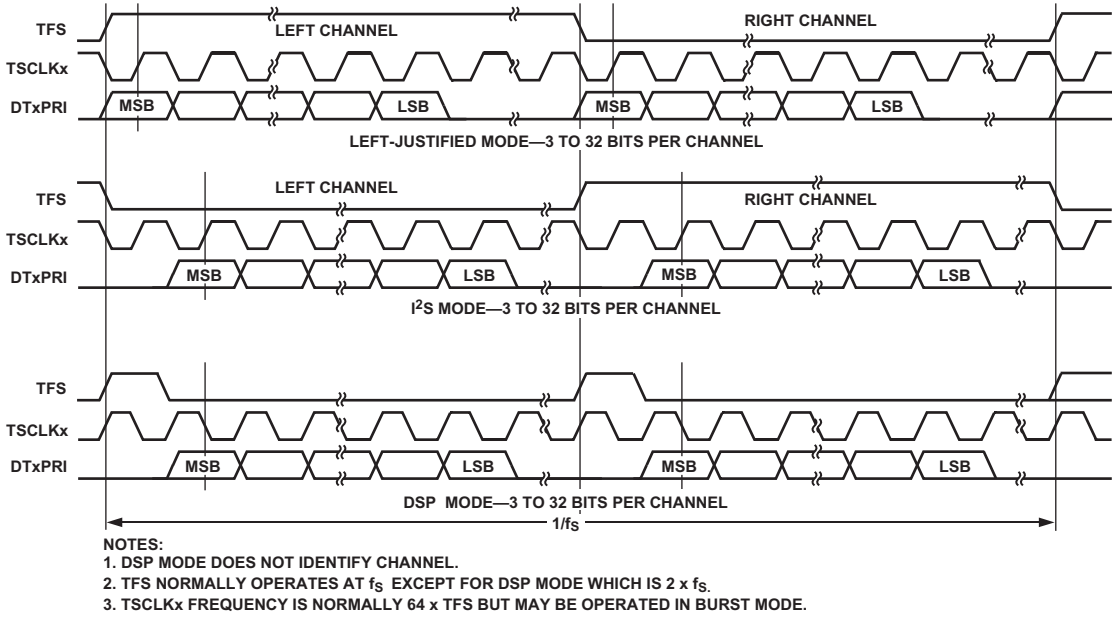


Figure 12-21. SPORT Stereo Serial Modes, Transmit

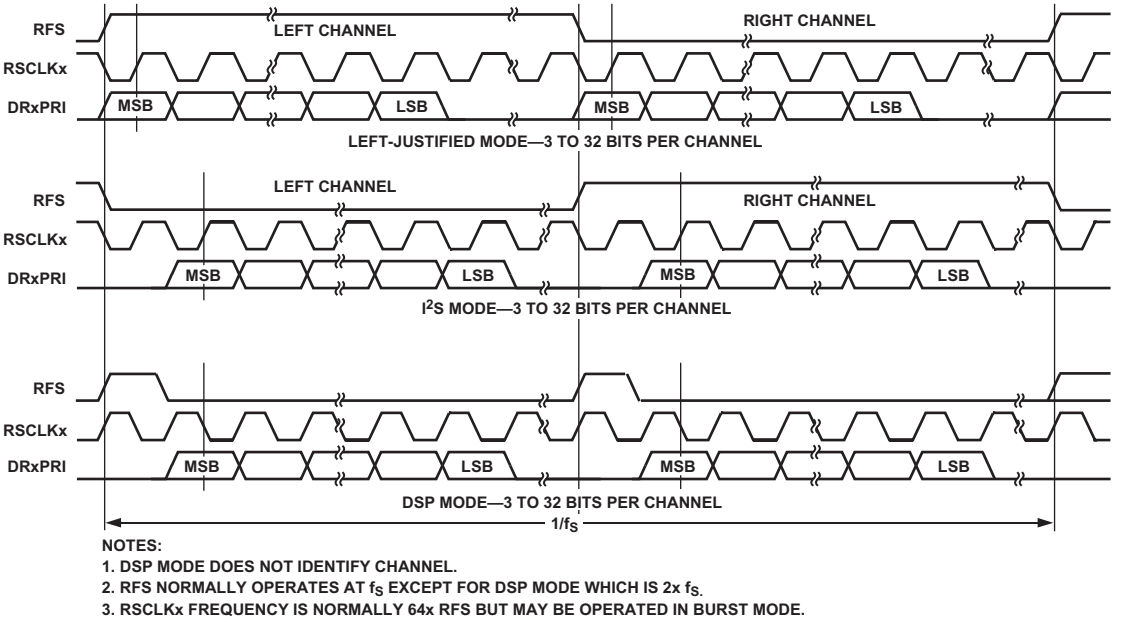


Figure 12-22. SPORT Stereo Serial Modes, Receive

The Blackfin processor's SPORTs are designed such that in I^2S master mode, the LRCLK signal is held at the last driven logic level. The LRCLK signal does not transition (provide an edge) after the final data word is driven out. While transmitting a fixed number of words to an I^2S receiver that expects an LRCLK edge to receive the incoming data word, the SPORT should send a dummy word after transmitting the fixed number of words. The transmission of this dummy word toggles LRCLK, generating an edge. Transmission of the dummy word is not required when the I^2S receiver is a Blackfin processor SPORT.

Multichannel Operation

The SPORTs offer a Multichannel mode of operation which allows the SPORT to communicate in a Time-Division-Multiplexed (TDM) serial system. In multichannel communications, each data word of the serial bitstream 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.

The SPORT can automatically select words for particular channels while ignoring the others. Up to 128 channels are available for transmitting or receiving; each SPORT can receive and transmit data selectively from any of the 128 channels. These 128 channels can be any 128 out of the 1024 total channels. RX and TX must use the same 128-channel region to selectively enable channels. The SPORT can do any of the following on each channel:

- Transmit data
- Receive data
- Transmit and receive data
- Do nothing

Data companding and DMA transfers can also be used in multichannel mode.

The DTPRI pin is always driven (not three-stated) if the SPORT is enabled (TSPEN = 1 in the SPORTx_TCR1 register), unless it is in Multichannel mode and an inactive time slot occurs. The DTSEC pin is always driven (not three-stated) if the SPORT is enabled and the secondary transmit is enabled (TXSE = 1 in the SPORTx_TCR2 register), unless the SPORT is in Multichannel mode and an inactive time slot occurs.

In Multichannel mode, RSCLK can either be provided externally or generated internally by the SPORT, and it is used for both transmit and receive functions. Leave TSCLK disconnected if the SPORT is used only in multichannel mode. If RSCLK is externally or internally provided, it will be internally distributed to both the receiver and transmitter circuitry.

The SPORT Multichannel Transmit Select register and the SPORT Multichannel Receive Select register must be programmed before enabling SPORTx_TX or SPORTx_RX operation for Multichannel Mode. This is especially important in "DMA data unpacked mode," since SPORT FIFO operation begins immediately after RSPEN and TSPEN are set, enabling both RX and TX. The MCMEN bit (in SPORTx_MCMC2) must be enabled prior to enabling SPORTx_TX or SPORTx_RX operation. When disabling the SPORT from multichannel operation, first disable TXEN and then disable RXEN. Note both TXEN and RXEN must be disabled before reenabling. Disabling only TX or RX is not allowed.

Figure 12-23 shows example timing for a multichannel transfer that has these characteristics:

- Use TDM method where serial data is sent or received on different channels sharing the same serial bus
- Can independently select transmit and receive channels
- RFS signals start of frame
- TFS is used as "Transmit Data Valid" for external logic, true only during transmit channels
- Receive on channels 0 and 2, transmit on channels 1 and 2
- Multichannel frame delay is set to 1

See "Timing Examples" on page 12-66 for more examples.

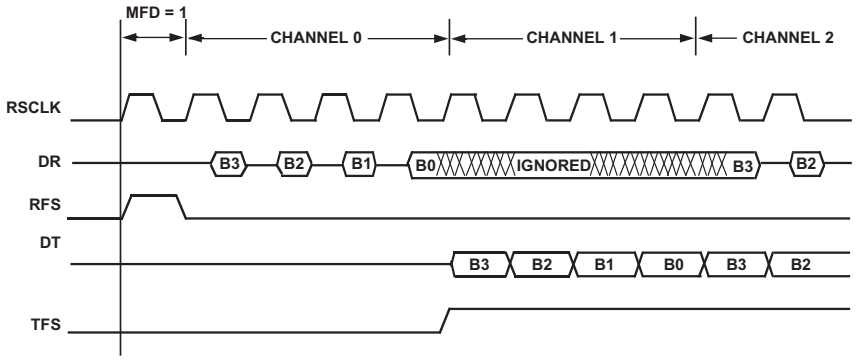


Figure 12-23. Multichannel Operation

SPORTx_MCMCn Registers

There are two SPORTx Multichannel Configuration registers (`SPORTx_MCMCn`) for each SPORT. The `SPORTx_MCMCn` registers are used to configure the multichannel operation of the SPORT. The two control registers are shown in Figure 12-24 and Figure 12-25.

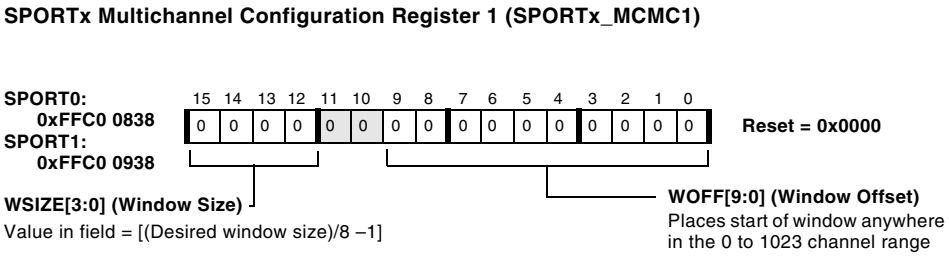


Figure 12-24. SPORTx Multichannel Configuration Register 1

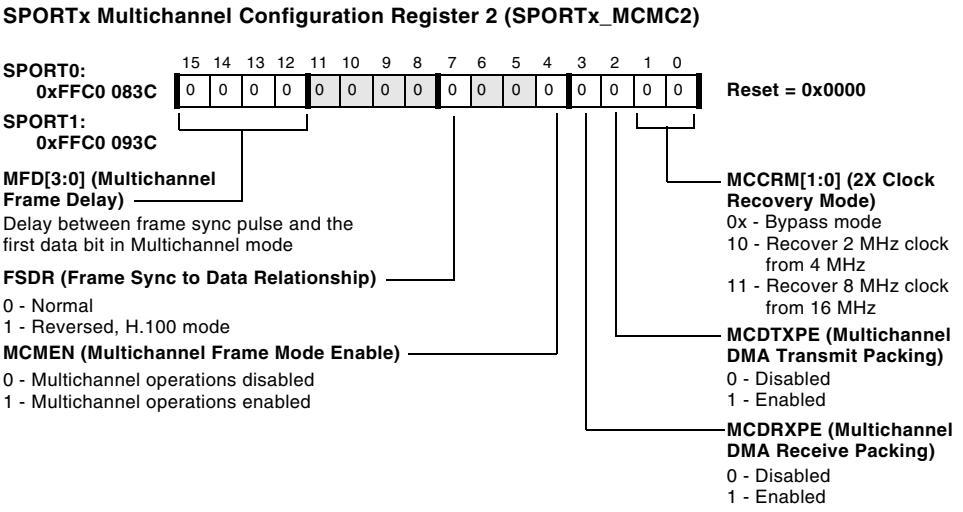


Figure 12-25. SPORTx Multichannel Configuration Register 2

Multichannel Enable

Setting the MCMEN bit in the SPORTx_MCM2 register enables Multichannel mode. When MCMEN = 1, multichannel operation is enabled; when MCMEN = 0, all multichannel operations are disabled.

Setting the MCMEN bit enables multichannel operation for *both* the receive and transmit sides of the SPORT. Therefore, if a receiving SPORT is in Multichannel mode, the transmitting SPORT must also be in Multichannel mode.

When in multichannel mode, do not enable the stereo serial frame sync modes or the late frame sync feature, as these features are incompatible with multichannel mode.

Table 12-4 shows the dependencies of bits in the SPORT Configuration register when the SPORT is in Multichannel Mode.

Table 12-4. Multichannel Mode Configuration

SPORTx_RCR1 or SPORTx_RCR2	SPORTx_TCR1 or SPORTx_TCR2	Notes
RSPEN	TSPEN	Set or clear both
IRCLK	-	Independent
-	ITCLK	Ignored
RDTYPE	TDTYPE	Independent
RLSBIT	TLSBIT	Independent
IRFS	-	Independent
-	ITFS	Ignored
RFSR	TFSR	Ignored
-	DITFS	Ignored
LRFS	LTFS	Independent
LARFS	LATFS	Both must be 0
RCKFE	TCKFE	Set or clear both to same value
SLEN	SLEN	Set or clear both to same value
RXSE	TXSE	Independent
RSFSE	TSFSE	Both must be 0
RRFST	TRFST	Ignored

Frame Syncs in Multichannel Mode

All receiving and transmitting devices in a multichannel system must have the same timing reference. The RFS signal is used for this reference, indicating the start of a block or frame of multichannel data words.

When Multichannel mode is enabled on a SPORT, both the transmitter and the receiver use RFS as a frame sync. This is true whether RFS is generated internally or externally. The RFS signal is used to synchronize the channels and restart each multichannel sequence. Assertion of RFS indicates the beginning of the channel 0 data word.

Since RFS is used by both the SPORTx_TX and SPORTx_RX channels of the SPORT in Multichannel mode configuration, the corresponding bit pairs in SPORTx_RCR1 and SPORTx_TCR1, and in SPORTx_RCR2 and SPORTx_TCR2, should always be programmed identically, with the possible exception of the RXSE and TXSE pair and the RDTYPE and TDTYPE pair. This is true even if SPORTx_RX operation is not enabled.

In Multichannel mode, RFS timing similar to late (alternative) frame mode is entered automatically; the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted, provided that MFD is set to 0.

The TFS signal is used as a transmit data valid signal which is active during transmission of an enabled word. The SPORT's data transmit pin is three-stated when the time slot is not active, and the TFS signal serves as an output-enabled signal for the data transmit pin. The SPORT drives TFS in Multichannel mode whether or not ITFS is cleared. The TFS pin in Multichannel mode still obeys the LTFS bit. If LTFS is set, the transmit data valid signal will be active low—a low signal on the TFS pin indicates an active channel.

Once the initial RFS is received, and a frame transfer has started, all other RFS signals are ignored by the SPORT until the complete frame has been transferred.

If MFD > 0, the RFS may occur during the last channels of a previous frame. This is acceptable, and the frame sync is not ignored as long as the delayed channel 0 starting point falls outside the complete frame.

In Multichannel mode, the RFS signal is used for the block or frame start reference, after which the word transfers are performed continuously with no further RFS signals required. Therefore, internally generated frame syncs are always data independent.

The Multichannel Frame

A multichannel frame contains more than one channel, as specified by the window size and window offset. A complete multichannel frame consists of 1 – 1024 channels, starting with channel 0. The particular channels of the multichannel frame that are selected for the SPORT are a combination of the window offset, the window size, and the Multichannel Select registers.

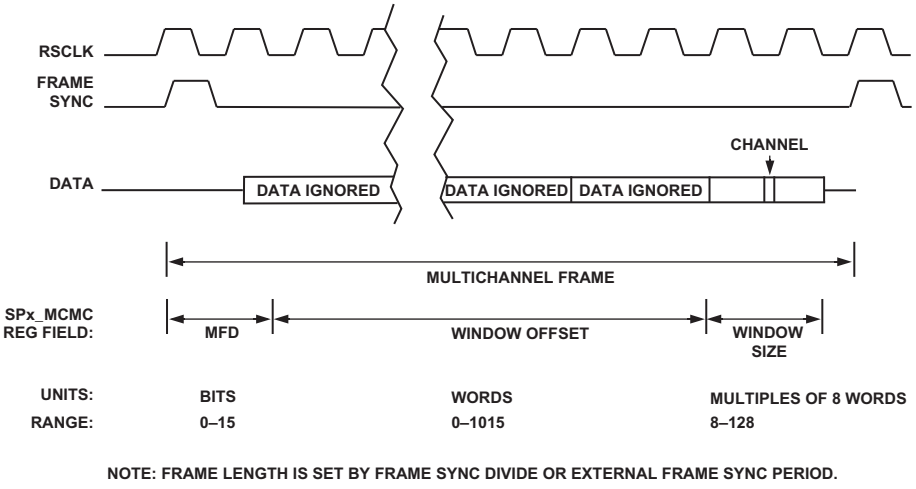


Figure 12-26. Relationships for Multichannel Parameters

Multichannel Frame Delay

The 4-bit MFD field in SPORTx_MCMC2 specifies a delay between the frame sync pulse and the first data bit in Multichannel mode. The value of MFD is the number of serial clock cycles of the delay. Multichannel frame delay allows the processor to work with different types of interface devices.

A value of 0 for MFD causes the frame sync to be concurrent with the first data bit. The maximum value allowed for MFD is 15. A new frame sync may occur before data from the last frame has been received, because blocks of data occur back-to-back.

Window Size

The window size (WSIZE[3:0]) defines the number of channels that can be enabled/disabled by the Multichannel Select registers. This range of words is called the active window. The number of channels can be any value in the range of 0 to 15, corresponding to active window size of 8 to 128, in increments of 8; the default value of 0 corresponds to a minimum active window size of 8 channels. To calculate the active window size from the WSIZE register, use this equation:

```
Number of words in active window = 8 x (WSIZE + 1)
```

Since the DMA buffer size is always fixed, it is possible to define a smaller window size (for example, 32 words), resulting in a smaller DMA buffer size (in this example, 32 words instead of 128 words) to save DMA bandwidth. The window size cannot be changed while the SPORT is enabled.

Multichannel select bits that are enabled but fall outside the window selected are ignored.

Window Offset

The window offset (WOFF[9:0]) specifies where in the 1024-channel range to place the start of the active window. A value of 0 specifies no offset and 896 is the largest value that permits using all 128 channels. As an example, a program could define an active window with a window size of 8 (WSIZE = 0) and an offset of 93 (WOFF = 93). This 8-channel window would reside in the range from 93 to 100. Neither the window offset nor the window size can be changed while the SPORT is enabled.

If the combination of the window size and the window offset would place any portion of the window outside of the range of the channel counter, none of the out-of-range channels in the frame are enabled.

SPORTx_CHNL Register

The 10-bit CHNL field in the SPORTx Current Channel register (SPORTx_CHNL) indicates which channel is currently being serviced during multichannel operation. This field is a read-only status indicator. The CHNL[9:0] field increments by one as each channel is serviced. The counter stops at the upper end of the defined window. The Channel Select register restarts at 0 at each frame sync. As an example, for a window size of 8 and an offset of 148, the counter displays a value between 0 and 156.

Once the window size has completed, the channel counter resets to 0 in preparation for the next frame. Because there are synchronization delays between RSCLK and the processor clock, the Channel register value is approximate. It is never ahead of the channel being served, but it may lag behind.

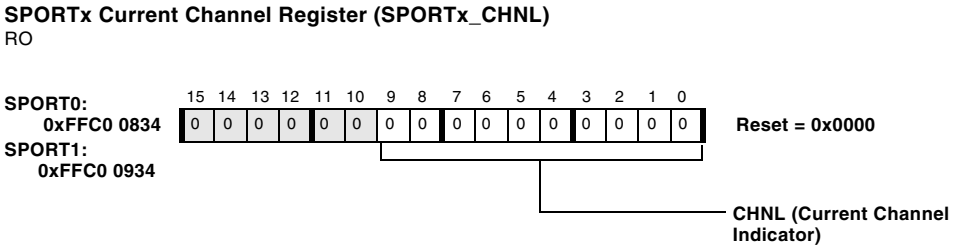


Figure 12-27. SPORTx Current Channel Register

Other Multichannel Fields in SPORTx_MCMC2

The FSDR bit in the SPORTx_MCMC2 register changes the timing relationship between the frame sync and the clock received. This change enables the SPORT to comply with the H.100 protocol.

Normally (When FSDR = 0), the data is transmitted on the same edge that the TFS is generated. For example, a positive edge on TFS causes data to be transmitted on the positive edge of the TSCLK—either the same edge or the following one, depending on when LATFS is set.

When the frame sync/data relationship is used (FSDR = 1), the frame sync is expected to change on the falling edge of the clock and is sampled on the rising edge of the clock. This is true even though data received is sampled on the negative edge of the receive clock.

Channel Selection Register

A channel is a multibit word from 3 to 32 bits in length that belongs to one of the TDM channels. Specific channels can be individually enabled or disabled to select which words are received and transmitted during multichannel communications. Data words from the enabled channels are received or transmitted, while disabled channel words are ignored. Up to 128 contiguous channels may be selected out of 1024 available channels.

The SPORTx_MRCSn and SPORTx_MTCSn Multichannel Select registers are used to enable and disable individual channels; the SPORTx_MRCSn registers specify the active receive channels, and the SPORTx_MTCSn registers specify the active transmit channels.

Four registers make up each Multichannel Select register. Each of the four registers has 32 bits, corresponding to 32 channels. Setting a bit enables that channel, so the SPORT selects its word from the multiple word block of data (for either receive or transmit).

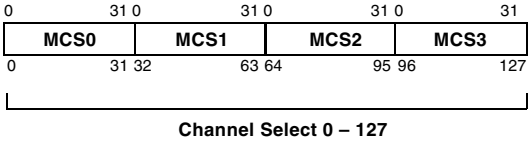


Figure 12-28. Multichannel Select Registers

Channel Select bit 0 always corresponds to the first word of the active window. To determine a channel's absolute position in the frame, add the window offset words to the channel select position. For example, setting bit 7 in MCS2 selects word 71 of the active window to be enabled. Setting bit 2 in MCS1 selects word 34 of the active window, and so on.

Setting a particular bit in the SPORTx_MTCSn register causes the SPORT to transmit the word in that channel's position of the datastream. Clearing the bit in the SPORTx_MTCSn register causes the SPORT's data transmit pin to three-state during the time slot of that channel.

Setting a particular bit in the SPORTx_MRCSn register causes the SPORT to receive the word in that channel's position of the datastream; the received word is loaded into the SPORTx_RX buffer. Clearing the bit in the SPORTx_MRCSn register causes the SPORT to ignore the data.

Companding may be selected for all channels or for no channels. A-law or μ-law companding is selected with the TDTYPE field in the SPORTx_TCR1 register and the RDTYPE field in the SPORTx_RCR1 register, and applies to all active channels. (See "Companding" on page 12-35 for more information about companding.)

SPORTx_MRCSn Registers

The Multichannel Selection registers are used to enable and disable individual channels. The SPORTx Multichannel Receive Select registers (SPORTx_MRCSn) specify the active receive channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the serial port selects that word for receive from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in the SPORTx_MRCSn register causes the serial port to receive the word in that channel's position of the datastream; the received word is loaded into the RX buffer. Clearing the bit in the SPORTx_MRCSn register causes the serial port to ignore the data.

SPORTx Multichannel Receive Select Registers (SPORTx_MRCSn)

For all bits, 0 - Channel disabled, 1 - Channel enabled, so SPORT selects that word from multiple word block of data.

For Memory-mapped addresses, see Table 12-5.

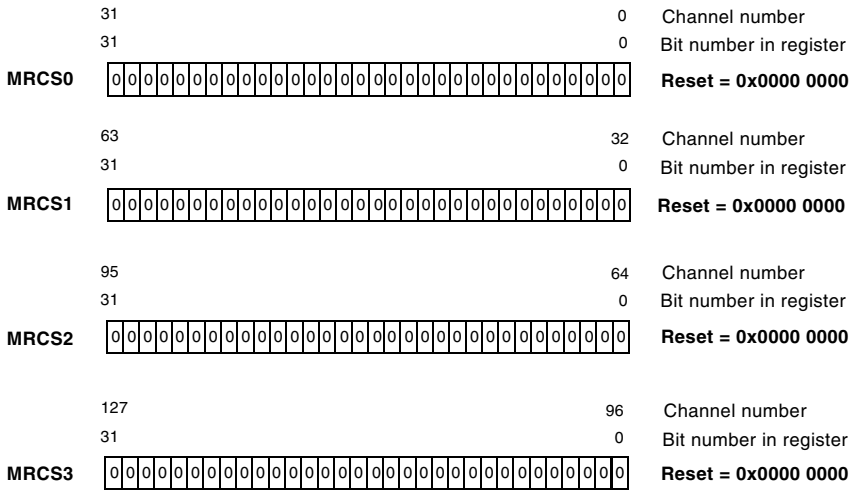


Figure 12-29. SPORTx Multichannel Receive Select Registers

Table 12-5. SPORTx Multichannel Receive Select Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
SPORT0_MRCS0	0xFFC0 0850
SPORT0_MRCS1	0xFFC0 0854
SPORT0_MRCS2	0xFFC0 0858
SPORT0_MRCS3	0xFFC0 085C
SPORT1_MRCS0	0xFFC0 0950
SPORT1_MRCS1	0xFFC0 0954

Table 12-5. SPORTx Multichannel Receive Select Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
SPORT1_MRCS2	0xFFC0 0958
SPORT1_MRCS3	0xFFC0 095C

SPORTx_MTCSn Registers

The Multichannel Selection registers are used to enable and disable individual channels. The four SPORTx Multichannel Transmit Select registers (`SPORTx_MTCSn`) specify the active transmit channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the serial port selects that word for transmit from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in a `SPORTx_MTCSn` register causes the serial port to transmit the word in that channel's position of the datastream. Clearing the bit in the `SPORTx_MTCSn` register causes the serial port's data transmit pin to three-state during the time slot of that channel.

SPORTx Multichannel Transmit Select Registers (SPORTx_MTCSn)

For all bits, 0 - Channel disabled, 1 - Channel enabled, so SPORT selects that word from multiple word block of data.

For Memory-mapped addresses, see Table 12-6.

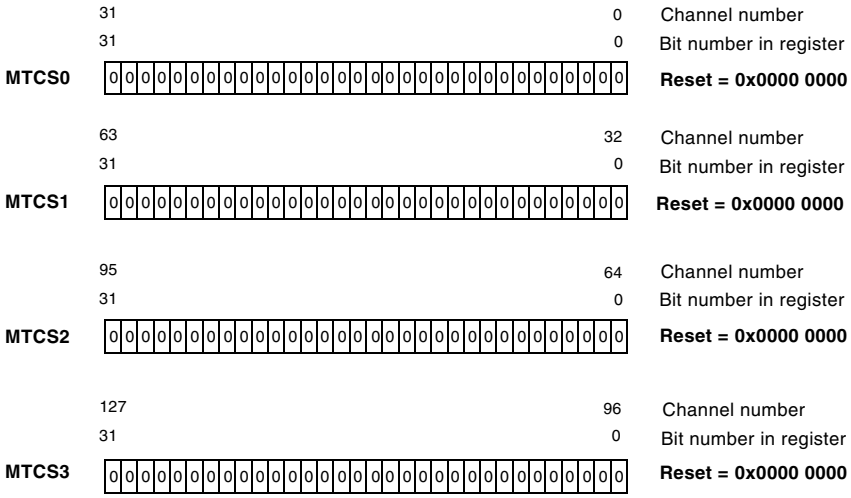


Figure 12-30. SPORTx Multichannel Transmit Select Registers

Table 12-6. SPORTx Multichannel Transmit Select Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
SPORT0_MTCS0	0xFFC0 0840
SPORT0_MTCS1	0xFFC0 0844
SPORT0_MTCS2	0xFFC0 0848
SPORT0_MTCS3	0xFFC0 084C
SPORT1_MTCS0	0xFFC0 0940
SPORT1_MTCS1	0xFFC0 0944

Table 12-6. SPORTx Multichannel Transmit Select Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
SPORT1_MTCS2	0xFFC0 0948
SPORT1_MTCS3	0xFFC0 094C

Multichannel DMA Data Packing

Multichannel DMA data packing and unpacking are specified with the MCDTXPE and MCDRXPE bits in the SPORTx_MCMC2 Multichannel Configuration register.

If the bits are set, indicating that data is packed, the SPORT expects the data contained by the DMA buffer corresponds only to the enabled SPORT channels. For example, if an MCM frame contains 10 enabled channels, the SPORT expects the DMA buffer to contain 10 consecutive words for each frame. It is not possible to change the total number of enabled channels without changing the DMA buffer size, and reconfiguring is not allowed while the SPORT is enabled.

If the bits are cleared (the default, indicating that data is not packed), the SPORT expects the DMA buffer to have a word for each of the channels in the active window, whether enabled or not, so the DMA buffer size must be equal to the size of the window. For example, if channels 1 and 10 are enabled, and the window size is 16, the DMA buffer size would have to be 16 words. The data to be transmitted or received would be placed at addresses 1 and 10 of the buffer, and the rest of the words in the DMA buffer would be ignored. This mode allows changing the number of enabled channels while the SPORT is enabled, with some caution. First read the Channel register to make sure that the active window is not being serviced. If the channel count is 0, then the Multichannel Select registers can be updated.

Support for H.100 Standard Protocol

The processor supports the H.100 standard protocol. The following SPORT parameters must be set to support this standard.

- Set for external frame sync. Frame sync generated by external bus master.
- TFSR/RFSR set (frame syncs required)
- LTFS/LRFS set (active low frame syncs)
- Set for external clock
- MCMEN set (Multichannel mode selected)
- MFD = 0 (no frame delay between frame sync and first data bit)
- SLEN = 7 (8-bit words)
- FSDR = 1 (set for H.100 configuration, enabling half-clock-cycle early frame sync)

2X Clock Recovery Control

The SPORTs can recover the data rate clock from a provided 2X input clock. This enables the implementation of H.100 compatibility modes for MVIP-90 (2 Mbps data) and HMVIP (8 Mbps data), by recovering 2 MHz from 4 MHz or 8 MHz from the 16 MHz incoming clock with the proper phase relationship. A 2-bit mode signal (MCCRM[1:0] in the SPORTx_MCMC2 register) chooses the applicable clock mode, which includes a non-divide or bypass mode for normal operation. A value of MCCRM = 00 chooses non-divide or bypass mode (H.100-compatible), MCCRM = 10 chooses MVIP-90 clock divide (extract 2 MHz from 4 MHz), and MCCRM = 11 chooses HMVIP clock divide (extract 8 MHz from 16 MHz).

SPORT Pin/Line Terminations

The processor has very fast drivers on all output pins, including the SPORTs. If connections on the data, clock, or frame sync lines are longer than six inches, consider using a series termination for strip lines on point-to-point connections. This may be necessary even when using low speed serial clocks, because of the edge rates.

Timing Examples

Several timing examples are included within the text of this chapter (in the sections "Framed Versus Unframed" on page 12-36, "Early Versus Late Frame Syncs (Normal Versus Alternate Timing)" on page 12-41, and "Frame Syncs in Multichannel Mode" on page 12-53). This section contains additional examples to illustrate other possible combinations of the framing options.

These timing examples show the relationships between the signals but are not scaled to show the actual timing parameters of the processor. Consult the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* for actual timing parameters and values.

These examples assume a word length of four bits (`SLEN = 3`). Framing signals are active high (`LRFS = 0` and `LTFS = 0`).

Figure 12-31 through Figure 12-36 show framing for receiving data.

In Figure 12-31 and Figure 12-32, the normal framing mode is shown for non-continuous data (any number of TSCLK or RSCLK cycles between words) and continuous data (no TSCLK or SCLK cycles between words).

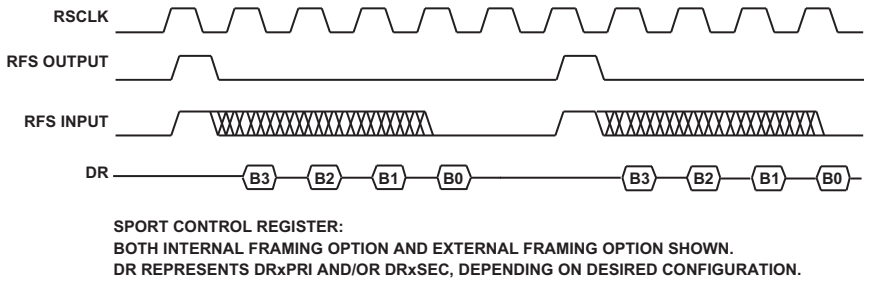


Figure 12-31. SPORT Receive, Normal Framing

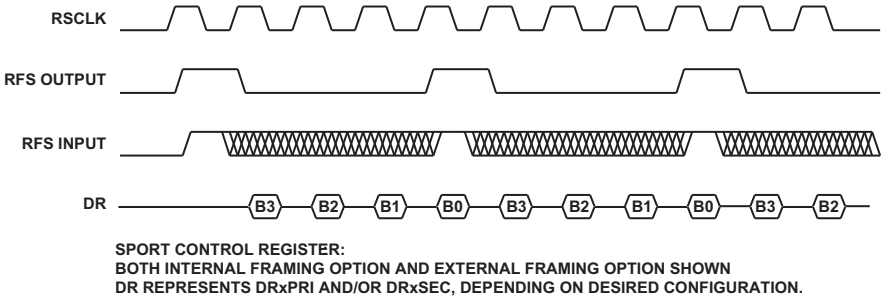


Figure 12-32. SPORT Continuous Receive, Normal Framing

Figure 12-33 and Figure 12-34 show non-continuous and continuous receiving in the alternate framing mode. These four figures show the input timing requirement for an externally generated frame sync and also the output timing characteristic of an internally generated frame sync. Note

the output meets the input timing requirement; therefore, with two SPORT channels used, one SPORT channel could provide RFS for the other SPORT channel.

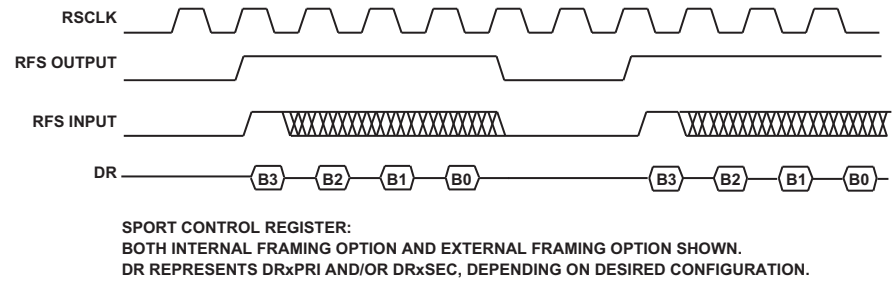


Figure 12-33. SPORT Receive, Alternate Framing

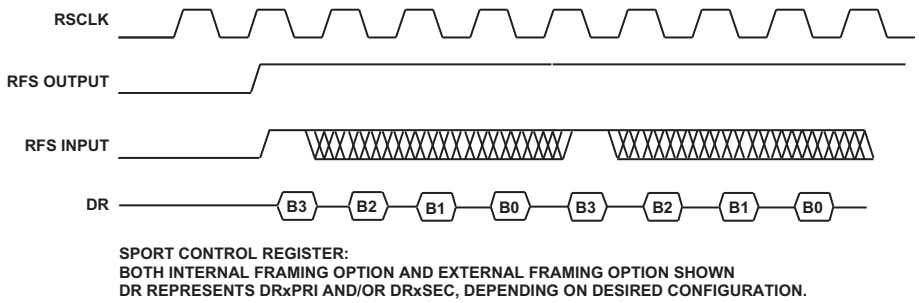


Figure 12-34. SPORT Continuous Receive, Alternate Framing

Figure 12-35 and Figure 12-36 show the receive operation with normal framing and alternate framing, respectively, in the unframed mode. A single frame sync signal occurs only at the start of the first word, either one

RSCLK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode). This mode is appropriate for multiword bursts (continuous reception).

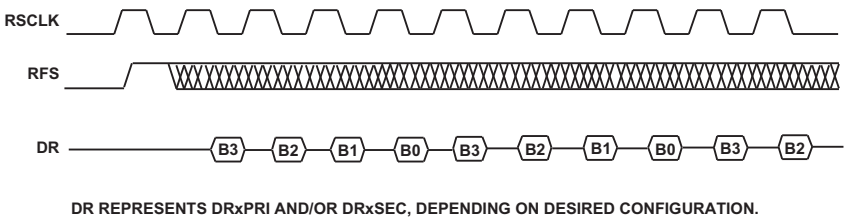


Figure 12-35. SPORT Receive, Unframed Mode, Normal Framing

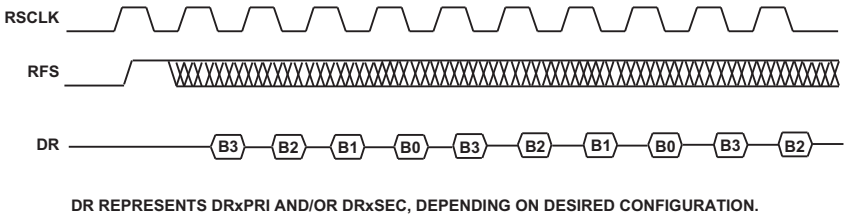


Figure 12-36. SPORT Receive, Unframed Mode, Alternate Framing

Figure 12-37 through Figure 12-42 show framing for transmitting data and are very similar to Figure 12-31 through Figure 12-36. In Figure 12-37 and Figure 12-38, the normal framing mode is shown for non-continuous data (any number of TSCLK cycles between words) and continuous data (no TSCLK cycles between words). Figure 12-39 and

Figure 12-40 show non-continuous and continuous transmission in the alternate framing mode. As noted previously for the receive timing diagrams, the RFS output meets the RFS input timing requirement.

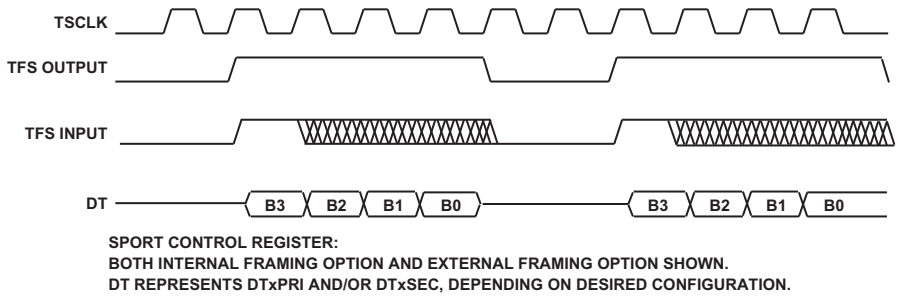


Figure 12-37. SPORT Transmit, Normal Framing

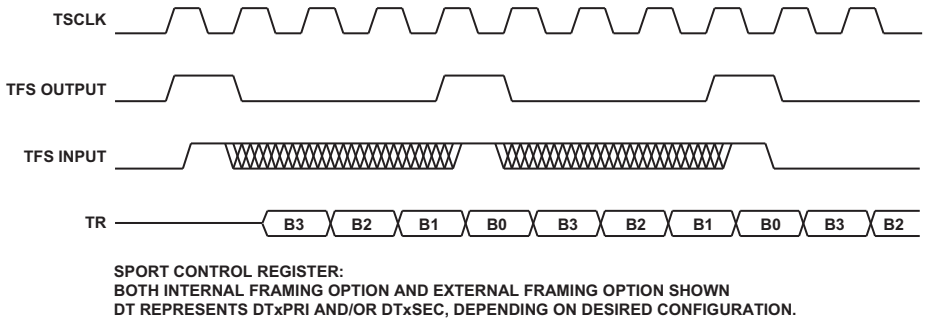


Figure 12-38. SPORT Continuous Transmit, Normal Framing

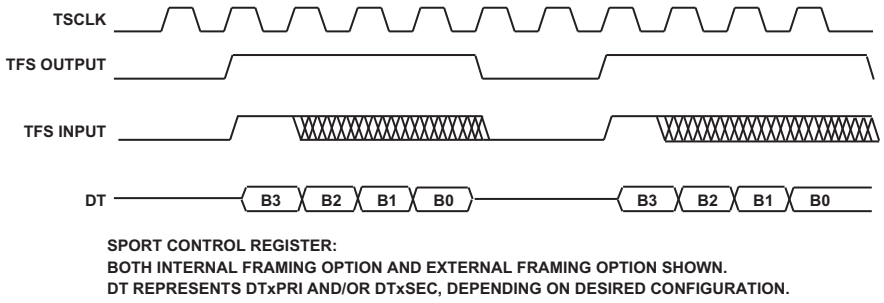


Figure 12-39. SPORT Transmit, Alternate Framing

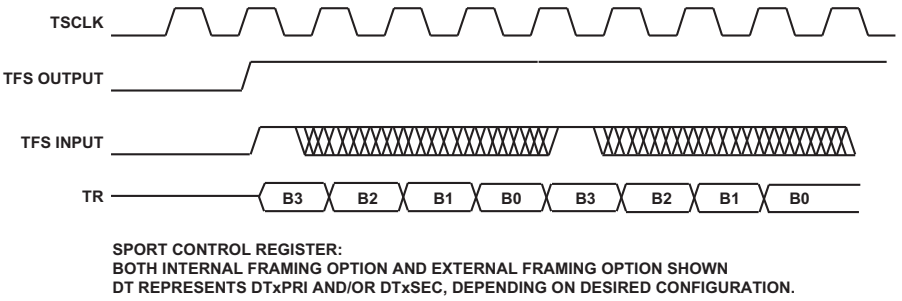


Figure 12-40. SPORT Continuous Transmit, Alternate Framing

Figure 12-41 and Figure 12-42 show the transmit operation with normal framing and alternate framing, respectively, in the unframed mode. A single frame sync signal occurs only at the start of the first word, either one TSCLK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode).

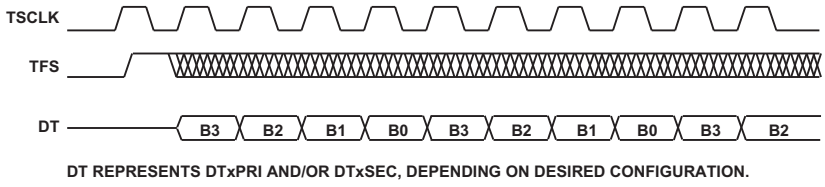


Figure 12-41. SPORT Transmit, Unframed Mode, Normal Framing

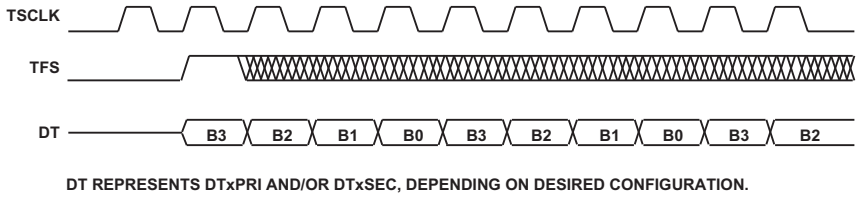


Figure 12-42. SPORT Transmit, Unframed Mode, Alternate Framing

13 UART PORT CONTROLLER

The Universal Asynchronous Receiver/Transmitter (UART) is a full-duplex peripheral compatible with PC-style industry-standard UARTs. The UART converts data between serial and parallel formats. The serial communication follows an asynchronous protocol that supports various word length, stop bits, and parity generation options. The UART includes interrupt handling hardware. Interrupts can be generated from 12 different events.

The UART supports the half-duplex IrDA® (Infrared Data Association) SIR (9.6/115.2 Kbps rate) protocol. This is a mode-enabled feature.

Modem status and control functionality is not supported by the UART module, but may be implemented using General-Purpose I/O (GPIO) pins.

The UART is a DMA-capable peripheral with support for separate TX and RX DMA master channels. It can be used in either DMA or programmed non-DMA mode of operation. The non-DMA mode requires software management of the data flow using either interrupts or polling. The DMA method requires minimal software intervention as the DMA engine itself moves the data. See Chapter 9, “Direct Memory Access” for more information on DMA.

Either one of the peripheral timers can be used to provide a hardware assisted autobaud detection mechanism for use with the UART. See Chapter 15, “Timers,” for more information.

Serial Communications

The UART follows an asynchronous serial communication protocol with these options:

- 5 – 8 data bits
- 1, 1½, or 2 stop bits
- None, even, or odd parity
- Baud rate = `SCLK/(16 × Divisor)`, where `SCLK` is the system clock frequency and Divisor can be a value from 1 to 65536

All data words require a start bit and at least one stop bit. With the optional parity bit, this creates a 7- to 12-bit range for each word. The format of received and transmitted character frames is controlled by the Line Control register (`UART_LCR`). Data is always transmitted and received least significant bit (LSB) first.

Figure 13-1 shows a typical physical bitstream measured on the TX pin.

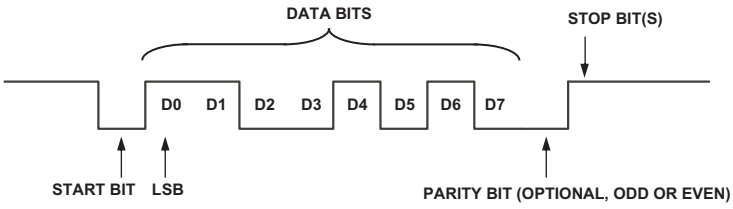


Figure 13-1. Bitstream on the TX Pin

UART Control and Status Registers

The processor provides a set of PC-style industry-standard control and status registers for each UART. These Memory-mapped Registers (MMRs) are byte-wide registers that are mapped as half words with the most significant byte zero filled.

Consistent with industry-standard interfaces, multiple registers are mapped to the same address location. The Divisor Latch registers (UART_DLH and UART_DLL) share their addresses with the Transmit Holding register (UART_THR), the Receive Buffer register (UART_RBR), and the Interrupt Enable register (UART_IER). The Divisor Latch Access bit (DLAB) in the Line Control Register (UART_LCR) controls which set of registers is accessible at a given time. Software must use 16-bit word load/store instructions to access these registers.

Transmit and receive channels are both buffered. The UART_THR register buffers the Transmit Shift register (TSR) and the UART_RBR register buffers the Receive Shift register (RSR). The shift registers are not directly accessible by software.

UART_LCR Register

The UART Line Control register (UART_LCR) controls the format of received and transmitted character frames. The SB bit functions even when the UART clock is disabled. Since the TX pin normally drives high, it can be used as a flag output pin, if the UART is not used.

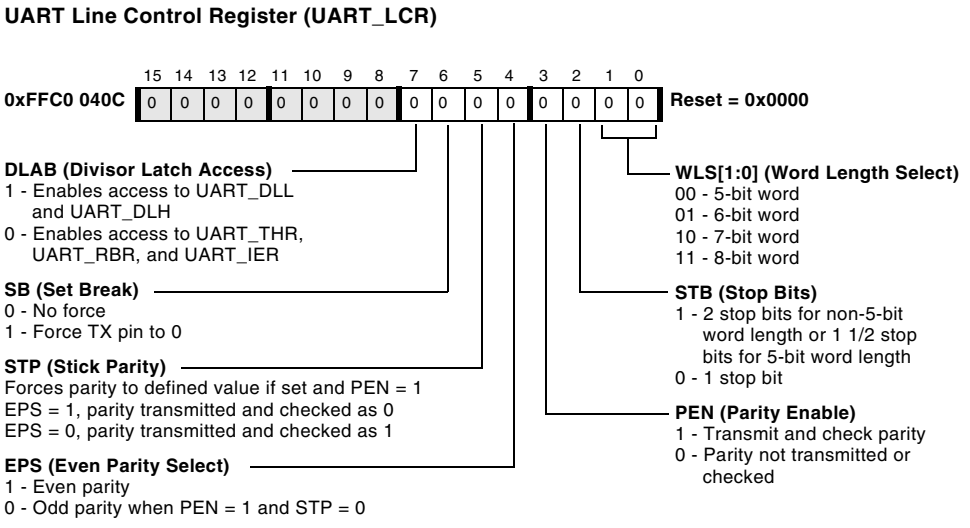


Figure 13-2. UART Line Control Register

UART_MCR Register

The Modem Control register (UART_MCR) controls the UART port, as shown in Figure 13-3. Even if modem functionality is not supported, the Modem Control register is available in order to support the loopback mode.

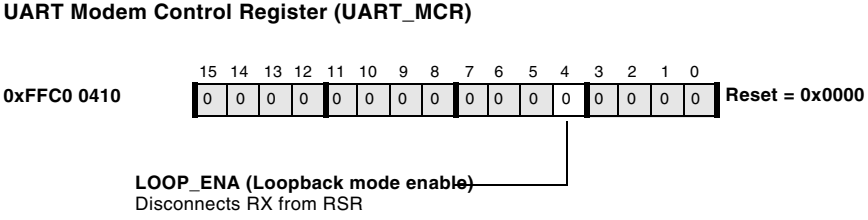


Figure 13-3. UART Modem Control Register

Loopback mode disconnects the receiver's input from the RX pin, but redirects it to the transmit output internally.

UART_LSR Register

The UART Line Status register (UART_LSR) contains UART status information as shown in Figure 13-4.

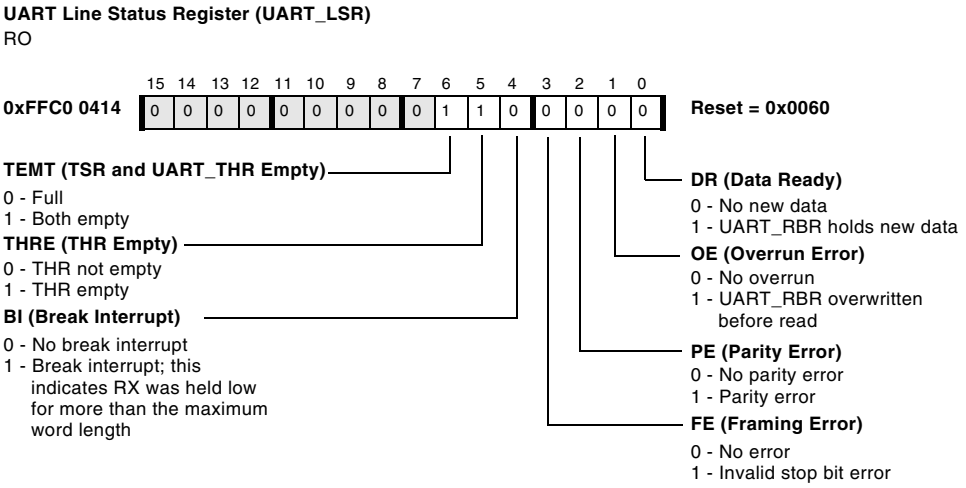


Figure 13-4. UART Line Status Register

The Break Interrupt (BI), Overrun Error (OE), Parity Error (PE) and Framing Error (FE) bits are cleared when the UART Line Status register (UART_LSR) is read. The Data Ready (DR) bit is cleared when the UART Receive Buffer register (UART_RBR) is read.

Because of the destructive nature of these read operations, special care should be taken. For more information, see "Speculative Load Execution" on page 6-69 and "Conditional Load Behavior" on page 6-70.

The THRE bit indicates that the UART transmit channel is ready for new data and software can write to UART_THR. Writes to UART_THR clear the THRE bit. It is set again when data is copied from UART_THR to the Transmit Shift register (TSR). The TEMT bit can be evaluated to determine whether a recently initiated transmit operation has been completed.

UART_THR Register

A write to the UART Transmit Holding register (UART_THR) initiates the transmit operation. The data is moved to the internal Transmit Shift register (TSR) where it is shifted out at a baud rate equal to SCLK/(16 × Divisor) with start, stop, and parity bits appended as required. All data words begin with a 1-to-0-transition start bit. The transfer of data from UART_THR to the Transmit Shift register sets the Transmit Holding Register Empty (THRE) status flag in the UART Line Status register (UART_LSR).

The write-only UART_THR register is mapped to the same address as the read-only UART_RBR and UART_DLL registers. To access UART_THR, the DLAB bit in UART_LCR must be cleared. When the DLAB bit is cleared, writes to this address target the UART_THR register, and reads from this address return the UART_RBR register.

Note data is transmitted and received least significant bit (LSB) first (bit 0) followed by the most significant bits (MSBs).

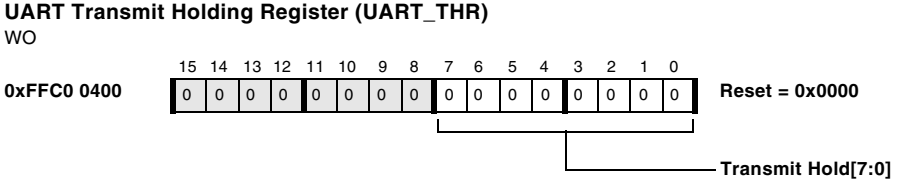


Figure 13-5. UART Transmit Holding Register

UART_RBR Register

The receive operation uses the same data format as the transmit configuration, except that the number of stop bits is always assumed to be 1. After detection of the start bit, the received word is shifted into the Receive Shift register (RSR) at a baud rate of SCLK/(16 x Divisor). After the appropriate number of bits (including stop bit) is received, the data and any status are updated and the Receive Shift register is transferred to the UART Receive Buffer register (UART_RBR), shown in Figure 13-6. After the transfer of the received word to the UART_RBR buffer and the appropriate synchronization delay, the Data Ready (DR) status flag is updated.

A sampling clock equal to 16 times the baud rate samples the data as close to the midpoint of the bit as possible. Because the internal sample clock may not exactly match the asynchronous receive data rate, the sampling point drifts from the center of each bit. The sampling point is synchronized again with each start bit, so the error accumulates only over the length of a single word. A receive filter removes spurious pulses of less than two times the sampling clock period.

The read-only UART_RBR register is mapped to the same address as the write-only UART_THR and UART_DLL registers. To access UART_RBR, the DLAB bit in UART_LCR must be cleared. When the DLAB bit is cleared, writes to this address target the UART_THR register, while reads from this address return the UART_RBR register.

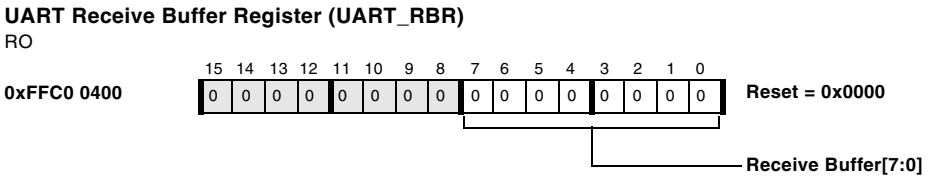


Figure 13-6. UART Receive Buffer Register

UART_IER Register

The UART Interrupt Enable register (UART_IER) is used to enable requests for system handling of empty or full states of UART data registers. Unless polling is used as a means of action, the ERBFI and/or ETBEI bits in this register are normally set.

Setting this register without enabling system DMA causes the UART to notify the processor of data inventory state by means of interrupts. For proper operation in this mode, system interrupts must be enabled, and appropriate interrupt handling routines must be present. For backward compatibility, the UART_IIR still reflects the correct interrupt status.

The UART features three separate interrupt channels to handle data transmit, data receive, and line status events independently, regardless whether DMA is enabled or not.

With system DMA enabled, the UART uses DMA to transfer data to or from the processor. Dedicated DMA channels are available to receive and transmit operation. Line error handling can be configured completely independently from the receive/transmit setup.

The UART_IER register is mapped to the same address as UART_DLH. To access UART_IER, the DLAB bit in UART_LCR must be cleared.

UART's DMA is enabled by first setting up the system DMA control registers and then enabling the UART ERBFI and/or ETBEI interrupts in the UART_IER register. This is because the interrupt request lines double as DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. However, UART's error interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.

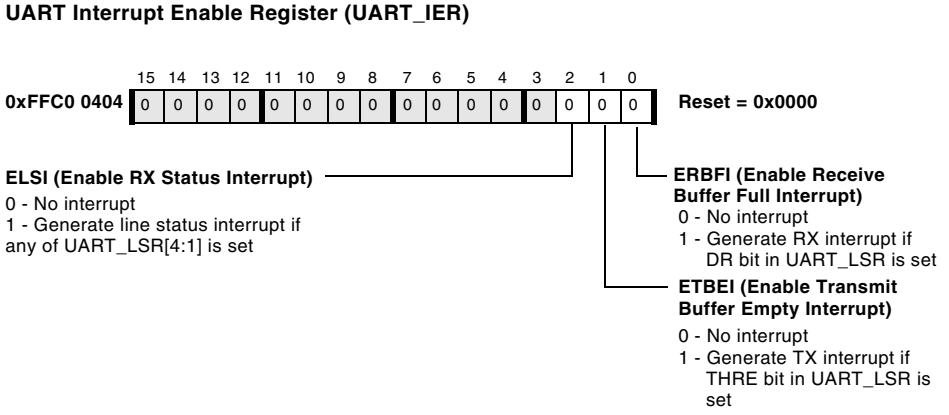


Figure 13-7. UART Interrupt Enable Register

The ELSI bit enables interrupt generation on an independent interrupt channel when any of the following conditions are raised by the respective bit in the UART Line Status register (UART_LSR):

- Receive Overrun Error (OE)
- Receive Parity Error (PE)
- Receive Framing Error (FE)
- Break Interrupt (BI)

When the ETBEI bit is set in the UART_IER register, the UART module immediately issues an interrupt or DMA request. When initiating the transmission of a string, no special handling of the first character is required. Set the ETBEI bit and let the interrupt service routine load the first character from memory and write it to the UART_THR register in the normal manner. Accordingly, the ETBEI bit should be cleared if the string transmission has completed.

UART_IIR Register

For legacy reasons, the UART Interrupt Identification register (UART_IIR) still reflects the UART interrupt status. Legacy operation may require bundling all UART interrupt sources to a single interrupt channel and servicing them all by the same software routine. This can be established by globally assigning all UART interrupts to the same interrupt priority, by using the System Interrupt Controller (SIC).

When cleared, the Pending Interrupt bit (NINT) signals that an interrupt is pending. The STATUS field indicates the highest priority pending interrupt. The receive line status has the highest priority; the UART_THR empty interrupt has the lowest priority. In the case where both interrupts are signalling, the UART_IIR reads 0x06.

When a UART interrupt is pending, the interrupt service routine (ISR) needs to clear the interrupt latch explicitly. The following figure shows how to clear any of the three latches.

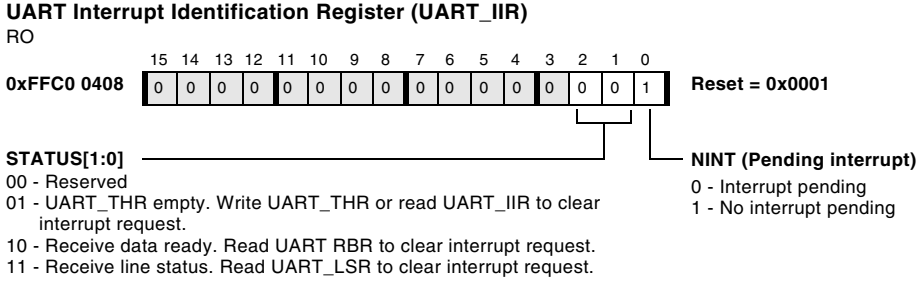


Figure 13-8. UART Interrupt Identification Register

The TX interrupt request is cleared by writing new data to the UART_THR register or by reading the UART_IIR register. Please note the special role of the UART_IIR register read in the case where the service routine does not want to transmit further data.

If software stops transmission, it must read the UART_IIR register to reset the interrupt request. As long as the UART_IIR register reads 0x04 or 0x06 (indicating that another interrupt of higher priority is pending), the UART_THR empty latch cannot be cleared by reading UART_IIR.

If either the Line Status interrupt or the Receive Data interrupt has been assigned a lower interrupt priority by the SIC, a deadlock condition can occur. To avoid this, always assign the lowest priority of the enabled UART interrupts to the UART_THR empty event.

Because of the destructive nature of these read operations, special care should be taken. For more information, see "Speculative Load Execution" on page 6-69 and "Conditional Load Behavior" on page 6-70.

UART_DLL and UART_DLH Registers

The bit rate is characterized by the system clock (SCLK) and the 16-bit Divisor. The Divisor is split into the UART Divisor Latch Low Byte register (UART_DLL) and the UART Divisor Latch High Byte register (UART_DLH). These registers form a 16-bit Divisor. The baud clock is divided by 16 so that:

```
BAUD RATE = SCLK/(16 x Divisor)
```

Divisor = 65,536 when UART_DLL = UART_DLH = 0

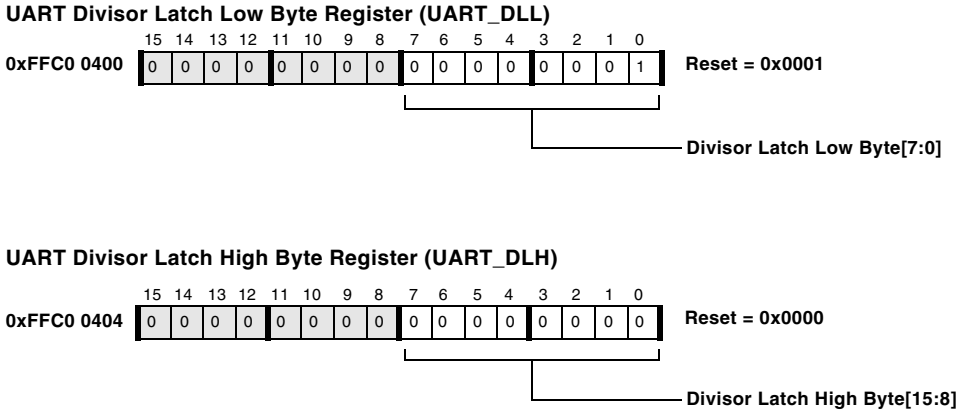


Figure 13-9. UART Divisor Latch Registers

The UART_DLL register is mapped to the same address as the UART_THR and UART_RBR registers. The UART_DLH register is mapped to the same address as the Interrupt Enable register (UART_IER). The DLAB bit in UART_LCR must be set before the UART Divisor Latch registers can be accessed.

Note the 16-bit Divisor formed by UART_DLH and UART_DLL resets to 0x0001, resulting in the highest possible clock frequency by default. If the UART is not used, disabling the UART clock will save power. The UART_DLH and UART_DLL registers can be programmed by software before or after setting the UCEN bit.

Table 13-1 provides example divide factors required to support most standard baud rates.

Table 13-1. UART Baud Rate Examples With 100 MHz SCLK

Baud Rate	DL	Actual	% Error
2400	2604	2400.15	.006
4800	1302	4800.31	.007
9600	651	9600.61	.006
19200	326	19171.78	.147
38400	163	38343.56	.147
57600	109	57339.45	.452
115200	54	115740.74	.469
921600	7	892857.14	3.119
6250000	1	6250000	-

Careful selection of SCLK frequencies, that is, even multiples of desired baud rates, can result in lower error percentages.

UART_SCR Register

The contents of the 8-bit UART Scratch register (UART_SCR) is reset to 0x00. It is used for general-purpose data storage and does not control the UART hardware in any way.

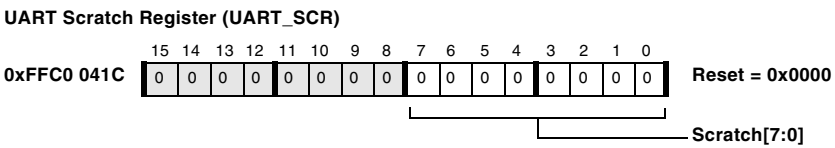


Figure 13-10. UART Scratch Register

UART_GCTL Register

The UART Global Control register (UART_GCTL) contains the enable bit for internal UART clocks and for the IrDA mode of operation of the UART.

Please note that the UCEN bit was not present in previous UART implementations. It has been introduced to save power if the UART is not used. When porting code, be sure to enable this bit.

The IrDA TX Polarity Change bit and the IrDA RX Polarity Change bit are effective only in IrDA mode. The two force error bits, FPE and FFE, are intended for test purposes. They are useful for debugging software, especially in loopback mode.

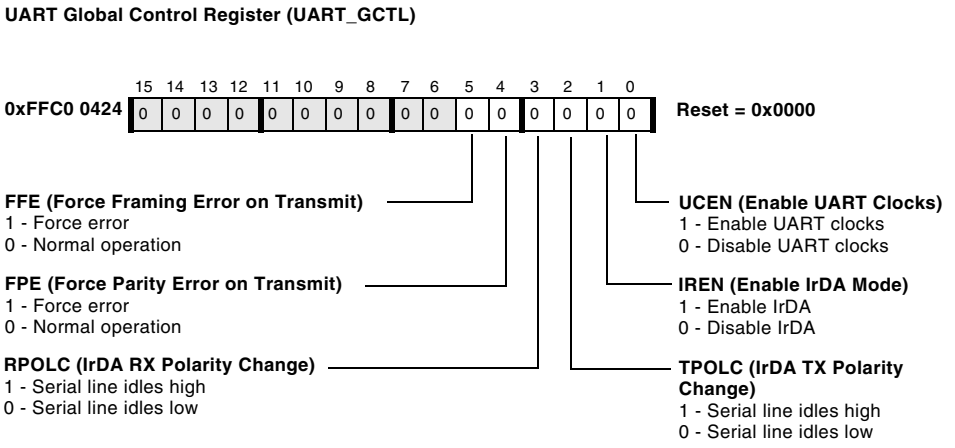


Figure 13-11. UART Global Control Register

Non-DMA Mode

In non-DMA mode, data is moved to and from the UART by the processor core. To transmit a character, load it into UART_THR. Received data can be read from UART_RBR. The processor must write and read one character at time.

To prevent any loss of data and misalignments of the serial datastream, the UART Line Status register (UART_LSR) provides two status flags for handshaking—THRE and DR.

The THRE flag is set when UART_THR is ready for new data and cleared when the processor loads new data into UART_THR. Writing UART_THR when it is not empty overwrites the register with the new value and the previous character is never transmitted.

The DR flag signals when new data is available in UART_RBR. This flag is cleared automatically when the processor reads from UART_RBR. Reading UART_RBR when it is not full returns the previously received value. When UART_RBR is not read in time, newly received data overwrites UART_RBR and the Overrun (OE) flag is set.

With interrupts disabled, these status flags can be polled to determine when data is ready to move. Note that because polling is processor intensive, it is not typically used in real-time signal processing environments. Software can write up to two words into the UART_THR register before enabling the UART clock. As soon as the UCEN bit is set, those two words are sent.

Alternatively, UART writes and reads can be accomplished by interrupt service routines (ISRs). Separate interrupt lines are provided for UART TX, UART RX, and UART Error. The independent interrupts can be enabled individually by the UART_IER register.

The ISRs can evaluate the Status bit field within the UART Interrupt Identification register (UART_IIR) to determine the signalling interrupt source. If more than one source is signalling, the status field displays the one with the highest priority. Interrupts also must be assigned and unmasked by the processor's interrupt controller. The ISRs must clear the interrupt latches explicitly. See Figure 13-8.

DMA Mode

In this mode, separate receive (RX) and transmit (TX) DMA channels move data between the UART and memory. The software does not have to move data, it just has to set up the appropriate transfers either through the descriptor mechanism or through Autobuffer mode.

No additional buffering is provided in the UART DMA channel, so the latency requirements are the same as in non-DMA mode. However, the latency is determined by the bus activity and arbitration mechanism and not by the processor loading and interrupt priorities. For more information, see Chapter 9, "Direct Memory Access."

DMA interrupt routines must explicitly write 1s to the corresponding DMA IRQ status registers to clear the latched request of the pending interrupt.

The UART's DMA is enabled by first setting up the system DMA control registers and then enabling the UART ERBFI and/or ETBEI interrupts in the UART_IER register. This is because the interrupt request lines double as DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. However, the UART's error interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.

The UART's DMA supports 8-bit operation.

Mixing Modes

Non-DMA and DMA modes use different synchronization mechanisms. Consequently, any serial communication must be complete before switching from non-DMA to DMA mode or vice versa. In other words, before switching from non-DMA transmission to DMA transmission, make sure both UART_THR and the internal Transmit Shift register (TSR) are empty by testing the THRE and the TEMT status bits in UART_LSR.

When switching from DMA to non-DMA operation, make sure both the receive (RX) and transmit (TX) DMA channels have completely transferred their data, including data contained in the DMA FIFOs. While the DMA RX interrupt indicates the last data word has been written to memory (and has left the DMA FIFO), the DMA TX interrupt indicates the last data word has left memory (and has entered the DMA FIFO). The processor must wait until the TX FIFO is empty, by testing that the DMA_RUN status bit in the TX channel's IRQ_STATUS register is clear, before it is safe to disable the DMA channel.

IrDA Support

Aside from the standard UART functionality, the UART also supports half-duplex serial data communication via infrared signals, according to the recommendations of the Infrared Data Association (IrDA). The physical layer known as IrDA SIR (9.6/115.2 Kbps rate) is based on return-to-zero-inverted (RZI) modulation. Pulse position modulation is not supported.

Using the 16x data rate clock, RZI modulation is achieved by inverting and modulating the non-return-to-zero (NRZ) code normally transmitted by the UART. On the receive side, the 16x clock is used to determine an IrDA pulse sample window, from which the RZI-modulated NRZ code is recovered.

IrDA support is enabled by setting the IREN bit in the UART Global Control register. The IrDA application requires external transceivers.

IrDA Transmitter Description

To generate the IrDA pulse transmitted by the UART, the normal NRZ output of the transmitter is first inverted, so a 0 is transmitted as a high pulse of 16 UART clock periods and a 1 is transmitted as a low pulse for 16 UART clock periods. The leading edge of the pulse is then delayed by six UART clock periods. Similarly, the trailing edge of the pulse is truncated by eight UART clock periods. This results in the final representation of the original 0 as a high pulse of only 3/16 clock periods in a 16-cycle UART clock period. The pulse is centered around the middle of the bit time, as shown in Figure 13-12. The final IrDA pulse is fed to the off-chip infrared driver.

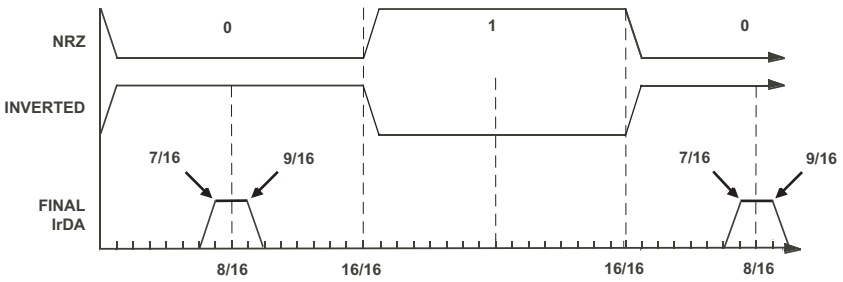


Figure 13-12. IrDA Transmit Pulse

This modulation approach ensures a pulse width output from the UART of three cycles high out of every 16 UART clock cycles. As shown in Table 13-1, the error terms associated with the baud rate generator are very small and well within the tolerance of most infrared transceiver specifications.

IrDA Receiver Description

The IrDA receiver function is more complex than the transmit function. The receiver must discriminate the IrDA pulse and reject noise. To do this, the receiver looks for the IrDA pulse in a narrow window centered around the middle of the expected pulse.

Glitch filtering is accomplished by counting 16 system clocks from the time an initial pulse is seen. If the pulse is absent when the counter expires, it is considered a glitch. Otherwise, it is interpreted as a 0. This is acceptable because glitches originating from on-chip capacitive cross-coupling typically do not last for more than a fraction of the system clock period. Sources outside of the chip and not part of the transmitter can be avoided by appropriate shielding. The only other source of a glitch is the transmitter itself. The processor relies on the transmitter to perform within specification. If the transmitter violates the specification, unpredictable results may occur. The 4-bit counter adds an extra level of protection at a minimal cost. Note because the system clock can change across systems, the longest glitch tolerated is inversely proportional to the system clock frequency.

The receive sampling window is determined by a counter that is clocked at the 16x bit-time sample clock. The sampling window is re-synchronized with each start bit by centering the sampling window around the start bit.

The polarity of receive data is selectable, using the IRPOL bit. Figure 13-13 gives examples of each polarity type.

- IRPOL = 0 assumes that the receive data input idles 0 and each active 1 transition corresponds to a UART NRZ value of 0.
- IRPOL = 1 assumes that the receive data input idles 1 and each active 0 transition corresponds to a UART NRZ value of 0.

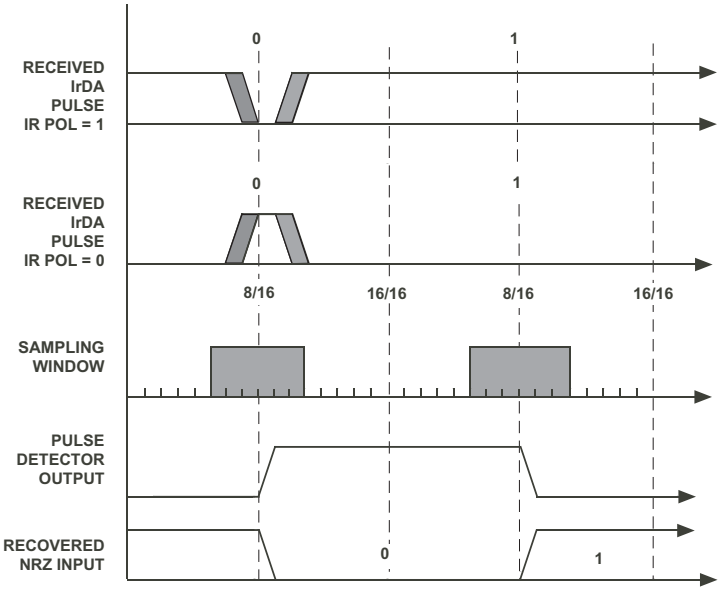


Figure 13-13. IrDA Receiver Pulse Detection

14 PROGRAMMABLE FLAGS

The processor supports 16 bidirectional programmable flags (PFx) or general-purpose I/O pins, PF[15:0]. Each pin can be individually configured as either an input or an output by using the Flag Direction register (FIO_DIR). When configured as output, the Flag Data register (FIO_FLAG_D) can be directly written to specify the state of all PFx pins. When configured as an output, the state written to the Flag Set (FIO_FLAG_S), Flag Clear (FIO_FLAG_C), and Flag Toggle (FIO_FLAG_T) registers determines the state driven by the output PFx pin. Regardless of how the pins are configured, as an input or an output, reading any of these registers (FIO_FLAG_D, FIO_FLAG_S, FIO_FLAG_C, FIO_FLAG_T) returns the state of each pin.

Each PFx pin can be configured to generate an interrupt. When a PFx pin is configured as an input, an interrupt can be generated according to the state of the pin (either high or low), an edge transition (low to high or high to low), or on both edge transitions (low to high *and* high to low). Input sensitivity is defined on a per-bit basis by the Flag Polarity register (FIO_POLAR), the Flag Interrupt Sensitivity register (FIO_EDGE) and the Flag Set on Both Edges register (FIO_BOTH). Input polarity is defined on a per-bit basis by the Flag Polarity register. When the PFx inputs are enabled and a PFx pin is configured as an output, enabling interrupts for the pin allows an interrupt to be generated by setting the PFx pin.

The processor provides two independent interrupt channels for the PFx pins. Identical in functionality, these are called Interrupt A and Interrupt B. Each interrupt channel has four mask registers associated with it, a Flag

Interrupt Mask Data register (FIO_MASKx_D), a Flag Interrupt Mask Set register (FIO_MASKx_S), a Flag Interrupt Mask Clear register (FIO_MASKx_C), and a Flag Interrupt Mask Toggle register (FIO_MASKx_T).

Each PFx pin is represented by a bit in each of these eight registers. Writing a 1 to a bit in a Mask Set register enables interrupt generation for that PFx pin, while writing a 1 to a bit in a Mask Clear register disables interrupt generation for that PFx pin.

The interrupt masking can be toggled by writing a 1 to a bit in the Mask Toggle register. Additionally, the mask bits can be directly written by writing to the Mask Data register. This flexible mechanism allows each bit to generate Flag Interrupt A, Flag Interrupt B, both Flag Interrupts A and B, or neither.

When a PF pin is not used in a system, the input buffer can be disabled so that no external pull-ups or pull-downs are required on the unused pins. By default, the input buffers are disabled. They can be enabled via bits in the Flag Input Enable register (FIO_INEN).

The PFx pins are multiplexed for use by the Parallel Peripheral Interface (PPI), Timers, and Serial Peripheral Interface (SPI). Table 14-1 shows the programmable flag pins and their multiplexed functionality.

Table 14-1. Programmable Flag Pins and Functionality

PF Pin	Peripheral That Shares the PF Pin		
	PPI	SPI	Timers 0, 1, 2
0		Slave Select Input ($\overline{\text{SPISS}}$)	
1		Slave Select Enable 1 (SPISEL1)	Input clock
2		Slave Select Enable 2 (SPISEL2)	

Table 14-1. Programmable Flag Pins and Functionality (Cont'd)

PF Pin	Peripheral That Shares the PF Pin		
	PPI	SPI	Timers 0, 1, 2
3	Frame Sync 3	Slave Select Enable 3 (SPISEL3)	
4	I/O #15	Slave Select Enable 4 (SPISEL4)	
5	I/O #14	Slave Select Enable 5 (SPISEL5)	
6	I/O #13	Slave Select Enable 6 (SPISEL6)	
7	I/O #12	Slave Select Enable 7 (SPISEL7)	
8	I/O #11		
9	I/O #10		
10	I/O #9		
11	I/O #8		
12	I/O #7		
13	I/O #6		
14	I/O #5		
15	I/O #4		

Table 14-2 describes how to use the peripheral function that shares the PF pin.

For more information, see Chapter 11, "Parallel Peripheral Interface," Chapter 10, "SPI Compatible Port Controllers," and Chapter 15, "Timers."

Table 14-2. Using Peripheral Function That Shares the PF Pin

PF Pin	To Use the Peripheral Function That Shares the PF Pin… (Assuming Peripheral is Enabled)		
	PPI	**SPI**	**Timers 0,1,2**
0		Write ‘1’ to PSSE in SPI_CTL	
1		Write ‘1’ to FLS1 in SPI_FLG	Write ‘1’ to CLK_SEL in TIMERx_ CONFIG
2		Write ‘1’ to FLS2 in SPI_FLG	
3	Write ‘01’ to PORT_CFG in PPI_CTL (if PORT_DIR = ‘1’), or write ‘10’ to PORT_CFG (if PORT_DIR = ‘0’)	Write ‘1’ to FLS3 in SPI_FLG	
4	Write ‘111’ to DLEN in PPI_CTL	Write ‘1’ to FLS4 in SPI_FLG	
5	Write ‘110’ to DLEN in PPI_CTL	Write ‘1’ to FLS5 in SPI_FLG	
6	Write ‘101’ to DLEN in PPI_CTL	Write ‘1’ to FLS6 in SPI_FLG	
7	Write ‘100’ to DLEN in PPI_CTL	Write ‘1’ to FLS7 in SPI_FLG	
8	Write ‘011’ to DLEN in PPI_CTL		
9	Write ‘010’ to DLEN in PPI_CTL		
10	Write ‘001’ to DLEN in PPI_CTL		
11	Write ‘001’ to DLEN in PPI_CTL		
12	Always enabled when PPI enabled		

Table 14-2. Using Peripheral Function That Shares the PF Pin (Cont'd)

PF Pin	To Use the Peripheral Function That Shares the PF Pin… (Assuming Peripheral is Enabled)		
	PPI	SPI	Timers 0,1,2
13	Always enabled when PPI enabled		
14	Always enabled when PPI enabled		
15	Always enabled when PPI enabled		

Programmable Flag Registers (MMRs)

The programmable flag registers are part of the system memory-mapped registers (MMRs). The addresses of the programmable flag MMRs appear in Appendix B. Core access to the Flag Configuration registers is through the system bus.

FIO_DIR Register

The Flag Direction register (FIO_DIR) is a read-write register. Each bit position corresponds to a PFx pin. A logic 1 configures a PFx pin as an output, driving the state contained in the FIO_FLAG_D register. A logic 0 configures a PFx pin as an input. The reset value of this register is 0x0000, making all PF pins inputs upon reset.

Note when using the PFx pin as an input, the corresponding bit should also be set in the Flag Input Enable register.

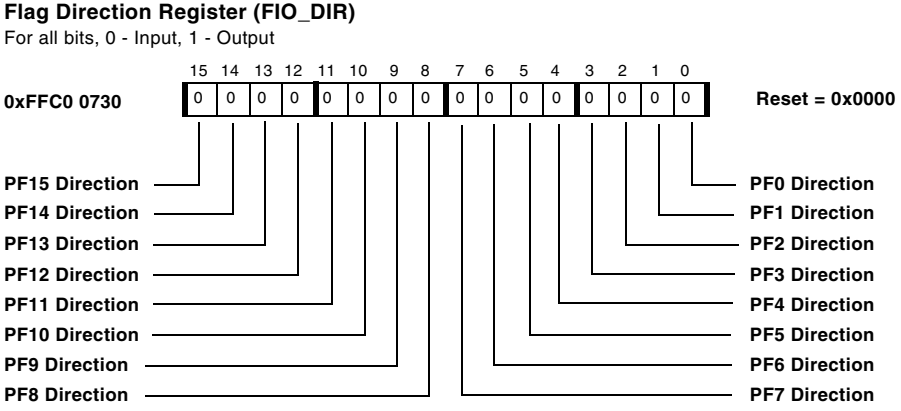


Figure 14-1. Flag Direction Register

Flag Value Registers Overview

The processor has four Flag Value registers:

- Flag Data register (FIO_FLAG_D)
- Flag Set register (FIO_FLAG_S)
- Flag Clear register (FIO_FLAG_C)
- Flag Toggle Direct register (FIO_FLAG_T)

These registers are used to:

- Sense the value of the PFx pins defined as inputs
- Specify the state of PFx pins defined as outputs
- Clear interrupts generated by the PFx pins

Each PFx pin is represented by a bit in each of the four registers.

Reading any of the Flag Data, Flag Set, Flag Clear, or Flag Toggle registers returns the value of the PFx pins. The value returned shows the state of the PFx pins defined as outputs and the sense of PFx pins defined as inputs, based on the polarity and sensitivity settings of each pin.

Reading the Flag Data, Flag Set, Flag Clear, or Flag Toggle register after reset results in 0x0000, because although the pins are inputs, the input buffers are not enabled. See Table 14-3 for guidance on how to interpret a value read from one of these registers, based on the settings of the FIO_POLAR, FIO_EDGE, and FIO_BOTH registers.

For pins configured as edge-sensitive, a readback of 1 from one of these registers is sticky. That is, once it is set it remains set until cleared by user code. For level-sensitive pins, the pin state is checked every cycle, so the readback value will change when the original level on the pin changes.

Table 14-3. Flag Value Register Pin Interpretation

FIO_POLAR	FIO_EDGE	FIO_BOTH	Effect of MMR Settings
0	0	X	Pin that is high reads as 1; pin that is low reads as 0
0	1	0	If rising edge occurred, pin reads as 1; otherwise, pin reads as 0
1	0	X	Pin that is low reads as 1; pin that is high reads as 0
1	1	0	If falling edge occurred, pin reads as 1; otherwise, pin reads as 0
X	1	1	If any edge occurred, pin reads as 1; otherwise, pin reads as 0

For more information about Flag Set, Flag Clear, and Flag Toggle registers, see "FIO_FLAG_S, FIO_FLAG_C, and FIO_FLAG_T Registers".

FIO_FLAG_D Register

When written, the Flag Data register (FIO_FLAG_D), shown in Figure 14-2, directly specifies the state of all PFx pins. When read, the register returns the value of the PFx pins.

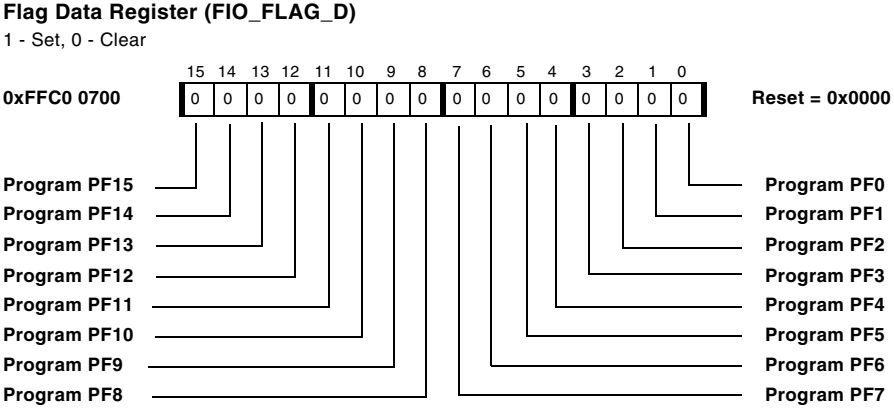


Figure 14-2. Flag Data Register

FIO_FLAG_S, FIO_FLAG_C, and FIO_FLAG_T Registers

The Flag Set register (FIO_FLAG_S), Flag Clear register (FIO_FLAG_C), and Flag Toggle register (FIO_FLAG_T) are used to:

- Set, clear or toggle the output state associated with each output PFx pin
- Clear the latched interrupt state captured from each input PFx pin

This mechanism is used to avoid the potential issues with more traditional read-modify-write mechanisms. Reading any of the these registers shows the flag pin state.

Figure 14-3 and Figure 14-4 represent the Flag Set and Flag Clear registers, respectively. Figure 14-5 represents the Flag Toggle register.

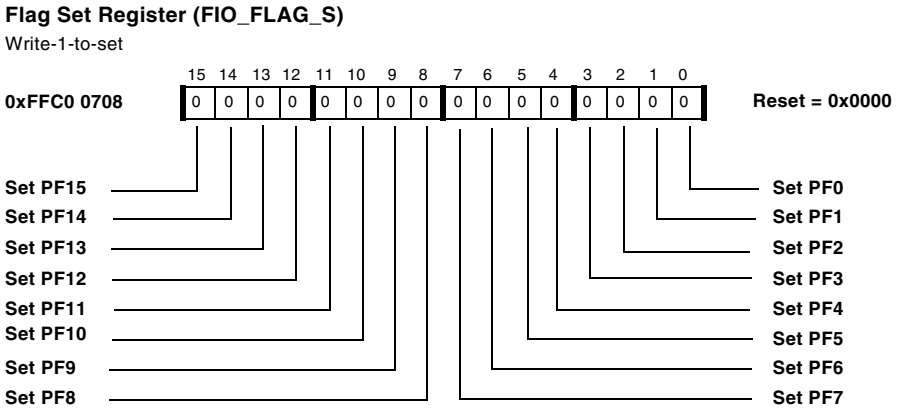


Figure 14-3. Flag Set Register

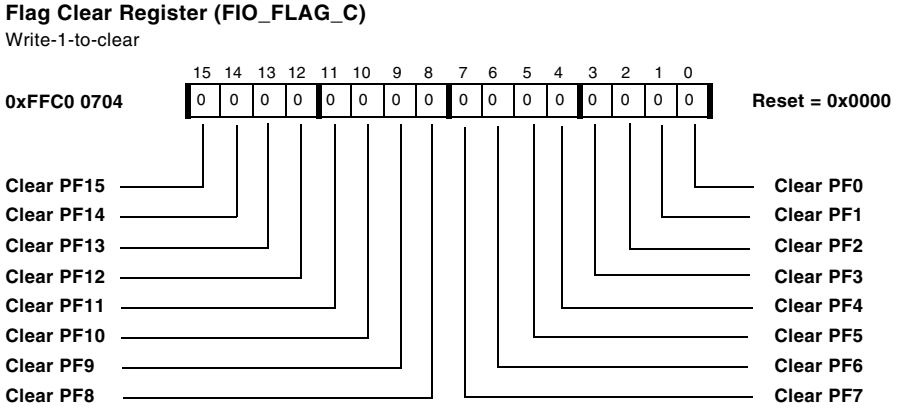


Figure 14-4. Flag Clear Register

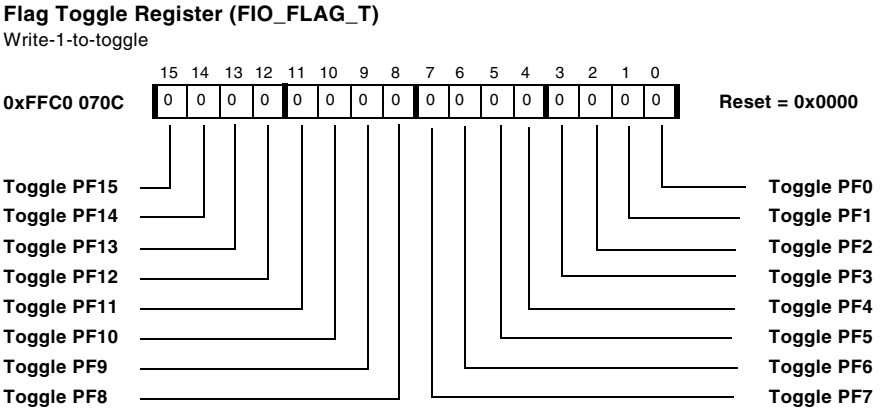


Figure 14-5. Flag Toggle Register

As an example of how these registers work, assume that PF[0] is configured as an output. Writing 0x0001 to the Flag Set register drives a logic 1 on the PF[0] pin without affecting the state of any other PFx pins. Writing 0x0001 to the Flag Clear register drives a logic 0 on the PF[0] pin without affecting the state of any other PFx pins. Writing a 0x0001 to the Flag Toggle register changes the pin state on PF[0] from logic 0 to logic 1 or from logic 1 to logic 0, depending upon the existing pin state.

Writing a 0 to the Flag Set, Flag Clear, or Flag Toggle register has no effect on the value of the flag pin and is, therefore, ignored.

Reading the Flag Set or Flag Clear register shows:

- 0s for PFx pins defined as outputs and driven low
- 1s for pins (including PF[0] in the example above) defined as outputs and driven high
- The present sense of PFx pins defined as inputs

Input sense is based on FIO_POLAR and FIO_EDGE settings, as well as the logic level at each pin.

FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T, FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, FIO_MASKB_T Registers

The Flag Mask Interrupt registers (FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T, FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, and FIO_MASKB_T) are implemented as complementary pairs of write-1-to-set, write-1-to-clear, and write-1-to-toggle registers. This implementation provides the ability to enable or disable a PFx pin to act as a processor interrupt without requiring read-modify-write accesses—or to directly specify the mask value with the data register.

Both Flag Interrupt A and Flag Interrupt B are supported by a set of four dedicated registers:

- Flag Mask Interrupt Data register
- Flag Mask Interrupt Set register
- Flag Mask Interrupt Clear register
- Flag Interrupt Toggle register

For diagrams of the registers that support Flag Interrupt A, see "FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T Registers".

For diagrams of the registers that support Flag Interrupt B, see "FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, FIO_MASKB_T Registers".

Each PFx pin is represented by a bit in each of the eight registers. Table 14-4 shows the effect of writing 1 to a bit in a Mask Set, Mask Clear, or Mask Toggle register.

Reading any of the [A&B] mask data, set, clear, or toggle registers returns the value of the current mask [A&B] data.

Table 14-4. Effect of Writing 1 to a Bit

Register	Effect of Writing 1 to a Bit in the Register
Mask Set	Enables interrupt generation for that PFx pin
Mask Clear	Disables interrupt generation for that PFx pin
Mask Toggle	Changes the state of interrupt generation capability

Interrupt A and Interrupt B operate independently. For example, writing 1 to a bit in the Flag Mask Interrupt A Set register does not affect Flag Interrupt B. This facility allows PFx pins to generate Flag Interrupt A, Flag Interrupt B, both Flag Interrupts A and B, or neither.

Note a Flag Interrupt is generated by a logical OR of all unmasked PF pins for that interrupt. For example, if PF[0] and PF[1] are both unmasked for Flag Interrupt A, Flag Interrupt A will be generated when triggered by PF[0] or PF[1].

When using either rising or falling edge-triggered interrupts, the interrupt condition must be cleared each time a corresponding interrupt is serviced by writing 1 to the appropriate FIO_FLAG_C bit.

At reset, all interrupts are masked.

Flag Interrupt Generation Flow

Figure 14-6 shows the process by which a Flag Interrupt A or a Flag Interrupt B event can be generated. Note the flow is shown for only one programmable flag, “FlagN.” However, a Flag Interrupt is generated by a logical OR of all unmasked PFx pins for that interrupt. For example, if only PF[0] and PF[1] are unmasked for Flag Interrupt A, this interrupt is generated when triggered by either PF[0] or PF[1].

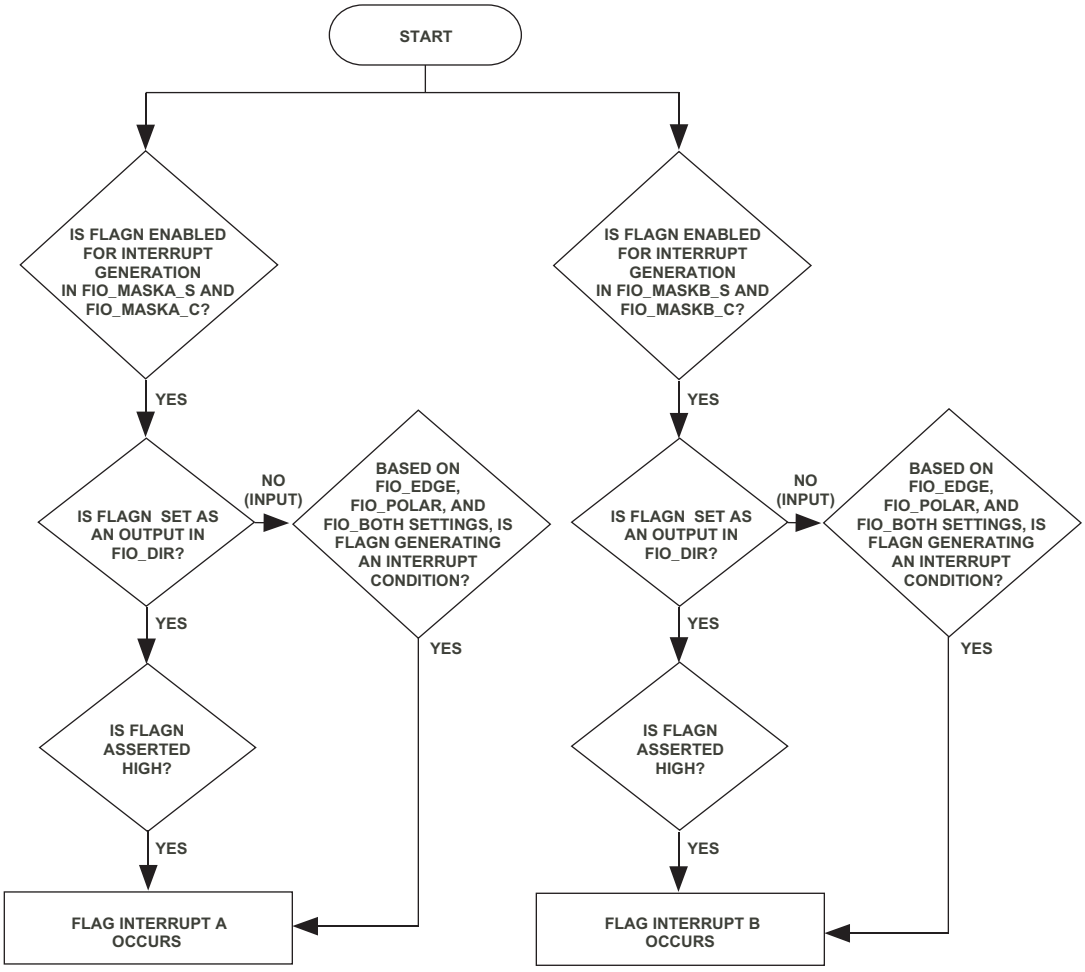


Figure 14-6. Flag Interrupt Generation Flow

FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T Registers

The following four registers support Flag Interrupt A. For details, see page 14-11.

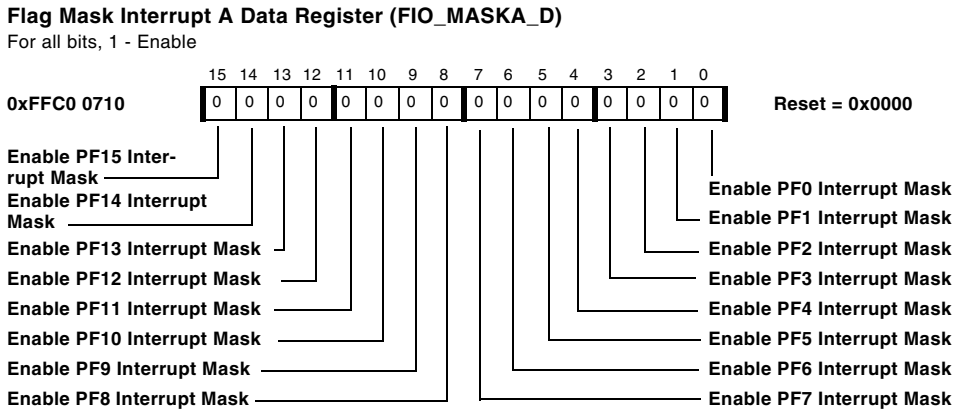


Figure 14-7. Flag Mask Interrupt A Data Register

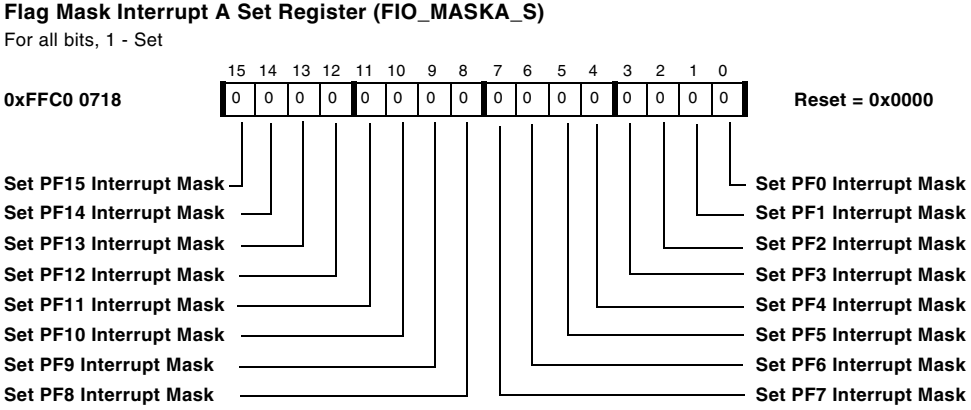


Figure 14-8. Flag Mask Interrupt A Set Register

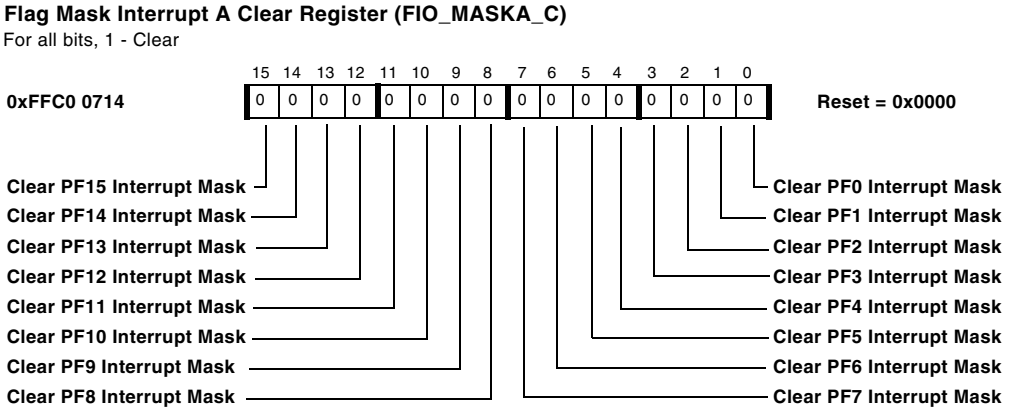


Figure 14-9. Flag Mask Interrupt A Clear Register

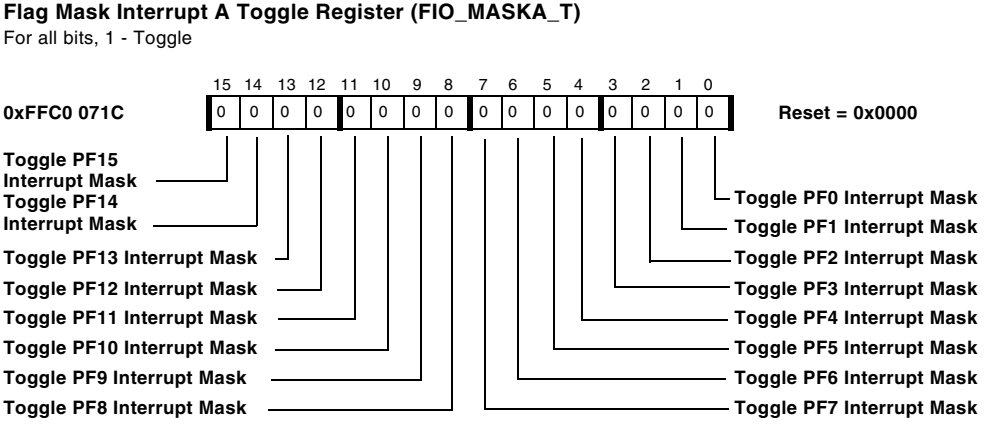


Figure 14-10. Flag Mask Interrupt A Toggle Register

FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, FIO_MASKB_T Registers

The following four registers support Flag Interrupt B. For details, see page 14-11.

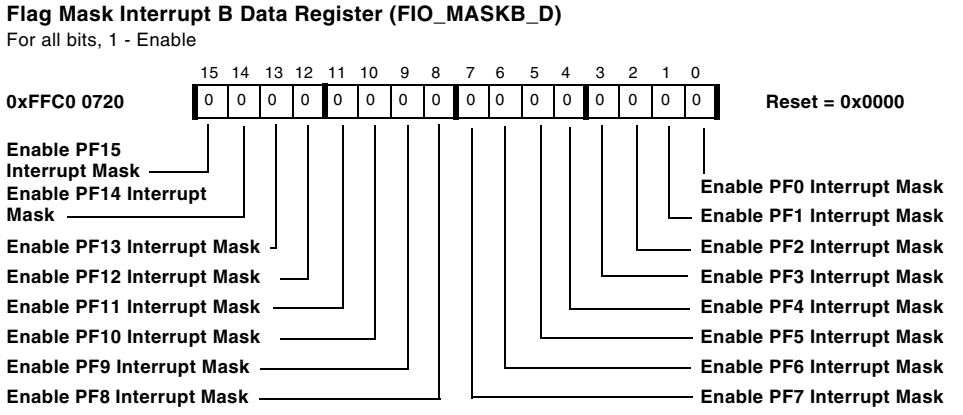


Figure 14-11. Flag Mask Interrupt B Data Register

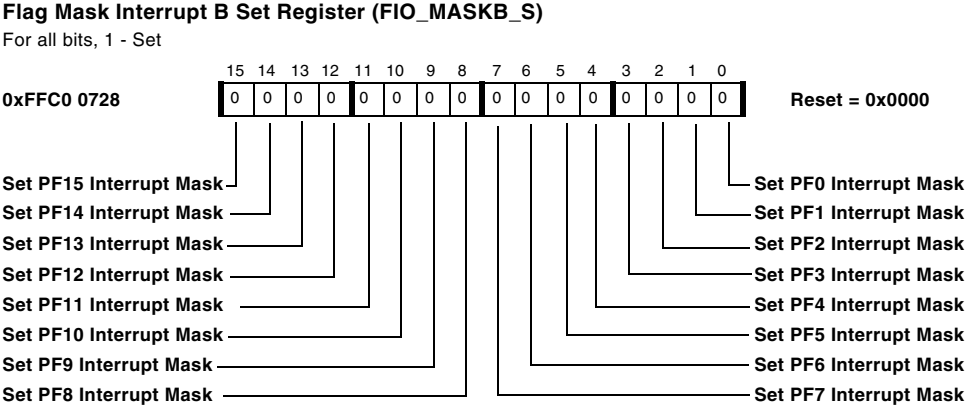


Figure 14-12. Flag Mask Interrupt B Set Register

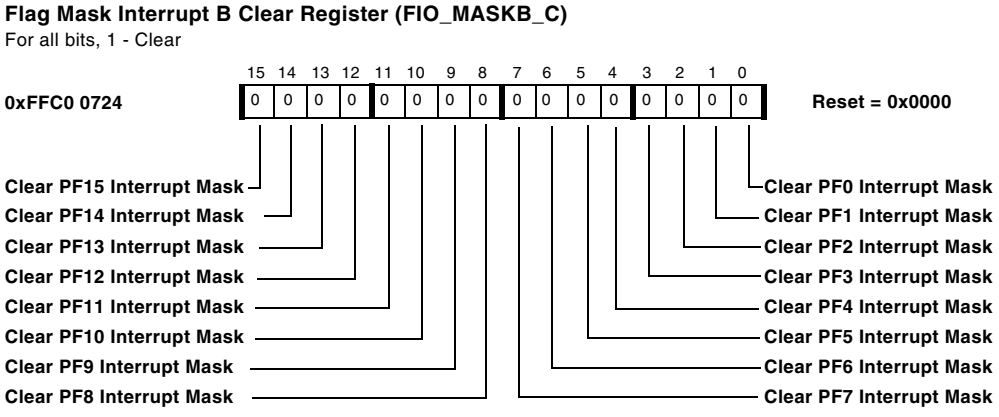


Figure 14-13. Flag Mask Interrupt B Clear Register

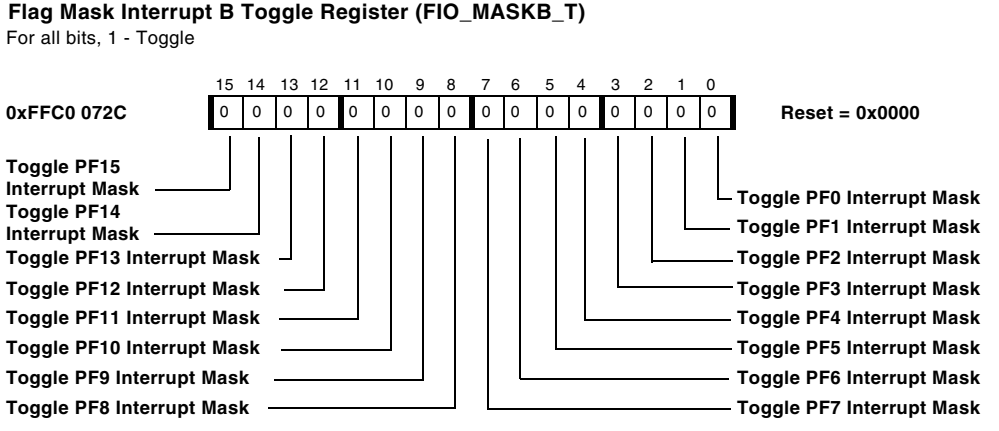


Figure 14-14. Flag Mask Interrupt B Toggle Register

FIO_POLAR Register

The Flag Polarity register (FIO_POLAR) is used to configure the polarity of the flag input source. To select active high or rising edge, set the bits in this register to 0. To select active low or falling edge, set the bits in this register to 1.

This register has no effect on PFx pins that are defined as outputs. The contents of this register are cleared at reset, defaulting to active high polarity.

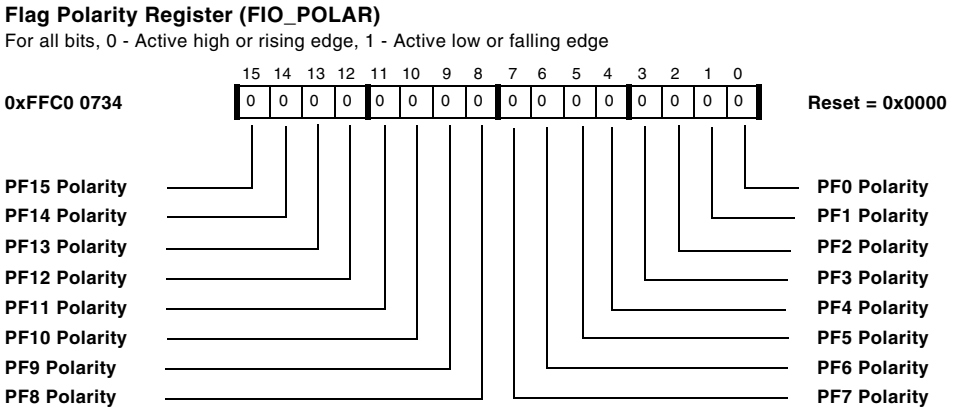


Figure 14-15. Flag Polarity Register

FIO_EDGE Register

The Flag Interrupt Sensitivity register (FIO_EDGE) is used to configure each of the flags as either a level-sensitive or an edge-sensitive source. When using an edge-sensitive mode, an edge detection circuit is used to prevent a situation where a short event is missed because of the system clock rate. This register has no effect on PFx pins that are defined as outputs.

The contents of this register are cleared at reset, defaulting to level sensitivity.

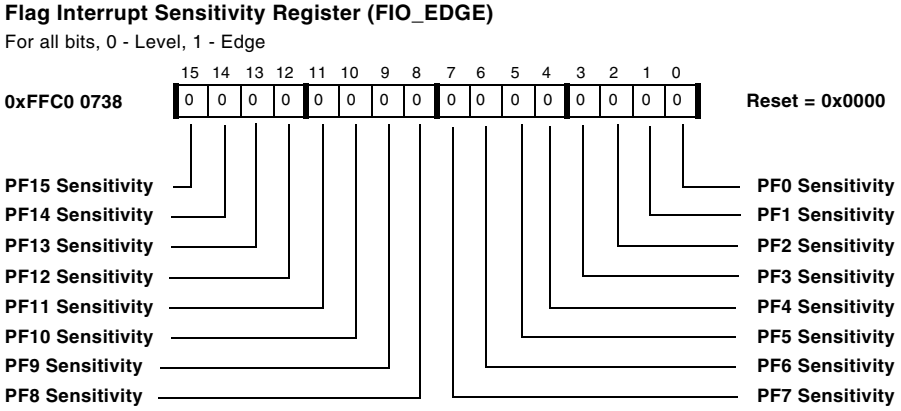


Figure 14-16. Flag Interrupt Sensitivity Register

FIO_BOTH Register

The Flag Set on Both Edges register (FIO_BOTH) is used to enable interrupt generation on both rising and falling edges.

When a given PFx pin has been set to edge-sensitive in the Flag Interrupt Sensitivity register, setting the PFx pin's bit in the Flag Set on Both Edges register to Both Edges results in an interrupt being generated on both the rising and falling edges. This register has no effect on PFx pins that are defined as level-sensitive or as outputs.

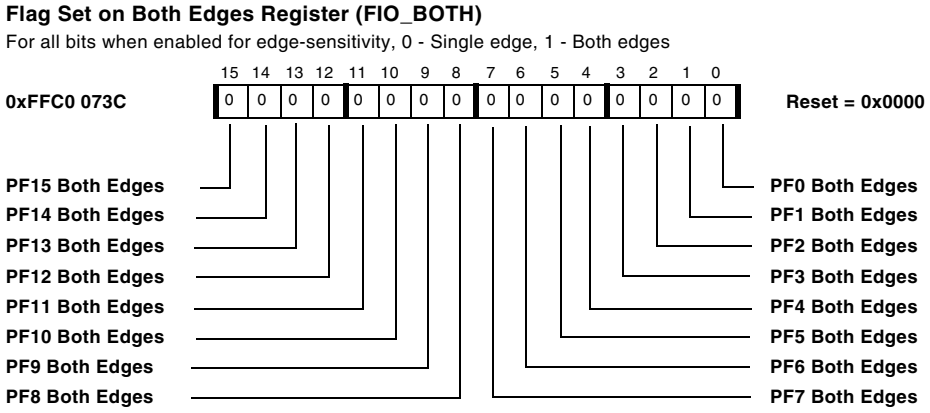


Figure 14-17. Flag Set on Both Edges Register

FIO_INEN Register

The Flag Input Enable register (FIO_INEN) is used to enable the input buffers on any flag pin that is being used as an input. Leaving the input buffer disabled eliminates the need for pull-ups and pull-downs when a particular PFx pin is not used in the system. By default, the input buffers are disabled.

Note that if the PFx pin is being used as an input, the corresponding bit in the Flag Input Enable register must be set. Otherwise, changes at the flag pins will not be recognized by the processor.

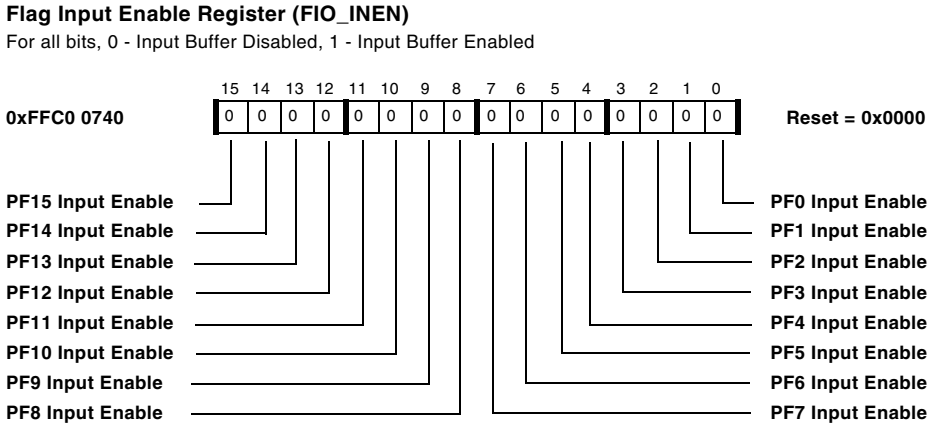


Figure 14-18. Flag Input Enable Register

Performance/Throughput

The PFx pins are synchronized to the system clock (SCLK). When configured as outputs, the programmable flags can transition once every system clock cycle.

When configured as inputs, the overall system design should take into account the potential latency between the core and system clocks. Changes in the state of PFx pins have a latency of 3 SCLK cycles before being detectable by the processor. When configured for level-sensitive interrupt generation, there is a minimum latency of 4 SCLK cycles between the time the flag is asserted and the time that program flow is interrupted. When configured for edge-sensitive interrupt generation, an additional SCLK cycle of latency is introduced, giving a total latency of 5 SCLK cycles between the time the edge is asserted and the time that the core program flow is interrupted.

15 TIMERS

The processor features three identical 32-bit general-purpose timers, a core timer, and a watchdog timer.

The general-purpose timers can be individually configured in any of three modes:

- Pulse Width Modulation (PWM_OUT) mode
- Pulse Width Count and Capture (WDTH_CAP) mode
- External Event (EXT_CLK) mode

The core timer is available to generate periodic interrupts for a variety of system timing functions.

The watchdog timer can be used to implement a software watchdog function. A software watchdog can improve system availability by generating an event to the Blackfin processor core if the timer expires before being updated by software.

General-Purpose Timers

Each general-purpose timer has one dedicated bidirectional chip pin, TMRx. This pin functions as an output pin in the PWM_OUT mode and as an input pin in the WDTH_CAP and EXT_CLK modes. To provide these functions, each timer has four registers. For range and precision, the Timer Counter (TIMERx_COUNTER), Timer Period (TIMERx_PERIOD), and Timer Pulse Width (TIMERx_WIDTH) registers are 32 bits wide. See Figure 15-1.

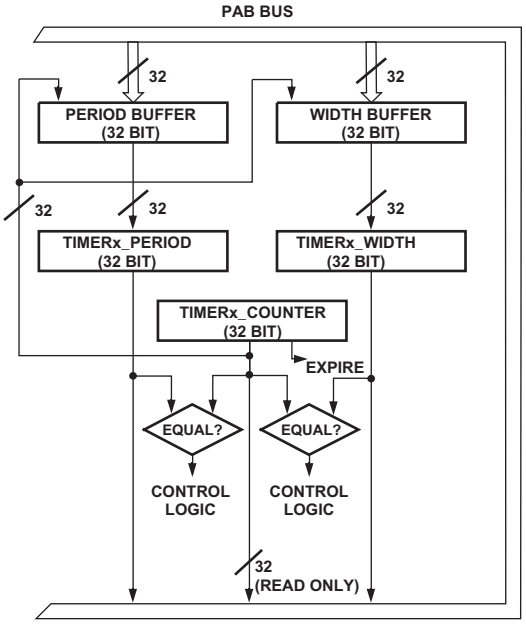


Figure 15-1. Timer Block Diagram

The registers for each general-purpose timer are:

- Timer Configuration (TIMERx_CONFIG) registers
- Timer Counter (TIMERx_COUNTER) registers
- Timer Period (TIMERx_PERIOD) registers
- Timer Pulse Width (TIMERx_WIDTH) registers

When clocked internally, the clock source is the processor's peripheral clock (SCLK). Assuming the peripheral clock is running at 133 MHz, the maximum period for the timer count is ($(2^{32}$-1) / 133 MHz) = 32.2 seconds.

The Timer Enable (TIMER_ENABLE) register can be used to enable all three timers simultaneously. The register contains three “write-1-to-set” control bits, one for each timer. Correspondingly, the Timer Disable (TIMER_DISABLE) register contains three “write-1-to-clear” control bits to allow simultaneous or independent disabling of the three timers. Either the Timer Enable or the Timer Disable register can be read back to check the enable status of the timers. A 1 indicates that the corresponding timer is enabled. The timer starts counting three SCLK cycles after the TIMENx bit is set.

The Timer Status (TIMER_STATUS) register contains an Interrupt Latch bit (TIMILx) and an Overflow/Error Indicator bit (TOVF_ERRx) for each timer. These sticky bits are set by the timer hardware and may be polled by software. They need to be cleared by software explicitly, by writing a 1 to the bit.

To enable a timer’s interrupts, set the IRQ_ENA bit in the timer’s Configuration (TIMERx_CONFIG) register and unmask the timer’s interrupt by setting the corresponding bits of the IMASK and SIC_IMASK registers. With the IRQ_ENA bit cleared, the timer does not set its Timer Interrupt latch (TIMILx) bits. To poll the TIMILx bits without permitting a timer interrupt, programs can set the IRQ_ENA bit while leaving the timer’s interrupt masked.

With interrupts enabled, make sure that the interrupt service routine (ISR) clears the TIMILx latch before the RTI instruction, to ensure that the interrupt is not reissued. To make sure that no timer event is missed, the latch should be reset at the very beginning of the interrupt routine when in External Clock (EXT_CLK) mode. To enable timer interrupts, set the IRQ_ENA bit in the proper Timer Configuration (TIMERx_CONFIG) register.

Timer Registers

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

Each timer provides four registers:

- TIMERx_CONFIG[15:0] – Timer Configuration register
- TIMERx_WIDTH[31:0] – Timer Pulse Width register
- TIMERx_PERIOD[31:0] – Timer Period register
- TIMERx_COUNTER[31:0] – Timer Counter register

Three registers are shared between the three timers:

- TIMER_ENABLE[15:0] – Timer Enable register
- TIMER_DISABLE[15:0] – Timer Disable register
- TIMER_STATUS[15:0] – Timer Status register

The size of accesses is enforced. A 32-bit access to a Timer Configuration register or a 16-bit access to a Timer Pulse Width, Timer Period, or Timer Counter register results in a Memory-Mapped Register (MMR) error. Both 16- and 32-bit accesses are allowed for the Timer Enable, Timer Disable, and Timer Status registers. On a 32-bit read, the upper word returns all 0s.

TIMER_ENABLE Register

The Timer Enable register (TIMER_ENABLE) allows all three timers to be enabled simultaneously in order to make them run completely synchronously. For each timer there is a single W1S control bit. Writing a 1 enables the corresponding timer; writing a 0 has no effect. The three bits can be set individually or in any combination. A read of the Timer Enable register shows the status of the enable for the corresponding timer. A 1 indicates that the timer is enabled. All unused bits return 0 when read.

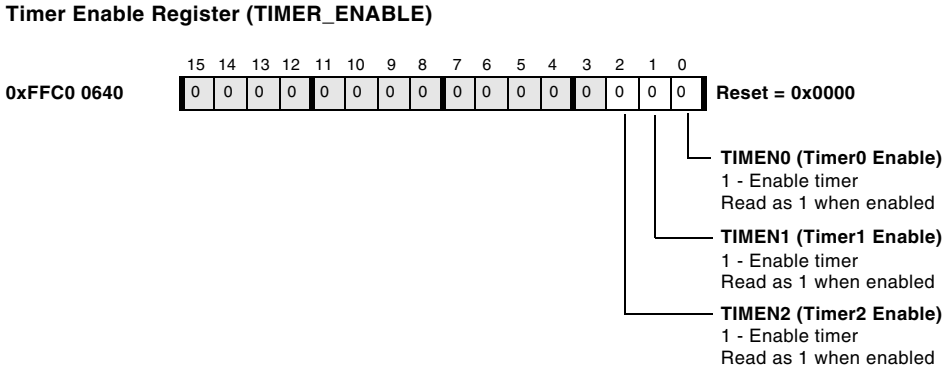


Figure 15-2. Timer Enable Register

TIMER_DISABLE Register

The Timer Disable register (TIMER_DISABLE) allows all three timers to be disabled simultaneously. For each timer there is a single W1C control bit. Writing a 1 disables the corresponding timer; writing a 0 has no effect. The three bits can be cleared individually or in any combination. A read of the Timer Disable register returns a value identical to a read of the Timer Enable register. A 1 indicates that the timer is enabled. All unused bits return 0 when read.

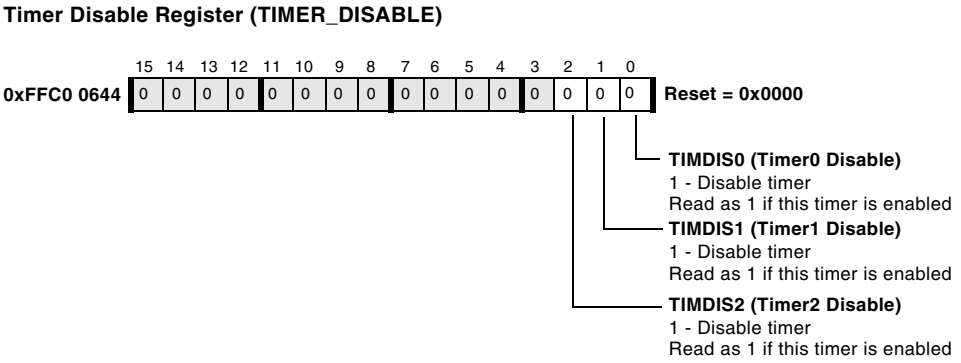


Figure 15-3. Timer Disable Register

In PWM_OUT mode, a write of a 1 to TIMER_DISABLE does not stop the corresponding timer immediately. Rather, the timer continues running and stops cleanly at the end of the current period (if PERIOD_CNT = 1) or pulse (if PERIOD_CNT = 0). If necessary, the processor can force a timer in PWM_OUT mode to stop immediately by first writing a 1 to the corresponding bit in TIMER_DISABLE, and then writing a 1 to the corresponding TRUNx bit in TIMER_STATUS. See "Stopping the Timer in PWM_OUT Mode" on page 15-19.

In WDTH_CAP and EXT_CLK modes, a write of a 1 to TIMER_DISABLE stops the corresponding timer immediately.

TIMER_STATUS Register

The Timer Status register (TIMER_STATUS) indicates the status of all three timers and is used to check the status of all three timers with a single read. Status bits are sticky and W1C. The TRUNx bits can clear themselves, which they do when a PWM_OUT mode timer stops at the end of a period. During a Status Register read access, all reserved or unused bits return a 0.

Each Timer generates a unique interrupt request signal, which is gated by the corresponding IRQ_ENA bit in the TIMERx_CONFIG register. The shared Timer Status register (TIMER_STATUS) latches these interrupts so the user can determine the interrupt source without reference to the unique interrupt signal (for example, in the case where all three timers have been assigned to the same interrupt priority). Interrupt bits are sticky and must be cleared by the interrupt service routine (ISR) to assure that the interrupt is not reissued.

The TIMILx bits work along with the IRQ_ENA bit of the Timer Configuration register to indicate interrupt requests. If an interrupt condition or error occurs and IRQ_ENA is set, then the TIMILx bit is set and the interrupt to the core is asserted. This interrupt may be masked by the system interrupt controller. If an interrupt condition or error occurs and IRQ_ENA is

cleared, then the TIMILx bit is not set and the interrupt is not asserted. If TIMILx is already set and IRQ_ENA is written to 0, TIMILx stays set and the interrupt stays asserted. See Figure 15-24.

The read value of the TRUNx bits reflects the timer slave enable status in all modes—TRUNx set indicates running and TRUNx cleared indicates stopped. While reading the TIMENx or TIMDISx bits in the TIMER_ENABLE and TIMER_DISABLE registers will reflect whether a timer is enabled, the TRUNx bits indicate whether the timer is actually running. In WDTH_CAP and EXT_CLK modes, reads from TIMENx and TRUNx always return the same value.

A W1C operation to the TIMER_DISABLE register disables the corresponding timer in all modes. In PWM_OUT mode, a disabled timer continues running until the ongoing period (PERIOD_CNT = 1) or pulse (PERIOD_CNT = 0) completes. During this final period the TIMENx bit returns 0, but the TRUNx bit still reads as a 1. See Figure 15-10. In this state only, TRUNx becomes a W1C bit. During this final period with the timer disabled, writing a 1 to TRUNx clears TRUNx and stops the timer immediately without waiting for the timer counter to reach the end of its current cycle.

Writing the TRUNx bits has no effect in other modes or when a timer has not been enabled. Writing the TRUNx bits to 1 in PWM_OUT mode has no effect on a timer that has not first been disabled.

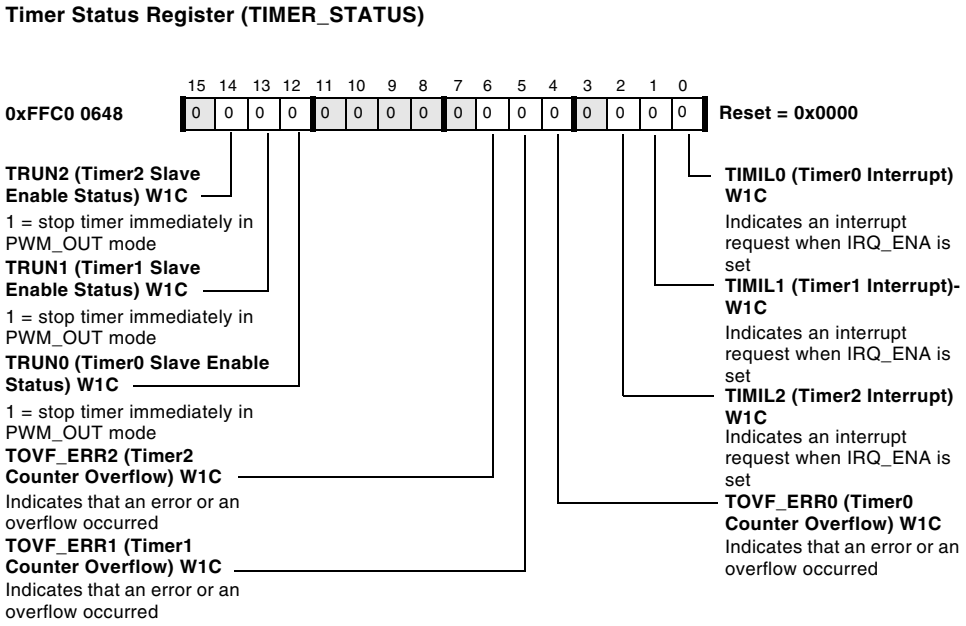


Figure 15-4. Timer Status Register

TIMERx_CONFIG Registers

The operating mode for each timer is specified by its Timer Configuration register (TIMERx_CONFIG). The TIMERx_CONFIG register may be written only when the timer is not running. After disabling the timer in PWM_OUT mode, make sure the timer has stopped running by checking its TRUNx bit in TIMER_STATUS before attempting to reprogram TIMERx_CONFIG. The TIMERx_CONFIG registers may be read at any time. The ERR_TYP field is read-only. It is cleared at reset and when the timer is enabled. Each time TOVF_ERRx is set, ERR_TYP[1:0] is loaded with a code that identifies the type of error that was detected. This value is held until the next error or timer enable occurs. For an overview of error conditions, see Table 15-1. The TIMERx_CONFIG register also controls the behavior of the TMRx pin, which becomes an output in PWM_OUT mode (TMODE = 01) when the OUT_DIS bit is cleared.

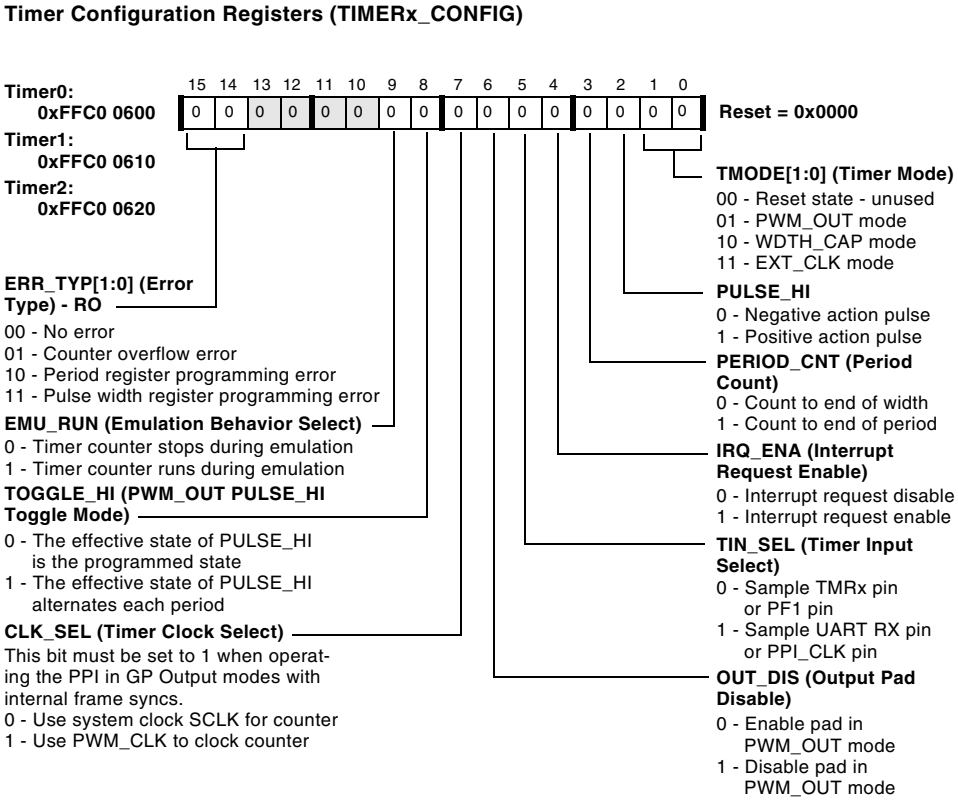


Figure 15-5. Timer Configuration Registers

TIMERx_COUNTER Registers

These read-only registers retain their state when disabled. When enabled, the Timer Counter register (TIMERx_COUNTER) is reinitialized by hardware based on configuration and mode. The Timer Counter register may be read at any time (whether the timer is running or stopped), and it returns a coherent 32-bit value. Depending on the operation mode, the incrementing counter can be clocked by four different sources: SCLK, the TMRx pin, the Programmable Flag pin PF1, or the parallel port clock PPI_CLK.

While the processor core is being accessed by an external emulator debugger, all code execution stops. By default, the TIMERx_COUNTER also halts its counting during an emulation access in order to remain synchronized with the software. While stopped, the count does not advance—in PWM_OUT mode, the TMRx pin waveform is "stretched"; in WDTH_CAP mode, measured values are incorrect; in EXT_CLK mode, input events on TMRx may be missed. All other timer functions such as register reads and writes, interrupts previously asserted (unless cleared), and the loading of TIMERx_PERIOD and TIMERx_WIDTH in WDTH_CAP mode remain active during an emulation stop.

Some applications may require the timer to continue counting asynchronously to the emulation-halted processor core. Set the EMU_RUN bit in TIMERx_CONFIG to enable this behavior.

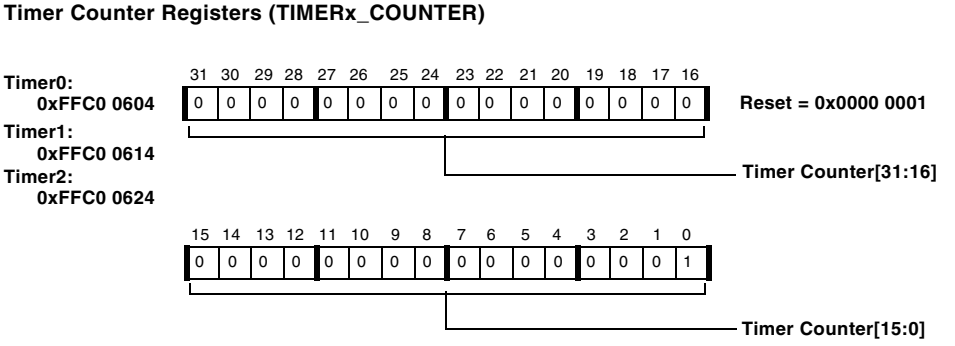


Figure 15-6. Timer Counter Registers

TIMERx_PERIOD and TIMERx_WIDTH Registers

When a timer is enabled and running, and the software writes new values to the Timer Period register and the Timer Pulse Width register, 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).

Usage of the Timer Period register (TIMERx_PERIOD) and the Timer Pulse Width register (TIMERx_WIDTH) varies depending on the mode of the timer:

- In Pulse Width Modulation mode (PWM_OUT), both the Timer Period and Timer Pulse Width register values can be updated "on-the-fly" since the Timer Period and Timer Pulse Width (duty cycle) register values change simultaneously.
- In Pulse Width and Period Capture mode (WDTH_CAP), the Timer Period and Timer Pulse Width buffer values are captured at the appropriate time. The Timer Period and Timer Pulse Width registers are then updated simultaneously from their respective buffers. Both registers are read-only in this mode.
- In External Event Capture mode (EXT_CLK), the Timer Period register is writable and can be updated "on-the-fly." The Timer Pulse Width register is not used.

If new values are not written to the Timer Period register or the Timer Pulse Width register, the value from the previous period is reused. Writes to the 32-bit Timer Period register and Timer Pulse Width register are atomic; it is not possible for the high word to be written without the low word also being written.

Values written to the Timer Period registers or Timer Pulse Width registers are always stored in the buffer registers. Reads from the Timer Period or Timer Pulse Width registers always return the current, active value of period or pulse width. Written values are not read back until they become active. When the timer is enabled, they do not become active until after the Timer Period and Timer Pulse Width registers are updated from their respective buffers at the end of the current period. See Figure 15-1.

When the timer is disabled, writes to the buffer registers are immediately copied through to the Timer Period or Timer Pulse Width register so that they will be ready for use in the first timer period. For example, to change

the values for the Timer Period and/or Timer Pulse Width registers in order to use a different setting for each of the first three timer periods after the timer is enabled, the procedure to follow is:

1. Program the first set of register values.
2. Enable the timer.
3. Immediately program the second set of register values.
4. Wait for the first timer interrupt.
5. Program the third set of register values.

Each new setting is then programmed when a timer interrupt is received.

In PWM_OUT mode with very small periods (less than 10 counts), there may not be enough time between updates from the buffer registers to write both the Timer Period register and the Timer Pulse Width register. The next period may use one old value and one new value. In order to prevent Pulse Width >= Period errors, write the Timer Pulse Width register before the Timer Period register when decreasing the values, and write the Timer Period register before the Timer Pulse Width register when increasing the value.

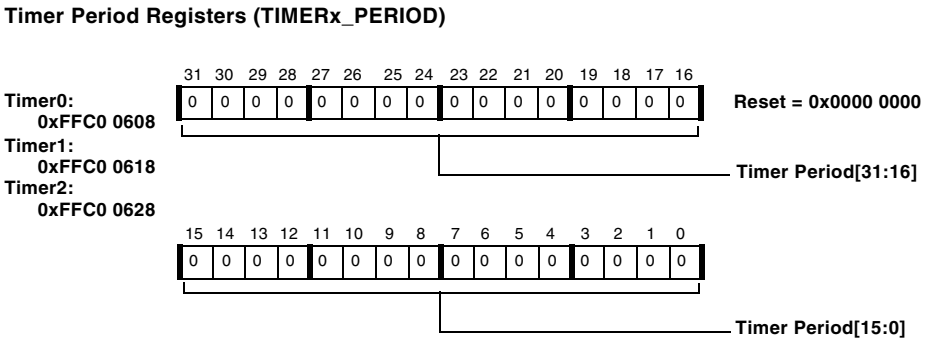


Figure 15-7. Timer Period Registers

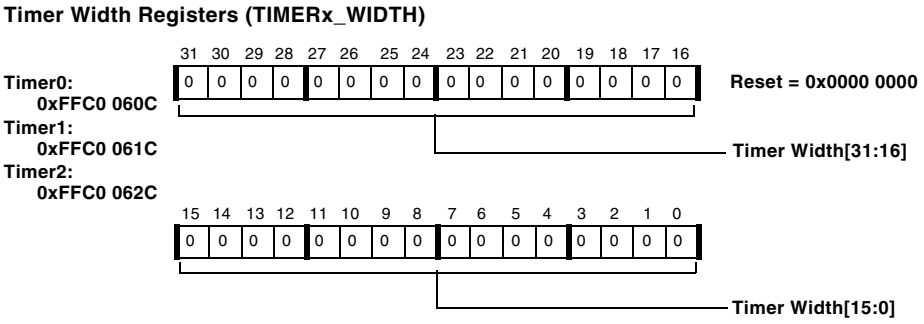


Figure 15-8. Timer Width Registers

Using the Timer

To enable an individual timer, set that timer's TIMEN bit in the TIMER_ENABLE register. To disable an individual timer, set that timer's TIMDIS bit in the TIMER_DISABLE register. To enable all three timers in parallel, set all three TIMEN bits in the TIMER_ENABLE register.

Before enabling a timer, always program the corresponding Timer Configuration (TIMERx_CONFIG) register. This register defines the timer operating mode, the polarity of the TMRx pin, and the timer interrupt behavior. Do not alter the operating mode while the timer is running.

Examples of timer enable and disable timing appear in Figure 15-9, Figure 15-10, and Figure 15-11.

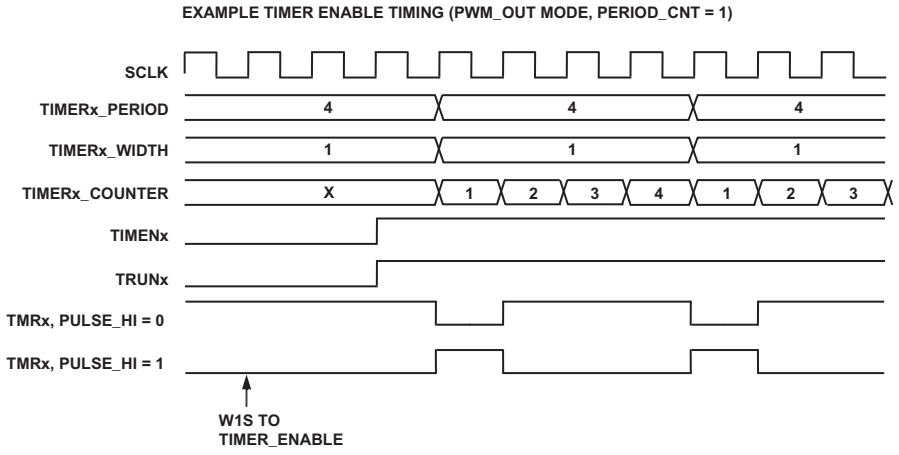


Figure 15-9. Timer Enable Timing

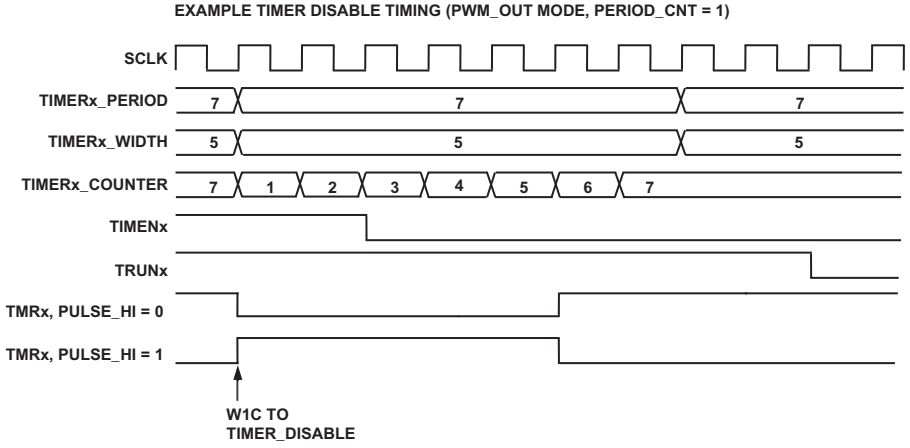


Figure 15-10. Timer Disable Timing

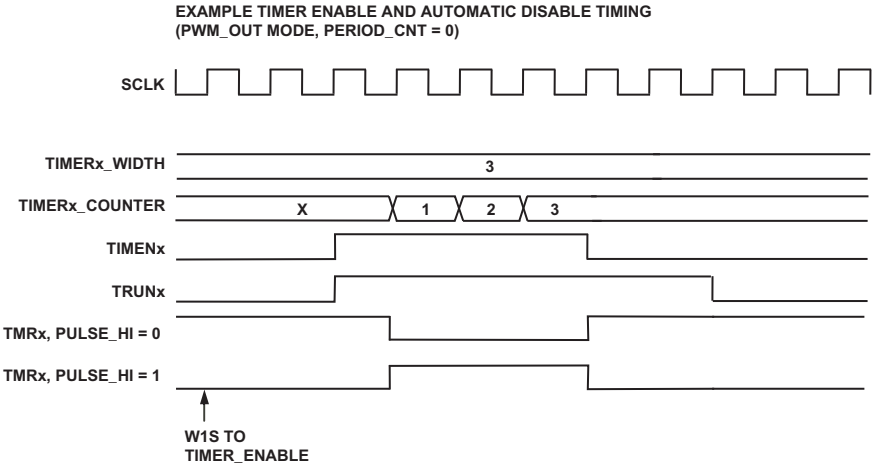


Figure 15-11. Timer Enable and Automatic Disable Timing

When timers are disabled, the Timer Counter registers retain their state; when a timer is re-enabled, the Timer Counter is reinitialized based on the operating mode. The Timer Counter registers are read-only. Software cannot overwrite or preset the Timer Counter value directly.

Pulse Width Modulation (PWM_OUT) Mode

Setting the TMODE field to b#01 in the Timer Configuration (TIMERx_CONFIG) register enables PWM_OUT mode. In PWM_OUT mode, the timer TMRx pin is an output. The output can be disabled by setting the OUT_DIS bit in the Timer Configuration register.

In PWM_OUT mode, the bits PULSE_HI, PERIOD_CNT, IRQ_ENA, OUT_DIS, CLK_SEL, EMU_RUN, and TOGGLE_HI enable orthogonal functionality. They may be set individually or in any combination, although some combinations are not useful (such as TOGGLE_HI = 1 with OUT_DIS = 1 or PERIOD_CNT = 0).

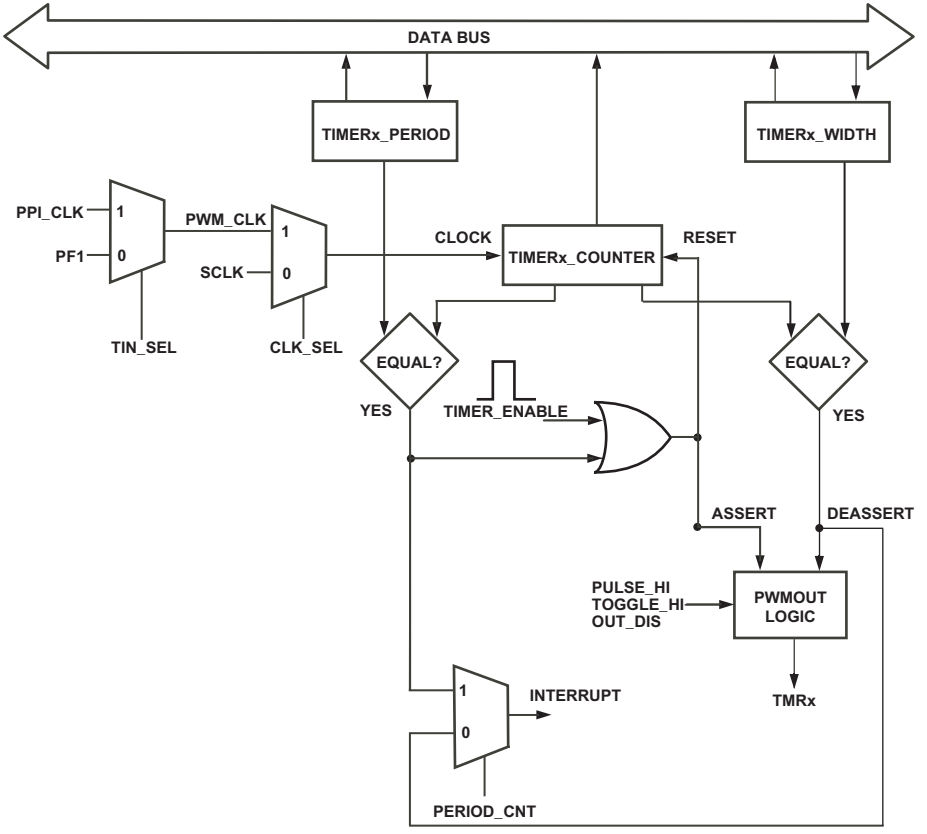


Figure 15-12. Timer Flow Diagram, PWM_OUT Mode

Once a timer has been enabled, the Timer Counter register is loaded with a starting value. If `CLK_SEL = 0`, the Timer Counter starts at 0x1. If `CLK_SEL = 1`, it is reset to 0x0 as in `EXT_CLK` mode. The timer counts upward to the value of the Timer Period register. For either setting of `CLK_SEL`, when the Timer Counter equals the Timer Period, the Timer Counter is reset to 0x1 on the next clock.

In PWM_OUT mode, the PERIOD_CNT bit controls whether the timer generates one pulse or many pulses. When PERIOD_CNT is cleared (PWM_OUT single pulse mode), the timer uses the TIMERx_WIDTH register, generates one asserting and one deasserting edge, then generates an interrupt (if enabled) and stops. When PERIOD_CNT is set (PWM_OUT continuous pulse mode), the timer uses both the TIMERx_PERIOD and TIMERx_WIDTH registers and generates a repeating (and possibly modulated) waveform. It generates an interrupt (if enabled) at the end of each period and stops only after it is disabled. A setting of PERIOD_CNT = 0 counts to the end of the Width; a setting of PERIOD_CNT = 1 counts to the end of the Period.

The TIMERx_PERIOD and TIMERx_WIDTH registers are read-only in some operation modes. Be sure to set the TMODE field in the TIMERx_CONFIG register to b#01 before writing to these registers.

Output Pad Disable

The output pin can be disabled in PWM_OUT mode by setting the OUT_DIS bit in the Timer Configuration register. The TMRx pin is then three-stated regardless of the setting of PULSE_HI and TOGGLE_HI. This can reduce power consumption when the output signal is not being used.

Single Pulse Generation

If the PERIOD_CNT bit is cleared, the PWM_OUT mode generates a single pulse on the TMRx pin. This mode can also be used to implement a precise delay. The pulse width is defined by the Timer Pulse Width register, and the Timer Period register is not used.

At the end of the pulse, the Timer Interrupt latch bit TIMILx gets set, and the timer is stopped automatically. If the PULSE_HI bit is set, an active high pulse is generated on the TMRx pin. If PULSE_HI is not set, the pulse is active low.

Pulse Width Modulation Waveform Generation

If the PERIOD_CNT bit is set, the internally clocked timer generates rectangular signals with well-defined period and duty cycle. This mode also generates periodic interrupts for real-time signal processing.

The 32-bit Timer Period (TIMERx_PERIOD) and Timer Pulse Width (TIMERx_WIDTH) 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 TMRx pin is pulled to a deasserted state each time the Timer Counter equals the value of the Timer Pulse Width register, and the pin is asserted again when the period expires (or when the timer gets started).

To control the assertion sense of the TMRx pin, the PULSE_HI bit in the corresponding TIMERx_CONFIG register is used. For a low assertion level, clear this bit. For a high assertion level, set this bit. When the timer is disabled in PWM_OUT mode, the TMRx pin is driven to the deasserted level.

If enabled, a timer interrupt is generated at the end of each period. An interrupt service routine (ISR) must clear the Interrupt Latch bit (TIMILx) and might alter period and/or width values. In pulse width modulation (PWM) applications, the software needs to update period and pulse width values while the timer is running. When software updates either period or pulse width registers, the new values are held by special buffer registers until the period expires. Then the new period and pulse width values become active simultaneously. New Timer Period and Timer Pulse Width register values are written while the old values are being used. The new values are loaded in to be used when the Timer Counter value equals the current Timer Period value. Reads from Timer Period and Timer Pulse Width registers return the old values until the period expires.

The TOVF_ERRx status bit signifies an error condition in PWM_OUT mode. The TOVF_ERRx bit is set if TIMERx_PERIOD = 0 or TIMERx_PERIOD = 1 at startup, or when the Timer Counter register rolls over. It is also set when

the Timer Counter register rolls over if the Timer Pulse Width register is greater than or equal to the Timer Period register. The ERR_TYP bits are set when the TOVF_ERRx bit is set.

To generate the maximum frequency on the TMRx output pin, set the period value to 2 and the pulse width to 1. This makes TMRx toggle each SCLK clock, producing a duty cycle of 50%. The period may be programmed to any value from 2 to (2^{32} – 1), inclusive. The pulse width may be programmed to any value from 1 to (Period – 1), inclusive. When PERIOD_CNT = 0, the pulse width may be programmed to any value from 1 to (2^{32} – 1), inclusive.

Although the hardware reports an error if the TIMERx_WIDTH value equals the TIMERx_PERIOD value, this is still a valid operation to implement PWM patterns with 100% duty cycle. If doing so, software must generally ignore the TOVL_ERRx flags. Pulse width values greater than the period value are not recommended. Similarly, TIMERx_WIDTH = 0 is not a valid operation. Duty cycles of 0% are not supported.

Stopping the Timer in PWM_OUT Mode

In all PWM_OUT mode variants, the timer treats a disable operation (W1C to TIMER_DISABLE) as a “stop is pending” condition. When disabled, it automatically completes the current waveform and then stops cleanly. This prevents truncation of the current pulse and unwanted PWM patterns at the TMRx pin. The processor can determine when the timer stops running by polling for the corresponding TRUNx bit in the TIMER_STATUS register to read 0 or by waiting for the last interrupt (if enabled). Note the timer cannot be reconfigured (TIMERx_CONFIG cannot be written to a new value) until after the timer stops and TRUNx reads 0.

In PWM_OUT single pulse mode (PERIOD_CNT = 0), it is not necessary to write TIMER_DISABLE to stop the timer. At the end of the pulse, the timer stops automatically, the corresponding bit in TIMER_ENABLE (and

TIMER_DISABLE) is cleared, and the corresponding TRUNx bit is cleared. See Figure 15-11. To generate multiple pulses, write a 1 to TIMER_ENABLE, wait for the timer to stop, then write another 1 to TIMER_ENABLE.

If necessary, the processor can force a timer in PWM_OUT mode to stop immediately. Do this by first writing a 1 to the corresponding bit in TIMER_DISABLE, and then writing a 1 to the corresponding TRUNx bit in TIMER_STATUS. This stops the timer whether the pending stop was waiting for the end of the current period (PERIOD_CNT = 1) or the end of the current pulse width (PERIOD_CNT = 0). This feature may be used to regain immediate control of a timer during an error recovery sequence.

Use this feature carefully, because it may corrupt the PWM pattern generated at the TMRx pin.

In PWM_OUT continuous pulse mode (PERIOD_CNT = 1), each timer samples its TIMENx bit at the end of each period. It stops cleanly at the end of the first period when TIMENx is low. This implies (barring any W1C to TRUNx) that a timer that is disabled and then re-enabled all before the end of the current period will continue to run as if nothing happened. Typically, software should disable a PWM_OUT timer and then wait for it to stop itself. The timer will always stop at the end of the first pulse when PERIOD_CNT = 0.

Externally Clocked PWM_OUT

By default, the timer is clocked internally by SCLK. Alternatively, if the CLK_SEL bit in the Timer Configuration (TIMERx_CONFIG) register is set, then the timer is clocked by PWM_CLK. The PWM_CLK is normally input from the PF1 pin, but may be taken from the PPI_CLK pin when the timers are configured to work with the PPI. Different timers may receive different signals on their PWM_CLK inputs, depending on configuration. As selected by the PERIOD_CNT bit, the PWM_OUT mode either generates pulse width modulation waveforms or generates a single pulse with pulse width defined by the TIMERx_WIDTH register.

When CLK_SEL is set, the counter resets to 0x0 at startup and increments on each rising edge of PWM_CLK. The TMRx pin transitions on rising edges of PWM_CLK. There is no way to select the falling edges of PWM_CLK. In this mode, the PULSE_HI bit controls only the polarity of the pulses produced. The timer interrupt may occur slightly before the corresponding edge on the TMRx pin (the interrupt occurs on an SCLK edge, the pin transitions on a later PWM_CLK edge). It is still safe to program new period and pulse width values as soon as the interrupt occurs. After a period expires, the counter rolls over to a value of 0x1.

The PWM_CLK clock waveform is not required to have a 50% duty cycle, but the minimum PWM_CLK clock low time is one SCLK period, and the minimum PWM_CLK clock high time is one SCLK period. This implies the maximum PWM_CLK clock frequency is SCLK/2.

The PF1 pin can only clock the timer when PF1 functions as an input pin. When any timer is in PWM_OUT mode with CLK_SEL = 1 and TIN_SEL = 0, then the PF1 bit in the FIO_DIR register is ignored and PF1 is forced to be an input.

PULSE_HI Toggle Mode

The waveform produced in PWM_OUT mode with PERIOD_CNT = 1 normally has a fixed assertion time and a programmable deassertion time (via the TIMERx_WIDTH register). When two timers are running synchronously by the same period settings, the pulses are aligned to the asserting edge as shown in Figure 15-13.

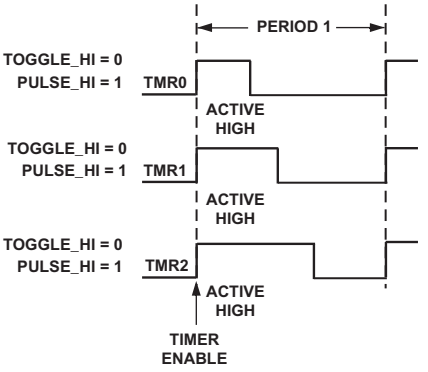


Figure 15-13. Timers With Pulses Aligned to Asserting Edge

The TOGGLE_HI mode enables control of the timing of both the asserting and deasserting edges of the output waveform produced. The phase between the asserting edges of two timer outputs is programmable. The effective state of the PULSE_HI bit alternates every period. The adjacent active low and active high pulses, taken together, create two halves of a fully arbitrary rectangular waveform. The effective waveform is still active high when PULSE_HI is set and active low when PULSE_HI is cleared. The value of TOGGLE_HI has no effect unless the mode is PWM_OUT and PERIOD_CNT = 1.

In TOGGLE_HI mode, when PULSE_HI is set, an active low pulse is generated in the first, third, and all odd-numbered periods, and an active high pulse is generated in the second, fourth, and all even-numbered periods. When PULSE_HI is cleared, an active high pulse is generated in the first, third, and all odd-numbered periods, and an active low pulse is generated in the second, fourth, and all even-numbered periods.

The deasserted state at the end of one period matches the asserted state at the beginning of the next period, so the output waveform only transitions when Count = Pulse Width. The net result is an output waveform pulse that repeats every two counter periods and is centered around the end of the first period (or the start of the second period).

Figure 15-14 shows an example with all three timers running with the same period settings. When software does not alter the PWM settings at runtime, the duty cycle is 50%. The values of the TIMERx_WIDTH registers control the phase between the signals.

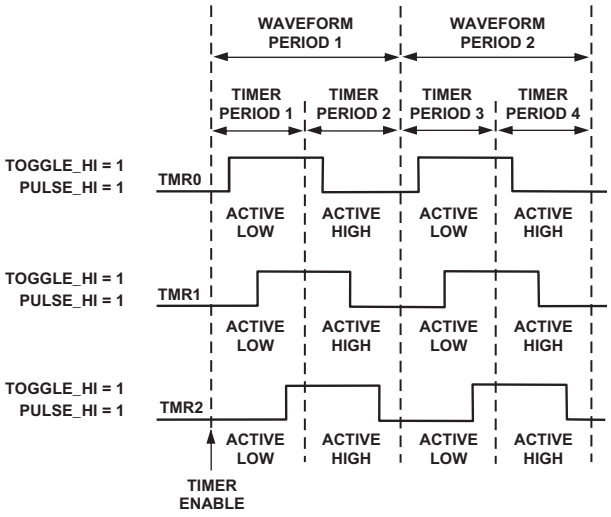


Figure 15-14. Three Timers With Same Period Settings

Similarly, two timers can generate non-overlapping clocks, by center-aligning the pulses while inverting the signal polarity for one of the timers (See Figure 15-15).

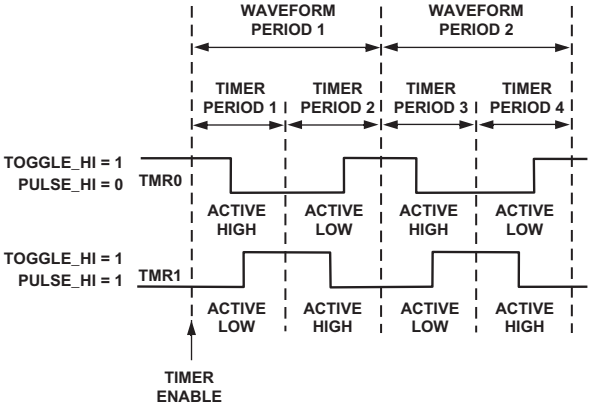


Figure 15-15. Two Timers With Non-Overlapping Clocks

When `TOGGLE_HI = 0`, software updates the Timer Period and Timer Pulse Width registers once per waveform period. When `TOGGLE_HI = 1`, software updates the Timer Period and Timer Pulse Width registers twice per waveform period with values that are half as large. In odd-numbered periods, write (`Period - Width`) instead of `Width` to the Timer Pulse Width register in order to obtain center-aligned pulses.

For example, if the pseudo-code when `TOGGLE_HI = 0` is:

```
int period, width ;
for (;;) {
     period = generate_period(...) ;
     width = generate_width(...) ;

     waitfor (interrupt) ;

     write(TIMERx_PERIOD, period) ;
     write(TIMERx_WIDTH, width) ;
}
```

Then when TOGGLE_HI = 1, the pseudo-code would be:

```
int period, width ;
int per1, per2, wid1, wid2 ;

for (;;)  {
     period = generate_period(...) ;
     width = generate_width(...) ;

     per1 = period/2 ;
     wid1 = width/2 ;

     per2 = period/2 ;
     wid2 = width/2 ;

     waitfor (interrupt) ;

     write(TIMERx_PERIOD, per1) ;
     write(TIMERx_WIDTH, per1 - wid1) ;

     waitfor (interrupt) ;

     write(TIMERx_PERIOD, per2) ;
     write(TIMERx_WIDTH, wid2) ;

}
```

As shown in this example, the pulses produced do not need to be symmetric (wid1 does not need to equal wid2). The period can be offset to adjust the phase of the pulses produced (per1 does not need to equal per2).

The Timer Slave Enable bit (TRUNx bit in the TIMER_STATUS register) is updated only at the end of even-numbered periods in TOGGLE_HI mode. When TIMER_DISABLE is written to 1, the current pair of counter periods (one waveform period) completes before the timer is disabled.

As when TOGGLE_HI = 0, errors are reported if:

TIMERx_WIDTH >= TIMERx_PERIOD, TIMERx_PERIOD = 0, or
TIMERx_PERIOD = 1

Pulse Width Count and Capture (WDTH_CAP) Mode

In WDTH_CAP mode, the TMRx pin is an input pin. The internally clocked timer is used to determine the period and pulse width of externally applied rectangular waveforms. Setting the TMODE field to b#10 in the TIMERx_CONFIG (Timer Configuration register) enables this mode.

When enabled in this mode, the timer resets the count in the TIMERx_COUNTER register to 0x0000 0001 and does not start counting until it detects a leading edge on the TMRx pin.

When the timer detects the first leading edge, it starts incrementing. When it detects a trailing edge of a waveform, the timer captures the current 32-bit value of the TIMERx_COUNTER register into the width buffer register. At the next leading edge, the timer transfers the current 32-bit value of the TIMERx_COUNTER register into the period buffer register. The count register is reset to 0x0000 0001 again, and the timer continues counting and capturing until it is disabled.

In this mode, software can measure both the pulse width and the pulse period of a waveform. To control the definition of leading edge and trailing edge of the TMRx pin, the PULSE_HI bit in the TIMERx_CONFIG register is set or cleared. If the PULSE_HI bit is cleared, the measurement is initiated by a falling edge, the Timer Counter register is captured to the Timer Pulse Width buffer register on the rising edge, and the Timer Period is captured on the next falling edge. When the PULSE_HI bit is set, the measurement is initiated by a rising edge, the Timer Counter register is captured to the Timer Pulse Width buffer register on the falling edge, and the Timer Period is captured on the next rising edge.

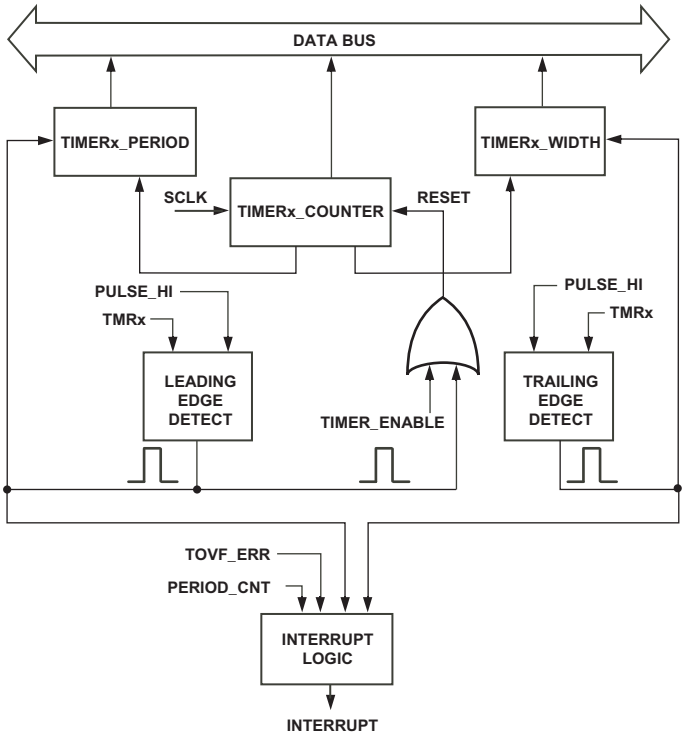


Figure 15-16. Timer Flow Diagram, WDTH_CAP Mode

In WDTH_CAP mode, these three events always occur at the same time as one unit:

1. The TIMERx_PERIOD register is updated from the period buffer register.
2. The TIMERx_WIDTH register is updated from the width buffer register.
3. The Timer Interrupt latch bit (TIMILx) gets set (if enabled) but does not generate an error.

The PERIOD_CNT bit in the TIMERx_CONFIG register controls the point in time at which this set of transactions is executed. Taken together, these three events are called a measurement report. The Timer Counter Overflow error latch bit (TOVF_ERRx) does not get set at a measurement report. A measurement report occurs at most once per input signal period.

The current timer counter value is always copied to the width buffer and period buffer registers at the trailing and leading edges of the input signal, respectively, but these values are not visible to software. A measurement report event samples the captured values into visible registers and sets the timer interrupt to signal that TIMERx_PERIOD and TIMERx_WIDTH are ready to be read. When the PERIOD_CNT bit is set, the measurement report occurs just after the period buffer register captures its value (at a leading edge). When the PERIOD_CNT bit is cleared, the measurement report occurs just after the width buffer register captures its value (at a trailing edge).

If the PERIOD_CNT bit is set and a leading edge occurred (See Figure 15-17), then the TIMERx_PERIOD and TIMERx_WIDTH registers report the pulse period and pulse width measured in the period that just ended. If the PERIOD_CNT bit is cleared and a trailing edge occurred (See Figure 15-18), then the TIMERx_WIDTH register reports the pulse width measured in the pulse that just ended, but the TIMERx_PERIOD register reports the pulse period measured at the end of the previous period.

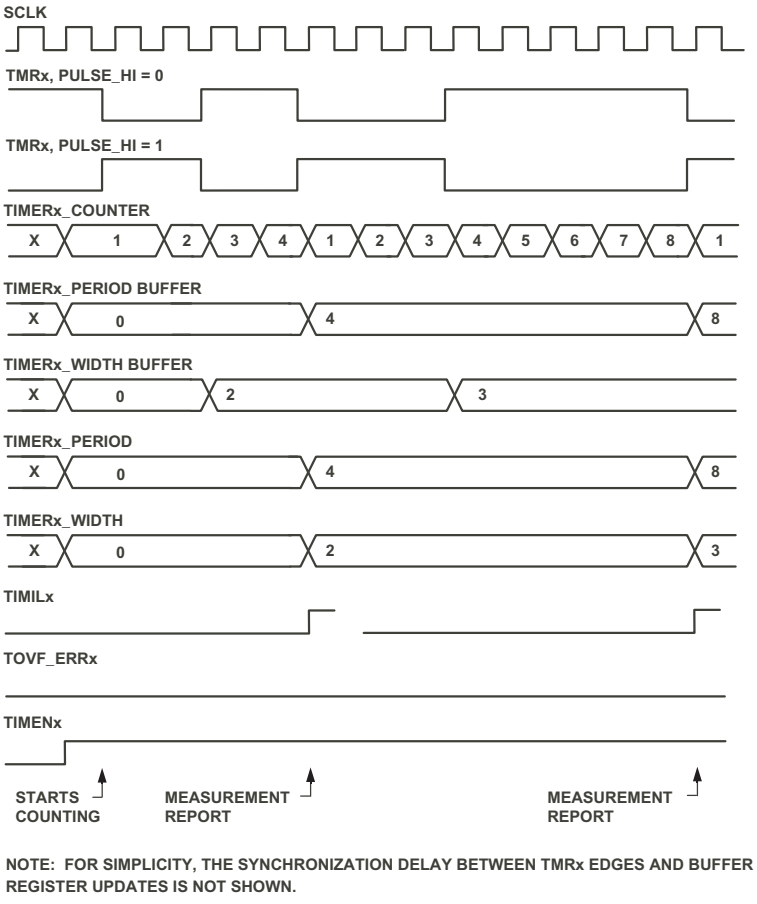


Figure 15-17. Example of Period Capture Measurement Report Timing (WDTH_CAP Mode, PERIOD_CNT = 1)

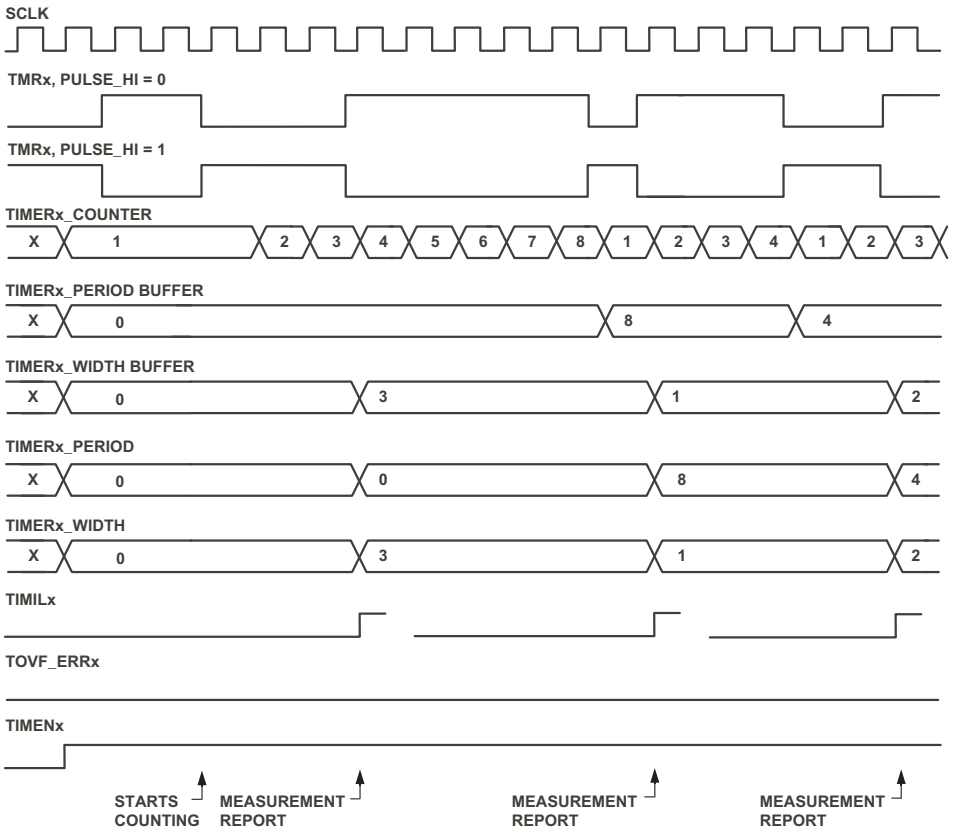


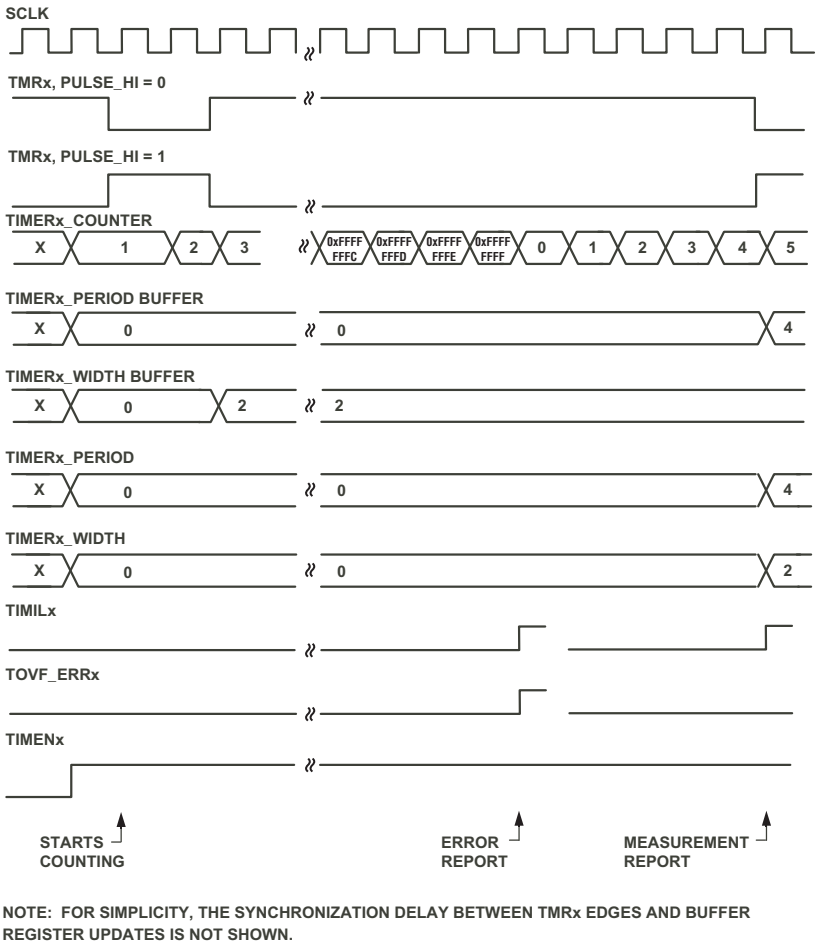
Figure 15-18. Example of Width Capture Measurement Report Timing (WDTH_CAP Mode, PERIOD_CNT = 0)

If the PERIOD_CNT bit is cleared and the first trailing edge occurred, then the first period value has not yet been measured at the first measurement report, so the period value is not valid. Reading the TIMERx_PERIOD value in this case returns 0, as shown in Figure 15-18. To measure the pulse width of a waveform that has only one leading edge and one trailing edge,

set PERIOD_CNT = 0. If PERIOD_CNT = 1 for this case, no period value is captured in the period buffer register. Instead, an error report interrupt is generated (if enabled) when the counter range is exceeded and the counter wraps around. In this case, both TIMERx_WIDTH and TIMERx_PERIOD read 0 (because no measurement report occurred to copy the value captured in the width buffer register to TIMERx_WIDTH). See the first interrupt in Figure 15-19.

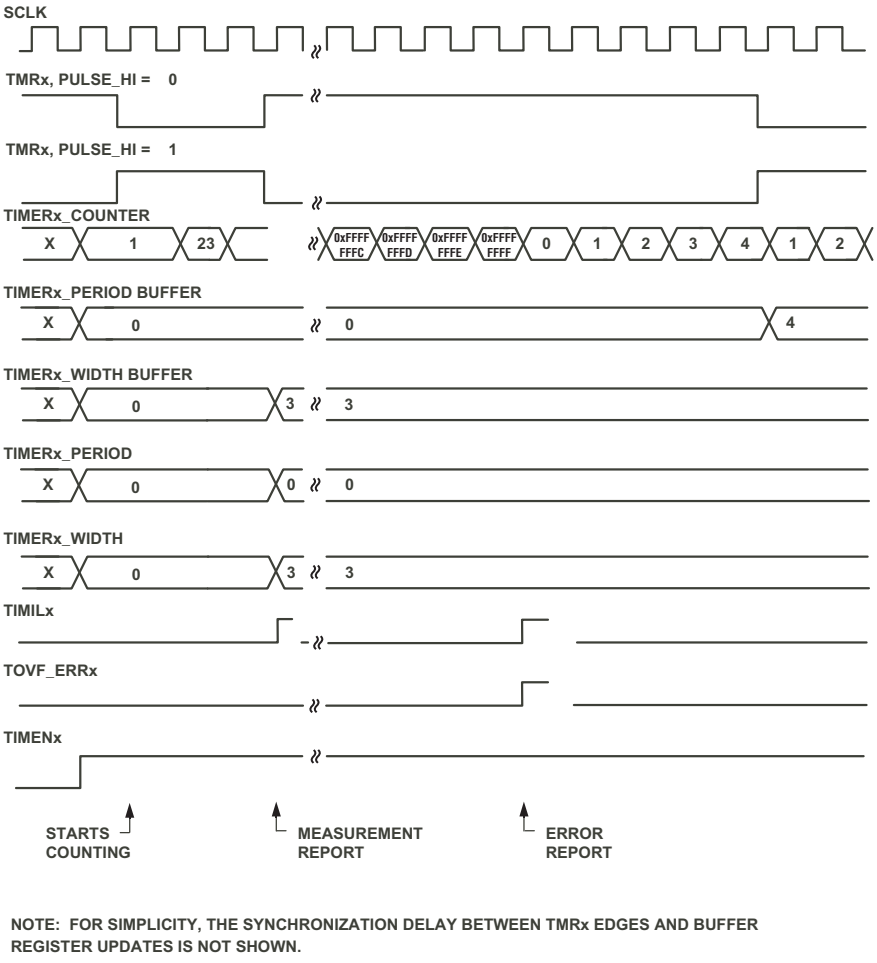
When using the PERIOD_CNT = 0 mode described above to measure the width of a single pulse, it is recommended to disable the timer after taking the interrupt that ends the measurement interval. If desired, the timer can then be reenabled as appropriate in preparation for another measurement. This procedure prevents the timer from free-running after the width measurement and logging errors generated by the timer count overflowing.

A timer interrupt (if enabled) is generated if the Timer Counter register wraps around from 0xFFFF FFFF to 0 in the absence of a leading edge. At that point, the TOVF_ERRx bit in the TIMER_STATUS register and the ERR_TYP bits in the TIMERx_CONFIG register are set, indicating a count overflow due to a period greater than the counter's range. This is called an error report. When a timer generates an interrupt in WDTH_CAP mode, either an error has occurred (an error report) or a new measurement is ready to be read (a measurement report), but never both at the same time. The TIMERx_PERIOD and TIMERx_WIDTH registers are never updated at the time an error is signaled. Refer to Figure 15-19 and Figure 15-20 for more information.



NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMRx EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 15-19. Example Timing for Period Overflow Followed by Period Capture (WDTH_CAP Mode, PERIOD_CNT = 1)



NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMRx EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 15-20. Example Timing for Width Capture Followed by Period Overflow (WDTH_CAP Mode, PERIOD_CNT = 0)

Both TIMILx and TOVF_ERRx are sticky bits, and software has to explicitly clear them. If the timer overflowed and PERIOD_CNT = 1, neither the TIMERx_PERIOD nor the TIMERx_WIDTH register were updated. If the timer

overflowed and PERIOD_CNT = 0, the TIMERx_PERIOD and TIMERx_WIDTH registers were updated only if a trailing edge was detected at a previous measurement report.

Software can count the number of error report interrupts between measurement report interrupts to measure input signal periods longer than 0xFFFF FFFF. Each error report interrupt adds a full 2^{32} SCLK counts to the total for the period, but the width is ambiguous. For example, in Figure 15-19 the period is 0x1 0000 0004 but the pulse width could be either 0x0 0000 0002 or 0x1 0000 0002.

The waveform applied to the TMRx pin is not required to have a 50% duty cycle, but the minimum TMRx low time is one SCLK period and the minimum TMRx high time is one SCLK period. This implies the maximum TMRx input frequency is SCLK/2 with a 50% duty cycle. Under these conditions, the WDTH_CAP mode timer would measure Period = 2 and Pulse Width = 1.

Autobaud Mode

Any one of the three timers may provide autobaud detection for the Universal Asynchronous Receiver/Transmitter (UART). The Timer Input Select (TIN_SEL) bit in the TIMERx_CONFIG register causes the timer to sample the UART port receive data (RX) pin instead of the TMRx pin when enabled for WDTH_CAP mode.

Do not enable the UART until after autobaud detection is complete.

A software routine can detect the pulse widths of serial stream bit cells. Because the sample base of the timers is synchronous with the UART operation—all derived from the Phase Locked Loop (PLL) clock—the pulse widths can be used to calculate the baud rate divider for the UART.

```
DIVISOR = ((TIMERx_WIDTH) / (16 x Number of captured UART bits))
```

In order to increase the number of timer counts and therefore the resolution of the captured signal, it is recommended not to measure just the pulse width of a single bit, but to enlarge the pulse of interest over more bits. Typically a NULL character (ASCII 0x00) is used in autobaud detection, as shown in Figure 15-21.

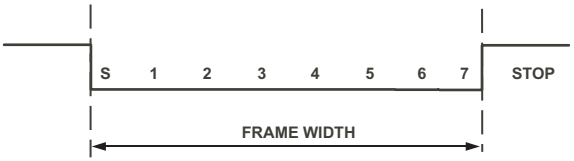


Figure 15-21. Autobaud Detection Character 0x00

Because the example frame in Figure 15-21 encloses 8 data bits and 1 start bit, apply the formula:

```
DIVISOR = TIMERx_WIDTH/(16 x 9)
```

Real UART RX signals often have asymmetrical falling and rising edges, and the sampling logic level is not exactly in the middle of the signal voltage range. At higher bit rates, such pulse width-based autobaud detection might not return adequate results without additional analog signal conditioning. Measuring signal periods works around this issue and is strongly recommended.

For example, predefine ASCII character “@” (40h) as an autobaud detection byte and measure the period between two subsequent falling edges. As shown in Figure 15-22, measure the period between the falling edge of the start bit and the falling edge after bit 6. Since this period encloses 8 bits, apply the formula:

```
DIVISOR = TIMERx_PERIOD/(16 x 8)
```

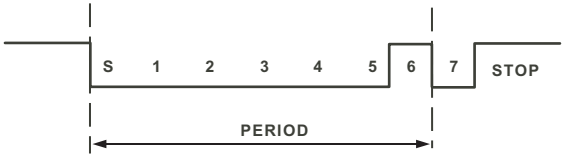


Figure 15-22. Autobaud Detection Character 0x40

External Event (EXT_CLK) Mode

In EXT_CLK mode, the TMRx pin is an input. The timer works as a counter clocked by an external source, which can also be asynchronous to the system clock. The current count in TIMERx_COUNTER represents the number of leading edge events detected. Setting the TMODE field to b#11 in the TIMERx_CONFIG register enables this mode. The TIMERx_PERIOD register is programmed with the value of the maximum timer external count.

The waveform applied to the TMRx pin is not required to have a 50% duty cycle, but the minimum TMRx low time is one SCLK period, and the minimum TMRx high time is one SCLK period. This implies the maximum TMRx input frequency is SCLK/2.

Period may be programmed to any value from 1 to (2^{32} – 1), inclusive.

After the timer has been enabled, it resets the Timer Counter register to 0x0 and then waits for the first leading edge on the TMRx pin. This edge causes the Timer Counter register to be incremented to the value 0x1. Every subsequent leading edge increments the count register. After reaching the period value, the TIMILx bit is set, and an interrupt is generated. The next leading edge reloads the Timer Counter register again with 0x1. The timer continues counting until it is disabled. The PULSE_HI bit determines whether the leading edge is rising (PULSE_HI set) or falling (PULSE_HI cleared).

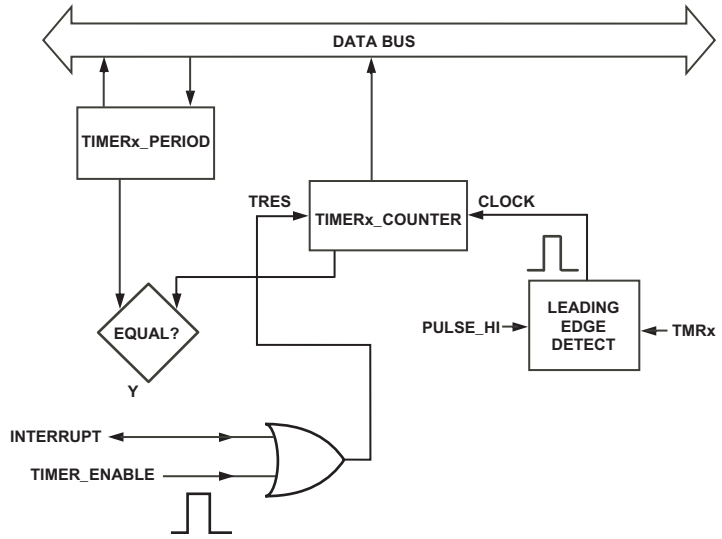


Figure 15-23. Timer Flow Diagram, EXT_CLK Mode

The configuration bits, `TIN_SEL` and `PERIOD_CNT`, have no effect in this mode. The `TOVF_ERRx` and `ERR_TYP` bits are set if the Timer Counter register wraps around from 0xFFFF FFFF to 0 or if `Period = 0` at startup or when the Timer Counter register rolls over (from `Count = Period` to `Count = 0x1`). The Timer Pulse Width register is unused.

Using the Timers With the PPI

Up to two timers are used to generate frame sync signals for certain PPI modes. For detailed instructions on how to configure the timers for use with the PPI, refer to "Frame Synchronization in GP Modes" on page 11-28 of the PPI chapter.

Interrupts

Each of the three timers can generate a single interrupt. The three resulting interrupt signals are routed to the System Interrupt Controller block for prioritization and masking. The Timer Status (TIMER_STATUS) register latches the timer interrupts to provide a means for software to determine the interrupt source. These bits are W1C and must be cleared prior to a RTI to assure that the interrupt is not reissued.

To enable interrupt generation, set the IRQ_ENA bit and unmask the interrupt source in the System Interrupt Mask register (SIC_IMASK). To poll the TIMILx bit without interrupt generation, set IRQ_ENA but leave the interrupt masked. If enabled by IRQ_ENA, interrupt requests are also generated by error conditions.

The system interrupt controller enables flexible interrupt handling. All timers may or may not share the same interrupt channel, so that a single interrupt routine services more than one timer. In PWM mode, more timers may run with the same period settings and issue their interrupt requests simultaneously. In this case, the service routine might clear all TIMILx latch bits at once by writing 0x07 to the TIMER_STATUS register.

If interrupts are enabled, make sure that the interrupt service routine (ISR) clears the TIMILx bit in the TIMERx_STATUS register before the RTI instruction executes. This ensures that the interrupt is not reissued. Remember that writes to system registers are delayed. If only a few instructions separate the TIMILx clear command from the RTI instruction, an extra SSYNC instruction may be inserted. In EXT_CLK mode, reset the TIMILx bit in the TIMERx_STATUS register at the very beginning of the interrupt service routine (ISR) to avoid missing any timer events.

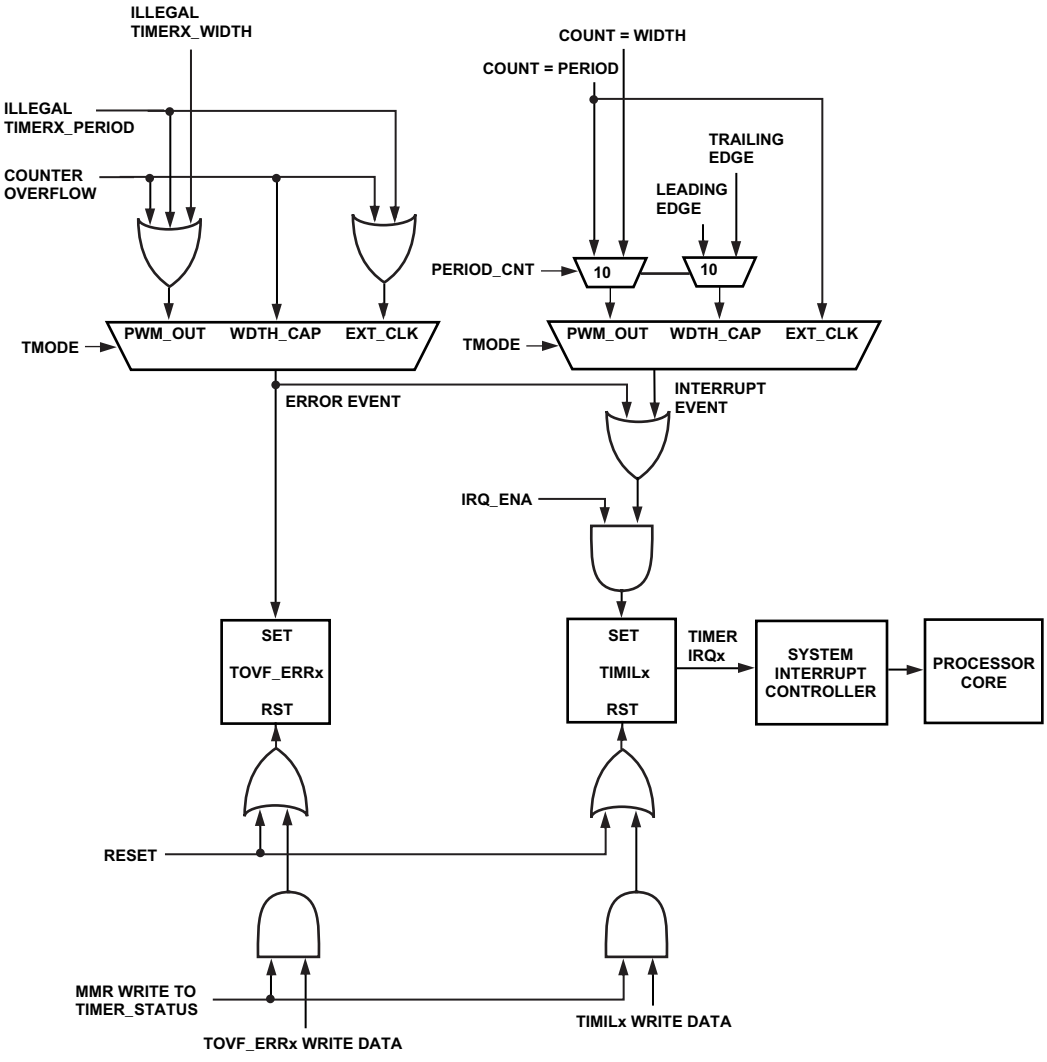


Figure 15-24. Timers Interrupt Structure

Illegal States

For Table 15-1, these definitions are used:

- **Startup**. The first clock period during which the timer counter is running after the timer is enabled by writing TIMER_ENABLE.
- **Rollover**. The time when the current count matches the value in TIMERx_PERIOD and the counter is reloaded with the value 1.
- **Overflow**. The timer counter was incremented instead of doing a rollover when it was holding the maximum possible count value of 0xFFFF FFFF. The counter does not have a large enough range to express the next greater value and so erroneously loads a new value of 0x0000 0000.
- **Unchanged**. No new error.
    - When ERR_TYP is unchanged, it displays the previously reported error code or 00 if there has been no error since this timer was enabled.
    - When TOVF_ERR is unchanged, it reads 0 if there has been no error since this timer was enabled, or if software has performed a W1C to clear any previous error. If a previous error has not been acknowledged by software, TOVF_ERR reads 1.

Software should read TOVF_ERR to check for an error. If TOVF_ERR is set, software can then read ERR_TYP for more information. Once detected, software should write 1 to clear TOVF_ERR to acknowledge the error.

The following table can be read as: “In mode __ at event __, if TIMERx_PERIOD is __ and TIMERx_WIDTH is __, then ERR_TYP is __ and TOVF_ERR is __.”

Startup error conditions do not prevent the timer from starting. Similarly, overflow and rollover error conditions do not stop the timer. Illegal cases may cause unwanted behavior of the TMRx pin.

Table 15-1. Overview of Illegal States

Mode	Event	TIMERx_ PERIOD	TIMERx_ WIDTH	ERR_TYP	TOVF_ERR
PWM_OUT, PERIOD_ CNT = 1	Startup (No boundary condition tests performed on TIMERx_ WIDTH)	== 0	Anything	b#10	Set
		== 1	Anything	b#10	Set
		>= 2	Anything	Unchanged	Unchanged
	Rollover	== 0	Anything	b#10	Set
		== 1	Anything	b#11	Set
		>= 2	== 0	b#11	Set
		>= 2	< TIMERx_ PERIOD	Unchanged	Unchanged
		>= 2	>= TIMERx_ PERIOD	b#11	Set
	Overflow, not possible unless there is also another error, such as TIMERx_ PERIOD == 0.	Anything	Anything	b#01	Set

Table 15-1. Overview of Illegal States (Cont'd)

Mode	Event	TIMERx_ PERIOD	TIMERx_ WIDTH	ERR_TYP	TOVF_ERR
PWM_OUT, PERIOD_ CNT = 0	Startup	Anything	== 0	b#01	Set
		This case is not detected at startup, but results in an overflow error once the counter counts through its entire range.			
		Anything	>= 1	Unchanged	Unchanged
	Rollover	Rollover is not possible in this mode.			
	Overflow, not possible unless there is also another error, such as TIMERx_ WIDTH == 0.	Anything	Anything	b#01	Set
WDTH_CAP	Startup	TIMERx_PERIOD and TIMERx_WIDTH are read-only in this mode, no error possible.			
	Rollover	TIMERx_PERIOD and TIMERx_WIDTH are read-only in this mode, no error possible.			
	Overflow	Anything	Anything	b#01	Set
EXT_CLK	Startup	== 0	Anything	b#10	Set
		>= 1	Anything	Unchanged	Unchanged
	Rollover	== 0	Anything	b#10	Set
		>= 1	Anything	Unchanged	Unchanged
	Overflow, not possible unless there is also another error, such as TIMERx_ PERIOD == 0.	Anything	Anything	b#01	Set

Summary

Table 15-2 summarizes control bit and register usage in each timer mode.

Table 15-2. Control Bit and Register Usage Chart

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TIMER_ENABLE	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect
TIMER_DISABLE	1 - Disable timer at end of period 0 - No effect	1 - Disable timer 0 - No effect	1 - Disable timer 0 - No effect
TMODE	b#01	b#10	b#11
PULSE_HI	1 - Generate high width 0 - Generate low width	1 - Measure high width 0 - Measure low width	1 - Count rising edges 0 - Count falling edges
PERIOD_CNT	1 - Generate PWM 0 - Single width pulse	1 - Interrupt after measuring period 0 - Interrupt after measuring width	Unused
IRQ_ENA	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt
TIN_SEL	Depends on CLK_SEL: If CLK_SEL = 1, 1 - Count PPI_CLKs 0 - Count PF1 clocks If CLK_SEL = 0, Unused	1 - Select RX input 0 - Select TMRx input	Unused
OUT_DIS	1 - Disable TMRx pin 0 - Enable TMRx pin	Unused	Unused
CLK_SEL	1 - PWM_CLK clocks timer 0 - SCLK clocks timer	Unused	Unused

Table 15-2. Control Bit and Register Usage Chart (Cont'd)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TOGGLE_HI	1 - One waveform period every two counter periods 0 - One waveform period every one counter period	Unused	Unused
ERR_TYP	Reports b#00, b#01, b#10, or b#11, as appropriate	Reports b#00 or b#01, as appropriate	Reports b#00, b#01, or b#10, as appropriate
EMU_RUN	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation
TMR Pin	Depends on OUT_DIS: 1 - Three-state 0 - Output	Depends on TIN_SEL: 1 - Unused 0 - Input	Input
Period	R/W: Period value	RO: Period value	R/W: Period value
Width	R/W: Width value	RO: Width value	Unused
Counter	RO: Counts up on SCLK or PWM_CLK	RO: Counts up on SCLK	RO: Counts up on event
TRUNx	Read: Timer slave enable status Write: 1 - Stop timer if disabled 0 - No effect	Read: Timer slave enable status Write: 1 - No effect 0 - No effect	Read: Timer slave enable status Write: 1 - No effect 0 - No effect

Table 15-2. Control Bit and Register Usage Chart (Cont'd)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TOVF_ERR	Set at startup or roll-over if period = 0 or 1 Set at rollover if width >= Period Set if counter wraps	Set if counter wraps	Set if counter wraps or set at startup or roll-over if period = 0
IRQ	Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter equals period and PERIOD_CNT = 1 or when counter equals width and PERIOD_CNT = 0 0 - Not set	Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter captures period and PERIOD_CNT = 1 or when counter captures width and PERIOD_CNT = 0 0 - Not set	Depends on IRQ_ENA: 1 - Set when counter equals period or TOVF_ERR set 0 - Not set

Core Timer

The Core timer is a programmable interval timer which can generate periodic interrupts. The Core timer runs at the core clock (CCLK) rate. The timer includes four core Memory-Mapped Registers (MMRs), the Timer Control register (TCNTL), the Timer Count register (TCOUNT), the Timer Period register (TPERIOD), and the Timer Scale register (TSCALE).

Figure 15-25 provides a block diagram of the Core timer.

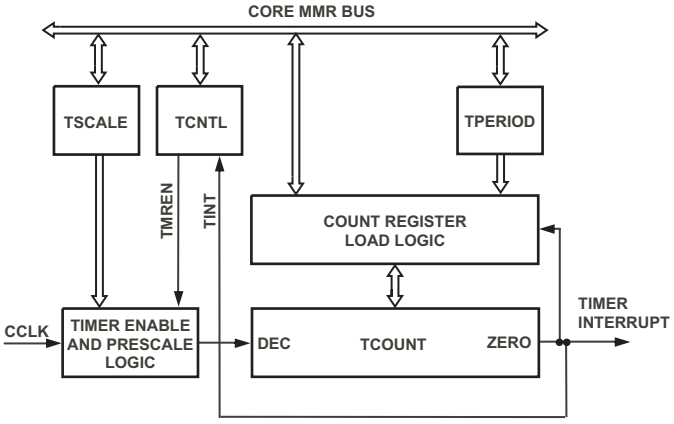


Figure 15-25. Core Timer Block Diagram

TCNTL Register

When the timer is enabled by setting the TMREN bit in the Core Timer Control register (TCNTL), the TCOUNT register is decremented once every TSCALE + 1 number of clock cycles. When the value of the TCOUNT register reaches 0, an interrupt is generated and the TINT bit is set in the TCNTL register. If the TAUTORLD bit in the TCNTL register is set, then the TCOUNT register is reloaded with the contents of the TPERIOD register and the count begins again.

The TINT bit in the TCNTL register indicates that an interrupt has been generated. Note that this is *not* a W1C bit. Write a 0 to clear it. However, the write is optional. It is not required to clear interrupt requests. The core time module does not provide any further interrupt enable bit. When the timer is enabled, interrupts can be masked in the CEC controller.

The Core timer can be put into low power mode by clearing the TMPWR bit in the TCNTL register. Before using the timer, set the TMPWR bit. This restores clocks to the timer unit. When TMPWR is set, the Core timer may then be enabled by setting the TMREN bit in the TCNTL register.

Hardware behavior is undefined if TMREN is set when TMPWR = 0.

Core Timer Control Register (TCNTL)

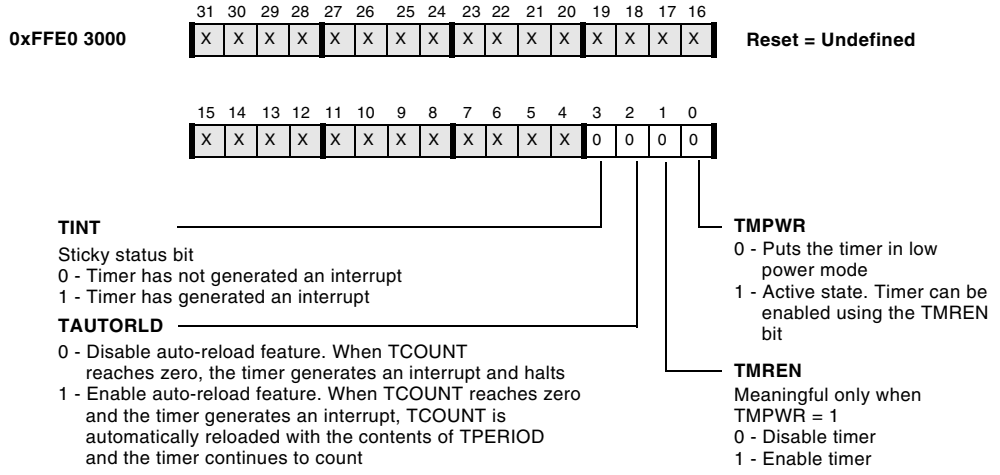


Figure 15-26. Core Timer Control Register

TCOUNT Register

The Core Timer Count register (TCOUNT) decrements once every TSCALE + 1 clock cycles. When the value of TCOUNT reaches 0, an interrupt is generated and the TINT bit of the TCNTL register is set.

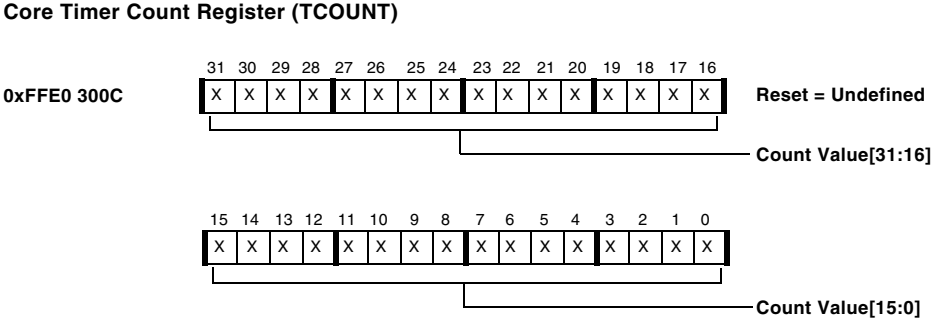


Figure 15-27. Core Timer Count Register

TPERIOD Register

When auto-reload is enabled, the TCOUNT register is reloaded with the value of the Core Timer Period register (TPERIOD) whenever TCOUNT reaches 0.

To ensure that there is valid data in the TPERIOD register, the TPERIOD and TCOUNT registers are initialized simultaneously on the first write to either register. If a different value is desired for the first count period, write the data to TCOUNT after writing to TPERIOD.

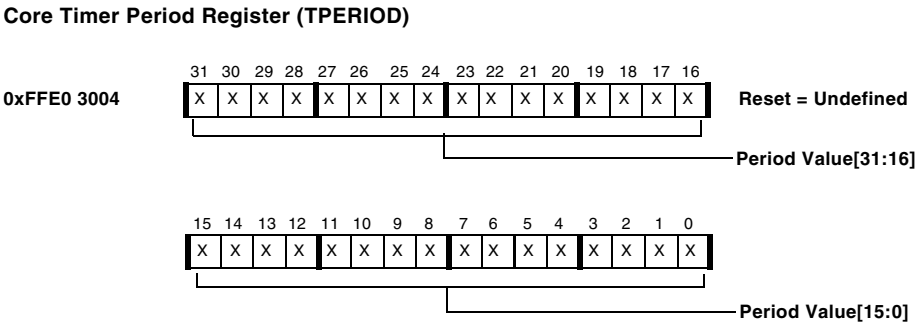


Figure 15-28. Core Timer Period Register

TSCALE Register

The Core Timer Scale register (TSCALE) stores the scaling value that is one less than the number of cycles between decrements of TCOUNT. For example, if the value in the TSCALE register is 0, the counter register decrements once every clock cycle. If TSCALE is 1, the counter decrements once every two cycles.

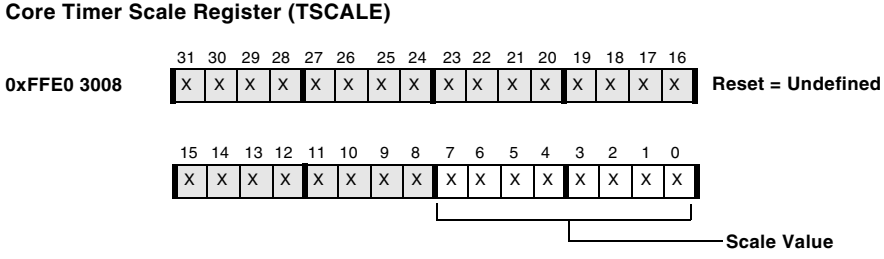


Figure 15-29. Core Timer Scale Register

Watchdog Timer

The processor includes a 32-bit timer that can be used to implement a software watchdog function. A software watchdog can improve system reliability by generating an event to the processor core if the timer expires before being updated by software. Depending on how the watchdog timer is programmed, the event that is generated may be a reset, a nonmaskable interrupt, or a general-purpose interrupt. The watchdog timer is clocked by the system clock (SCLK).

Watchdog Timer Operation

To use the watchdog timer:

1. Set the count value for the watchdog timer by writing the count value into the Watchdog Count register (WDOG_CNT). Note that loading the WDOG_CNT register while the watchdog timer is not enabled will also pre-load the WDOG_STAT register.
2. In the Watchdog Control register (WDOG_CTL), select the event to be generated upon timeout.
3. Enable the watchdog timer in WDOG_CTL. The watchdog timer then begins counting down, decrementing the value in the WDOG_STAT register. When the WDOG_STAT reaches 0, the programmed event is generated. To prevent the event from being generated, software must reload the count value from WDOG_CNT to WDOG_STAT by executing a write (of any value) to WDOG_STAT, or must disable the watchdog timer in WDOG_CTL before the watchdog timer expires.

WDOG_CNT Register

The Watchdog Count register (WDOG_CNT) holds the 32-bit unsigned count value. The WDOG_CNT register must be accessed with 32-bit read/writes only.

The Watchdog Count register holds the programmable count value. A valid write to the Watchdog Count register also preloads the Watchdog counter. For added safety, the Watchdog Count register can be updated only when the watchdog timer is disabled. A write to the Watchdog Count register while the timer is enabled does not modify the contents of this register.

Watchdog Count Register (WDOG_CNT)

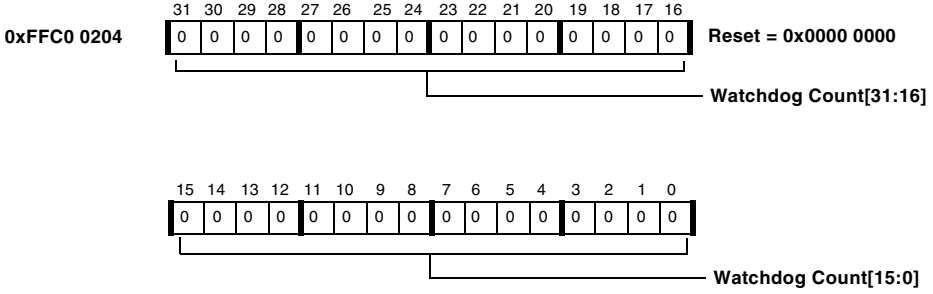


Figure 15-30. Watchdog Count Register

WDOG_STAT Register

The 32-bit Watchdog Status register (WDOG_STAT) contains the current count value of the watchdog timer. Reads to WDOG_STAT return the current count value. When the watchdog timer is enabled, WDOG_STAT is decremented by 1 on each SCLK cycle. When WDOG_STAT reaches 0, the watchdog timer stops counting and the event selected in the Watchdog Control register (WDOG_CTL) is generated.

Values cannot be stored directly in WDOG_STAT, but are instead copied from WDOG_CNT. This can happen in two ways.

- While the watchdog timer is disabled, writing the WDOG_CNT register pre-loads the WDOG_STAT register.
- While the watchdog timer is enabled, writing the WDOG_STAT register loads it with the value in WDOG_CNT.

When the processor executes a write (of an arbitrary value) to WDOG_STAT, the value in WDOG_CNT is copied into WDOG_STAT. Typically, software sets the value of WDOG_CNT at initialization, then periodically writes to WDOG_STAT before the watchdog timer expires. This reloads the watchdog timer with the value from WDOG_CNT and prevents generation of the selected event.

The WDOG_STAT register is a 32-bit unsigned system memory-mapped register that must be accessed with 32-bit reads and writes.

If the user does not reload the counter before SCLK * Count register cycles, a Watchdog interrupt or reset is generated and the WDRO bit in the Watchdog Control register is set. When this happens the counter stops decrementing and remains at zero.

If the counter is enabled with a zero loaded to the counter, the WDRO bit of the Watchdog Control register is set immediately and the counter remains at zero and does not decrement.

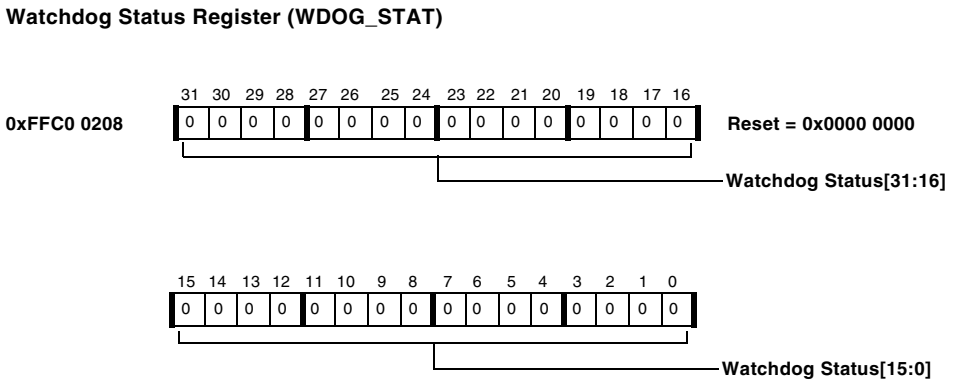


Figure 15-31. Watchdog Status Register

WDOG_CTL Register

The Watchdog Control register (WDOG_CTL) is a 16-bit system memory-mapped register used to control the watchdog timer.

The WDEV[1:0] field is used to select the event that is generated when the watchdog timer expires. Note that if the general-purpose interrupt option is selected, the System Interrupt Mask register (SIC_IMASK) should be appropriately configured to unmask that interrupt. If the generation of watchdog events is disabled, the watchdog timer operates as described, except that no event is generated when the watchdog timer expires.

The WDEN[7:0] field is used to enable and disable the watchdog timer. Writing any value other than the disable value into this field enables the watchdog timer. This multibit disable key minimizes the chance of inadvertently disabling the watchdog timer.

Software can determine whether the timer has rolled over by interrogating the WDRO status bit of the Watchdog Control register. This is a sticky bit that is set whenever the watchdog timer count reaches 0 and cleared only by disabling the watchdog timer and then writing a 1 to the bit.

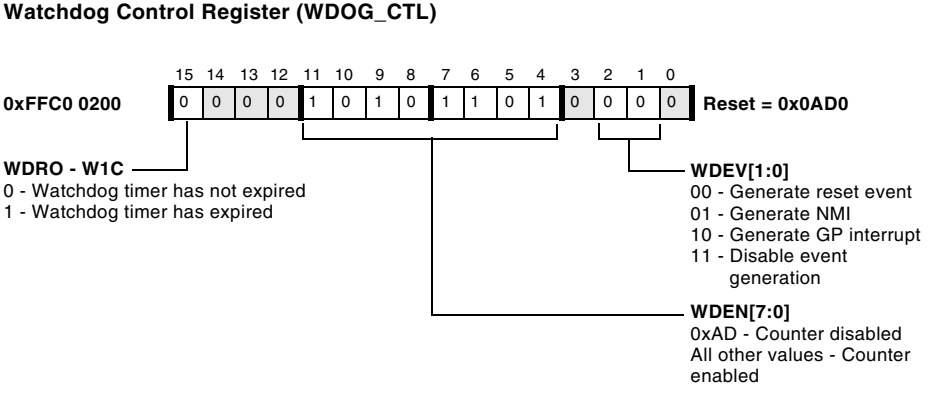


Figure 15-32. Watchdog Control Register

Note that when the processor is in Emulation mode, the watchdog timer counter will not decrement even if it is enabled.

16 REAL-TIME CLOCK

The Real-Time Clock (RTC) provides a set of digital watch features to the processor, including time of day, alarm, and stopwatch countdown. It is typically used to implement either a real-time watch or a life counter.

The RTC watch features are clocked by a 32.768 kHz crystal external to the processor. The RTC uses dedicated power supply pins and is independent of any reset, which enables it to maintain functionality even when the rest of the processor is powered down.

The RTC input clock is divided down to a 1 Hz signal by a prescaler, which can be bypassed. When bypassed, the RTC is clocked at the 32.768 kHz crystal rate. In normal operation, the prescaler is enabled.

The primary function of the RTC is to maintain an accurate day count and time of day. The RTC accomplishes this by means of four counters:

- 60-second counter
- 60-minute counter
- 24-hour counter
- 32768-day counter

The RTC increments the 60-second counter once per second and increments the other three counters when appropriate. The 32768-day counter is incremented each day at midnight (0 hours, 0 minutes, 0 seconds). Interrupts can be issued periodically, either every second, every minute, every hour, or every day. Each of these interrupts can be independently controlled.

The RTC provides two alarm features, programmed with the RTC Alarm register (RTC_ALARM). The first is a time of day alarm (hour, minute, and second). When the alarm interrupt is enabled, the RTC generates an interrupt each day at the time specified. The second alarm feature allows the application to specify a day as well as a time. When the day alarm interrupt is enabled, the RTC generates an interrupt on the day and time specified. The alarm interrupt and day alarm interrupt can be enabled or disabled independently.

The RTC provides a stopwatch function that acts as a countdown timer. The application can program a second count into the RTC Stopwatch Count register (RTC_SWCNT). When the stopwatch interrupt is enabled and the specified number of seconds have elapsed, the RTC generates an interrupt.

Interfaces

The RTC external interface consists of two clock pins, which together with the external components form the reference clock circuit for the RTC. The RTC interfaces internally to the processor system through the Peripheral Access bus (PAB), and through the interrupt interface to the SIC (System Interrupt Controller).

The RTC has dedicated power supply pins that power the clock functions at all times, including when the core power supply is turned off.

RTC Clock Requirements

The RTC timer is clocked by a 32.768 kHz crystal external to the processor. The RTC system memory-mapped registers (MMRs) are clocked by this crystal. When the prescaler is disabled, the RTC MMRs are clocked at the 32.768 kHz crystal frequency. When the prescaler is enabled, the RTC MMRs are clocked at the 1 Hz rate.

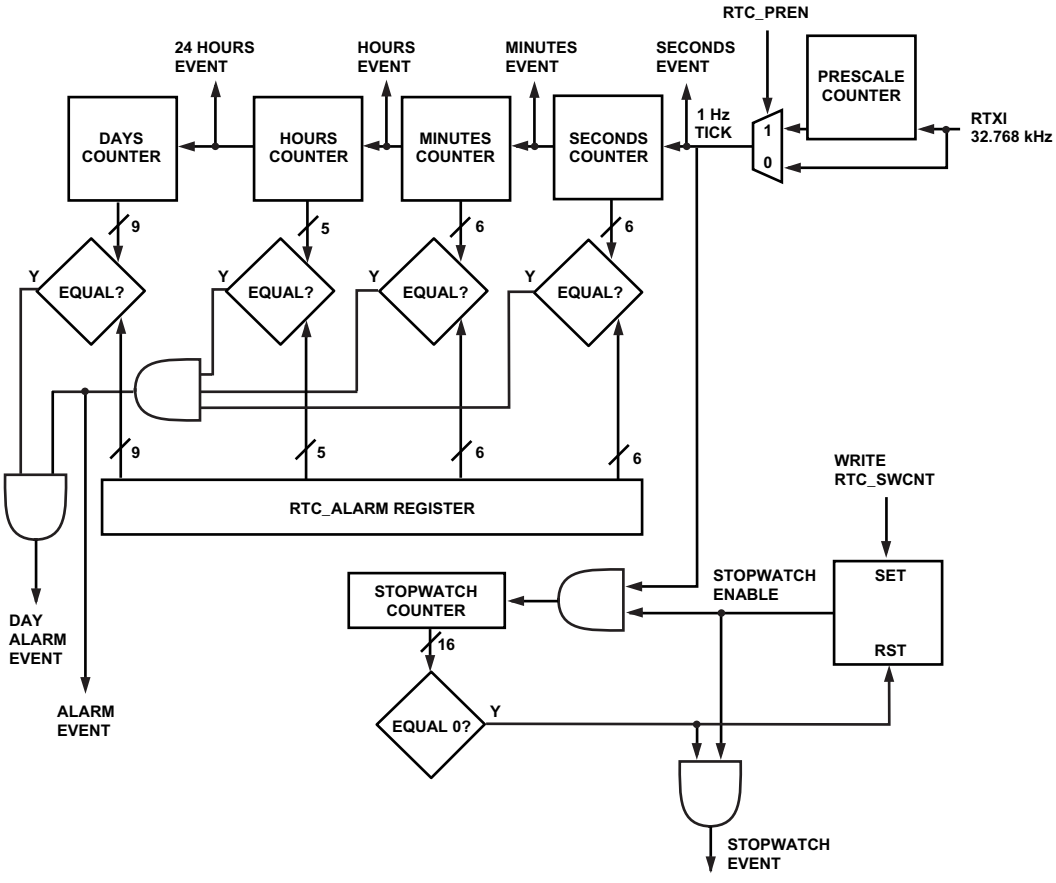


Figure 16-1. RTC Block Diagram

There is no way to disable the RTC counters from software. If a given system does not require the RTC functionality, then it may be disabled with hardware tie-offs. Tie the RTXI pin to EGND, tie the RTCVDD pin to EVDD, and leave the RTXO pin unconnected.

RTC Programming Model

The RTC programming model consists of a set of system MMRs. Software can configure the RTC and can determine the status of the RTC through reads and writes to these registers. The RTC Interrupt Control register (RTC_ICTL) and the RTC Interrupt Status register (RTC_ISTAT) provide RTC interrupt management capability.

Note that software cannot disable the RTC counting function. However, all RTC interrupts can be disabled, or masked. At reset, all interrupts are disabled. The RTC state can be read via the system MMR status registers at any time.

The primary Real-Time Clock functionality, shown in Figure 16-1, consists of registers and counters that are powered by an independent RTC V_{dd} supply. This logic is never reset; it comes up in an unknown state when RTC V_{dd} is first powered on.

The RTC also contains logic powered by the same internal V_{dd} as the processor core and other peripherals. This logic contains some control functionality, holding registers for PAB write data, and prefetched PAB read data shadow registers for each of the five RTC V_{dd}-powered registers. This logic is reset by the same system reset and clocked by the same SCLK as the other peripherals.

Figure 16-2 shows the connections between the RTC V_{dd}-powered RTC MMRs and their corresponding internal V_{dd}-powered write holding registers and read shadow registers. In the figure, “REG” means each of the RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, and RTC_PREN registers. The RTC_ISTAT register connects only to the PAB.

The rising edge of the 1 Hz RTC clock is the “1 Hz tick”. Software can synchronize to the 1 Hz tick by waiting for the Seconds Event flag to set or by waiting for the Seconds Interrupt (if enabled).

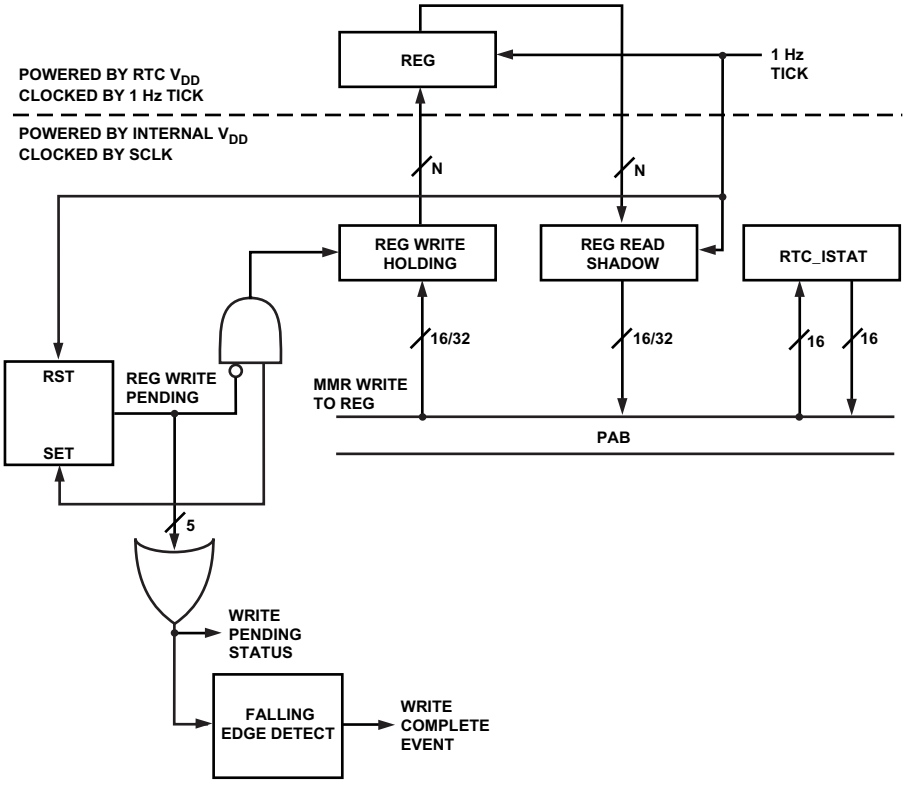


Figure 16-2. RTCT Register Architecture

Register Writes

Writes to all RTC MMRs, except the RTC Interrupt Status register (RTC_ISTAT), are saved in write holding registers and then are synchronized to the RTC 1 Hz clock. The Write Pending Status bit in RTC_ISTAT indicates the progress of the write. The Write Pending Status bit is set when a write is initiated and is cleared when all writes are complete. The falling edge of the Write Pending Status bit causes the Write Complete flag in RTC_ISTAT to be set. This flag can be configured in RTC_ICTL to

cause an interrupt. Software does not have to wait for writes to one RTC MMR to complete before writing to another RTC MMR. The Write Pending Status bit is set if any writes are in progress, and the Write Complete flag is set only when all writes are complete.

Any writes in progress when peripherals are reset will be aborted. Do not stop SCLK (enter Deep Sleep mode) or remove Internal V_{dd} power until all RTC writes have completed.

Do not attempt another write to the same register without waiting for the previous write to complete. Subsequent writes to the same register are ignored if the previous write is not complete.

Reading a register that has been written before the Write Complete flag is set will return the old value. Always check the Write Pending Status bit before attempting a read or write.

Write Latency

Writes to the RTC MMRs are synchronized to the 1 Hz RTC clock. When setting the time of day, do not factor in the delay when writing to the RTC MMRs. The most accurate method of setting the Real-Time Clock is to monitor the Seconds (1 Hz) Event flag or to program an interrupt for this event and then write the current time to the RTC Status register (RTC_STAT) in the interrupt service routine (ISR). The new value is inserted ahead of the incrementer. Hardware adds one second to the written value (with appropriate carries into minutes, hours and days) and loads the incremented value at the next 1 Hz tick, when it represents the then-current time.

Writes posted at any time are properly synchronized to the 1 Hz clock. Writes complete at the rising edge of the 1 Hz clock. A write posted just before the 1 Hz tick may not be completed until the 1 Hz tick one second later. Any write posted in the first 990 ms after a 1 Hz tick will complete on the next 1 Hz tick, but the simplest, most predictable and

recommended technique is to only post writes to RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, or RTC_PREN immediately after a Seconds Interrupt or Event. All five registers may be written in the same second.

W1C bits in the RTC_ISTAT register take effect immediately.

Register Reads

There is no latency when reading RTC MMRs, as the values come from the read shadow registers. The shadows are updated and ready for reading by the time any RTC interrupts or Event flags for that second are asserted. Once the internal V_{dd} logic completes its initialization sequence after SCLK starts, there is no point in time when it is unsafe to read the RTC MMRs for synchronization reasons. They always return coherent values, although the values may be unknown.

Deep Sleep

When the Dynamic Power Management Controller (DPMC) state is Deep Sleep, all clocks in the system (except RTXI and the RTC 1 Hz tick) are stopped. In this state, the RTC V_{dd} counters continue to increment. The internal V_{dd} shadow registers are not updated, but neither can they be read.

During Deep Sleep state, all bits in RTC_ISTAT are cleared. Events that occur during Deep Sleep are not recorded in RTC_ISTAT. The internal V_{dd} RTC control logic generates a virtual 1 Hz tick within one RTXI period (30.52 µs) after SCLK restarts. This loads all shadow registers with up-to-date values and sets the Seconds Event flag. Other Event flags may also be set. When the system wakes up from Deep Sleep, whether by an RTC event or a hardware reset, all of the RTC events that occurred during that second (and only that second) are reported in RTC_ISTAT.

When the system wakes up from Deep Sleep state, software does not need to W1C the bits in RTC_ISTAT. All W1C bits are already cleared by hardware. The Seconds Event flag is set when the RTC internal V_{dd} logic has completed its restart sequence. Software should wait until the Seconds Event flag is set and then may begin reading or writing any RTC register.

Prescaler Enable

The single active bit of the RTC Prescaler Enable register (RTC_PREN) is written using a synchronization path. Clearing of the bit is synchronized to the 32.768 kHz clock. This faster synchronization allows the module to be put into high-speed mode (bypassing the prescaler) without waiting the full 1 second for the write to complete that would be necessary if the module were already running with the prescaler enabled.

When setting the RTC_PREN bit, the first positive edge of the 1 Hz clock occurs 1 to 2 cycles of the 32.768 kHz clock after the prescaler is enabled. The Write Complete Status/Interrupt works as usual when enabling or disabling the prescale counter. The new RTC clock rate is in effect before the Write Complete Status is set.

Event Flags

The unknown values in the registers at powerup can cause Event flags to set before the correct value is written into each of the registers. By catching the 1 Hz clock edge, the write to RTC_STAT can occur a full second before the write to RTC_ALARM. This would cause an extra second of delay between the validity of RTC_STAT and RTC_ALARM, if the value of the RTC_ALARM out of reset is the same as the value written to RTC_STAT. Wait for the writes to complete on these registers before using the flags and interrupts associated with their values.

The following is a list of flags along with the conditions under which they are valid:

- Seconds (1 Hz) Event flag

  Always set on the positive edge of the 1 Hz clock and after shadow registers have updated after waking from Deep Sleep. This is valid as long as the RTC 1 Hz clock is running. Use this flag or interrupt to validate the other flags.

- Write Complete

  Always valid.

- Write Pending Status

  Always valid.

- Minutes Event flag

  Valid only after the second field in `RTC_STAT` is valid. Use the Write Complete and Write Pending Status flags or interrupts to validate the `RTC_STAT` value before using this flag value or enabling the interrupt.

- Hours Event flag

  Valid only after the minute field in `RTC_STAT` is valid. Use the Write Complete and Write Pending Status flags or interrupts to validate the `RTC_STAT` value before using this flag value or enabling the interrupt.

- 24 Hours Event flag

  Valid only after the hour field in RTC_STAT is valid. Use the Write Complete and Write Pending Status flags or interrupts to validate the RTC_STAT value before using this flag value or enabling the interrupt.

- Stopwatch Event flag

  Valid only after the RTC_SWCNT register is valid. Use the Write Complete and Write Pending Status flags or interrupts to validate the RTC_SWCNT value before using this flag value or enabling the interrupt.

- Alarm Event flag

  Valid only after the RTC_STAT and RTC_ALARM registers are valid. Use the Write Complete and Write Pending Status flags or interrupts to validate the RTC_STAT and RTC_ALARM values before using this flag value or enabling its interrupt.

- Day Alarm Event flag

  Same as Alarm.

Writes posted together at the beginning of the same second take effect together at the next 1 Hz tick. The following sequence is safe and does not result in any spurious interrupts from a previous state.

1. Wait for 1 Hz tick.
2. Write 1s to clear the RTC_ISTAT flags for Alarm, Day Alarm, Stopwatch, and/or per-interval.
3. Write new values for RTC_STAT, RTC_ALARM, and/or RTC_SWCNT.

4. Write new value for RTC_ICTL with Alarm, Day Alarm, Stopwatch, and/or per-interval interrupts enabled.
5. Wait for 1 Hz tick.
6. New values have now taken effect simultaneously.

Interrupts

The RTC can provide interrupts at several programmable intervals, including:

- Per second
- Per minute
- Per hour
- Per day
- On countdown from a programmable value
- Daily at a specific time
- On a specific day and time

The RTC can be programmed to provide an interrupt at the completion of all pending writes to any of the 1 Hz registers (RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, and RTC_PREN). Interrupts can be individually enabled or disabled using the RTC Interrupt Control register (RTC_ICTL). Interrupt status can be determined by reading the RTC Interrupt Status register (RTC_ISTAT).

The RTC interrupt is set whenever an event latched into the RTC_ISTAT register is enabled in the RTC_ICTL register. The pending RTC interrupt is cleared whenever all enabled and set bits in RTC_ISTAT are cleared, or when all bits in RTC_ICTL corresponding to pending events are cleared.

As shown in Figure 16-3, the RTC generates an interrupt request (IRQ) to the processor core for event handling and wakeup from a Sleep state. The RTC generates a separate signal for wakeup from a Deep Sleep or from an internal V_{dd} power-off state. The Deep Sleep wakeup signal is asserted at the 1 Hz tick when any RTC interval event enabled in `RTC_ICTL` occurs. The assertion of the Deep Sleep wakeup signal causes the processor core clock (`CCLK`) and the system clock (`SCLK`) to restart. Any enabled event that asserts the RTC Deep Sleep wakeup signal also causes the RTC IRQ to assert once `SCLK` restarts.

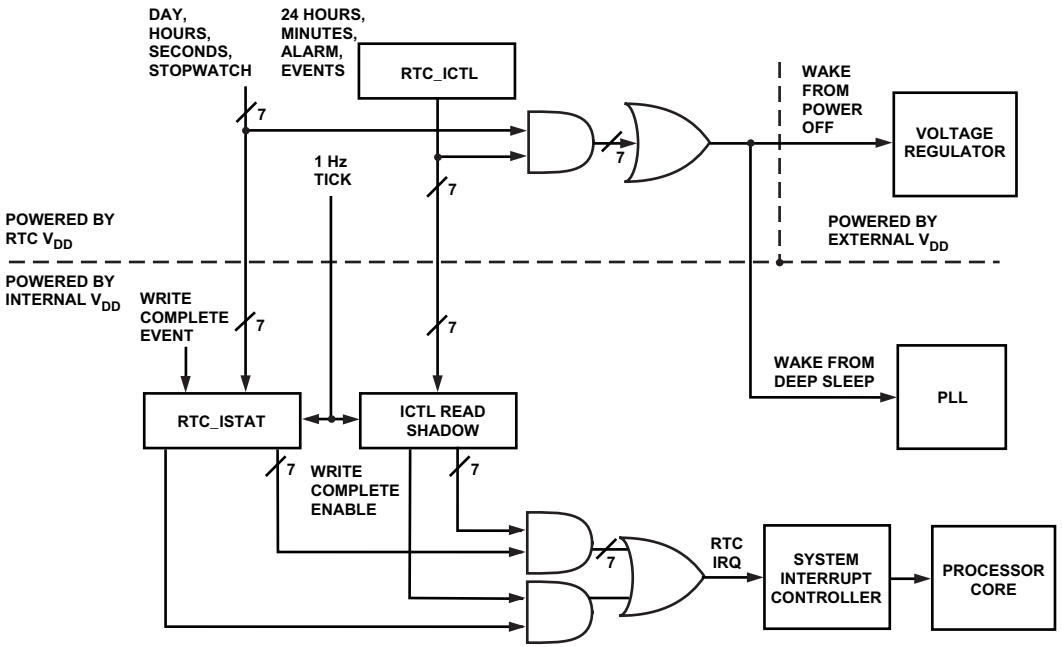


Figure 16-3. RTC Interrupt Structure

RTC_STAT Register

The RTC Status register (RTC_STAT) is used to read or write the current time. Reads return a 32-bit value that always reflects the current state of the days, hours, minutes, and seconds counters. Reads and writes must be 32-bit transactions; attempted 16-bit transactions result in an MMR error. Reads always return a coherent 32-bit value. The hours, minutes, and seconds fields are usually set to match the real time of day. The day counter value is incremented every day at midnight to record how many days have elapsed since it was last modified. Its value does not correspond to a particular calendar day. The 15-bit day counter provides a range of 89 years, 260 or 261 days (depending on leap years) before it overflows.

After the 1 Hz tick, program RTC_STAT with the current time. At the next 1 Hz tick, RTC_STAT takes on the new, incremented value. For example:

1. Wait for 1 Hz tick.
2. Read RTC_STAT, get 10:45:30.
3. Write RTC_STAT to current time, 13:10:59.
4. Read RTC_STAT, still get old time 10:45:30.
5. Wait for 1 Hz tick.
6. Read RTC_STAT, get new current time, 13:11:00.

RTC_ICTL Register

The eight RTC interrupt events can be individually masked or enabled by the RTC Interrupt Control register (RTC_ICTL). The seconds interrupt is generated on each 1 Hz clock tick, if enabled. The minutes interrupt is generated at the 1 Hz clock tick that advances the seconds counter from 59 to 0. The hour interrupt is generated at the 1 Hz clock tick that advances the minute counter from 59 to 0. The 24-hour interrupt occurs

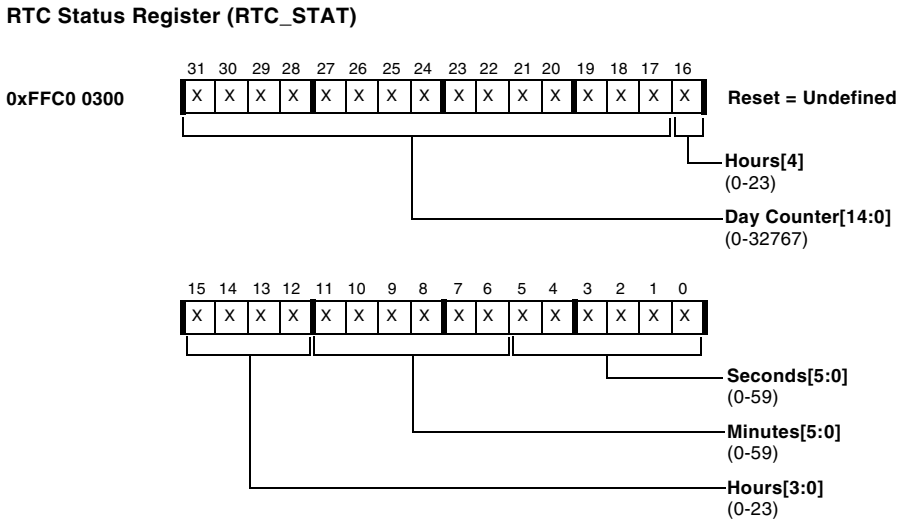


Figure 16-4. RTC Status Register

once per 24-hour period at the 1 Hz clock tick that advances the time to midnight (00:00:00). Any of these interrupts can generate a wakeup request to the processor, if enabled. All implemented bits are read/write.

This register is only partially cleared at reset, so some events may appear to be enabled initially. However, the RTC Interrupt and the RTC Wakeup to the PLL are handled specially and are masked (forced low) until after the first write to the RTC_ICTL register is complete. Therefore, all interrupts act as if they were disabled at system reset (as if all bits of RTC_ICTL were zero), even thought some bits of RTC_ICTL may read as nonzero. If no RTC interrupts are needed immediately after reset, it is recommended to write RTC_ICTL to 0x0000 so that later read-modify-write accesses will function as intended.

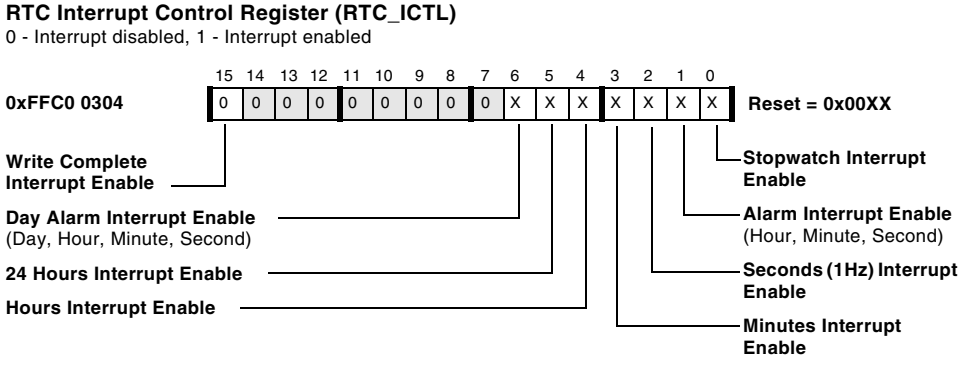


Figure 16-5. RTC Interrupt Control Register

RTC_ISTAT Register

The RTC Interrupt Status register (RTC_ISTAT) provides the status of all RTC interrupts. These bits are sticky. Once set by the corresponding event, each bit remains set until cleared by a software write to this register. Event flags are always set; they are not masked by the interrupt enable bits in RTC_ICTL. Values are cleared by writing a 1 to the respective bit location, except for the Write Pending Status bit, which is read-only. Writes of 0 to any bit of the register have no effect. This register is cleared at reset and during Deep Sleep.

RTC_SWCNT Register

The RTC Stopwatch Count register (RTC_SWCNT) contains the countdown value for the stopwatch. The stopwatch counts down seconds from the programmed value and generates an interrupt (if enabled) when the count reaches 0. The counter stops counting at this point and does not resume counting until a new value is written to RTC_SWCNT. Once running, the counter may be overwritten with a new value. This allows the stopwatch

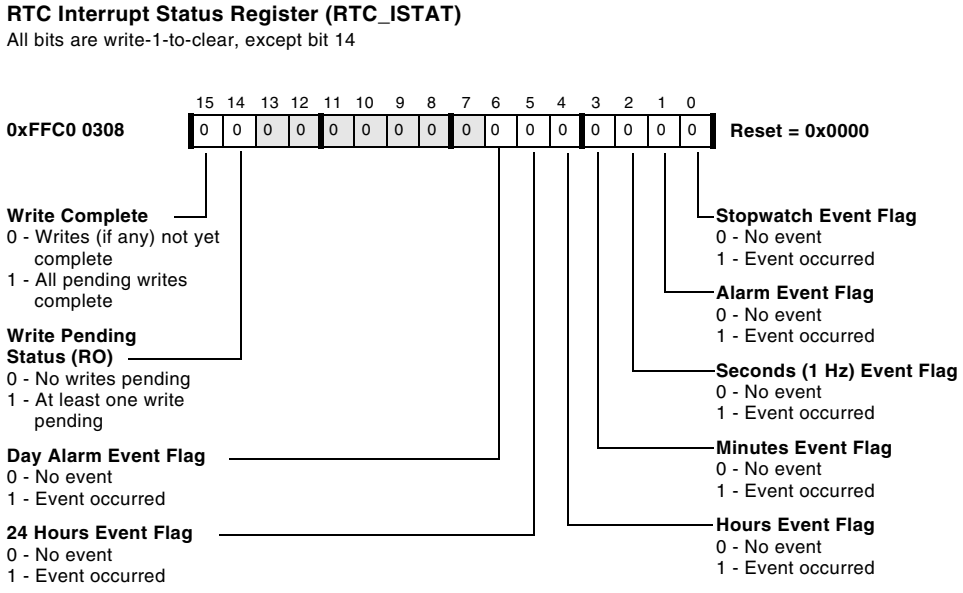


Figure 16-6. RTC Interrupt Status Register

to be used as a watchdog timer with a precision of one second. Writing the running stopwatch to 0 forces it to stop and interrupt early. The Stopwatch Event flag is set at the 1 Hz tick at which any of these occur:

- The stopwatch counter decrements to 0x0000
- A write of 0x0000 to `RTC_SWCNT` completes and the stopwatch was running (current stopwatch count was greater than 0)
- A write of 0x0000 to `RTC_SWCNT` completes and the stopwatch was stopped (current stopwatch count was equal to 0)

The register can be programmed to any value between 0 and (2^{16} – 1) seconds. This is a range of 18 hours, 12 minutes, and 15 seconds.

Typically, software should wait for a 1 Hz tick, then write RTC_SWCNT. One second later, RTC_SWCNT changes to the new value and begins decrementing. Because the register write occupies nearly one second, the time from writing a value of N until the stopwatch interrupt is nearly N + 1 seconds. To produce an exact delay, software can compensate by writing N - 1 to get a delay of nearly N seconds. This implies that you cannot achieve a delay of 1 second with the stopwatch. Writing a value of 1 immediately after a 1 Hz tick results in a stopwatch interrupt nearly two seconds later. To wait one second, software should just wait for the next 1 Hz tick.

The RTC Stopwatch Count register is not reset. After initial powerup, it may be running. When the stopwatch is not used, writing it to 0 to force it to stop saves a small amount of power.

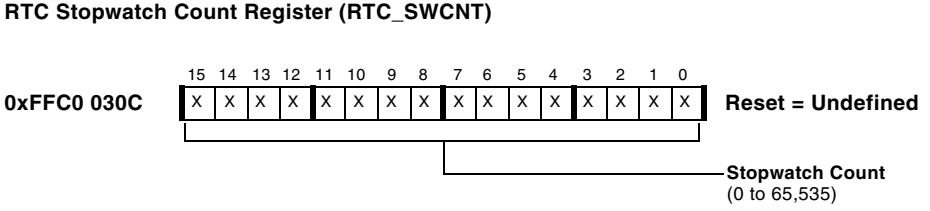


Figure 16-7. RTC Stopwatch Count Register

RTC_ALARM Register

The RTC Alarm register (RTC_ALARM) is programmed by software for the time (in hours, minutes, and seconds) the alarm interrupt occurs. Reads and writes can occur at any time. The alarm interrupt occurs whenever the hour, minute, and second fields first match those of the RTC Status register. The day interrupt occurs whenever the day, hour, minute, and second fields first match those of the RTC Status register.

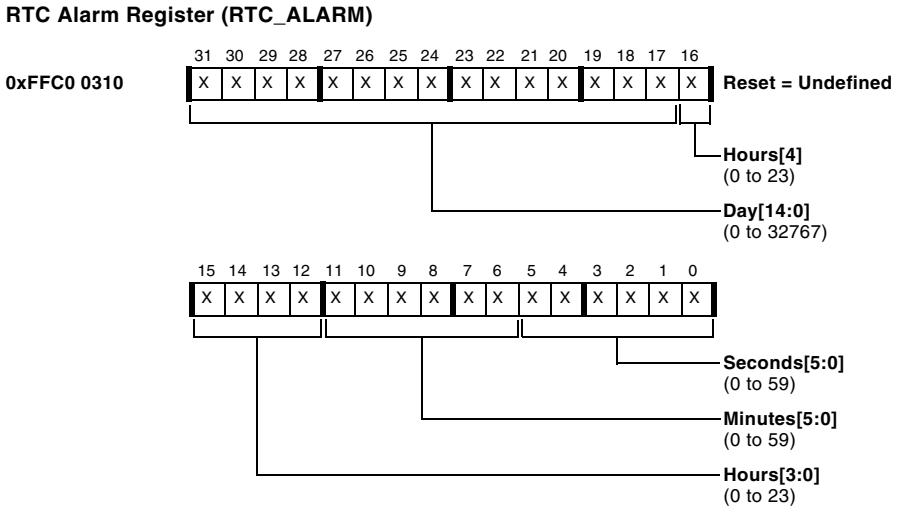


Figure 16-8. RTC Alarm Register

RTC_PREN Register

The RTC Prescaler Enable register (RTC_PREN) has one active bit. When this bit is set, the prescaler is enabled, and the RTC runs at a frequency of 1 Hz. When this bit is cleared, the prescaler is disabled, and the RTC runs at the 32.768 kHz crystal frequency.

In order for the RTC to operate at the proper rate, software must set the Prescaler Enable bit after initial powerup. Write RTC_PREN and then wait for the Write Complete event before programming the other registers. It is safe to write RTC_PREN to 1, once the power is reapplied to the RTC. The first time sets the bit, and subsequent writes will have no effect, as no state is changed.

Do not disable the prescaler by clearing the bit in RTC_PREN without making sure that there are no writes to RTC MMRs in progress. Do not switch between fast and slow mode during normal

operation by setting and clearing this bit, as this disrupts the accurate tracking of real time by the counters. To avoid these potential errors, initialize the RTC during startup via RTC_PREN and do not dynamically alter the state of the prescaler during normal operation.

Running without the prescaler enabled is provided primarily as a test mode. All functionality works, just 32,768 times as fast. Typical software should never program RTC_PREN to 0. The only reason to do so is to synchronize the 1 Hz tick to a more precise external event, as the 1 Hz tick predictably occurs a few RTXI cycles after a 0 → 1 transition of RTC_PREN. Use the following sequence to achieve synchronization to within 100 µs.

1. Write RTC_PREN to 0.
2. Wait for the write to complete.
3. Wait for the external event.
4. Write RTC_PREN to 1.
5. Wait for the write to complete.
6. Reprogram the time into RTC_STAT.

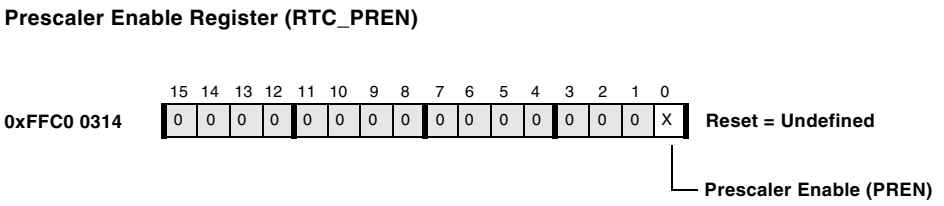


Figure 16-9. Prescaler Enable Register

State Transitions Summary

Table 16-1 shows how each RTC MMR is affected by the system states. The phase locked loop (PLL) states (Reset, Full On, Active, Sleep, and Deep Sleep) are defined in Chapter 8, "Dynamic Power Management." "No Power" means none of the processor power supply pins are connected to a source of energy. "Off" means the processor core, peripherals, and memory are not powered (Internal V_{dd} is off), while the RTC is still powered and running. External V_{dd} may still be powered. Registers described as "As written" are holding the last value software wrote to the register. If the register has not been written since RTC V_{dd} power was applied, then the state is unknown (for all bits of `RTC_STAT`, `RTC_ALARM`, and `RTC_SWCNT`, and for some bits of `RTC_ISTAT`, `RTC_PREN`, and `RTC_ICTL`).

Table 16-1. Effect of States on RTC MMRs

RTC V_{dd}	IV_{dd}	System State	RTC_ICTL	RTC_ISTAT	RTC_STAT RTC_SWCNT	RTC_ALARM RTC_PREN
Off	Off	No Power	X	X	X	X
On	On	Reset	As written	0	Counting	As written
On	On	Full On	As written	Events	Counting	As written
On	On	Sleep	As written	Events	Counting	As written
On	On	Active	As written	Events	Counting	As written
On	On	Deep Sleep	As written	0	Counting	As written
On	Off	Off	As written	X	Counting	As written

Table 16-2 summarizes software's responsibilities with respect to the RTC at various system state transition events.

Table 16-2. RTC System State Transition Events

At This Event:	Execute This Sequence:
Power On from No Power	Write RTC_PREN = 1. Wait for Write Complete. Write RTC_STAT to current time. Write RTC_ALARM, if needed. Write RTC_SWCNT. Write RTC_ISTAT to clear any pending RTC events. Write RTC_ICTL to enable any desired RTC interrupts or to disable all RTC interrupts.
Full On after Reset or Full On after Power On from Off	Wait for Seconds Event, or write RTC_PREN = 1 and wait for Write Complete. Write RTC_ISTAT to clear any pending RTC events. Write RTC_ICTL to enable any desired RTC interrupts or to disable all RTC interrupts. Read RTC MMRs as required.
Wake from Deep Sleep	Wait for Seconds Event flag to set. Write RTC_ISTAT to acknowledge RTC Deep Sleep wakeup. Read RTC MMRs as required. The PLL state is now Active. Transition to Full On as needed.
Wake from Sleep	If wakeup came from RTC, Seconds Event flag will be set. In this case, write RTC_ISTAT to acknowledge RTC wakeup IRQ. Always, read RTC MMRs as required.
Before Going to Sleep	If wakeup by RTC is desired: Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable the desired RTC interrupt sources for wakeup. Wait for Write Complete. Enable RTC for wakeup in the System Interrupt Wakeup-Enable register (SIC_IWR).

Table 16-2. RTC System State Transition Events (Cont'd)

At This Event:	Execute This Sequence:
Before Going to Deep Sleep	Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable the desired RTC event sources for Deep Sleep wakeup. Wait for Write Complete.
Before Going to Off	Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable any desired RTC event sources for powerup wakeup. Wait for Write Complete. Set the Wake bit in the Voltage Regulator Control register (VR_CTL).

17 EXTERNAL BUS INTERFACE UNIT

The External Bus Interface Unit (EBIU) provides glueless interfaces to external memories. The processor supports synchronous DRAM (SDRAM) and is compliant with the PC100 and PC133 SDRAM standards. The EBIU also supports asynchronous interfaces such as SRAM, ROM, FIFOs, flash memory, and ASIC/FPGA designs.

Overview

The EBIU services requests for external memory from the core or from a DMA channel. The priority of the requests is determined by the External Bus Controller. The address of the request determines whether the request is serviced by the EBIU SDRAM Controller or the EBIU Asynchronous Memory Controller.

The EBIU is clocked by the system clock (`SCLK`). All synchronous memories interfaced to the processor operate at the `SCLK` frequency. The ratio between core frequency and `SCLK` frequency is programmable using a phase-locked loop (PLL) system memory-mapped register (MMR). For more information, see “Core Clock/System Clock Ratio Control” on page 8-5.

The external memory space is shown in Figure 17-1. One memory region is dedicated to SDRAM support. SDRAM interface timing and the size of the SDRAM region are programmable. The SDRAM memory space can range in size from 16 to 128M byte.

For information on how to connect to SDRAMs smaller than 16M byte, please see "Using SDRAMs Smaller Than 16M Byte" on page 18-8.

The start address of the SDRAM memory space is 0x0000 0000. The area from the end of the SDRAM memory space up to address 0x2000 0000 is reserved.

The next four regions are dedicated to supporting asynchronous memories. Each asynchronous memory region can be independently programmed to support different memory device characteristics. Each region has its own memory select output pin from the EBIU.

The next region is reserved memory space. References to this region do not generate external bus transactions. Writes have no effect on external memory values, and reads return undefined values. The EBIU generates an error response on the internal bus, which will generate a hardware exception for a core access or will optionally generate an interrupt from a DMA channel.

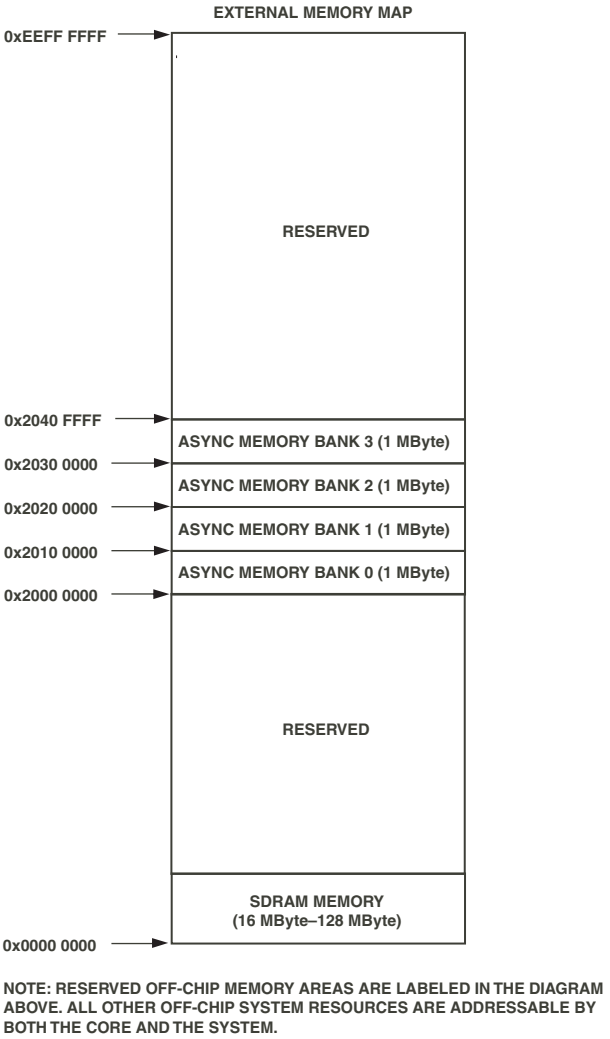


Figure 17-1. External Memory Map

Block Diagram

Figure 17-2 is a conceptual block diagram of the EBIU and its interfaces. Signal names shown with an overbar are active low signals.

Since only one external memory device can be accessed at a time, control, address, and data pins for each memory type are multiplexed together at the pins of the device. The Asynchronous Memory Controller (AMC) and the SDRAM Controller (SDC) effectively arbitrate for the shared pin resources.

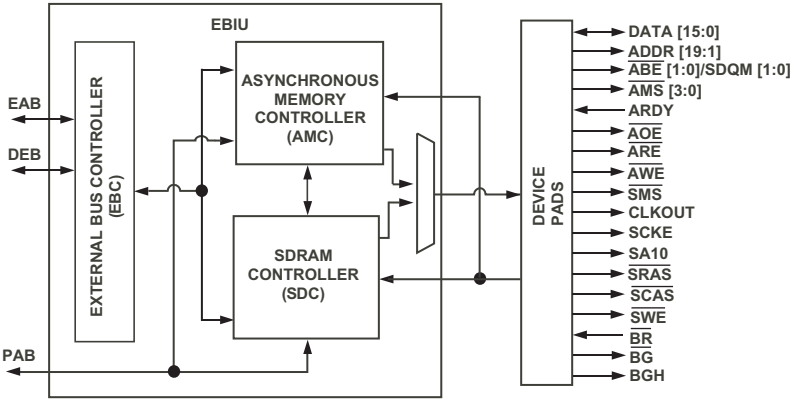


Figure 17-2. External Bus Interface Unit (EBIU)

Internal Memory Interfaces

The EBIU functions as a slave on three buses internal to the processor:

- External Access Bus (EAB), mastered by the core memory management unit on behalf of external bus requests from the core
- DMA External Bus (DEB), mastered by the DMA controller on behalf of external bus requests from any DMA channel
- Peripheral Access Bus (PAB), mastered by the core on behalf of system MMR requests from the core

These are synchronous interfaces, clocked by SCLK, as are the EBIU and Pads registers. The EAB provides access to both asynchronous external memory and synchronous DRAM external memory. The external access is controlled by either the Asynchronous Memory Controller (AMC) or the SDRAM Controller (SDC), depending on the internal address used to access the EBIU. Since the AMC and SDC share the same interface to the external pins, access is sequential and must be arbitrated based on requests from the EAB.

The third bus (PAB) is used only to access the memory-mapped control and status registers of the EBIU. The PAB connects separately to the AMC and SDC; it does not need to arbitrate with or take access cycles from the EAB bus.

The External Bus Controller (EBC) logic must arbitrate access requests for external memory coming from the EAB and DEB buses. The EBC logic routes read and write requests to the appropriate memory controller based on the bus selects. The AMC and SDC compete for access to the shared resources in the Pads logic. This competition is resolved in a pipelined fashion, in the order dictated by the EBC arbiter. Transactions from the core have priority over DMA accesses in most circumstances. However, if the DMA controller detects an excessive backup of transactions, it can request its priority to be temporarily raised above the core.

External Memory Interfaces

Both the AMC and the SDC share the external interface address and data pins, as well as some of the control signals. These pins are shared:

- ADDR[19:1], address bus
- DATA[15:0], data bus
- $\overline{\text{ABE[1:0]}}$/SDQM[1:0], AMC byte enables/SDC data masks
- $\overline{\text{BR}}$, $\overline{\text{BG}}$, $\overline{\text{BGH}}$, external bus access control signals

No other signals are multiplexed between the two controllers.

The following tables describe the signals associated with each interface.

Table 17-1. Asynchronous Memory Interface Signals

Pad	Pin Type [1]	Description
DATA[15:0]	I/O	External Data Bus
ADDR[19:1]	O	External Address Bus
$\overline{\text{AMS[3:0]}}$	O	Asynchronous Memory Selects
$\overline{\text{AWE}}$	O	Asynchronous Memory Write Enable
$\overline{\text{ARE}}$	O	Asynchronous Memory Read Enable
$\overline{\text{AOE}}$	O	Asynchronous Memory Output Enable In most cases, the $\overline{\text{AOE}}$ pin should be connected to the $\overline{\text{OE}}$ pin of an external memory-mapped asynchronous device. Please refer to the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* for specific timing information between the $\overline{\text{AOE}}$ and $\overline{\text{ARE}}$ signals to determine which interface signal should be used in your system.

Table 17-1. Asynchronous Memory Interface Signals (Cont'd)

Pad	Pin Type [1]	Description
ARDY	I	Asynchronous Memory Ready Response Note this is a synchronous input
$\overline{\text{ABE[1:0]}}$/SDQM[1:0]	O	Byte Enables

1 Pin Types: I = Input, O = Output

Table 17-2. SDRAM Interface Signals

Pad	Pin Type [1]	Description
DATA[15:0]	I/O	External Data Bus
ADDR[19:18], ADDR[16:1]	O	External Address Bus Connect to SDRAM Address pins. Bank address is output on ADDR[19:18] and should be connected to SDRAM BA[1:0] pins.
$\overline{\text{SRAS}}$	O	SDRAM Row Address Strobe pin Connect to SDRAM's $\overline{\text{RAS}}$ pin.
$\overline{\text{SCAS}}$	O	SDRAM Column Address Strobe pin Connect to SDRAM's $\overline{\text{CAS}}$ pin.
$\overline{\text{SWE}}$	O	SDRAM Write Enable pin Connect to SDRAM's $\overline{\text{WE}}$ pin.
$\overline{\text{ABE[1:0]}}$/ SDQM[1:0]	O	SDRAM Data Mask pins Connect to SDRAM's DQM pins.
$\overline{\text{SMS}}$	O	Memory Select pin of external memory bank configured for SDRAM Connect to SDRAM's $\overline{\text{CS}}$ (Chip Select) pin. Active Low.
SA10	O	SDRAM A10 pin SDRAM interface uses this pin to be able to do refreshes while the AMC is using the bus. Connect to SDRAM's A[10] pin.

Table 17-2. SDRAM Interface Signals (Cont'd)

Pad	Pin Type [1]	Description
SCKE	O	SDRAM Clock Enable pin Connect to SDRAM's CKE pin.
CLKOUT	O	SDRAM Clock Output pin Switches at system clock frequency. Connect to the SDRAM's CLK pin.

1 Pin Types: I = Input, O = Output

EBIU Programming Model

This section describes the programming model of the EBIU. This model is based on system memory-mapped registers used to program the EBIU.

There are six control registers and one status register in the EBIU. They are:

- Asynchronous Memory Global Control register (`EBIU_AMGCTL`)
- Asynchronous Memory Bank Control 0 register (`EBIU_AMBCTL0`)
- Asynchronous Memory Bank Control 1 register (`EBIU_AMBCTL1`)
- SDRAM Memory Global Control register (`EBIU_SDGCTL`)
- SDRAM Memory Bank Control register (`EBIU_SDBCTL`)
- SDRAM Refresh Rate Control register (`EBIU_SDRRC`)
- SDRAM Control Status register (`EBIU_SDSTAT`)

Each of these registers is described in detail in the AMC and SDC sections later in this chapter.

Error Detection

The EBIU responds to any bus operation which addresses the range of 0x0000 0000 – 0xEEFF FFFF, even if that bus operation addresses reserved or disabled memory or functions. It responds by completing the bus operation (asserting the appropriate number of acknowledges as specified by the bus master) and by asserting the bus error signal for these error conditions:

- Any access to reserved off-chip memory space
- Any access to a disabled external memory bank
- Any access to an unpopulated area of an SDRAM memory bank

If the core requested the faulting bus operation, the bus error response from the EBIU is gated into the HWE interrupt internal to the core (this interrupt can be masked off in the core). If a DMA master requested the faulting bus operation, then the bus error is captured in that controller and can optionally generate an interrupt to the core.

Asynchronous Memory Interface

The asynchronous memory interface allows a glueless interface to a variety of memory and peripheral types. These include SRAM, ROM, EPROM, flash memory, and FPGA/ASIC designs. Four asynchronous memory regions are supported. Each has a unique memory select associated with it, shown in Table 17-3.

Table 17-3. Asynchronous Memory Bank Address Range

Memory Bank Select	Address Start	Address End
$\overline{\text{AMS[3]}}$	2030 0000	203F FFFF
$\overline{\text{AMS[2]}}$	2020 0000	202F FFFF
$\overline{\text{AMS[1]}}$	2010 0000	201F FFFF
$\overline{\text{AMS[0]}}$	2000 0000	200F FFFF

Asynchronous Memory Address Decode

The address range allocated to each asynchronous memory bank is fixed at 1M byte; however, not all of an enabled memory bank need be populated. Unlike the SDRAM memory, which may need to support very large memory structures spanning multiple memory banks, it should be relatively easy to constrain code and data structures to fit within one of the supported asynchronous memory banks, because of the nature of the types of code or data that is stored here.

Note accesses to unpopulated memory of partially populated AMC banks do not result in a bus error and will alias to valid AMC addresses.

The asynchronous memory signals are defined in Table 17-1. The timing of these pins is programmable to allow a flexible interface to devices of different speeds. For example interfaces, see Chapter 18, “System Design.”

EBIU_AMGCTL Register

The Asynchronous Memory Global Control register (EBIU_AMGCTL) configures global aspects of the controller. It contains bank enables and other information as described in this section. This register should not be programmed while the AMC is in use. The EBIU_AMGCTL register should be the last control register written to when configuring the processor to access external memory-mapped asynchronous devices.

Asynchronous Memory Global Control Register (EBIU_AMGCTL)

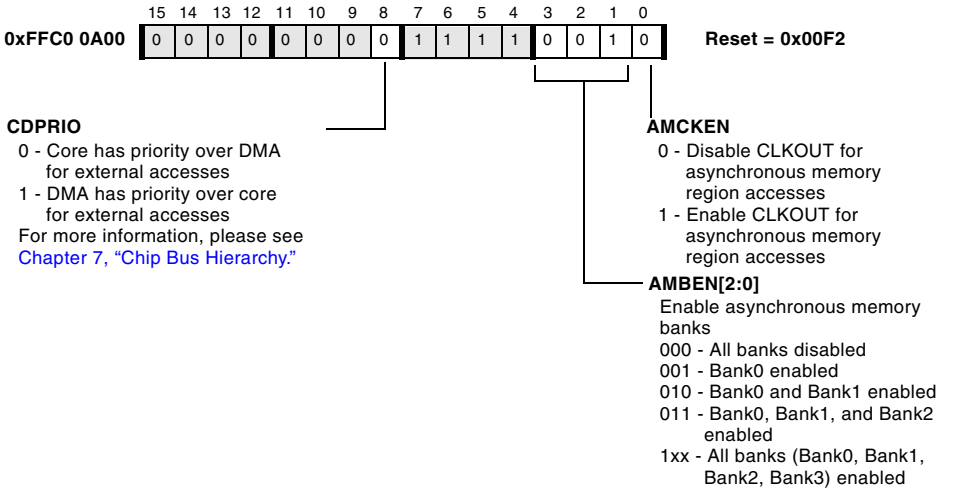


Figure 17-3. Asynchronous Memory Global Control Register

If a bus operation accesses a disabled asynchronous memory bank, the EBIU responds by acknowledging the transfer and asserting the error signal on the requesting bus. The error signal propagates back to the requesting bus master. This generates a hardware exception to the core, if it is the requester. For DMA mastered requests, the error is captured in the respective status register. If a bank is not fully populated with memory, then the memory likely aliases into multiple address regions within the bank. This aliasing condition is not detected by the EBIU, and no error response is asserted.

For external devices that need a clock, CLKOUT can be enabled by setting the AMCKEN bit in the EBIU_AMGCTL register. In systems that do not use CLKOUT, set the AMCKEN bit to 0.

EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers

The EBIU asynchronous memory controller has two Asynchronous Memory Bank Control registers (EBIU_AMBCTL0 and EBIU_AMBCTL1). They contain bits for counters for setup, strobe, and hold time; bits to determine memory type and size; and bits to configure use of ARDY. These registers should not be programmed while the AMC is in use.

The timing characteristics of the AMC can be programmed using these four parameters:

- Setup: the time between the beginning of a memory cycle ($\overline{\text{AMS[x]}}$ low) and the read-enable assertion ($\overline{\text{ARE}}$ low) or write-enable assertion ($\overline{\text{AWE}}$ low)
- Read Access: the time between read-enable assertion ($\overline{\text{ARE}}$ low) and deassertion ($\overline{\text{ARE}}$ high)
- Write Access: the time between write-enable assertion ($\overline{\text{AWE}}$ low) and deassertion ($\overline{\text{AWE}}$ high)
- Hold: the time between read-enable deassertion ($\overline{\text{ARE}}$ high) or write-enable deassertion ($\overline{\text{AWE}}$ high) and the end of the memory cycle ($\overline{\text{AMS[x]}}$ high)

Each of these parameters can be programmed in terms of EBIU clock cycles. In addition, there are minimum values for these parameters:

- Setup ≥ 1 cycle
- Read Access ≥ 1 cycle
- Write Access ≥ 1 cycle
- Hold ≥ 0 cycles

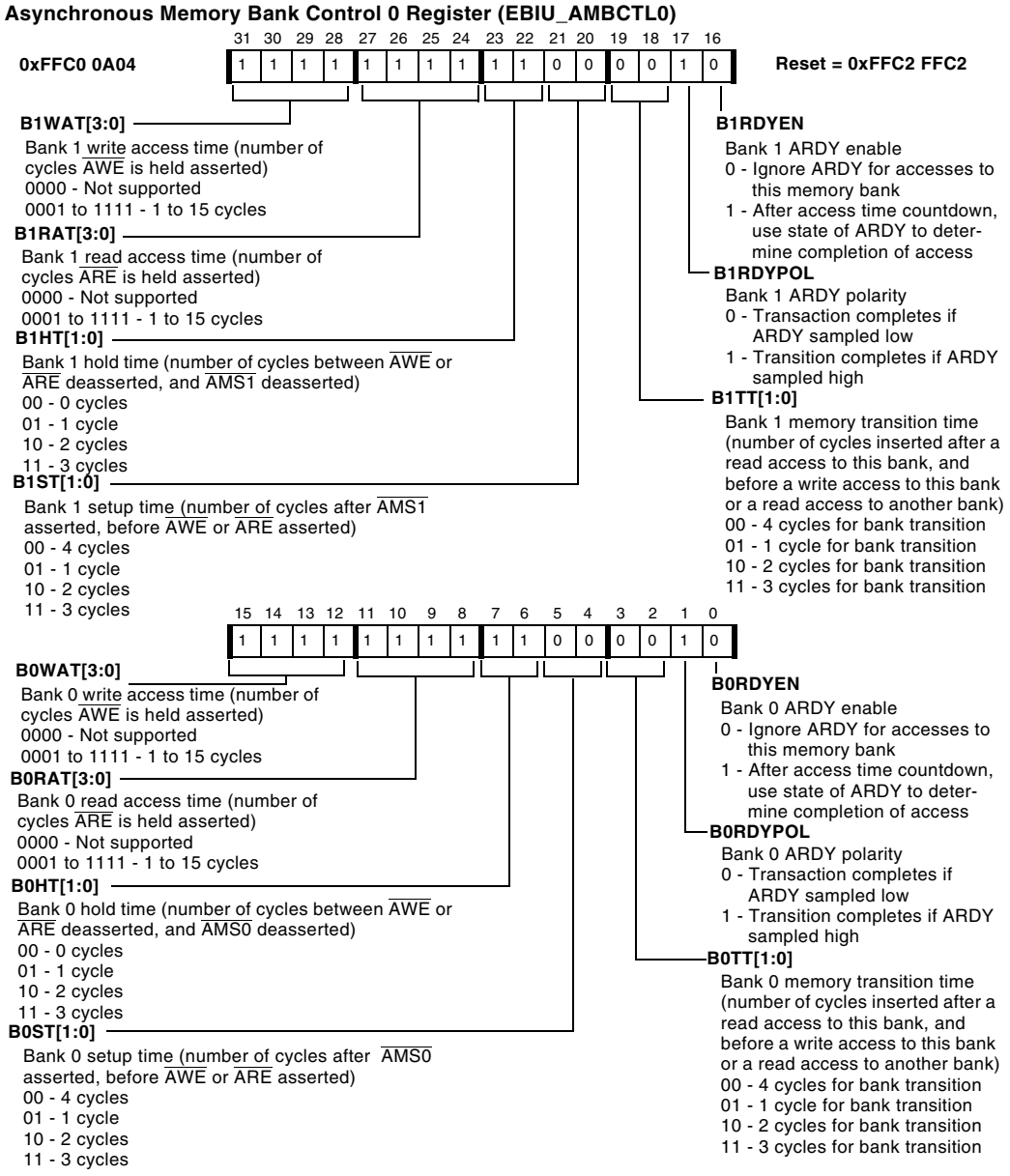


Figure 17-4. Asynchronous Memory Bank Control 0 Register

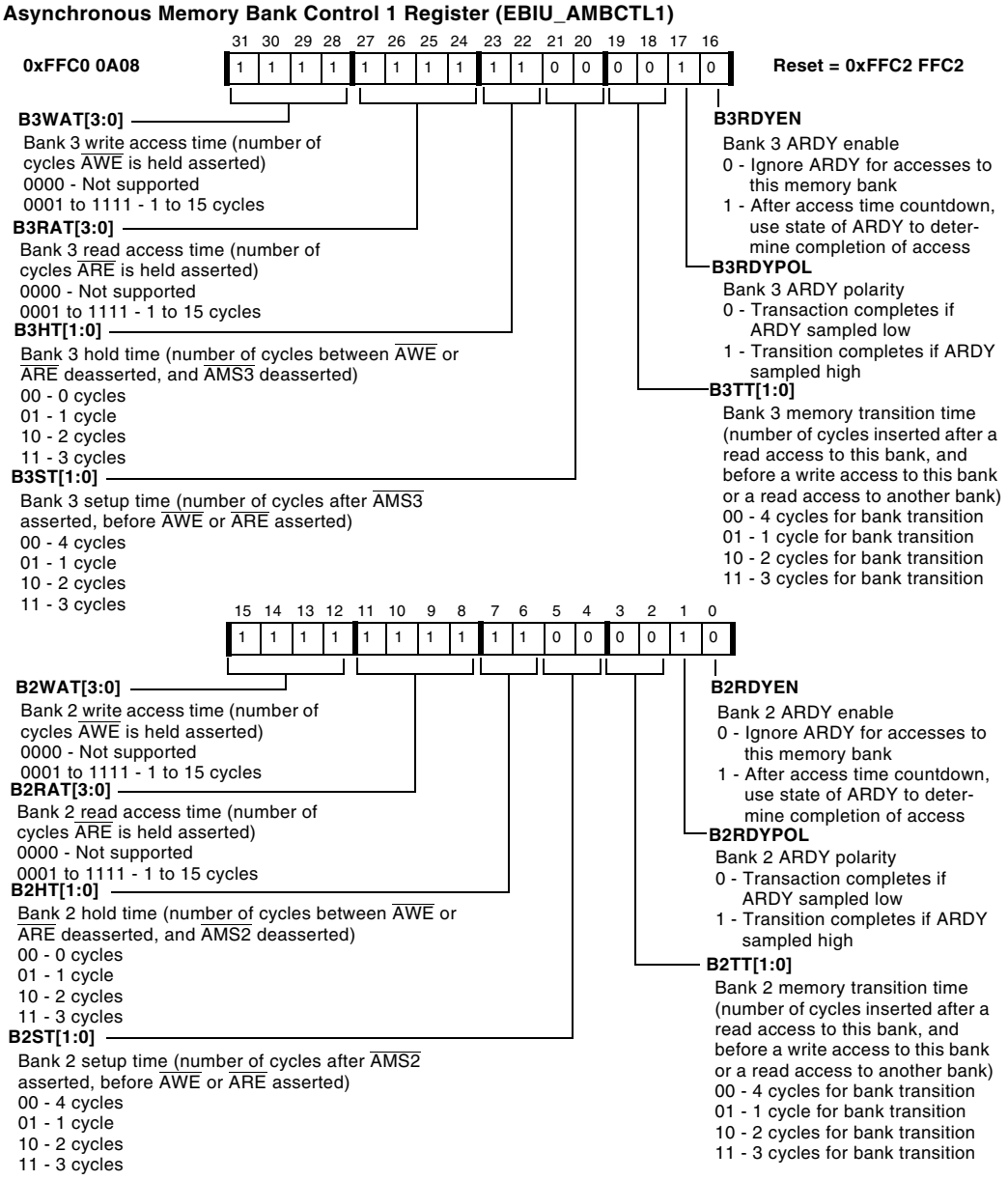


Figure 17-5. Asynchronous Memory Bank Control 1 Register

Avoiding Bus Contention

Because the three-stated data bus is shared by multiple devices in a system, be careful to avoid contention. Contention causes excessive power dissipation and can lead to device failure. Contention occurs during the time one device is getting off the bus and another is getting on. If the first device is slow to three-state and the second device is quick to drive, the devices contend.

There are two cases where contention can occur. The first case is a read followed by a write to the same memory space. In this case, the data bus drivers can potentially contend with those of the memory device addressed by the read. The second case is back-to-back reads from two different memory spaces. In this case, the two memory devices addressed by the two reads could potentially contend at the transition between the two read operations.

To avoid contention, program the turnaround time (Bank Transition Time) appropriately in the Asynchronous Memory Bank Control registers. This feature allows software to set the number of clock cycles between these types of accesses on a bank-by-bank basis. Minimally, the EBIU provides one cycle for the transition to occur.

ARDY Input Control

Each bank can be programmed to sample the ARDY input after the read or write access timer has counted down or to ignore this input signal. If enabled and disabled at the sample window, ARDY can be used to extend the access time as required. Note ARDY is synchronously sampled, therefore:

- Assertion and deassertion of ARDY to the processor must meet the data sheet setup and hold times. Failure to meet these synchronous specifications could result in meta-stable behavior internally. The processor's CLKOUT signal should be used to ensure synchronous transitions of ARDY.

- The ARDY pin must be stable (either asserted or deasserted) at the external interface on the cycle before the internal bank counter reaches 0; that is, more than one CLKOUT cycle before the scheduled rising edge of $\overline{\text{AWE}}$ or $\overline{\text{ARE}}$. This will determine whether the access is extended or not.
- Once the transaction has been extended by the assertion of ARDY, the transaction completes in the cycle after ARDY is sampled as asserted.

The polarity of ARDY is programmable on a per-bank basis. Since ARDY is not sampled until an access is in progress to a bank in which the ARDY enable is asserted, ARDY does not need to be driven by default. For more information, see "Adding Additional Wait States" on page 17-20.

Programmable Timing Characteristics

This section describes the programmable timing characteristics for the EBIU. Timing relationships depend on the programming of the AMC, whether initiation is from the core or from Memory DMA (MemDMA), and the sequence of transactions (read followed by read, read followed by write, and so on).

Asynchronous Accesses by Core Instructions

Some external memory accesses are caused by core instructions of the type:

```
R0.L = W[P0++] ; /* Read from external memory, where P0 points
to a location in external memory */
```

or:

```
W[P0++] = R0.L ; /* Write to external memory */
```

Asynchronous Reads

Figure 17-6 shows an asynchronous read bus cycle with timing programmed as setup = 2 cycles, read access = 2 cycles, hold = 1 cycle, and transition time = 1 cycle.

Asynchronous read bus cycles proceed as follows.

1. At the start of the setup period, $\overline{\texttt{AMS[x]}}$, the address bus, and $\overline{\texttt{ABE[1:0]}}$ become valid, and $\overline{\texttt{AOE}}$ asserts.
2. At the beginning of the read access period and after the 2 setup cycles, $\overline{\texttt{ARE}}$ asserts.
3. At the beginning of the hold period, read data is sampled on the rising edge of the EBIU clock. The $\overline{\texttt{ARE}}$ pin deasserts after this rising edge.
4. At the end of the hold period, $\overline{\texttt{AOE}}$ deasserts unless this bus cycle is followed by another asynchronous read to the same memory space. Also, $\overline{\texttt{AMS[x]}}$ deasserts unless the next cycle is to the same memory bank.
5. Unless another read of the same memory bank is queued internally, the AMC appends the programmed number of memory transition time cycles.

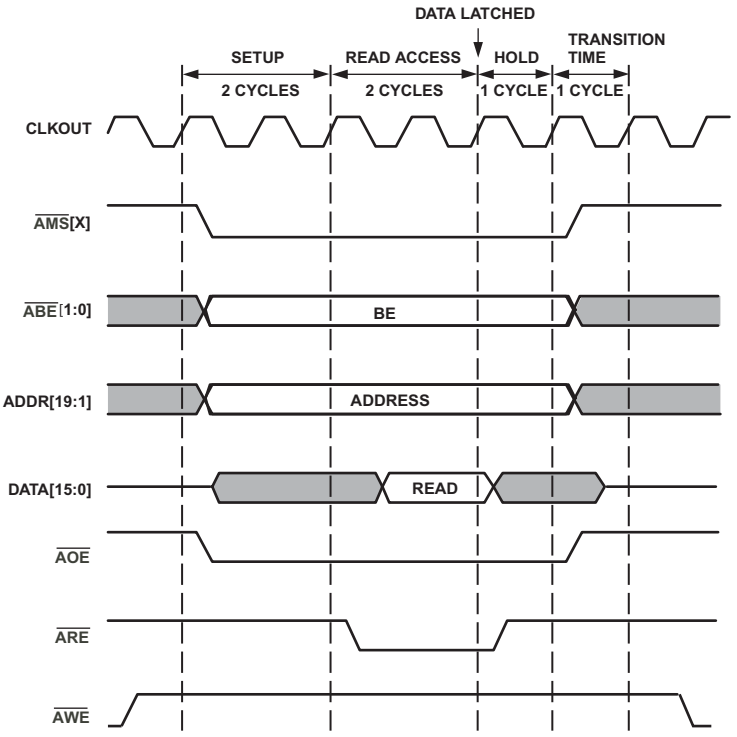


Figure 17-6. Asynchronous Read Bus Cycles

Read access is completed with the AMSx and AOE signals getting de-asserted. There are a few idle cycles before the next read operation starts. The number of idle cycles is a function of the CCLK/SCLK ratio. The number of idle cycles is 6 for a CCLK/SCLK ratio of 3, 4 for a CCLK/SCLK ratio of 5, and 3 for a CCLK/SCLK ratio of 10.

Asynchronous Writes

Figure 17-7 shows an asynchronous write bus cycle followed by an asynchronous read cycle to the same bank, with timing programmed as setup = 2 cycles, write access = 2 cycles, read access = 3 cycles, hold = 1 cycle, and transition time = 1 cycle.

Asynchronous write bus cycles proceed as follows.

1. At the start of the setup period, $\overline{\texttt{AMS[x]}}$, the address bus, data buses, and $\overline{\texttt{ABE[1:0]}}$ become valid.
2. At the beginning of the write access period, $\overline{\texttt{AWE}}$ asserts.
3. At the beginning of the hold period, $\overline{\texttt{AWE}}$ deasserts.

Asynchronous read bus cycles proceed as follows.

1. At the start of the setup period, $\overline{\texttt{AMS[x]}}$, the address bus, and $\overline{\texttt{ABE[1:0]}}$ become valid, and $\overline{\texttt{AOE}}$ asserts.
2. At the beginning of the read access period, $\overline{\texttt{ARE}}$ asserts.
3. At the beginning of the hold period, read data is sampled on the rising edge of the EBIU clock. The $\overline{\texttt{ARE}}$ signal deasserts after this rising edge.
4. At the end of the hold period, $\overline{\texttt{AOE}}$ deasserts unless this bus cycle is followed by another asynchronous read to the same memory space. Also, $\overline{\texttt{AMS[x]}}$ deasserts unless the next cycle is to the same memory bank.
5. Unless another read of the same memory bank is queued internally, the AMC appends the programmed number of memory transition time cycles.

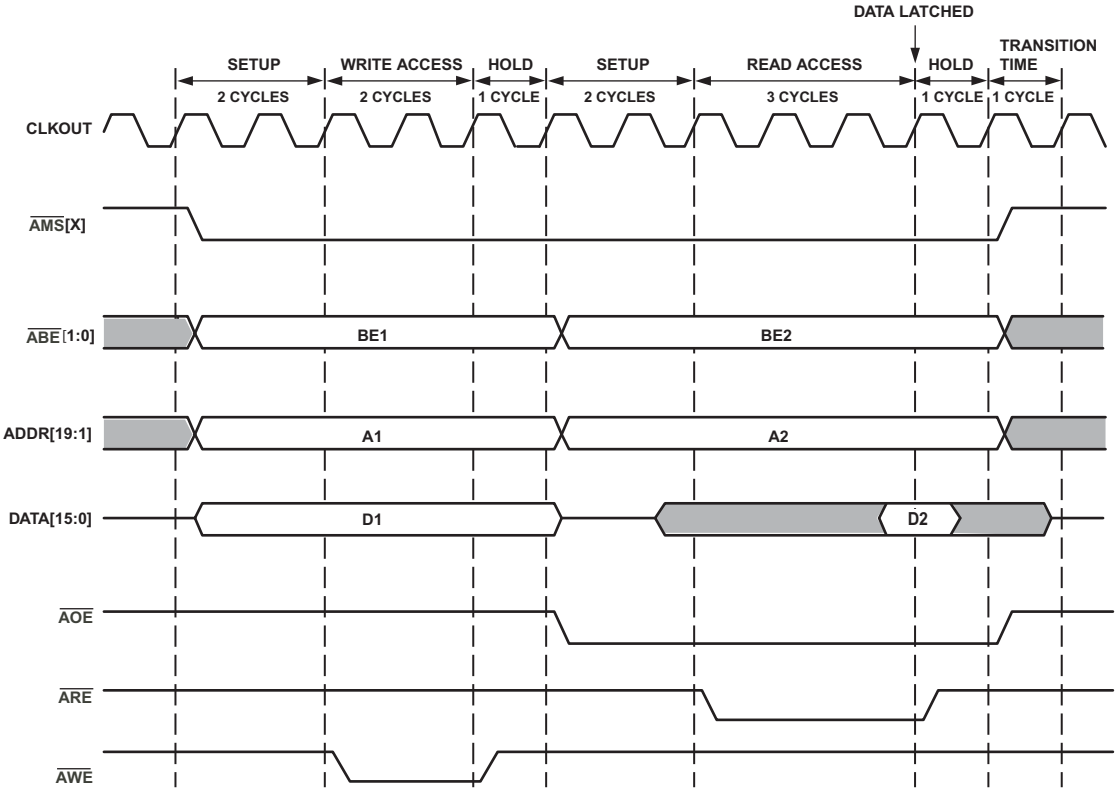


Figure 17-7. Asynchronous Write and Read Bus Cycles

Adding Additional Wait States

The ARDY pin is used to insert extra wait states. The input is sampled synchronously with the EBIU internal clock. The EBIU starts sampling ARDY on the clock cycle before the end of the programmed strobe period. If ARDY is sampled as deasserted, the access period is extended. The ARDY pin is then sampled on each subsequent clock edge. Read data is latched on the clock edge after ARDY is sampled as asserted. The read- or write-enable remains asserted for one clock cycle after ARDY is sampled as asserted. An

example of this behavior is shown in Figure 17-8, where setup = 2 cycles, read access = 4 cycles, and hold = 1 cycle. Note the read access period must be programmed to a minimum of two cycles to make use of the ARDY input.

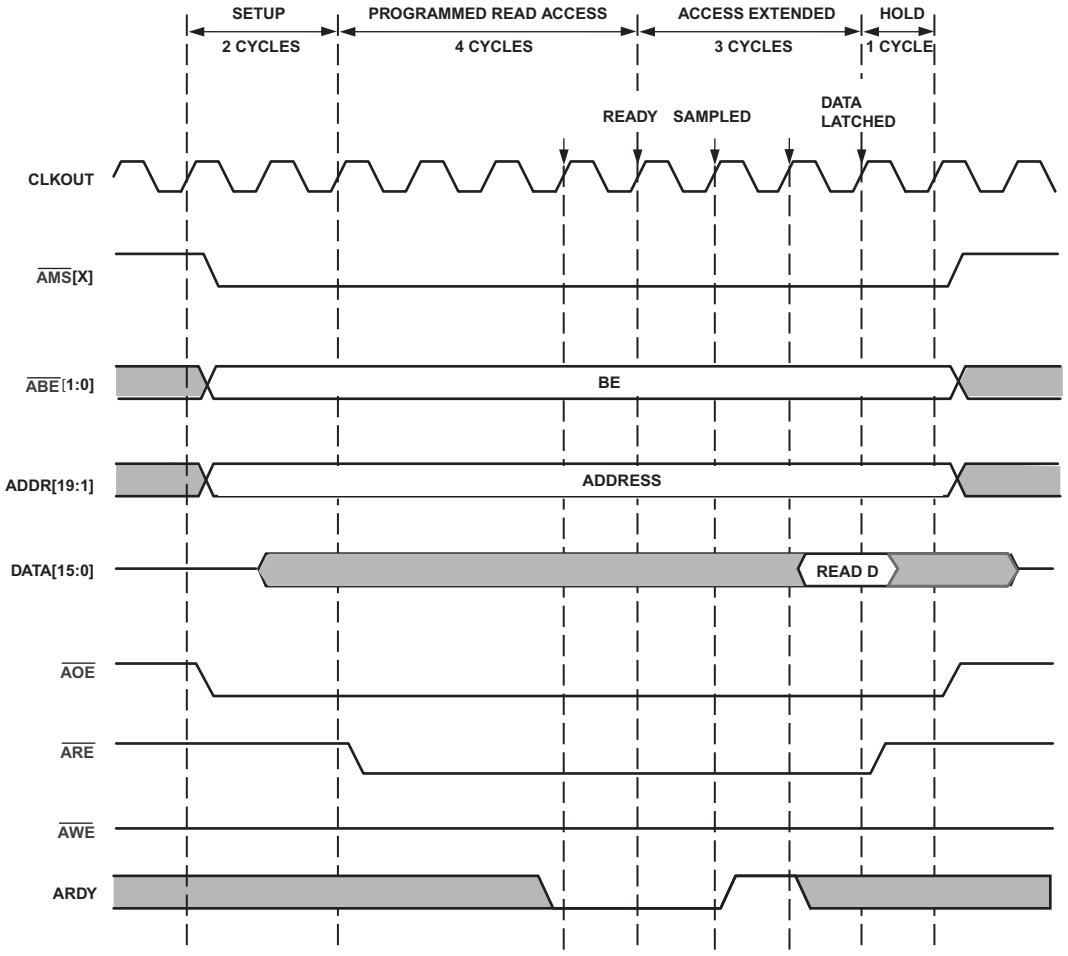


Figure 17-8. Inserting Wait States Using ARDY

Byte Enables

The $\overline{\texttt{ABE[1:0]}}$ pins are both low during all asynchronous reads and 16-bit asynchronous writes. When an asynchronous write is made to the upper byte of a 16-bit memory, $\overline{\texttt{ABE1}}$ = 0 and $\overline{\texttt{ABE0}}$ = 1. When an asynchronous write is made to the lower byte of a 16-bit memory, $\overline{\texttt{ABE1}}$ = 1 and $\overline{\texttt{ABE0}}$ = 0.

SDRAM Controller (SDC)

The SDRAM Controller (SDC) enables the processor to transfer data to and from Synchronous DRAM (SDRAM) with a maximum frequency specified in the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet*. The processor supports a glueless interface with one external bank of standard SDRAMs of 64 Mbit to 512 Mbit, with configurations x4, x8, and x16, up to a maximum total capacity of 128M bytes of SDRAM. This bank is controlled by the $\overline{\texttt{SMS}}$ Memory Select pin. The interface includes timing options to support additional buffers between the processor and SDRAM, to handle the capacitive loads of large memory arrays.

All inputs are sampled and all outputs are valid on the rising edge of the SDRAM clock output CLKOUT.

The EBIU SDC provides a glueless interface with standard SDRAMs. The SDRAM controller:

- Supports SDRAMs of 64M bit, 128M bit, 256M bit, and 512M bit with configurations of x4, x8, and x16
- Supports up to 128M byte of SDRAM in external SDRAM
- Supports SDRAM page sizes of 512 bytes, 1K byte, 2K byte, and 4K byte
- Supports four internal banks within the SDRAM

- Uses a programmable refresh counter to coordinate between varying clock frequencies and the SDRAM's required refresh rate
- Provides multiple timing options to support additional buffers between the processor and SDRAM
- Uses a separate pin (`SA10`) that enables the SDC to precharge SDRAM before issuing an Auto-Refresh or Self-Refresh command while the asynchronous memory controller has control of the EBIU port
- Supports self-refresh for standard SDRAMs and partial array self-refresh for mobile SDRAMs
- Provides two SDRAM powerup options
- Supports interleaved SDRAM bank accesses

Definition of Terms

The following are definitions used in the remainder of this chapter.

Bank Activate Command

The Bank Activate command causes the SDRAM to open an internal bank (specified by the bank address) in a row (specified by the row address). When the Bank Activate command is issued to the SDRAM, the SDRAM opens a new row address in the dedicated bank. The memory in the open internal bank and row is referred to as the open page. The Bank Activate command must be applied before a read or write command.

Burst Length

The burst length determines the number of words that the SDRAM device stores or delivers after detecting a single write or read command, respectively. The burst length is selected by writing certain bits in the SDRAM's Mode register during the SDRAM powerup sequence.

Although the SDC supports only Burst Length = 1 mode, during a burst to SDRAM, the SDC applies the read or write command every cycle and keeps accessing the data. Therefore, the effective burst length is much greater than 1. In other words, setting Burst Length = 1 does not reduce the performance throughput.

Burst Stop Command

The Burst Stop command is one of several ways to terminate or interrupt a burst read or write operation.

Since the SDRAM burst length is always hardwired to be 1, the SDC does not support the Burst Stop command.

Burst Type

The burst type determines the address order in which the SDRAM delivers burst data after detecting a read command or stores burst data after detecting a write command. The burst type is programmed in the SDRAM during the SDRAM powerup sequence.

Since the SDRAM burst length is always programmed to be 1, the burst type does not matter. However, the SDC always sets the burst type to sequential-accesses-only during the SDRAM powerup sequence.

CAS Latency (CL)

The Column Address Strobe (CAS) latency is the delay in clock cycles between when the SDRAM detects the read command and when it provides the data at its output pins. The CAS latency is programmed in the SDRAM Mode register during the powerup sequence.

The speed grade of the SDRAM and the application's clock frequency determine the value of the CAS latency. The SDC can support CAS latency of two or three clock cycles. The selected CAS latency value must be programmed into the SDRAM Memory Global Control register (EBIU_SDGCTL) before the SDRAM powerup sequence. See "EBIU_SDGCTL Register" on page 17-32.

CBR (CAS Before RAS) Refresh or Auto-Refresh

When the SDC refresh counter times out, the SDC precharges all four banks of SDRAM and then issues an Auto-Refresh command to them. This causes the SDRAMs to generate an internal CBR refresh cycle. When the internal refresh completes, all four internal SDRAM banks are precharged.

DQM Pin Mask Function

The SDQM[1:0] pins provide a byte-masking capability on 8-bit writes to SDRAM. The DQM pins are used to block the input buffer of the SDRAM during partial write operations. The SDQM[1:0] pins are not used to mask data on partial read cycles. For write cycles, the data masks have a latency of zero cycles, permitting data writes when the corresponding SDQM[x]] pin is sampled low and blocking data writes when the SDQM[x] pin is sampled high on a byte-by-byte basis.

Internal Bank

There are several internal memory banks on a given SDRAM. The SDC supports interleaved accesses among the internal banks. The bank address can be thought of as part of the row address. The SDC assumes that all SDRAMs to which it interfaces have four internal banks and allows each activated bank to have a unique row address.

Mode Register

SDRAM devices contain an internal configuration register which allows specification of the SDRAM device's functionality. After powerup and before executing a read or write to the SDRAM memory space, the application must trigger the SDC to write the SDRAM's Mode register. The write of the SDRAM's Mode register is triggered by writing a 1 to the PSSE bit in the SDRAM Memory Global Control register (EBIU_SDGCTL) and then issuing a read or write transfer to the SDRAM address space. The initial read or write triggers the SDRAM powerup sequence to be run, which programs the SDRAM's Mode register with the CAS latency from the EBIU_SDGCTL register. This initial read or write to SDRAM takes many cycles to complete.

Note for most applications, the SDRAM powerup sequence and writing of the Mode register needs to be done only once. Once the powerup sequence has completed, the PSSE bit should not be set again unless a change to the Mode register is desired. In this case, please refer to "Managing SDRAM Refresh During PLL Transitions" on page 18-8.

Low power SDRAM devices may also contain an Extended Mode register. The EBIU enables programming of the Extended Mode register during powerup via the EMREN bit in the EBIU_SDGCTL register.

Page Size

Page size is the amount of memory which has the same row address and can be accessed with successive read or write commands without needing to activate another row. The page size can be calculated for 16-bit SDRAM banks with this formula:

- 16-bit SDRAM banks: page size = $2^{(CAW + 1)}$

where CAW is the column address width of the SDRAM, plus 1 because the SDRAM bank is 16 bits wide (1 address bit = 2 bytes).

Precharge Command

The Precharge command closes a specific internal bank in the active page or all internal banks in the page.

SDRAM Bank

The SDRAM bank is a region of memory that can be configured to 16M byte, 32M byte, 64M byte, or 128M byte and is selected by the $\overline{\text{SMS}}$ pin.

Do not confuse the “SDRAM internal banks” which are internal to the SDRAM and are selected with the bank address, with the “SDRAM bank” or “external bank” that is enabled by the $\overline{\text{SMS}}$ pin.

Self-Refresh

When the SDRAM is in Self-Refresh mode, the SDRAM’s internal timer initiates Auto-Refresh cycles periodically, without external control input. The SDC must issue a series of commands including the Self-Refresh command to put the SDRAM into this low power mode, and it must issue another series of commands to exit Self-Refresh mode. Entering Self-Refresh mode is programmable in the SDRAM Memory Global

Control register (EBIU_SDGCTL) and any access to the SDRAM address space causes the SDC to exit the SDRAM from Self-Refresh mode. See "Entering and Exiting Self-Refresh Mode (SRFS)" on page 17-37.

t_{RAS}

This is the required delay between issuing a Bank Activate command and issuing a Precharge command, and between the Self-Refresh command and the exit from Self-Refresh. The TRAS bit field in the SDRAM Memory Global Control register (EBIU_SDGCTL) is 4 bits wide and can be programmed to be 1 to 15 clock cycles long. See "Selecting the Bank Activate Command Delay (TRAS)" on page 17-40.

t_{RC}

This is the required delay between issuing successive Bank Activate commands to the same SDRAM internal bank. This delay is not directly programmable. The t_{RC} delay must be satisfied by programming the TRAS and TRP fields to ensure that $t_{RAS} + t_{RP} \geq t_{RC}$.

t_{RCD}

This is the required delay between a Bank Activate command and the start of the first Read or Write command. The TRCD bit field in the SDRAM Memory Global Control register (EBIU_SDGCTL) is three bits wide and can be programmed to be from 1 to 7 clock cycles long.

t_{RFC}

This is the required delay between issuing an Auto-Refresh command and a Bank Activate command and between issuing successive Auto-Refresh commands. This delay is not directly programmable and is assumed to be equal to t_{RC}. The t_{RC} delay must be satisfied by programming the TRAS and TRP fields to ensure that $t_{RAS} + t_{RP} \geq t_{RC}$.

t_{RP}

This is the required delay between issuing a Precharge command and:

- issuing a Bank Activate command
- issuing an Auto-Refresh command
- issuing a Self-Refresh command

The TRP bit field in the SDRAM Memory Global Control register (EBIU_SDGCTL) is three bits wide and can be programmed to be 1 to 7 clock cycles long. See "Selecting the Precharge Delay (TRP)" on page 17-42.

t_{RRD}

This is the required delay between issuing a Bank A Activate command and a Bank B Activate command. This delay is not directly programmable and is assumed to be t_{RCD} + 1.

t_{WR}

This is the required delay between a Write command (driving write data) and a Precharge command. The TWR bit field in the SDRAM Memory Global Control register (EBIU_SDGCTL) is two bits wide and can be programmed to be from 1 to 3 clock cycles long.

t_{XSR}

This is the required delay between exiting Self-Refresh mode and issuing the Auto-Refresh command. This delay is not directly programmable and is assumed to be equal to t_{RC}. The t_{RC} delay must be satisfied by programming the TRAS and TRP fields to ensure that $t_{RAS} + t_{RP} \geq t_{RC}$.

SDRAM Configurations Supported

Table 17-4 shows all possible bank sizes, bank widths and SDRAM discrete component configurations that can be gluelessly interfaced to the SDC.

Table 17-4. SDRAM Discrete Component Configurations Supported

Bank Size (M byte)	Bank Width (Bits)	SDRAM		
		Size (M bit)	Configuration	Number of Chips
16	16	32	2M x 4 x 4 banks	4
16	16	64	2M x 8 x 4 banks	2
16	16	128	2M x 16 x 4 banks	1
32	16	64	4M x 4 x 4 banks	4
32	16	128	4M x 8 x 4 banks	2
32	16	256	4M x 16 x 4 banks	1
64	16	128	8M x 4 x 4 banks	4
64	16	256	8M x 8 x 4 banks	2
64	16	512	8M x 16 x 4 banks	1
128	16	256	16M x 4 x 4 banks	4
128	16	512	16M x 8 x 4 banks	2
128	16	1024	16M x 16 x 4 banks	1

Example SDRAM System Block Diagrams

Figure 17-9 shows a block diagram of the SDRAM interface. In this example, the SDRAM interface connects to two 64 Mbit (x8) SDRAM devices to form one external bank of 16M bytes of memory. The same address and control bus feeds both SDRAM devices.

The SDC includes a separate address pin (SA10) to enable the execution of Auto-Refresh commands in parallel with any asynchronous memory access. This separate pin allows the SDC to issue a Precharge command to the SDRAM before it issues an Auto-Refresh command.

In addition, the SA10 pin allows the SDC to enter and exit Self-Refresh mode in parallel with any asynchronous memory access. The SA10 pin (instead of the ADDR[11] pin) should be directly connected to the SDRAM's A10 pin. During the Precharge command, SA10 is used to indicate that a Precharge All should be done. During a Bank Activate command, SA10 outputs the internal row address bit, which should be multiplexed to the A10 SDRAM input. During Read and Write commands, SA10 is used to disable the auto-precharge function of SDRAMs.

SDRAM systems do not use the ADDR[11] pin.

Executing a Parallel Refresh Command

The SDC includes a separate address pin (`SA10`) to enable the execution of Auto-Refresh commands in parallel with any asynchronous memory access. This separate pin allows the SDC to issue a Precharge command to the SDRAM before it issues an Auto-Refresh command. In addition, the `SA10` pin allows the SDC to enter and exit Self-Refresh mode in parallel with any asynchronous memory access.

The `SA10` pin should be directly connected to the `A10` pin of the SDRAM (instead of to the `ADDR[10]` pin). During the Precharge command, `SA10` is used to indicate that a Precharge All should be done. During a Bank Activate command, `SA10` outputs the internal row address bit, which should be multiplexed to the `A10` SDRAM input. During Read and Write commands, `SA10` is used to disable the auto-precharge function of SDRAMs.

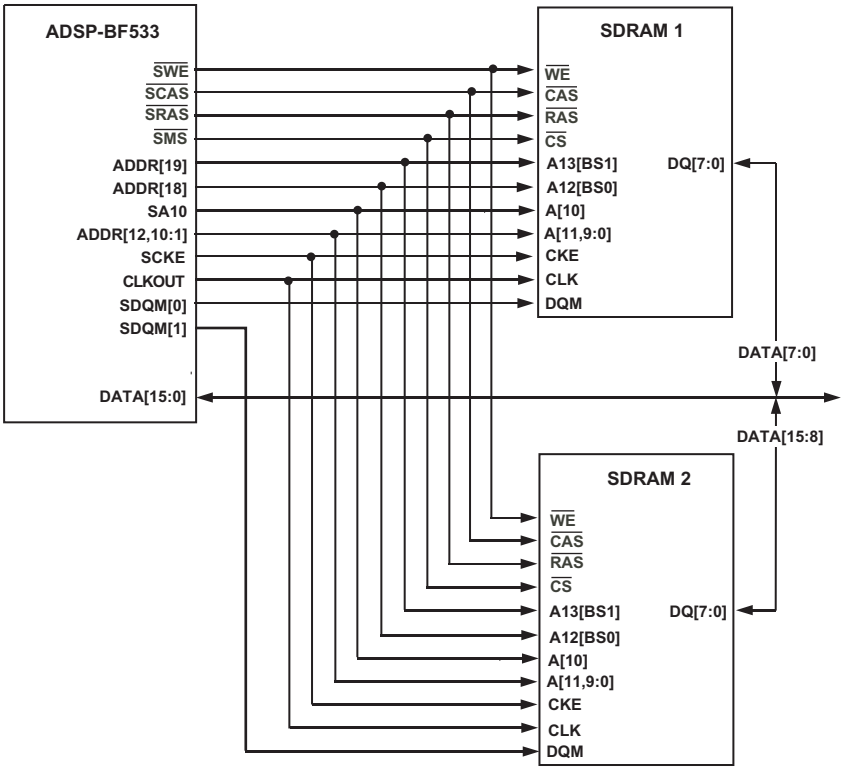


Figure 17-9. 16M Byte SDRAM System Example

EBIU_SDGCTL Register

The SDRAM Memory Global Control register (EBIU_SDGCTL) includes all programmable parameters associated with the SDRAM access timing and configuration. Figure 17-10 shows the EBIU_SDGCTL register bit definitions.

When using the hibernate state with the intent of preserving SDRAM contents during power-down, an application may issue an immediate read from SDRAM after enabling the controller. If this

is the case, the write to this register should be followed by an SSYNC instruction to prevent the subsequent read from happening before the controller is properly initialized.

The SCTLE bit is used to enable or disable the SDC. If SCTLE is disabled, any access to SDRAM address space generates an internal bus error, and the access does not occur externally. For more information, see "Error Detection" on page 17-9. When SCTLE is disabled, all SDC control pins are in their inactive states and the SDRAM clock is not running. The SCTLE bit must be enabled for SDC operation and is enabled by default at reset.

The CAS Latency (CL), SDRAM t_{RAS} Timing (TRAS), SDRAM t_{RP} Timing (TRP), SDRAM t_{RCD} Timing (TRCD), and SDRAM t_{WR} Timing (TWR) bits should be programmed based on the system clock frequency and the timing specifications of the SDRAM used.

The PSM and PSSE bits work together to specify and trigger an SDRAM powerup (initialization) sequence. If the PSM bit is set to 1, the SDC does a Precharge All command, followed by a Load Mode Register command, and then does eight Auto-Refresh cycles. If the PSM bit is cleared, the SDC does a Precharge All command, followed by eight Auto-Refresh cycles, and then a Load Mode Register command. Two events must occur before the SDC does the SDRAM powerup sequence:

- The PSSE bit must be set to 1 to enable the SDRAM powerup sequence.
- A read or write access must be done to enabled SDRAM address space in order to have the external bus granted to the SDC so that the SDRAM powerup sequence may occur.

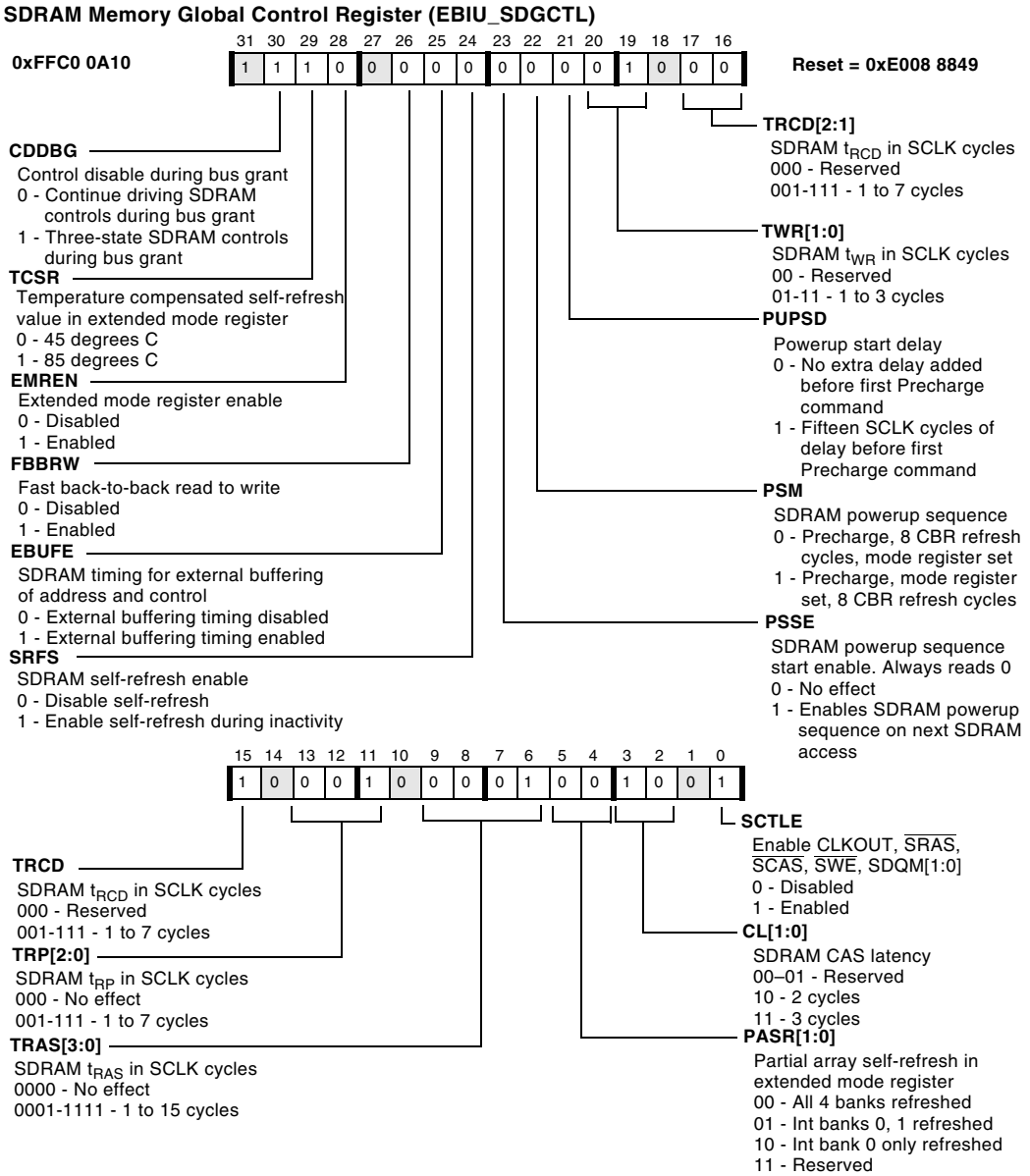


Figure 17-10. SDRAM Memory Global Control Register

The SDRAM powerup sequence occurs and is followed immediately by the read or write transfer to SDRAM that was used to trigger the SDRAM powerup sequence. Note there is a latency for this first access to SDRAM because the SDRAM powerup sequence takes many cycles to complete.

Before executing the SDC powerup sequence, ensure that the SDRAM receives stable power and is clocked for the proper amount of time, as specified by the SDRAM specification.

The Powerup Start Delay bit (PUPSD) optionally delays the powerup start sequence for 15 SCLK cycles. This is useful for multiprocessor systems sharing an external SDRAM. If the bus has been previously granted to the other processor before powerup and Self-Refresh mode is used when switching bus ownership, then the PUPSD bit can be used to guarantee a sufficient period of inactivity from self-refresh to the first Precharge command in the powerup sequence in order to meet the exit self-refresh time (t_{XSR}) of the SDRAM.

When the SRFS bit is set to 1, Self-Refresh mode is triggered. Once the SDC completes any active transfers, the SDC executes the sequence of commands to put the SDRAM into Self-Refresh mode. The next access to an enabled SDRAM bank causes the SDC to execute the commands to exit the SDRAM from Self-Refresh and execute the access. See "Entering and Exiting Self-Refresh Mode (SRFS)" on page 17-37 for more information about the SRFS bit.

The EBUFE bit is used to enable or disable external buffer timing. When buffered SDRAM modules or discrete register-buffers are used to drive the SDRAM control inputs, EBUFE should be set to 1. Using this setting adds a cycle of data buffering to read and write accesses. See "Setting the SDRAM Buffering Timing Option (EBUFE)" on page 17-39 for more information about the EBUFE bit.

The FBBRW bit enables an SDRAM read followed by write to occur on consecutive cycles. In many systems, this is not possible because the turn-off time of the SDRAM data pins is too long, leading to bus contention with the succeeding write from the processor. When this bit is 0, a clock cycle is inserted between read accesses followed immediately by write accesses.

The EMREN bit enables programming of the Extended Mode register during startup. The Extended Mode register is used to control SDRAM power consumption in certain mobile low power SDRAMs. If the EMREN bit is enabled, then the TCSR and PASR[1:0] bits control the value written to the Extended Mode register. The PASR bits determine how many SDRAM internal banks are refreshed during Self-Refresh. The TCSR bit signals to the SDRAM the worst case temperature range for the system, and thus how often the SDRAM internal banks need to be refreshed during Self-Refresh.

The CDDBG bit is used to enable or disable the SDRAM control signals when the external memory interface is granted to an external controller. If this bit is set to a 1, then the control signals are three-stated when bus grant is active. Otherwise, these signals continue to be driven during grant. If the bit is set and the external bus is granted, all SDRAM internal banks are assumed to have been changed by the external controller. This means a precharge is required on each bank prior to use after control of the external bus is re-established. The control signals affected by this pin are $\overline{\text{SRAS}}$, $\overline{\text{SCAS}}$, $\overline{\text{SWE}}$, $\overline{\text{SMS}}$, SA10, SCKE, and CLKOUT.

Note all reserved bits in this register must always be written with 0s.

Setting the SDRAM Clock Enable (SCTLE)

The SCTLE bit allows software to disable all SDRAM control pins. These pins are SDQM[3:0], $\overline{\text{SCAS}}$, $\overline{\text{SRAS}}$, $\overline{\text{SWE}}$, SCKE, and CLKOUT.

- SCTLE = 0

  Disable all SDRAM control pins (control pins negated, CLKOUT low)

- SCTLE = 1

  Enable all SDRAM control pins (CLKOUT toggles)

Note the CLKOUT function is also shared with the AMC. Even if SCTLE is disabled, CLKOUT can be enabled independently by the CLKOUT enable in the AMC (AMCKEN in the EBIU_AMGCTL register).

If the system does not use SDRAM, SCTLE should be set to 0.

If an access occurs to the SDRAM address space while SCTLE is 0, the access generates an internal bus error and the access does not occur externally. For more information, see “Error Detection” on page 17-9. With careful software control, the SCTLE bit can be used in conjunction with Self-Refresh mode to further lower power consumption. However, SCTLE must remain enabled at all times when the SDC is needed to generate Auto-Refresh commands to SDRAM.

Entering and Exiting Self-Refresh Mode (SRFS)

The SDC supports SDRAM Self-Refresh mode. In Self-Refresh mode, the SDRAM performs refresh operations internally—without external control—reducing the SDRAM’s power consumption.

The SRFS bit in EBIU_SDGCTL enables the start of Self-Refresh mode:

- SRFS = 0

  Disable Self-Refresh mode

- SRFS = 1

  Enable Self-Refresh mode

When SRFS is set to 1, once the SDC enters an idle state it issues a Pre-charge command if necessary, and then issues a Self-Refresh command. If an internal access is pending, the SDC delays issuing the Self-Refresh command until it completes the pending SDRAM access and any subsequent pending access requests. Refer to "SDC Commands" on page 17-55 for more information.

Once the SDRAM device enters into Self-Refresh mode, the SDRAM controller asserts the SDSRA bit in the SDRAM Control Status register (EBIU_SDSTAT).

The SDRAM device exits Self-Refresh mode only when the SDC receives a core or DMA access request. In conjunction with the SRFS bit, 2 possibilities are given to exit the self-refresh mode:

- If SRFS bit is set before the request, the SDC exits self-refresh and remains in auto-refresh mode.
- If SRFS bit is cleared before the request, the SDC exits self-refresh only for a single request and returns back to self-refresh mode until a new request is coming.

Note once the SRFS bit is set to 1, the SDC enters Self-Refresh mode when it finishes pending accesses. There is no way to cancel the entry into-Self-Refresh mode.

Setting the SDRAM Buffering Timing Option (EBUFE)

To meet overall system timing requirements, systems that employ several SDRAM devices connected in parallel may require buffering between the processor and multiple SDRAM devices. This buffering generally consists of a register and driver.

To meet such timing requirements and to allow intermediary registration, the SDC supports pipelining of SDRAM address and control signals.

The `EBUFE` bit in the `EBIU_SDGCTL` register enables this mode:

- `EBUFE = 0`

  Disable external buffering timing

- `EBUFE = 1`

  Enable external buffering timing

When `EBUFE = 1`, the SDRAM controller delays the data in write accesses by one cycle, enabling external buffer registers to latch the address and controls. In read accesses, the SDRAM controller samples data one cycle later to account for the one-cycle delay added by the external buffer registers. When external buffering timing is enabled, the latency of all accesses is increased by one cycle.

Selecting the CAS Latency Value (CL)

The CAS latency value defines the delay, in number of clock cycles, between the time the SDRAM detects the Read command and the time it provides the data at its output pins.

CAS latency does not apply to write cycles.

The CL bits in the SDRAM Memory Global Control register (EBIU_SDGCTL) select the CAS latency value:

- CL = 00

  Reserved

- CL = 01

  Reserved

- CL = 10

  2 clock cycles

- CL = 11

  3 clock cycles

Generally, the frequency of operation determines the value of the CAS latency. For specific information about setting this value, consult the SDRAM device documentation.

Selecting the Bank Activate Command Delay (TRAS)

The t_{RAS} value (Bank Activate command delay) defines the required delay, in number of clock cycles, between the time the SDC issues a Bank Activate command and the time it issues a Precharge command. The SDRAM must also remain in Self-Refresh mode for at least the time period specified by t_{RAS}. The t_{RP} and t_{RAS} values define the t_{RFC}, t_{RC}, and t_{XSR} values. See page 17-28 for more information.

The t_{RAS} parameter allows the processor to adapt to the timing requirements of the system's SDRAM devices.

The TRAS bits in the SDRAM Memory Global Control register (EBIU_SDGCTL) select the t_{RAS} value. Any value between 1 and 15 clock cycles can be selected. For example:

- TRAS = 0000

  No effect

- TRAS = 0001

  1 clock cycle

- TRAS = 0010

  2 clock cycles

- TRAS = 1111

  15 clock cycles

For specific information on setting this value, consult the SDRAM device documentation.

Selecting the RAS to CAS Delay (TRCD)

The t_{RCD} value (RAS to CAS delay) defines the delay for the first read or write command after a row activate command, in number of clock cycles. The t_{RCD} parameter allows the processor to adapt to the timing requirements of the system's SDRAM devices.

The t_{RCD} bits in the SDRAM Memory Global Control register (EBIU_SDGCTL) select the t_{RCD} value. Any value between 1 and 7 clock cycles may be selected. For example:

- `TRCD = reserved`

  No effect

- `TRCD = 001`

  1 clock cycle

- `TRCD = 010`

  2 clock cycles

- `TRCD = 111`

  7 clock cycles

Selecting the Precharge Delay (TRP)

The t_{RP} value (Precharge delay) defines the required delay, in number of clock cycles, between the time the SDC issues a Precharge command and the time it issues a Bank Activate command. The t_{RP} also specifies the time required between Precharge and Auto-Refresh, and between Precharge and Self-Refresh. The t_{RP} and t_{RAS} values define the t_{RFC}, t_{RC}, and t_{XSR} values.

This parameter enables the application to accommodate the SDRAM's timing requirements.

The TRP bits in the SDRAM Memory Global Control register (EBIU_SDGCTL) select the t_{RP} value. Any value between 1 and 7 clock cycles may be selected. For example:

- TRP = 000

  No effect

- TRP = 001

  1 clock cycle

- TRP = 010

  2 clock cycles

- TRP = 111

  7 clock cycles

Selecting the Write to Precharge Delay (TWR)

The t_{WR} value defines the required delay, in number of clock cycles, between the time the SDC issues a Write command (drives write data) and a Precharge command.

This parameter enables the application to accommodate the SDRAM's timing requirements.

The TWR bits in the SDRAM Memory Global Control register (EBIU_SDGCTL) select the t_{WR} value.

Any value between 1 and 3 clock cycles may be selected. For example:

- `TWR = 00`

  Reserved

- `TWR = 01`

  1 clock cycle

- `TWR = 10`

  2 clock cycles

- `TWR = 11`

  3 clock cycles

EBIU_SDBCTL Register

The SDRAM Memory Bank Control register (`EBIU_SDBCTL`) includes external bank-specific programmable parameters. It allows software to control some parameters of the SDRAM. The external bank can be configured for a different size of SDRAM. It uses the access timing parameters defined in the SDRAM Memory Global Control register (`EBIU_SDGCTL`). The `EBIU_SDBCTL` register should be programmed before powerup and should be changed only when the SDC is idle.

The `EBIU_SDBCTL` register stores the configuration information for the SDRAM bank interface. The EBIU supports 64 Mbit, 128 Mbit, 256 Mbit, and 512 Mbit SDRAM devices with x4, x8, x16 configurations. Table 17-4 maps SDRAM density and I/O width to the supported `EBSZ` encodings. See "SDRAM External Memory Size" on page 17-50 for more information on bank starting address decodes.

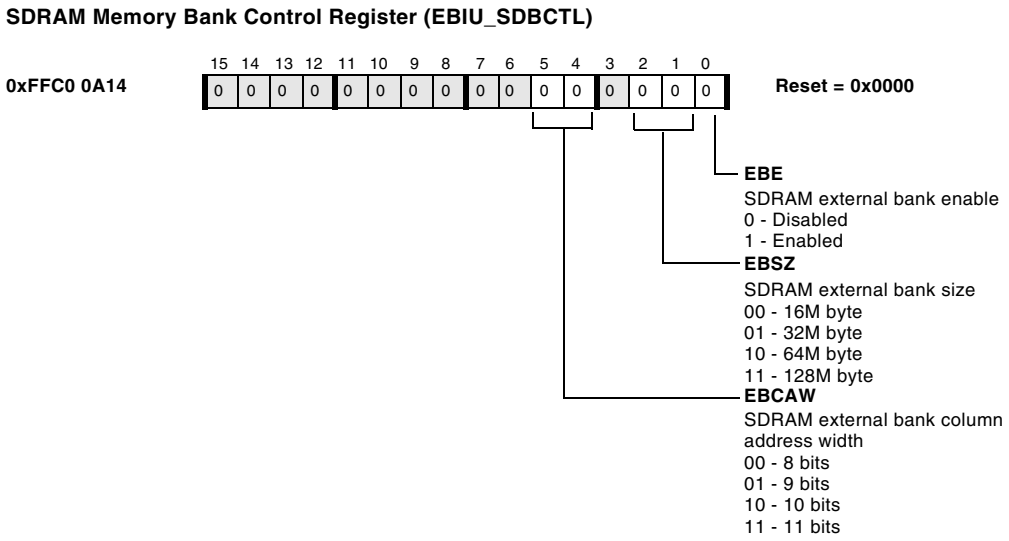


Figure 17-11. SDRAM Memory Bank Control Register

The SDC determines the internal SDRAM page size from the `EBCAW` parameters. Page sizes of 512 B, 1K byte, 2K byte, and 4K byte are supported. Table 17-5 shows the page size and breakdown of the internal address (`IA[31:0]`, as seen from the core or DMA) into the row, bank, column, and byte address. The column address and the byte address together make up the address inside the page.

The `EBE` bit in the `EBIU_SDBCTL` register is used to enable or disable the external SDRAM bank. If the SDRAM is disabled, any access to the SDRAM address space generates an internal bus error, and the access does not occur externally. For more information, see “Error Detection” on page 17-9.

For information on how to connect to SDRAMs smaller than 16M byte, please see “Using SDRAMs Smaller Than 16M Byte” on page 18-8.

Table 17-5. Internal Address Mapping

Bank Width (bits)	Bank Size (Mbyte)	Col. Addr. Width (CAW)	Page Size (K Byte)	Bank Address	Row Address	Page	
						Column Address	Byte Address
16	128	11	4	IA[26:25]	IA[24:12]	A[11:1]	IA[0]
16	128	10	2	IA[26:25]	IA[24:11]	IA[10:1]	IA[0]
16	128	9	1	1A[26:25]	IA[24:10]	IA[9:1]	IA[0]
16	128	8	.5	IA[26:25]	IA[24:9]	IA[8:1]	IA[0]
16	64	11	4	IA[25:24]	IA[23:12]	IA[11:1]	IA[0]
16	64	10	2	IA[25:24]	IA[23:11]	IA[10:1]	IA[0]
16	64	9	1	IA[25:24]	IA[23:10]	IA[9:1]	IA[0]
16	64	8	.5	IA[25:24]	IA[23:9]	IA[8:1]	IA[0]
16	32	11	4	IA[24:23]	IA[22:12]	IA[11:1]	IA[0]
16	32	10	2	IA[24:23]	IA[22:11]	IA[10:1]	IA[0]
16	32	9	1	IA[24:23]	IA[22:10]	IA[9:1]	IA[0]
16	32	8	.5	IA[24:23]	IA[22:9]	IA[8:1]	IA[0]
16	16	11	4	IA[23:22]	IA[21:12]	IA[11:1]	IA[0]
16	16	10	2	IA[23:22]	IA[21:11]	IA[10:1]	IA[0]
16	16	9	1	IA[23:22]	IA[21:10]	IA[9:1]	IA[0]
16	16	8	.5	IA[23:22]	IA[21:9]	IA[8:1]	IA[0]

EBIU_SDSTAT Register

The SDRAM Control Status register (EBIU_SDSTAT) provides information on the state of the SDC. This information can be used to determine when it is safe to alter SDC control parameters or it can be used as a debug aid. The SDEASE bit of this register is sticky. Once it has been set, software

must explicitly write a 1 to the bit to clear it. Writes have no effect on the other status bits, which are updated by the SDC only. This SDC MMR is 16 bits wide.

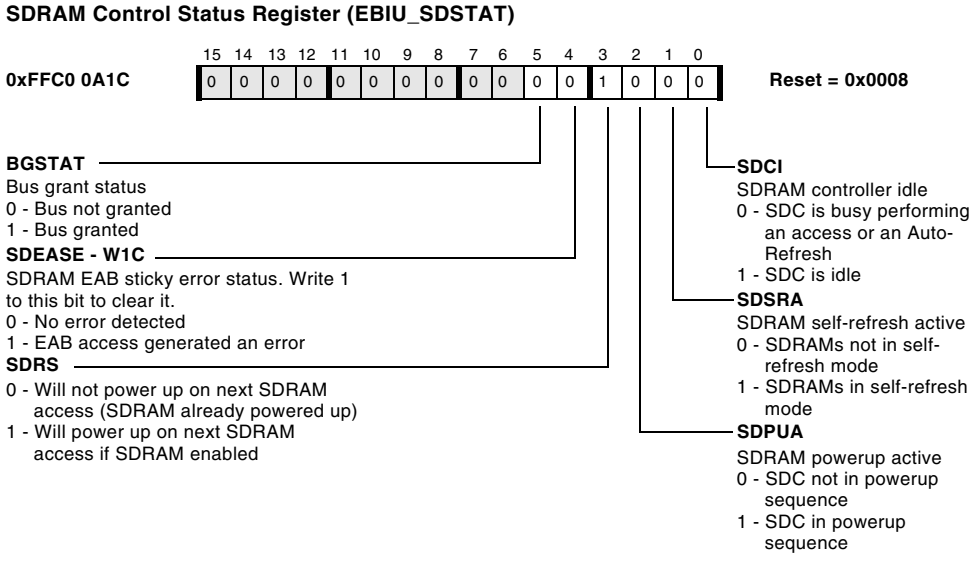


Figure 17-12. SDRAM Control Status Register

EBIU_SDRRC Register

The SDRAM Refresh Rate Control register (`EBIU_SDRRC`) provides a flexible mechanism for specifying the Auto-Refresh timing. Since the clock supplied to the SDRAM can vary, the SDC provides a programmable refresh counter, which has a period based on the value programmed into the `RDIV` field of this register. This counter coordinates the supplied clock rate with the SDRAM device's required refresh rate.

The desired delay (in number of SDRAM clock cycles) between consecutive refresh counter time-outs must be written to the `RDIV` field. A refresh counter time-out triggers an Auto-Refresh command to all external

SDRAM devices. Write the RDIV value to the EBIU_SDRRC register before the SDRAM powerup sequence is triggered. Change this value only when the SDC is idle.

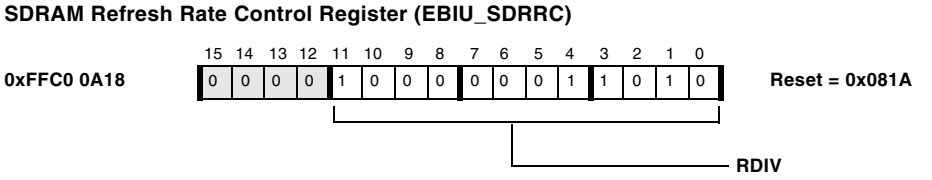


Figure 17-13. SDRAM Refresh Rate Control Register

To calculate the value that should be written to the EBIU_SDRRC register, use the following equation:

$$RDIV = ((f_{SCLK} \times t_{REF}) / NRA) - (t_{RAS} + t_{RP})$$

Where:

- f_{SCLK} = SDRAM clock frequency (system clock frequency)
- t_{REF} = SDRAM refresh period
- NRA = Number of row addresses in SDRAM (refresh cycles to refresh whole SDRAM)
- t_{RAS} = Active to Precharge time (TRAS in the SDRAM Memory Global Control register) in number of clock cycles
- t_{RP} = RAS to Precharge time (TRP in the SDRAM Memory Global Control register) in number of clock cycles

This equation calculates the number of clock cycles between required refreshes and subtracts the required delay between Bank Activate commands to the same internal bank ($t_{RC} = t_{RAS} + t_{RP}$). The t_{RC} value is subtracted, so that in the case where a refresh time-out occurs while an

SDRAM cycle is active, the SDRAM refresh rate specification is guaranteed to be met. The result from the equation should always be rounded down to an integer.

Below is an example of the calculation of RDIV for a typical SDRAM in a system with a 133 MHz clock:

f_{SCLK} = 133 MHz

t_{REF} = 64 ms

NRA = 4096 row addresses

t_{RAS} = 2

t_{RP} = 2

The equation for RDIV yields:

RDIV = ($(133 \times 10^{6} \times 64 \times 10^{-3}) / 4096) - (2 + 2) = 2074$ clock cycles

This means RDIV is 0x81A (hex) and the SDRAM Refresh Rate Control register should be written with 0x081A.

Note RDIV must be programmed to a nonzero value if the SDRAM controller is enabled. When RDIV = 0, operation of the SDRAM controller is not supported and can produce undesirable behavior. Values for RDIV can range from 0x001 to 0xFFF.

Please refer to "Managing SDRAM Refresh During PLL Transitions" on page 18-8 for a detailed discussion of the process for changing the PLL frequency when using SDRAM.

SDRAM External Memory Size

The total amount of external SDRAM memory addressed by the processor is controlled by the EBSZ bits of the EBIU_SDBCTL register (see Table 17-6). Accesses above the range shown for a specialized EBSZ value results in an internal bus error and the access does not occur. For more information, see “Error Detection” on page 17-9.

Table 17-6. Bank Size Encodings

EBSZ	Bank Size (Mbyte)	Valid SDRAM Addresses
00	16	0x0000 0000 – 0x00FF FFFF
01	32	0x0000 0000 – 0x01FF FFFF
10	64	0x0000 0000 – 0x03FF FFFF
11	128	0x0000 0000 – 0x07FF FFFF

SDRAM Address Mapping

To access SDRAM, the SDC multiplexes the internal 32-bit non-multiplexed address into a row address, a column address, a bank address, and the byte data masks for the SDRAM device. See Figure 17-14. The lowest bit is mapped to byte data masks, the next bits are mapped into the column address, the next bits are mapped into the row address, and the final two bits are mapped into the bank address. This mapping is based on the EBSZ and EBCAW parameters programmed into the SDRAM Memory Bank Control register.

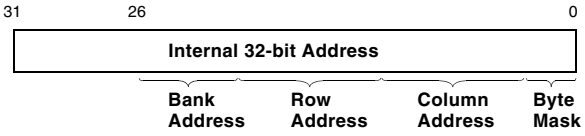


Figure 17-14. Multiplexed SDRAM Addressing Scheme

16-Bit Wide SDRAM Address Muxing

Table 17-7 shows the connection of the address pins with the SDRAM device pins.

Table 17-7. SDRAM Address Connections for 16-Bit Banks

External Address Pin	SDRAM Address Pin
ADDR[19]	BA[1]
ADDR[18]	BA[0]
ADDR[16]	A[15]
ADDR[15]	A[14]
ADDR[14]	A[13]
ADDR[13]	A[12]
ADDR[12]	A[11]
SA[10]	A[10]
ADDR[10]	A[9]
ADDR[9]	A[8]
ADDR[8]	A[7]
ADDR[7]	A[6]
ADDR[6]	A[5]
ADDR[5]	A[4]
ADDR[4]	A[3]
ADDR[3]	A[2]
ADDR[2]	A[1]
ADDR[1]	A[0]

Data Mask (SDQM[1:0]) Encodings

During write transfers to SDRAM, the SDQM[1:0] pins are used to mask writes to bytes that are not accessed. Table 17-8 shows the SDQM[1:0] encodings for 16-bit wide SDRAM based on the internal transfer address bit IA[0] and the transfer size.

During read transfers to SDRAM banks, reads are always done of all bytes in the bank regardless of the transfer size. This means for 16-bit SDRAM banks, SDQM[1:0] are all 0s.

The only time that the SDQM[1:0] pins are high is when bytes are masked during write transfers to the SDRAM. At all other times, the SDQM[1:0] pins are held low.

Table 17-8. SDQM[1:0] Encodings During Writes

Internal Address IA[0]	Internal Transfer Size		
	byte	2 bytes	4 bytes
0	SDQM[1] = 1 SDQM[0] = 0	SDQM[1] = 0 SDQM[0] = 0	SDQM[1] = 0 SDQM[0] = 0
1	SDQM[1] = 0 SDQM[0] = 1		

SDC Operation

The SDC uses a burst length = 1 for read and write operations. Whenever a page miss occurs, the SDC executes a Precharge command followed by a Bank Activate command before executing the Read or Write command. If there is a page hit, the Read or Write command can be given immediately without requiring the Precharge command.

For SDRAM Read commands, there is a latency from the start of the Read command to the availability of data from the SDRAM, equal to the CAS latency. This latency is always present for any single read transfer. Subsequent reads do not have latency.

A programmable refresh counter is provided. It can be programmed to generate background Auto-Refresh cycles at the required refresh rate based on the clock frequency used. The refresh counter period is specified with the RDIV field in the SDRAM Refresh Rate Control register.

To allow Auto-Refresh commands to execute in parallel with any AMC access, a separate A10 pin (SA10) is provided. All the SDRAM internal banks are precharged before issuing an Auto-Refresh command.

The internal 32-bit non-multiplexed address is multiplexed into a row address, a column address, a bank select address, and data masks. Bit0 for 16-bit wide SDRAMs is used to generate the data masks. The next lowest bits are mapped into the column address, next bits are mapped into the row address, and the final two bits are mapped into the internal bank address. This mapping is based on the EBCAW and EBSZ values programmed into the SDRAM Memory Bank Control register.

SDC Configuration

After a processor's hardware or software reset, the SDC clocks are enabled; however, the SDC must be configured and initialized. Before programming the SDC and executing the powerup sequence, ensure the clock to the SDRAM is enabled after the power has stabilized for the proper amount of time (as specified by the SDRAM). In order to set up the SDC and start the SDRAM powerup sequence for the SDRAMs, the SDRAM Refresh Rate Control register (EBIU_SDRRC), the SDRAM Memory Bank Control register (EBIU_SDBCTL), and SDRAM Memory Global Control register (EBIU_SDGCTL) must be written, and a transfer must be started to

SDRAM address space. The SDRS bit of the SDRAM Control Status register can be checked to determine the current state of the SDC. If this bit is set, the SDRAM powerup sequence has not been initiated.

The RDIV field of the EBIU_SDRRC register should be written to set the SDRAM refresh rate.

The EBIU_SDBCTL register should be written to describe the sizes and SDRAM memory configuration used (EBSZ and EBCAW) and to enable the external bank (EBE). Note until the SDRAM powerup sequence has been started, any access to SDRAM address space, regardless of the state of the EBE bit, generates an internal bus error, and the access does not occur externally. For more information, see “Error Detection” on page 17-9. After the SDRAM powerup sequence has completed, if the external bank is disabled, any transfer to it results in a hardware error interrupt, and the SDRAM transfer does not occur.

The EBIU_SDGCTL register is written:

- to set the SDRAM cycle timing options (CL, TRAS, TRP, TRCD, TWR, EBUFE)
- to enable the SDRAM clock (SCTLE)
- to select and enable the start of the SDRAM powerup sequence (PSM, PSSE)

Note if SCTLE is disabled, any access to SDRAM address space generates an internal bus error and the access does not occur externally. For more information, see “Error Detection” on page 17-9.

Once the PSSE bit in the EBIU_SDGCTL register is set to 1, and a transfer occurs to enabled SDRAM address space, the SDC initiates the SDRAM powerup sequence. The exact sequence is determined by the PSM bit in the EBIU_SDGCTL register. The transfer used to trigger the SDRAM powerup sequence can be either a read or a write. This transfer occurs when the

SDRAM powerup sequence has completed. This initial transfer takes many cycles to complete since the SDRAM powerup sequence must take place.

SDC Commands

This section provides a description of each of the commands that the SDC uses to manage the SDRAM interface. These commands are initiated automatically upon a memory read or memory write. A summary of the various commands used by the on-chip controller for the SDRAM interface is as follows.

- Precharge All: Precharges all banks
- Single Precharge: Precharges a single bank
- Bank Activate: Activates a page in the required SDRAM internal bank
- Load Mode Register: Initializes the SDRAM operation parameters during the powerup sequence
- Load Extended Mode Register: Initializes mobile SDRAM operation parameters during the powerup sequence
- Read/Write
- Auto-Refresh: Causes the SDRAM to execute a CAS before RAS refresh
- Self-Refresh: Places the SDRAM in self-refresh mode, in which the SDRAM powers down and controls its refresh operations internally
- NOP/Command Inhibit: No operation

Table 17-9 shows the SDRAM pin state during SDC commands.

Table 17-9. Pin State During SDC Commands

Command	$\overline{\text{SMS}}$	$\overline{\text{SCAS}}$	$\overline{\text{SRAS}}$	$\overline{\text{SWE}}$	SCKE	SA10
Precharge All	low	high	low	low	high	high
Single Precharge	low	high	low	low	high	low
Bank Activate	low	high	low	high	high	
Load Mode Register	low	low	low	low	high	
Load Extended Mode Register	low	low	low	low	high	low
Read	low	low	high	high	high	low
Write	low	low	high	low	high	low
Auto-Refresh	low	low	low	high	high	
Self-Refresh	low	low	low	high	low	
NOP	low	high	high	high	high	
Command Inhibit	high	high	high	high	high	

Precharge Commands

The Precharge All command is given to precharge all internal banks at the same time before executing an auto-refresh. For a page miss during reads or writes in a specific internal SDRAM bank, the SDC uses the Single Precharge command to that bank.

Bank Activate Command

The Bank Activate command is required if the next data access is in a different page. The SDC executes the Precharge command, followed by a Bank Activate command, to activate the page in the desired SDRAM internal bank.

The SDC supports bank interleaving (opening up to 4 internal SDRAM banks at a time). This results in an effective size of 4 pages. The address mapping indicates the start address of each internal bank.

Bank interleaving is accomplished by switching between 4 internal SDRAM banks without any stalls between the pages.

Load Mode Register Command

The Load Mode Register command initializes SDRAM operation parameters. This command is a part of the SDRAM powerup sequence. The Load Mode Register command uses the address bus of the SDRAM as data input. The powerup sequence is initiated by writing 1 to the `PSSE` bit in the SDRAM Memory Global Control register (`EBIU_SDGCTL`) and then writing or reading from any enabled address within the SDRAM address space to trigger the powerup sequence. The exact order of the powerup sequence is determined by the `PSM` bit of the `EBIU_SDGCTL` register.

The Load Mode Register command initializes these parameters:

- Burst length = 1, bits 2–0, always 0
- Wrap type = sequential, bit 3, always 0
- Ltmode = latency mode (CAS latency), bits 6–4, programmable in the `EBIU_SDGCTL` register
- Bits 14–7, always 0

While executing the Load Mode Register command, the unused address pins are set to 0. During the two clock cycles following the Load Mode Register command, the SDC issues only NOP commands.

For low power mobile SDRAMs that include an Extended Mode register, this register is programmed during powerup sequence if the EMREN bit is set in the EBIU_SDGCTL register.

The Extended Mode register is initialized with these parameters:

- Partial Array Self-Refresh, bits 2–0, bit 2 always 0, bits 1–0 programmable in EBIU_SDGCTL
- Temperature Compensated Self-Refresh, bits 4–3, bit 3 always 1, bit 4 programmable in EBIU_SDGCTL
- Bits 12–5, always 0, and bit 13 always 1

Read/Write Command

A Read/Write command is executed if the next read/write access is in the present active page. During the Read command, the SDRAM latches the column address. The delay between Activate and Read commands is determined by the t_{RCD} parameter. Data is available from the SDRAM after the CAS latency has been met.

In the Write command, the SDRAM latches the column address. The write data is also valid in the same cycle. The delay between Activate and Write commands is determined by the t_{RCD} parameter.

The SDC does not use the auto-precharge function of SDRAMs, which is enabled by asserting SA10 high during a Read or Write command.

Auto-Refresh Command

The SDRAM internally increments the refresh address counter and causes a CAS before RAS (CBR) refresh to occur internally for that address when the Auto-Refresh command is given. The SDC generates an Auto-Refresh command after the SDC refresh counter times out. The RDIV value in the SDRAM Refresh Rate Control register must be set so that all addresses are refreshed within the t_{REF} period specified in the SDRAM timing specifications. This command is issued to the external bank whether or not it is enabled (EBE in the SDRAM Memory Global Control register). Before executing the Auto-Refresh command, the SDC executes a Precharge All command to the external bank. The next Activate command is not given until the t_{RFC} specification ($t_{RFC} = t_{RAS} + t_{RP}$) is met.

Auto-Refresh commands are also issued by the SDC as part of the powerup sequence and also after exiting Self-Refresh mode.

Self-Refresh Command

The Self-Refresh command causes refresh operations to be performed internally by the SDRAM, without any external control. This means that the SDC does not generate any Auto-Refresh cycles while the SDRAM is in Self-Refresh mode. Before executing the Self-Refresh command, all internal banks are precharged. Self-Refresh mode is enabled by writing a 1 to the SRFS bit of the SDRAM Memory Global Control register (EBIU_SDGCTL). After issuing the Self-Refresh command, the SDC drives SCKE low. This puts the SDRAM into a power down mode (SCKE = 0, $\overline{\text{SRAS}}/\overline{\text{SMS}}/\overline{\text{SCAS}}/\overline{\text{SWE}} = \overline{1}$) Before exiting Self-Refresh mode, the SDC asserts SCKE. The SDRAM remains in Self-Refresh mode for at least t_{RAS} and until an internal access to SDRAM space occurs. When an internal access occurs causing the SDC to exit the SDRAM from Self-Refresh mode, the SDC waits to meet the t_{XSR} specification ($t_{XSR} = t_{RAS} + t_{RP}$) and then issues an Auto-Refresh command. After the Auto-Refresh command, the SDC waits for the t_{RFC} specification ($t_{RFC} = t_{RAS} + t_{RP}$) to be met before executing the Activate command for the transfer that caused the SDRAM

to exit Self-Refresh mode. Therefore, the latency from when a transfer is received by the SDC while in Self-Refresh mode, until the Activate command occurs for that transfer, is 2 x (t_{RAS} + t_{RP}).

Note CLKOUT is not disabled by the SDC during Self-Refresh mode. However, software may disable the clock by clearing the SCTLE bit in EBIU_SDGCTL. The application software should ensure that all applicable clock timing specifications are met before the transfer to SDRAM address space which causes the controller to exit Self-Refresh mode. If a transfer occurs to SDRAM address space when the SCTLE bit is cleared, an internal bus error is generated, and the access does not occur externally, leaving the SDRAM in Self-Refresh mode. For more information, see "Error Detection" on page 17-9.

No Operation/Command Inhibit Commands

The No Operation (NOP) command to the SDRAM has no effect on operations currently in progress. The Command Inhibit command is the same as a NOP command; however, the SDRAM is not chip-selected. When the SDC is actively accessing the SDRAM but needs to insert additional commands with no effect, the NOP command is given. When the SDC is not accessing the SDRAM, the Command Inhibit command is given.

SDRAM Timing Specifications

To support key timing requirements and powerup sequences for different SDRAM vendors, the SDC provides programmability for t_{RAS}, t_{RP}, t_{RCD}, t_{WR}, and the powerup sequence mode. (For more information, see "EBIU_SDGCTL Register" on page 17-32.) CAS latency should be programmed in the EBIU_SDGCTL register based on the frequency of operation. (Please refer to the SDRAM vendor's data sheet for more information.)

For other parameters, the SDC assumes:

- Bank Cycle Time: $t_{RC} = t_{RAS} + t_{RP}$
- Refresh Cycle Time: $t_{RFC} = t_{RAS} + t_{RP}$
- Exit Self-Refresh Time: $t_{XSR} = t_{RAS} + t_{RP}$
- Load Mode Register to Activate Time: t_{MRD} or t_{RSC} = 3 clock cycles
- Page-Miss Penalty = $t_{RP} + t_{RCD}$
- Row (Bank A) to Row (Bank B) Active Time: $t_{RRD} = t_{RCD} + 1$

SDRAM Performance

Table 7-2 lists the data throughput rates for the core or DMA read/write accesses to 16-bit wide SDRAM. For this example, assume all cycles are SCLK cycles and the following SCLK frequency and SDRAM parameters are used:

- SCLK frequency = 133 MHz
- CAS latency = 2 cycles (CL = 2)
- No SDRAM buffering (EBUFE = 0)
- RAS precharge (t_{RP}) = 2 cycles (TRP = 2)
- RAS to CAS delay (t_{RCD}) = 2 cycles (TRCD = 2)
- Active command time (t_{RAS}) = 5 cycles (TRAS = 5)

When the external buffer timing (EBUFE = 1 in the SDRAM Memory Global Control register) and/or CAS latency of 3 (CL = 11 in the SDRAM Memory Global Control register) is used, all accesses take one extra cycle for each feature selected.

Bus Request and Grant

The processor can relinquish control of the data and address buses to an external device. The processor three-states its memory interface to allow an external controller to access either external asynchronous or synchronous memory parts.

Operation

When the external device requires access to the bus, it asserts the Bus Request ($\overline{\text{BR}}$) signal. The $\overline{\text{BR}}$ signal is arbitrated with EAB requests. If no internal request is pending, the external bus request will be granted. The processor initiates a bus grant by:

- Three-stating the data and address buses and the asynchronous memory control signals. The synchronous memory control signals can optionally be three-stated.
- Asserting the Bus Grant ($\overline{\text{BG}}$) signal.

The processor may halt program execution if the bus is granted to an external device and an instruction fetch or data read/write request is made to external memory. When the external device releases $\overline{\text{BR}}$, the processor deasserts BG and continues execution from the point at which it stopped.

The processor asserts the $\overline{\text{BGH}}$ pin when it is ready to start another external port access, but is held off because the bus was previously granted.

When the bus has been granted, the BGSTAT bit in the SDSTAT register is set. This bit can be used by the processor to check the bus status to avoid initiating a transaction that would be delayed by the external bus grant.

18 SYSTEM DESIGN

This chapter provides hardware, software, and system design information to aid users in developing systems based on the Blackfin processor. The design options implemented in a system are influenced by cost, performance, and system requirements. In many cases, design issues cited here are discussed in detail in other sections of this manual. In such cases, a reference appears to the corresponding section of the text, instead of repeating the discussion in this chapter.

Pin Descriptions

Refer to the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* for pin information, including pin numbers for the 160-lead PBGA package.

Recommendations for Unused Pins

Refer to the *ADSP-BF531/ADSP-BF532/ADSP-BF533 Embedded Processor Data Sheet* for detailed pin descriptions.

Resetting the Processor

In addition to the Hardware Reset mode provided via the $\overline{\text{RESET}}$ pin, the processor supports several software reset modes. For detailed information on the various modes, see "System Reset and Powerup" on page 3-12.

The processor state after reset is described in "Reset State" on page 3-10.

Booting the Processor

The processor can be booted via a variety of methods. These include executing from external 16-bit memory, booting from a ROM configured to load code from 8-bit flash memory, or booting from a serial ROM (8-bit, 16-bit, or 24-bit address range). For more information on boot modes, see "Booting Methods" on page 3-18.

Figure 18-1 and Figure 18-2 show the connections necessary for 8-bit and 16-bit booting, respectively. Notice that the address connections are made in the same manner for both 8- and 16-bit peripherals. Only the lower byte of each 16-bit word is accessed if byte-wide memory is used.

For example, on core reads of the form:

```
R0 = W[P0] (Z) ;   //P0 points to a 16-bit aligned ASYNC memory
location
```

only the lower 8 bits of `R0` contain the actual value read from the 8-bit device.

For core writes of the form:

```
W[P0] = R0.L ;   //P0 points to a 16-bit aligned ASYNC memory
location
```

The 8-bit value to be written to the 8-bit device should be first loaded into the lower byte of `R0`.

Managing Clocks

Systems can drive the clock inputs with a crystal oscillator or a buffered, shaped clock derived from an external clock oscillator. The external clock connects to the processor's `CLKIN` pin. It is not possible to halt, change, or

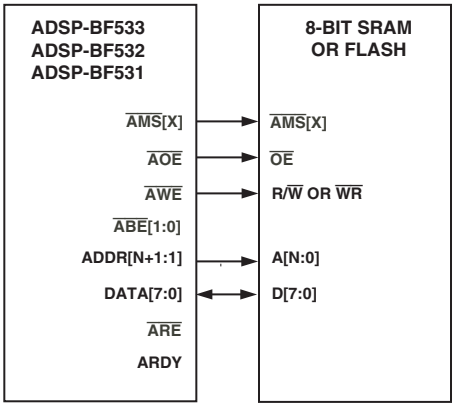


Figure 18-1. Interface to 8-Bit SRAM or Flash

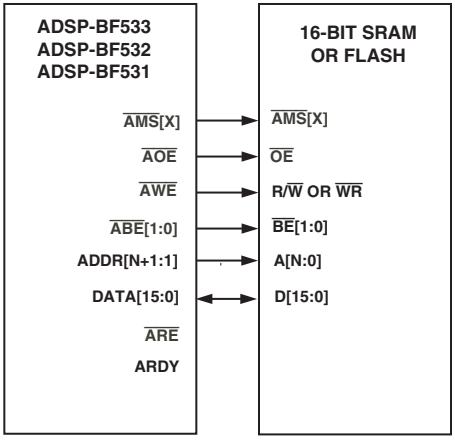


Figure 18-2. Interface to 16-Bit SRAM or Flash

operate CLKIN below the specified frequency during normal operation. The processor uses the clock input (CLKIN) to generate on-chip clocks. These include the core clock (CCLK) and the peripheral clock (SCLK).

Managing Core and System Clocks

The processor produces a multiplication of the clock input provided on the CLKIN pin to generate the PLL VCO clock. This VCO clock is divided to produce the core clock (CCLK) and the system clock (SCLK). The core clock is based on a divider ratio that is programmed via the CSEL bit settings in the PLL_DIV register. The system clock is based on a divider ratio that is programmed via the SSEL bit settings in the PLL_DIV register. For detailed information about how to set and change CCLK and SCLK frequencies, see Chapter 8, "Dynamic Power Management."

Configuring and Servicing Interrupts

A variety of interrupts are available. They include both core and peripheral interrupts. The processor assigns default core priorities to system-level interrupts. However, these system interrupts can be remapped via the System Interrupt Assignment registers (SIC_IARx). For more information, see "System Interrupt Assignment Registers (SIC_IARx)" on page 4-30.

The processor core supports nested and non-nested interrupts, as well as self-nested interrupts. For explanations of the various modes of servicing events, please see "Nesting of Interrupts" on page 4-50.

Semaphores

Semaphores provide a mechanism for communication between multiple processors or processes/threads running in the same system. They are used to coordinate resource sharing. For instance, if a process is using a particular resource and another process requires that same resource, it must wait until the first process signals that it is no longer using the resource. This signalling is accomplished via semaphores.

Semaphore coherency is guaranteed by using the Test and Set Byte (Atomic) instruction (TESTSET). The TESTSET instruction performs these functions.

- Loads the half word at memory location pointed to by a P-register. The P-register must be aligned on a half-word boundary.
- Sets CC if the value is equal to zero.
- Stores the value back in its original location (but with the most significant bit (MSB) of the low byte set to 1).

The events triggered by TESTSET are atomic operations. The bus for the memory where the address is located is acquired and not relinquished until the store operation completes. In multithreaded systems, the TESTSET instruction is required to maintain semaphore consistency.

To ensure that the store operation is flushed through any store or write buffers, issue an SSYNC instruction immediately after semaphore release.

The TESTSET instruction can be used to implement binary semaphores or any other type of mutual exclusion method. The TESTSET instruction supports a system-level requirement for a multicycle bus lock mechanism.

The processor restricts use of the TESTSET instruction to the external memory region only. Use of the TESTSET instruction to address any other area of the memory map may result in unreliable behavior.

Example Code for Query Semaphore

Listing 18-1 provides an example of a query semaphore that checks the availability of a shared resource.

Listing 18-1. Query Semaphore

```
/* Query semaphore. Denotes "Busy" if its value is nonzero. Wait
until free (or reschedule thread-- see note below). P0 holds
address of semaphore. */
QUERY:
TESTSET ( P0 ) ;
IF !CC JUMP QUERY ;
/* At this point, semaphore has been granted to current thread,
and all other contending threads are postponed because semaphore
value at [P0] is nonzero. Current thread could write thread_id to
semaphore location to indicate current owner of resource. */
R0.L = THREAD_ID ;
B[P0] = R0 ;
/* When done using shared resource, write a zero byte to [P0] */
R0 = 0 ;
B[P0] = R0 ;
SSYNC ;
/* NOTE: Instead of busy idling in the QUERY loop, one can use an
operating system call to reschedule the current thread. */
```

Data Delays, Latencies and Throughput

For detailed information on latencies and performance estimates on the DMA and External Memory buses, refer to Chapter 7, “Chip Bus Hierarchy.”

Bus Priorities

For an explanation of prioritization between the various internal buses, refer to Chapter 7, “Chip Bus Hierarchy.”

External Memory Design Issues

This section describes design issues related to external memory.

Example Asynchronous Memory Interfaces

This section shows glueless connections to 16-bit wide SRAM. Note this interface does not require external assertion of ARDY, since the internal wait state counter is sufficient for deterministic access times of memories.

Figure 18-3 shows the system interconnect required to support 16-bit memories. The programming model must ensure that data is only accessed on 16-bit boundaries.

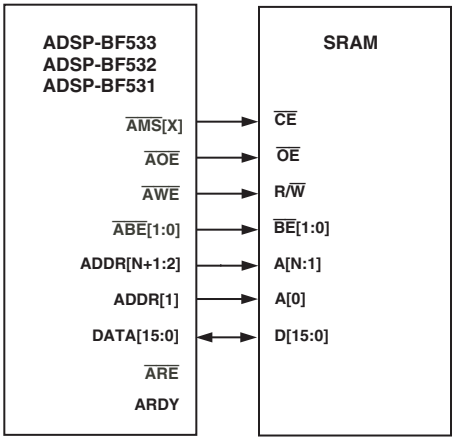


Figure 18-3. Interface to 16-Bit SRAM

Using SDRAMs Smaller Than 16M Byte

It is possible to use SDRAMs smaller than 16M byte on the ADSP-BF531/ADSP-BF532/ADSP-BF533, as long as it is understood how the resulting memory map is altered. Figure 18-4 shows an example where a 2M byte SDRAM (512K x 16 bits x 2 banks) is mapped to the external memory interface. In this example, there are 11 row addresses and 8 column addresses per bank. Referring to Table 17-5, the lowest available bank size (16M byte) for a device with 8 column addresses has 2 Bank Address lines (IA[23:22]) and 13 Row Address lines (IA[21:9]). Therefore, 1 processor Bank Address line and 2 Row Address lines are unused when hooking up to the SDRAM in the example. This causes aliasing in the processor's external memory map, which results in the SDRAM being mapped into noncontiguous regions of the processor's memory space.

Referring to the table in Figure 18-4, note that each line in the table corresponds to 2^{19} bytes, or 512K byte. Thus, the mapping of the 2M byte SDRAM is noncontiguous in Blackfin memory, as shown by the memory mapping in the left side of the figure.

Managing SDRAM Refresh During PLL Transitions

Since the processor's SDRAM refresh rate is based on the SCLK frequency, lowering SCLK after configuring SDRAM can result in an improper refresh rate, which could compromise the data stored in SDRAM. Raising SCLK after configuring SDRAM, however, would merely result in a less efficient use of SDRAM, since the processor would just refresh the memory at an unnecessarily fast rate.

In systems where SDRAM is used, the recommended procedure for changing the PLL VCO frequency is:

1. Issue an SSYNC instruction to ensure all pending memory operations have completed.

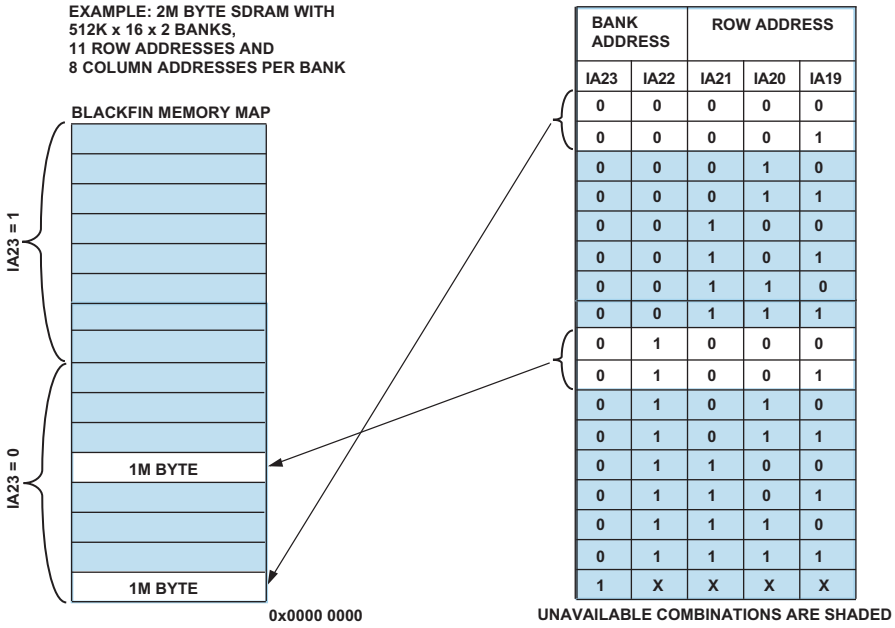


Figure 18-4. Using Small SDRAMs

2. Set the SDRAM to Self-Refresh mode by writing a `1` to the `SRFS` bit of `EBIU_SDGCTL`.

3. Execute the desired PLL programming sequence (refer to Chapter 8, "Dynamic Power Management," for details).

4. After the wakeup occurs that signifies the PLL has settled to the new `VCO` frequency, reprogram the SDRAM Refresh Rate Control register (`EBIU_SDRRC`) with a value appropriate to the new `SCLK` frequency.

5. Bring the SDRAM out of Self-Refresh mode by clearing the `SRFS` bit of `EBIU_SDGCTL`. If it is desired to change the SDRAM Mode register, write these changes to `EBIU_SDGCTL` as well, making sure the `PSSE` bit is set.

Changing the SCLK frequency using the SSEL bits in PLL_DIV, as opposed to actually changing the VCO frequency, should be done using these steps:

1. Issue an SSYNC instruction to ensure all pending memory operations have completed.
2. Set the SDRAM to Self-Refresh mode by writing a 1 to the SRFS bit of EBIU_SDGCTL.
3. Execute the desired write to the SSEL bits.
4. Reprogram the SDRAM Refresh Rate Control register (EBIU_SDRRC) with a value appropriate to the new SCLK frequency.
5. Bring the SDRAM out of Self-Refresh mode by clearing the SRFS bit of EBIU_SDGCTL. If it is desired to change the SDRAM Mode register, write these changes to EBIU_SDGCTL as well, making sure the PSSE bit is set.

Note steps 2 and 4 are not strictly necessary if changing SCLK to a higher value, but they should always be performed when decreasing SCLK.

For more information on SDRAM refresh, refer to "SDRAM Controller (SDC)" in Chapter 17, External Bus Interface Unit.

Avoiding Bus Contention

Because the three-stated data bus is shared by multiple devices in a system, be careful to avoid contention. Contention causes excessive power dissipation and can lead to device failure. Contention occurs during the time one device is getting off the bus and another is getting on. If the first device is slow to three-state and the second device is quick to drive, the devices contend.

There are two cases where contention can occur. The first case is a read followed by a write to the same memory space. In this case, the data bus drivers can potentially contend with those of the memory device addressed

by the read. The second case is back-to-back reads from two different memory spaces. In this case, the two memory devices addressed by the two reads can potentially contend at the transition between the two read operations.

To avoid contention, program the turnaround time (Bank Transition Time) appropriately in the Asynchronous Memory Bank Control registers. This feature allows software to set the number of clock cycles between these types of accesses on a bank-by-bank basis. Minimally, the External Bus Interface Unit (EBIU) provides one cycle for the transition to occur.

High Frequency Design Considerations

Because the processor can operate at very fast clock frequencies, signal integrity and noise problems must be considered for circuit board design and layout. The following sections discuss these topics and suggest various techniques to use when designing and debugging signal processing systems.

Point-to-Point Connections on Serial Ports

Although the serial ports may be operated at a slow rate, the output drivers still have fast edge rates and for longer distances the drivers may require source termination.

You can add a series termination resistor near the pin for point-to-point connections. Typically, serial port applications use this termination method when distances are greater than 6 inches. For details, see the reference source in "Recommended Reading" on page 18-14 for suggestions on transmission line termination. Also, see the processor data sheet for rise and fall time data for the output drivers.

Signal Integrity

The capacitive loading on high-speed signals should be reduced as much as possible. Loading of buses can be reduced by using a buffer for devices that operate with wait states (for example, DRAMs). This reduces the capacitance on signals tied to the zero-wait-state devices, allowing these signals to switch faster and reducing noise-producing current spikes.

Signal run length (inductance) should also be minimized to reduce ringing. Extra care should be taken with certain signals such as external memory, read, write, and acknowledge strobes.

Other recommendations and suggestions to promote signal integrity:

- Use more than one ground plane on the Printed Circuit Board (PCB) to reduce crosstalk. Be sure to use lots of vias between the ground planes. These planes should be in the center of the PCB.
- Keep critical signals such as clocks, strobes, and bus requests on a signal layer next to a ground plane and away from or laid out perpendicular to other non-critical signals to reduce crosstalk.
- Design for lower transmission line impedances to reduce crosstalk and to allow better control of impedance and delay.
- Experiment with the board and isolate crosstalk and noise issues from reflection issues. This can be done by driving a signal wire from a pulse generator and studying the reflections while other components and signals are passive.

Decoupling Capacitors and Ground Planes

Ground planes must be used for the ground and power supplies. The capacitors should be placed very close to the `VDDEXT` and `VDDINT` pins of the package as shown in Figure 18-5. Use short and fat traces for this. The ground end of the capacitors should be tied directly to the ground plane

inside the package footprint of the processor (underneath it, on the bottom of the board), not outside the footprint. A surface-mount capacitor is recommended because of its lower series inductance.

Connect the power plane to the power supply pins directly with minimum trace length. The ground planes must not be densely perforated with vias or traces as their effectiveness is reduced. In addition, there should be several large tantalum capacitors on the board.

Designs can use either bypass placement case or combinations of the two. Designs should try to minimize signal feedthroughs that perforate the ground plane.

Oscilloscope Probes

When making high-speed measurements, be sure to use a “bayonet” type or similarly short (< 0.5 inch) ground clip, attached to the tip of the oscilloscope probe. The probe should be a low-capacitance active probe with 3 pF or less of loading. The use of a standard ground clip with 4 inches of ground lead causes ringing to be seen on the displayed trace and makes the signal appear to have excessive overshoot and undershoot. To see the signals accurately, a 1 GHz or better sampling oscilloscope is needed.

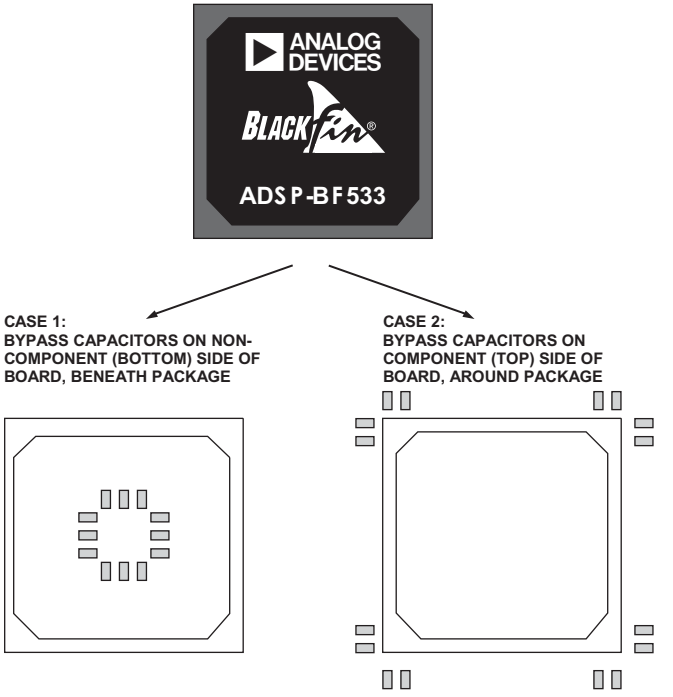


Figure 18-5. Bypass Capacitor Placement

Recommended Reading

For more information, refer to *High-Speed Digital Design: A Handbook of Black Magic*, Johnson & Graham, Prentice Hall, Inc., ISBN 0-13-395724-1.

This book is a technical reference that covers the problems encountered in state of the art, high-frequency digital circuit design. It is an excellent source of information and practical ideas. Topics covered in the book include:

- High-speed Properties of Logic Gates
- Measurement Techniques
- Transmission Lines
- Ground Planes and Layer Stacking
- Terminations
- Vias
- Power Systems
- Connectors
- Ribbon Cables
- Clock Distribution
- Clock Oscillators

A BLACKFIN PROCESSOR CORE MMR ASSIGNMENTS

The Blackfin processor's memory-mapped registers (MMRs) are in the address range 0xFFE0 0000 – 0xFFFF FFFF.

All core MMRs must be accessed with a 32-bit read or write access.

This appendix lists core MMR addresses and register names. To find more information about an MMR, refer to the page shown in the "See Section" column. When viewing the PDF version of this document, click a reference in the "See Section" column to jump to additional information about the MMR.

L1 Data Memory Controller Registers

L1 Data Memory Controller registers (0xFFE0 0000 – 0xFFE0 0404)

Table A-1. L1 Data Memory Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFE0 0004	DMEM_CONTROL	"DMEM_CONTROL Register" on page 6-26
0xFFE0 0008	DCPLB_STATUS	"DCPLB_STATUS and ICPLB_STATUS Registers" on page 6-61
0xFFE0 000C	DCPLB_FAULT_ADDR	"DCPLB_FAULT_ADDR and ICPLB_FAULT_ADDR Registers" on page 6-63
0xFFE0 0100	DCPLB_ADDR0	"DCPLB_ADDRx Registers" on page 6-59

Table A-1. L1 Data Memory Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFE0 0104	DCPLB_ADDR1	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0108	DCPLB_ADDR2	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 010C	DCPLB_ADDR3	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0110	DCPLB_ADDR4	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0114	DCPLB_ADDR5	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0118	DCPLB_ADDR6	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 011C	DCPLB_ADDR7	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0120	DCPLB_ADDR8	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0124	DCPLB_ADDR9	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0128	DCPLB_ADDR10	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 012C	DCPLB_ADDR11	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0130	DCPLB_ADDR12	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0134	DCPLB_ADDR13	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0138	DCPLB_ADDR14	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 013C	DCPLB_ADDR15	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0200	DCPLB_DATA0	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0204	DCPLB_DATA1	"DCPLB_DATAx Registers" on page 6-57
0 xFFE0 0208	DCPLB_DATA2	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 020C	DCPLB_DATA3	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0210	DCPLB_DATA4	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0214	DCPLB_DATA5	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0218	DCPLB_DATA6	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 021C	DCPLB_DATA7	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0220	DCPLB_DATA8	"DCPLB_DATAx Registers" on page 6-57

Table A-1. L1 Data Memory Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFE0 0224	DCPLB_DATA9	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0228	DCPLB_DATA10	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 022C	DCPLB_DATA11	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0230	DCPLB_DATA12	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0234	DCPLB_DATA13	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0238	DCPLB_DATA14	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 023C	DCPLB_DATA15	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0300	DTEST_COMMAND	"DTEST_COMMAND Register" on page 6-42
0xFFE0 0400	DTEST_DATA0	"DTEST_DATA0 Register" on page 6-45
0xFFE0 0404	DTEST_DATA1	"DTEST_DATA1 Register" on page 6-44

L1 Instruction Memory Controller Registers

L1 Instruction Memory Controller registers (0xFFE0 1004 – 0xFFE0 1404)

Table A-2. L1 Instruction Memory Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFE0 1004	IMEM_CONTROL	"IMEM_CONTROL Register" on page 6-8
0xFFE0 1008	ICPLB_STATUS	"DCPLB_STATUS and ICPLB_STATUS Registers" on page 6-61
0xFFE0 100C	ICPLB_FAULT_ADDR	"DCPLB_FAULT_ADDR and ICPLB_FAULT_ADDR Registers" on page 6-63
0xFFE0 1100	ICPLB_ADDR0	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1104	ICPLB_ADDR1	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1108	ICPLB_ADDR2	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 110C	ICPLB_ADDR3	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1110	ICPLB_ADDR4	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1114	ICPLB_ADDR5	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1118	ICPLB_ADDR6	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 111C	ICPLB_ADDR7	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1120	ICPLB_ADDR8	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1124	ICPLB_ADDR9	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1128	ICPLB_ADDR10	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 112C	ICPLB_ADDR11	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1130	ICPLB_ADDR12	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1134	ICPLB_ADDR13	"ICPLB_ADDRx Registers" on page 6-60

Table A-2. L1 Instruction Memory Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFE0 1138	ICPLB_ADDR14	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 113C	ICPLB_ADDR15	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1200	ICPLB_DATA0	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1204	ICPLB_DATA1	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1208	ICPLB_DATA2	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 120C	ICPLB_DATA3	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1210	ICPLB_DATA4	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1214	ICPLB_DATA5	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1218	ICPLB_DATA6	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 121C	ICPLB_DATA7	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1220	ICPLB_DATA8	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1224	ICPLB_DATA9	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1228	ICPLB_DATA10	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 122C	ICPLB_DATA11	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1230	ICPLB_DATA12	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1234	ICPLB_DATA13	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1238	ICPLB_DATA14	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 123C	ICPLB_DATA15	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1300	ITEST_COMMAND	"ITEST_COMMAND Register" on page 6-23
0XFFE0 1400	ITEST_DATA0	"ITEST_DATA0 Register" on page 6-25
0XFFE0 1404	ITEST_DATA1	"ITEST_DATA1 Register" on page 6-24

Interrupt Controller Registers

Interrupt Controller registers (0xFFE0 2000 – 0xFFE0 2110)

Table A-3. Interrupt Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFE0 2000	EVT0 (EMU)	"Core Event Vector Table" on page 4-38
0xFFE0 2004	EVT1 (RST)	"Core Event Vector Table" on page 4-38
0xFFE0 2008	EVT2 (NMI)	"Core Event Vector Table" on page 4-38
0xFFE0 200C	EVT3 (EVX)	"Core Event Vector Table" on page 4-38
0xFFE0 2010	EVT4	"Core Event Vector Table" on page 4-38
0xFFE0 2014	EVT5 (IVHW)	"Core Event Vector Table" on page 4-38
0xFFE0 2018	EVT6 (TMR)	"Core Event Vector Table" on page 4-38
0xFFE0 201C	EVT7 (IVG7)	"Core Event Vector Table" on page 4-38
0xFFE0 2020	EVT8 (IVG8)	"Core Event Vector Table" on page 4-38
0xFFE0 2024	EVT9 (IVG9)	"Core Event Vector Table" on page 4-38
0xFFE0 2028	EVT10 (IVG10)	"Core Event Vector Table" on page 4-38
0xFFE0 202C	EVT11 (IVG11)	"Core Event Vector Table" on page 4-38
0xFFE0 2030	EVT12 (IVG12)	"Core Event Vector Table" on page 4-38

Table A-3. Interrupt Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFE0 2034	EVT13 (IVG13)	"Core Event Vector Table" on page 4-38
0xFFE0 2038	EVT14 (IVG14)	"Core Event Vector Table" on page 4-38
0xFFE0 203C	EVT15 (IVG15)	"Core Event Vector Table" on page 4-38
0xFFE0 2104	IMASK	"IMASK Register" on page 4-34
0xFFE0 2108	IPEND	"IPEND Register" on page 4-36
0xFFE0 2110	IPRIO	"IPRIO Register and Write Buffer Depth" on page 6-38
0xFFE0 210C	ILAT	"ILAT Register" on page 4-35

Core Timer Registers

Core Timer registers (0xFFE0 3000 – 0xFFE0 300C)

Table A-4. Core Timer Registers

Memory-Mapped Address	Register Name	See Section
0xFFE0 3000	TCNTL	"TCNTL Register" on page 15-46
0xFFE0 3004	TPERIOD	"TPERIOD Register" on page 15-48
0xFFE0 3008	TSCALE	"TSCALE Register" on page 15-49
0xFFE0 300C	TCOUNT	"TCOUNT Register" on page 15-48

Debug, MP, and Emulation Unit Registers

Debug, MP, and Emulation Unit registers (0xFFE0 5000 – 0xFFE0 5008)

For further details about these registers, see Chapter 21, “Debug” of the *Blackfin Processor Programming Reference*.

Table A-5. Debug and Emulation Unit Registers

Memory-Mapped Address	Register Name
0xFFE0 5000	DSPID

Trace Unit Registers

Trace Unit registers (0xFFE0 6000 – 0xFFE0 6100)

For further details about these registers, see Chapter 21, “Debug” of the *Blackfin Processor Programming Reference*.

Table A-6. Trace Unit Registers

Memory-Mapped Address	Register Name
0xFFE0 6000	TBUFCTL
0xFFE0 6004	TBUFSTAT
0xFFE0 6100	TBUF

Watchpoint and Patch Registers

Watchpoint and Patch registers (0xFFE0 7000 – 0xFFE0 7200)

For further details about these registers, see Chapter 21, “Debug” of the *Blackfin Processor Programming Reference*.

Table A-7. Watchpoint and Patch Registers

Memory-Mapped Address	Register Name
0xFFE0 7000	WPIACTL
0xFFE0 7040	WPIA0
0xFFE0 7044	WPIA1
0xFFE0 7048	WPIA2
0xFFE0 704C	WPIA3
0xFFE0 7050	WPIA4
0xFFE0 7054	WPIA5
0xFFE0 7080	WPIACNT0
0xFFE0 7084	WPIACNT1
0xFFE0 7088	WPIACNT2
0xFFE0 708C	WPIACNT3
0xFFE0 7090	WPIACNT4
0xFFE0 7094	WPIACNT5
0xFFE0 7100	WPDACTL
0xFFE0 7140	WPDA0
0xFFE0 7144	WPDA1
0xFFE0 7180	WPDACNT0

Table A-7. Watchpoint and Patch Registers (Cont'd)

Memory-Mapped Address	Register Name
0xFFE0 7184	WPDACNT1
0xFFE0 7200	WPSTAT

Performance Monitor Registers

Performance Monitor registers (0xFFE0 8000 – 0xFFE0 8104)

For further details about these registers, see Chapter 21, "Debug" of the *Blackfin Processor Programming Reference*.

Table A-8. Performance Monitor Registers

Memory-Mapped Address	Register Name
0xFFE0 8000	PFCTL
0xFFE0 8100	PFCNTR0
0xFFE0 8104	PFCNTR1

B SYSTEM MMR ASSIGNMENTS

These notes provide general information about the system memory-mapped registers (MMRs):

- The system MMR address range is 0xFFC0 0000 – 0xFFDF FFFF.
- All system MMRs are either 16 bits or 32 bits wide. MMRs that are 16 bits wide must be accessed with 16-bit read or write operations. MMRs that are 32 bits wide must be accessed with 32-bit read or write operations. Check the description of the MMR to determine whether a 16-bit or a 32-bit access is required.
- All system MMR space that is not defined in this appendix is reserved for internal use only.

This appendix lists MMR addresses and register names. To find more information about an MMR, refer to the page shown in the “See Section” column. When viewing the PDF version of this document, click a reference in the “See Section” column to jump to additional information about the MMR.

Dynamic Power Management Registers

Dynamic Power Management registers (0xFFC0 0000 – 0xFFC0 00FF)

Table B-1. Dynamic Power Management Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0000	PLL_CTL	"PLL_CTL Register" on page 8-7
0xFFC0 0004	PLL_DIV	"PLL_DIV Register" on page 8-7
0xFFC0 0008	VR_CTL	"VR_CTL Register" on page 8-26
0xFFC0 000C	PLL_STAT	"PLL_STAT Register" on page 8-10
0xFFC0 0010	PLL_LOCKCNT	"PLL_LOCKCNT Register" on page 8-11

System Reset and Interrupt Control Registers

System Reset and Interrupt Controller registers (0xFFC0 0100 – 0xFFC0 01FF)

Table B-2. System Interrupt Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0100	SWRST	"SWRST Register" on page 3-15
0xFFC0 0104	SYSCR	"SYSCR Register" on page 3-14
0xFFC0 010C	SIC_IMASK	"SIC_IMASK Register" on page 4-29
0xFFC0 0110	SIC_IAR0	"System Interrupt Assignment Registers (SIC_IARx)" on page 4-30
0xFFC0 0114	SIC_IAR1	"System Interrupt Assignment Registers (SIC_IARx)" on page 4-30

Table B-2. System Interrupt Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 0118	SIC_IAR2	"System Interrupt Assignment Registers (SIC_IARx)" on page 4-30
0xFFC0 0120	SIC_ISR	"SIC_ISR Register" on page 4-28
0xFFC0 0124	SIC_IWR	"SIC_IWR Register" on page 4-26

Watchdog Timer Registers

Watchdog Timer registers (0xFFC0 0200 – 0xFFC0 02FF)

Table B-3. Watchdog Timer Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0200	WDOG_CTL	"WDOG_CTL Register" on page 15-53
0xFFC0 0204	WDOG_CNT	"WDOG_CNT Register" on page 15-50
0xFFC0 0208	WDOG_STAT	"WDOG_STAT Register" on page 15-51

Real-Time Clock Registers

Real-Time Clock registers (0xFFC0 0300 – 0xFFC0 03FF)

Table B-4. Real-Time Clock Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0300	RTC_STAT	"RTC_STAT Register" on page 16-13
0xFFC0 0304	RTC_ICTL	"RTC_ICTL Register" on page 16-13
0xFFC0 0308	RTC_ISTAT	"RTC_ISTAT Register" on page 16-15

Table B-4. Real-Time Clock Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 030C	RTC_SWCNT	"RTC_SWCNT Register" on page 16-15
0xFFC0 0310	RTC_ALARM	"RTC_ALARM Register" on page 16-17
0xFFC0 0314	RTC_PREN	"RTC_PREN Register" on page 16-18

Parallel Peripheral Interface (PPI) Registers

Parallel Peripheral Interface (PPI) registers (0xFFC0 1000 – 0xFFC0 10FF)

Table B-5. Parallel Peripheral Interface (PPI) Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 1000	PPI_CONTROL	"PPI_CONTROL Register" on page 11-3
0xFFC0 1004	PPI_STATUS	"PPI_STATUS Register" on page 11-8
0xFFC0 1008	PPI_COUNT	"PPI_COUNT Register" on page 11-11
0xFFC0 100C	PPI_DELAY	"PPI_DELAY Register" on page 11-10
0xFFC0 1010	PPI_FRAME	"PPI_FRAME Register" on page 11-12

UART Controller Registers

UART Controller registers (0xFFC0 0400 – 0xFFC0 04FF)

Table B-6. UART Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0400	UART_THR	"UART_THR Register" on page 13-6
0xFFC0 0400	UART_RBR	"UART_RBR Register" on page 13-7
0xFFC0 0400	UART_DLL	"UART_DLL and UART_DLH Registers" on page 13-11
0xFFC0 0404	UART_DLH	"UART_DLL and UART_DLH Registers" on page 13-11
0xFFC0 0404	UART_IER	"UART_IER Register" on page 13-8
0xFFC0 0408	UART_IIR	"UART_IIR Register" on page 13-10
0xFFC0 040C	UART_LCR	"UART_LCR Register" on page 13-3
0xFFC0 0410	UART_MCR	"UART_MCR Register" on page 13-4
0xFFC0 0414	UART_LSR	"UART_LSR Register" on page 13-5
0xFFC0 041C	UART_SCR	"UART_SCR Register" on page 13-13
0xFFC0 0424	UART_GCTL	"UART_GCTL Register" on page 13-14

SPI Controller Registers

SPI Controller registers (0xFFC0 0500 – 0xFFC0 05FF)

Table B-7. SPI Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0500	SPI_CTL	"SPI_CTL Register" on page 10-8
0xFFC0 0504	SPI_FLG	"SPI_FLG Register" on page 10-11
0xFFC0 0508	SPI_STAT	"SPI_STAT Register" on page 10-15
0xFFC0 050C	SPI_TDBR	"SPI_TDBR Register" on page 10-17
0xFFC0 0510	SPI_RDBR	"SPI_RDBR Register" on page 10-18
0xFFC0 0514	SPI_BAUD	"SPI_BAUD Register" on page 10-7
0xFFC0 0518	SPI_SHADOW	"SPI_SHADOW Register" on page 10-18

Timer Registers

Timer registers (0xFFC0 0600 – 0xFFC0 06FF)

Table B-8. Timer Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0600	TIMER0_CONFIG	"TIMERx_CONFIG Registers" on page 15-8
0xFFC0 0604	TIMER0_COUNTER	"TIMERx_COUNTER Registers" on page 15-9
0xFFC0 0608	TIMER0_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 15-10
0xFFC0 060C	TIMER0_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 15-10

Table B-8. Timer Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 0610	TIMER1_CONFIG	"TIMERx_CONFIG Registers" on page 15-8
0xFFC0 0614	TIMER1_COUNTER	"TIMERx_COUNTER Registers" on page 15-9
0xFFC0 0618	TIMER1_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 15-10
0xFFC0 061C	TIMER1_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 15-10
0xFFC0 0620	TIMER2_CONFIG	"TIMERx_CONFIG Registers" on page 15-8
0xFFC0 0624	TIMER2_COUNTER	"TIMERx_COUNTER Registers" on page 15-9
0xFFC0 0628	TIMER2_PERIOD	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 15-10
0xFFC0 062C	TIMER2_WIDTH	"TIMERx_PERIOD and TIMERx_WIDTH Registers" on page 15-10
0xFFC0 0640	TIMER_ENABLE	"TIMER_ENABLE Register" on page 15-4
0xFFC0 0644	TIMER_DISABLE	"TIMER_DISABLE Register" on page 15-5
0xFFC0 0648	TIMER_STATUS	"TIMER_STATUS Register" on page 15-6

Programmable Flag Registers

Programmable Flag registers (0xFFC0 0700 – 0xFFC0 07FF)

Table B-9. Programmable Flags Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0700	FIO_FLAG_D	"FIO_FLAG_D Register" on page 14-8
0xFFC0 0704	FIO_FLAG_C	"FIO_FLAG_S, FIO_FLAG_C, and FIO_FLAG_T Registers" on page 14-8
0xFFC0 0708	FIO_FLAG_S	"FIO_FLAG_S, FIO_FLAG_C, and FIO_FLAG_T Registers" on page 14-8
0xFFC0 070C	FIO_FLAG_T	"FIO_FLAG_S, FIO_FLAG_C, and FIO_FLAG_T Registers" on page 14-8
0xFFC0 0710	FIO_MASKA_D	"FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T, FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, FIO_MASKB_T Registers" on page 14-11
0xFFC0 0714	FIO_MASKA_C	"FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T, FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, FIO_MASKB_T Registers" on page 14-11
0xFFC0 0718	FIO_MASKA_S	"FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T, FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, FIO_MASKB_T Registers" on page 14-11
0xFFC0 071C	FIO_MASKA_T	"FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T, FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, FIO_MASKB_T Registers" on page 14-11

Table B-9. Programmable Flags Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 0720	FIO_MASKB_D	"FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T, FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, FIO_MASKB_T Registers" on page 14-11
0xFFC0 0724	FIO_MASKB_C	"FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T, FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, FIO_MASKB_T Registers" on page 14-11
0xFFC0 0728	FIO_MASKB_S	"FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T, FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, FIO_MASKB_T Registers" on page 14-11
0xFFC0 072C	FIO_MASKB_T	"FIO_MASKA_D, FIO_MASKA_C, FIO_MASKA_S, FIO_MASKA_T, FIO_MASKB_D, FIO_MASKB_C, FIO_MASKB_S, FIO_MASKB_T Registers" on page 14-11
0xFFC0 0730	FIO_DIR	"FIO_DIR Register" on page 14-5
0xFFC0 0734	FIO_POLAR	"FIO_POLAR Register" on page 14-18
0xFFC0 0738	FIO_EDGE	"FIO_EDGE Register" on page 14-18
0xFFC0 073C	FIO_BOTH	"FIO_BOTH Register" on page 14-20
0xFFC0 0740	FIO_INEN	"FIO_INEN Register" on page 14-21

SPORT0 Controller Registers

SPORT0 Controller registers (0xFFC0 0800 – 0xFFC0 08FF)

Table B-10. SPORT0 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0800	SPORT0_TCR1	"SPORTx_TCR1 and SPORTx_TCR2 Registers" on page 12-11
0xFFC0 0804	SPORT0_TCR2	"SPORTx_TCR1 and SPORTx_TCR2 Registers" on page 12-11
0xFFC0 0808	SPORT0_TCLKDIV	"SPORTx_TCLKDIV and SPORTx_RCLKDIV Registers" on page 12-29
0xFFC0 080C	SPORT0_TFSDIV	"SPORTx_TFSDIV and SPORTx_RFSDIV Register" on page 12-30
0xFFC0 0810	SPORT0_TX	"SPORTx_TX Register" on page 12-22
0xFFC0 0818	SPORT0_RX	"SPORTx_RX Register" on page 12-24
0xFFC0 0820	SPORT0_RCR1	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 12-16
0xFFC0 0824	SPORT0_RCR2	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 12-16
0xFFC0 0828	SPORT0_RCLKDIV	"SPORTx_TCLKDIV and SPORTx_RCLKDIV Registers" on page 12-29
0xFFC0 082C	SPORT0_RFSDIV	"SPORTx_TFSDIV and SPORTx_RFSDIV Register" on page 12-30
0xFFC0 0830	SPORT0_STAT	"SPORTx_STAT Register" on page 12-27
0xFFC0 0834	SPORT0_CHNL	"SPORTx_CHNL Register" on page 12-57
0xFFC0 0838	SPORT0_MCMC1	"SPORTx_MCMCn Registers" on page 12-51
0xFFC0 083C	SPORT0_MCMC2	"SPORTx_MCMCn Registers" on page 12-51
0xFFC0 0840	SPORT0_MTCS0	"SPORTx_MTCSn Registers" on page 12-62
0xFFC0 0844	SPORT0_MTCS1	"SPORTx_MTCSn Registers" on page 12-62

Table B-10. SPORT0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 0848	SPORT0_MTCS2	"SPORTx_MTCSn Registers" on page 12-62
0xFFC0 084C	SPORT0_MTCS3	"SPORTx_MTCSn Registers" on page 12-62
0xFFC0 0850	SPORT0_MRCS0	"SPORTx_MRCSn Registers" on page 12-60
0xFFC0 0854	SPORT0_MRCS1	"SPORTx_MRCSn Registers" on page 12-60
0xFFC0 0858	SPORT0_MRCS2	"SPORTx_MRCSn Registers" on page 12-60
0xFFC0 085C	SPORT0_MRCS3	"SPORTx_MRCSn Registers" on page 12-60

SPORT1 Controller Registers

SPORT1 Controller registers (0xFFC0 0900 – 0xFFC0 09FF)

Table B-11. SPORT 1 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0900	SPORT1_TCR1	"SPORTx_TCR1 and SPORTx_TCR2 Registers" on page 12-11
0xFFC0 0904	SPORT1_TCR2	"SPORTx_TCR1 and SPORTx_TCR2 Registers" on page 12-11
0xFFC0 0908	SPORT1_TCLKDIV	"SPORTx_TCLKDIV and SPORTx_RCLKDIV Registers" on page 12-29
0xFFC0 090C	SPORT1_TFSDIV	"SPORTx_TFSDIV and SPORTx_RFSDIV Register" on page 12-30
0xFFC0 0910	SPORT1_TX	"SPORTx_TX Register" on page 12-22
0xFFC0 0918	SPORT1_RX	"SPORTx_RX Register" on page 12-24
0xFFC0 0920	SPORT1_RCR1	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 12-16

Table B-11. SPORT 1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 0924	SPORT1_RCR2	"SPORTx_RCR1 and SPORTx_RCR2 Registers" on page 12-16
0xFFC0 0928	SPORT1_RCLKDIV	"SPORTx_TCLKDIV and SPORTx_RCLKDIV Registers" on page 12-29
0xFFC0 092C	SPORT1_RFSDIV	"SPORTx_TFSDIV and SPORTx_RFSDIV Register" on page 12-30
0xFFC0 0930	SPORT1_STAT	"SPORTx_STAT Register" on page 12-27
0xFFC0 0934	SPORT1_CHNL	"SPORTx_CHNL Register" on page 12-57
0xFFC0 0938	SPORT1_MCMC1	"SPORTx_MCMCn Registers" on page 12-51
0xFFC0 093C	SPORT1_MCMC2	"SPORTx_MCMCn Registers" on page 12-51
0xFFC0 0940	SPORT1_MTCS0	"SPORTx_MRCSn Registers" on page 12-60
0xFFC0 0944	SPORT1_MTCS1	"SPORTx_MRCSn Registers" on page 12-60
0xFFC0 0948	SPORT1_MTCS2	"SPORTx_MRCSn Registers" on page 12-60
0xFFC0 094C	SPORT1_MTCS3	"SPORTx_MRCSn Registers" on page 12-60
0xFFC0 0950	SPORT1_MRCS0	"SPORTx_MRCSn Registers" on page 12-60
0xFFC0 0954	SPORT1_MRCS1	"SPORTx_MRCSn Registers" on page 12-60
0xFFC0 0958	SPORT1_MRCS2	"SPORTx_MRCSn Registers" on page 12-60
0xFFC0 095C	SPORT1_MRCS3	"SPORTx_MRCSn Registers" on page 12-60

DMA/Memory DMA Control Registers

DMA Control registers (0xFFC0 0B00 – 0xFFC0 0FFF)

Table B-12. DMA Traffic Control Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0B0C	DMA_TC_PER	"DMA_TC_PER and DMA_TC_CNT Registers" on page 9-55
0xFFC0 0B10	DMA_TC_CNT	"DMA_TC_PER and DMA_TC_CNT Registers" on page 9-55

Since each DMA channel has an identical MMR set, with fixed offsets from the base address associated with that DMA channel, it is convenient to view the MMR information as provided in Table B-13 and Table B-14. Table B-13 identifies the base address of each DMA channel, as well as the register prefix that identifies the channel. Table B-14 then lists the register suffix and provides its offset from the Base Address.

As an example, the DMA Channel 0 Y_MODIFY register is called `DMA0_Y_MODIFY`, and its address is 0xFFC0 0C1C. Likewise, the Memory DMA Stream 0 Source Current Address register is called `MDMA_S0_CURR_ADDR`, and its address is 0xFFC0 0E64.

Table B-13. DMA Channel Base Addresses

DMA Channel Identifier	MMR Base Address	Register Prefix
0	0xFFC0 0C00	DMA0_
1	0xFFC0 0C40	DMA1_
2	0xFFC0 0C80	DMA2_
3	0xFFC0 0CC0	DMA3_
4	0xFFC0 0D00	DMA4_

Table B-13. DMA Channel Base Addresses (Cont'd)

DMA Channel Identifier	MMR Base Address	Register Prefix
5	0xFFC0 0D40	DMA5_
6	0xFFC0 0D80	DMA6_
7	0xFFC0 0DC0	DMA7_
Mem DMA Stream 0 Destination	0xFFCO 0E00	MDMA_D0_
Mem DMA Stream 0 Source	0xFFC0 0E40	MDMA_S0_
Mem DMA Stream 1 Destination	0xFFCO 0E80	MDMA_D1_
Mem DMA Stream 1 Source	0xFFC0 0EC0	MDMA_S1_

Table B-14. DMA Register Suffix and Offset

Register Suffix	Offset From Base	See Section
NEXT_DESC_PTR	0x00	"DMAx_NEXT_DESC_PTR/MDMA_yy_NEXT_DESC_PTR Register" on page 9-8
START_ADDR	0x04	"DMAx_START_ADDR/MDMA_yy_START_ADDR Register" on page 9-10
CONFIG	0x08	"DMAx_CONFIG/MDMA_yy_CONFIG Register" on page 9-12
X_COUNT	0x10	"DMAx_X_COUNT/MDMA_yy_X_COUNT Register" on page 9-16
X_MODIFY	0x14	"DMAx_X_MODIFY/MDMA_yy_X_MODIFY Register" on page 9-17
Y_COUNT	0x18	"DMAx_Y_COUNT/MDMA_yy_Y_COUNT Register" on page 9-19
Y_MODIFY	0x1C	"DMAx_Y_MODIFY/MDMA_yy_Y_MODIFY Register" on page 9-20

Table B-14. DMA Register Suffix and Offset (Cont'd)

Register Suffix	Offset From Base	See Section
CURR_DESC_PTR	0x20	"DMAx_CURR_DESC_PTR/MDMA_yy_CURR_DESC_PTR Register" on page 9-22
CURR_ADDR	0x24	"DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR Register" on page 9-24
IRQ_STATUS	0x28	"DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS Register" on page 9-31
PERIPHERAL_MAP	0x2C	"DMAx_PERIPHERAL_MAP/MDMA_yy_PERIPHERAL_MAP Register" on page 9-28
CURR_X_COUNT	0x30	"DMAx_CURR_X_COUNT/MDMA_yy_CURR_X_COUNT Register" on page 9-25
CURR_Y_COUNT	0x38	"DMAx_CURR_Y_COUNT/MDMA_yy_CURR_Y_COUNT Register" on page 9-27

External Bus Interface Unit Registers

External Bus Interface Unit registers (0xFFC0 0A00 – 0xFFC0 0AFF)

Table B-15. External Bus Interface Unit Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0A00	EBIU_AMGCTL	"EBIU_AMGCTL Register" on page 17-10
0xFFC0 0A04	EBIU_AMBCTL0	"EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers" on page 17-12
0xFFC0 0A08	EBIU_AMBCTL1	"EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers" on page 17-12
0xFFC0 0A10	EBIU_SDGCTL	"EBIU_SDGCTL Register" on page 17-32
0xFFC0 0A14	EBIU_SDBCTL	"EBIU_SDBCTL Register" on page 17-44

Table B-15. External Bus Interface Unit Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 0A18	EBIU_SDRRC	"EBIU_SDRRC Register" on page 17-47
0xFFC0 0A1C	EBIU_SDSTAT	"EBIU_SDSTAT Register" on page 17-46

C TEST FEATURES

This chapter discusses the test features of the processor.

JTAG Standard

The processor is fully compatible with the IEEE 1149.1 standard, also known as the Joint Test Action Group (JTAG) standard.

The JTAG standard defines circuitry that may be built to assist in the test, maintenance, and support of assembled printed circuit boards.The circuitry includes a standard interface through which instructions and test data are communicated. A set of test features is defined, including a Boundary-Scan register, such that the component can respond to a minimum set of instructions designed to help test printed circuit boards.

The standard defines test logic that can be included in an integrated circuit to provide standardized approaches to:

- Testing the interconnections between integrated circuits once they have been assembled onto a printed circuit board
- Testing the integrated circuit itself
- Observing or modifying circuit activity during normal component operation

The test logic consists of a Boundary-Scan register and other building blocks. The test logic is accessed through a Test Access Port (TAP).

Full details of the JTAG standard can be found in the document *IEEE Standard Test Access Port and Boundary-Scan Architecture*, ISBN 1-55937-350-4.

Boundary-Scan Architecture

The boundary-scan test logic consists of:

- A TAP comprised of five pins (see Table C-1)
- A TAP controller that controls all sequencing of events through the test registers
- An Instruction register (IR) that interprets 5-bit instruction codes to select the test mode that performs the desired test operation
- Several data registers defined by the JTAG standard

Table C-1. Test Access Port Pins

Pin Name	Input/Output	Description
TDI	Input	Test Data Input
TMS	Input	Test Mode Select
TCK	Input	Test Clock
$\overline{\text{TRST}}$	Input	Test Reset
TDO	Output	Test Data Out

The TAP controller is a synchronous, 16-state, finite-state machine controlled by the TCK and TMS pins. Transitions to the various states in the diagram occur on the rising edge of TCK and are defined by the state of the TMS pin, here denoted by either a logic 1 or logic 0 state. For full details of the operation, see the JTAG standard.

Figure C-1 shows the state diagram for the TAP controller.

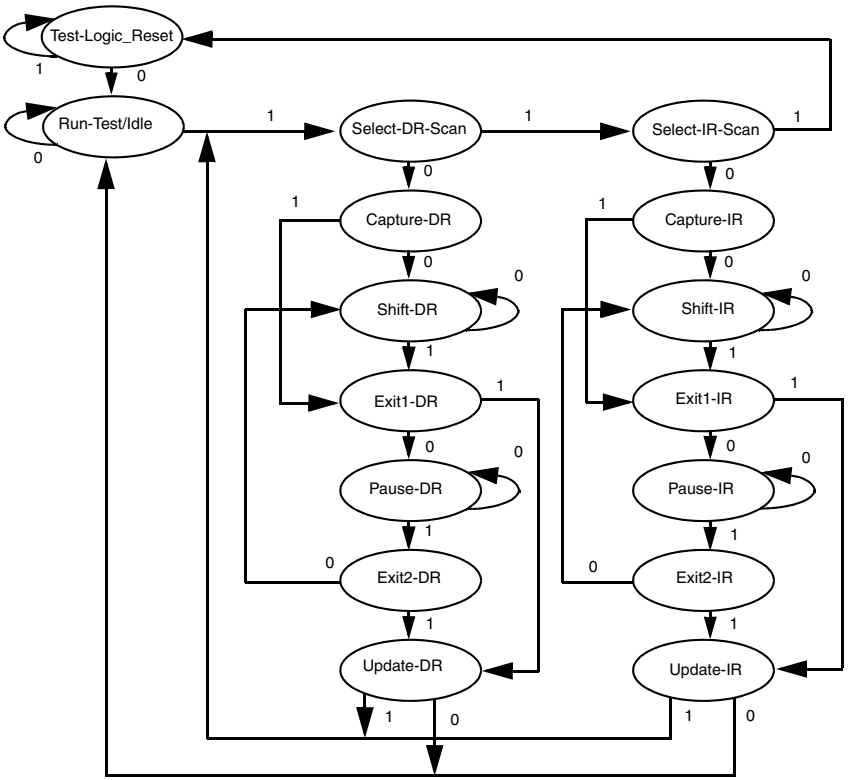


Figure C-1. TAP Controller State Diagram

Note:

- The TAP controller enters the Test-Logic-Reset state when TMS is held high after five TCK cycles.
- The TAP controller enters the Test-Logic-Reset state when TRST is asynchronously asserted.

- An external system reset does not affect the state of the TAP controller, nor does the state of the TAP controller affect an external system reset.

Instruction Register

The Instruction register is five bits wide and accommodates up to 32 boundary-scan instructions.

The Instruction register holds both public and private instructions. The JTAG standard requires some of the public instructions; other public instructions are optional. Private instructions are reserved for the manufacturer's use.

The binary decode column of Table C-2 lists the decode for the public instructions. The register column lists the serial scan paths.

Table C-2. Decode for Public JTAG-Scan Instructions

Instruction Name	Binary Decode 01234	Register
EXTEST	00000	Boundary-Scan
SAMPLE/PRELOAD	10000	Boundary-Scan
BYPASS	11111	Bypass

Figure C-2 shows the instruction bit scan ordering for the paths shown in Table C-2.

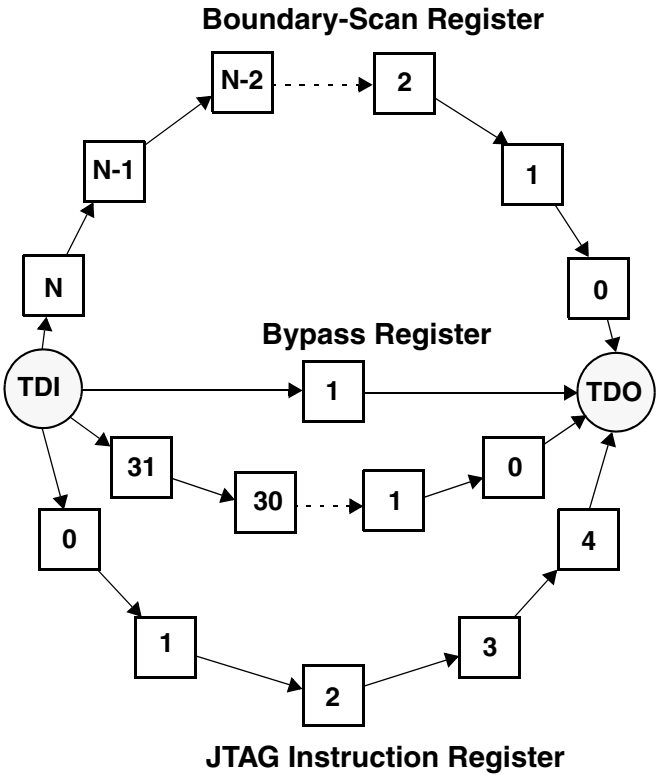


Figure C-2. Serial Scan Paths

Public Instructions

The following sections describe the public JTAG scan instructions.

EXTEST – Binary Code 00000

The EXTEST instruction selects the Boundary-Scan register to be connected between the TDI and TDO pins. This instruction allows testing of on-board circuitry external to the device.

The EXTEST instruction allows internal data to be driven to the boundary outputs and external data to be captured on the boundary inputs.

To protect the internal logic when the boundary outputs are overdriven or signals are received on the boundary inputs, make sure that nothing else drives data on the processor's output pins.

SAMPLE/PRELOAD – Binary Code 10000

The SAMPLE/PRELOAD instruction performs two functions and selects the Boundary-Scan register to be connected between TDI and TDO. The instruction has no effect on internal logic.

The SAMPLE part of the instruction allows a snapshot of the inputs and outputs captured on the boundary-scan cells. Data is sampled on the rising edge of TCK.

The PRELOAD part of the instruction allows data to be loaded on the device pins and driven out on the board with the EXTEST instruction. Data is preloaded on the pins on the falling edge of TCK.

BYPASS – Binary Code 11111

The BYPASS instruction selects the BYPASS register to be connected to TDI and TDO. The instruction has no effect on the internal logic. No data inversion should occur between TDI and TDO.

Boundary-Scan Register

The Boundary-Scan register is selected by the EXTEST and SAMPLE/PRELOAD instructions. These instructions allow the pins of the processor to be controlled and sampled for board-level testing.

D NUMERIC FORMATS

ADSP-BF53x Blackfin family processors support 8-, 16-, 32-, and 40-bit fixed-point data in hardware. Special features in the computation units allow support of other formats in software. This appendix describes various aspects of these data formats. It also describes how to implement a block floating-point format in software.

Unsigned or Signed: Two's-Complement Format

Unsigned integer numbers are positive, and no sign information is contained in the bits. Therefore, the value of an unsigned integer is interpreted in the usual binary sense. The least significant words of multiple-precision numbers are treated as unsigned numbers.

Signed numbers supported by the ADSP-BF53x Blackfin family are in two's-complement format. Signed-magnitude, one's-complement, binary-coded decimal (BCD) or excess-n formats are not supported.

Integer or Fractional

The ADSP-BF53x Blackfin family supports both fractional and integer data formats. In an integer, the radix point is assumed to lie to the right of the least significant bit (LSB), so that all magnitude bits have a weight of 1 or greater. This format is shown in Figure D-1. Note in two's-complement format, the sign bit has a negative weight.

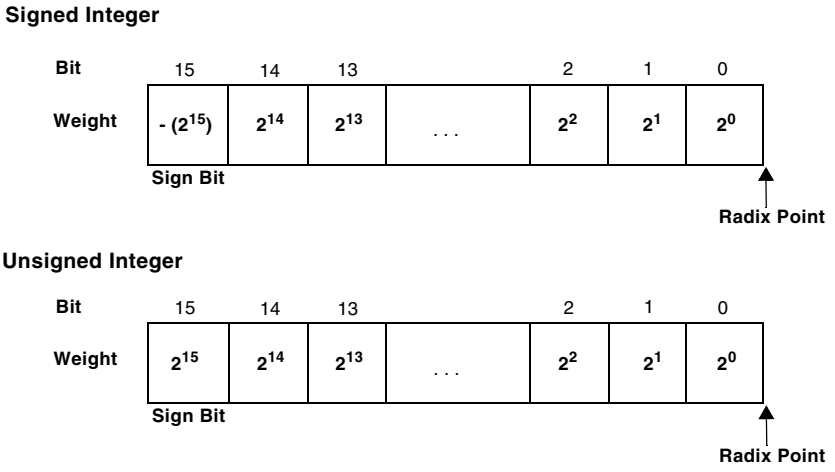


Figure D-1. Integer Format

In a fractional format, the assumed radix point lies within the number, so that some or all of the magnitude bits have a weight of less than 1. In the format shown in Figure D-2, the assumed radix point lies to the left of the three LSBs, and the bits have the weights indicated.

The native formats for the Blackfin processor family are a signed fractional 1.M format and an unsigned fractional 0.N format, where N is the number of bits in the data word and M = N – 1.

The notation used to describe a format consists of two numbers separated by a period (.); the first number is the number of bits to the left of the radix point, the second is the number of bits to the right of the radix point. For example, 16.0 format is an integer format; all bits lie to the left of the radix point. The format in Figure D-2 is 13.3.

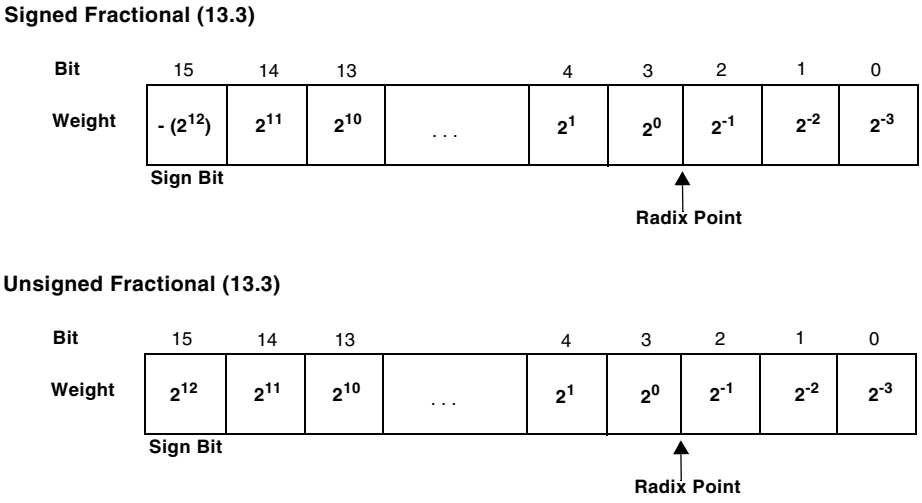


Figure D-2. Example of Fractional Format

Table D-1 shows the ranges of signed numbers representable in the fractional formats that are possible with 16 bits.

Table D-1. Fractional Formats and Their Ranges

Format	# of Integer Bits	# of Fractional Bits	Max Positive Value (0x7FFF) In Decimal	Max Negative Value (0x8000) In Decimal	Value of 1 LSB (0x0001) In Decimal
1.15	1	15	0.999969482421875	–1.0	0.000030517578125
2.14	2	14	1.999938964843750	–2.0	0.000061035156250
3.13	3	13	3.999877929687500	–4.0	0.000122070312500
4.12	4	12	7.999755859375000	–8.0	0.000244140625000
5.11	5	11	15.999511718750000	–16.0	0.000488281250000
6.10	6	10	31.999023437500000	–32.0	0.000976562500000
7.9	7	9	63.998046875000000	–64.0	0.001953125000000
8.8	8	8	127.996093750000000	–128.0	0.003906250000000
9.7	9	7	255.992187500000000	–256.0	0.007812500000000
10.6	10	6	511.984375000000000	–512.0	0.015625000000000
11.5	11	5	1023.968750000000000	–1024.0	0.031250000000000
12.4	12	4	2047.937500000000000	–2048.0	0.062500000000000
13.3	13	3	4095.875000000000000	–4096.0	0.125000000000000
14.2	14	2	8191.750000000000000	–8192.0	0.250000000000000
15.1	15	1	16383.500000000000000	–16384.0	0.500000000000000
16.0	16	0	32767.000000000000000	–32768.0	1.000000000000000

Binary Multiplication

In addition and subtraction, both operands must be in the same format (signed or unsigned, radix point in the same location), and the result format is the same as the input format. Addition and subtraction are performed the same way whether the inputs are signed or unsigned.

In multiplication, however, the inputs can have different formats, and the result depends on their formats. The ADSP-BF53x Blackfin family assembly language allows you to specify whether the inputs are both signed, both unsigned, or one of each (mixed-mode). The location of the radix point in the result can be derived from its location in each of the inputs. This is shown in Figure D-3. The product of two 16-bit numbers is a 32-bit number. If the inputs' formats are M.N and P.Q, the product has the format (M + P).(N + Q). For example, the product of two 13.3 numbers is a 26.6 number. The product of two 1.15 numbers is a 2.30 number.

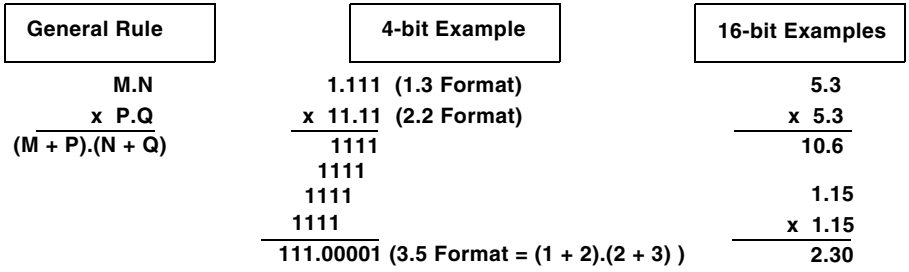


Figure D-3. Format of Multiplier Result

Fractional Mode And Integer Mode

A product of 2 two's-complement numbers has two sign bits. Since one of these bits is redundant, you can shift the entire result left one bit. Additionally, if one of the inputs was a 1.15 number, the left shift causes the result to have the same format as the other input (with 16 bits of additional precision). For example, multiplying a 1.15 number by a 5.11 number yields a 6.26 number. When shifted left one bit, the result is a 5.27 number, or a 5.11 number plus 16 LSBs.

The ADSP-BF53x Blackfin family provides a means (a signed fractional mode) by which the multiplier result is always shifted left one bit before being written to the result register. This left shift eliminates the extra sign bit when both operands are signed, yielding a result that is correctly formatted.

When both operands are in 1.15 format, the result is 2.30 (30 fractional bits). A left shift causes the multiplier result to be 1.31 which can be rounded to 1.15. Thus, if you use a signed fractional data format, it is most convenient to use the 1.15 format.

For more information about data formats, see the data formats listed in Table 2-2 on page 2-12.

Block Floating-Point Format

A block floating-point format enables a fixed-point processor to gain some of the increased dynamic range of a floating-point format without the overhead needed to do floating-point arithmetic. However, some additional programming is required to maintain a block floating-point format.

A floating-point number has an exponent that indicates the position of the radix point in the actual value. In block floating-point format, a set (block) of data values share a common exponent. A block of fixed-point

values can be converted to block floating-point format by shifting each value left by the same amount and storing the shift value as the block exponent.

Typically, block floating-point format allows you to shift out non-significant MSBs (most significant bits), increasing the precision available in each value. Block floating-point format can also be used to eliminate the possibility of a data value overflowing. See Figure D-4. Each of the three data samples shown has at least two non-significant, redundant sign bits. Each data value can grow by these two bits (two orders of magnitude) before overflowing. These bits are called guard bits.

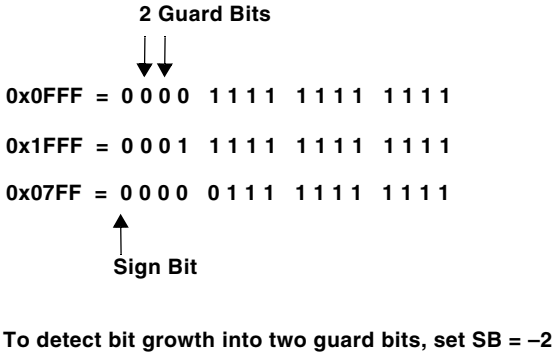


Figure D-4. Data With Guard Bits

If it is known that a process will not cause any value to grow by more than the two guard bits, then the process can be run without loss of data. Later, however, the block must be adjusted to replace the guard bits before the next process.

Figure D-5 shows the data after processing but before adjustment. The block floating-point adjustment is performed as follows.

- Assume the output of the SIGNBITS instruction is SB and SB is used as an argument in the EXPADJ instruction (see the *Blackfin Processor Programming Reference* for the usage and syntax of these instructions). Initially, the value of SB is +2, corresponding to the two guard bits. During processing, each resulting data value is inspected by the EXPADJ instruction, which counts the number of redundant sign bits and adjusts SB if the number of redundant sign bits is less than two. In this example, SB = +1 after processing, indicating the block of data must be shifted right one bit to maintain the two guard bits.
- If SB were 0 after processing, the block would have to be shifted two bits right. In either case, the block exponent is updated to reflect the shift.

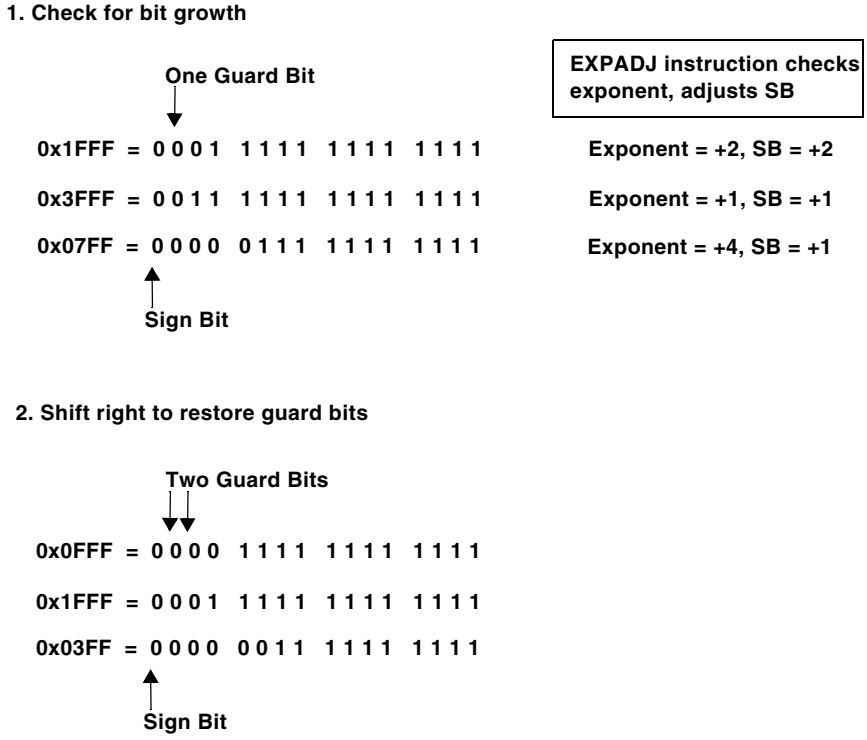


Figure D-5. Block Floating-Point Adjustment

G GLOSSARY

ALU.

See *Arithmetic/Logic Unit*

AMC (Asynchronous Memory Controller).

A configurable memory controller supporting multiple banks of asynchronous memory including SRAM, ROM, and flash, where each bank can be independently programmed with different timing parameters.

Arithmetic/Logic Unit (ALU).

A processor component that performs arithmetic, comparative, and logical functions.

Bank Activate command.

The Bank Activate command causes the SDRAM to open an internal bank (specified by the bank address) in a row (specified by the row address). When the Bank Activate command is issued to the SDRAM, the SDRAM opens a new row address in the dedicated bank. The memory in the open internal bank and row is referred to as the open page. The Bank Activate command must be applied before a read or write command.

base address.

The starting address of a circular buffer.

Base register.

A Data Address Generator (DAG) register that contains the starting address for a circular buffer.

bit-reversed addressing.

The addressing mode in which the Data Address Generator (DAG) provides a bit-reversed address during a data move without reversing the stored address.

Boot memory space.

Internal memory space designated for a program that is executed immediately after powerup or after a software reset.

burst length.

The burst length determines the number of words that the SDRAM device stores or delivers after detecting a single write or read command, respectively. The burst length is selected by writing certain bits in the SDRAM's Mode register during the SDRAM powerup sequence.

Burst Stop command.

The Burst Stop command is one of several ways to terminate a burst read or write operation.

burst type.

The burst type determines the address order in which the SDRAM delivers burst data after detecting a read command, or stores burst data after detecting a write command. The burst type is programmed in the SDRAM during the SDRAM powerup sequence.

cache block.

The smallest unit of memory that is transferred to/from the next level of memory from/to a cache as a result of a cache miss.

cache hit.

A memory access that is satisfied by a valid, present entry in the cache.

cache line.

Same as cache block. In this document, *cache line* is used for *cache block*.

cache miss.

A memory access that does not match any valid entry in the cache.

cache tag.

Upper address bits, stored along with the cached data line, to identify the specific address source in memory that the cached line represents.

Cacheability Protection Lookaside Buffer (CPLB).

Storage area that describes the access characteristics of the core memory map.

CAM (Content Addressable Memory).

Also called Associative Memory. A memory device that includes comparison logic with each bit of storage. A data value is broadcast to all words in memory; it is compared with the stored values; and values that match are flagged.

CAS (Column Address Strobe).

A signal sent from the SDC to a DRAM device to indicate that the column address lines are valid.

CAS latency (also t_{AA}, t_{CAC}, CL).

The Column Address Strobe (CAS) latency is the delay in clock cycles between when the SDRAM detects the read command and when it provides the data at its output pins.

CBR (CAS Before RAS) memory refresh.

DRAM devices have a built-in counter for the refresh row address. By activating Column Address Strobe (CAS) before activating Row Address Strobe (RAS), this counter is selected to supply the row address instead of the address inputs.

CEC.

See *Core Event Controller*

circular addressing.

The process by which the Data Address Generator (DAG) "wraps around" or repeatedly steps through a range of registers.

companding.

(Compressing/expanding). The process of logarithmically encoding and decoding data to minimize the number of bits that must be sent.

conditional branches.

Jump or call/return instructions whose execution is based on defined conditions.

core.

The core consists of these functional blocks: CPU, L1 memory, Event Controller, core timer, and Performance Monitoring registers.

Core Event Controller (CEC).

The CEC works with the System Interrupt Controller (SIC) to prioritize and control all system interrupts. The CEC handles general-purpose interrupts and interrupts routed from the SIC.

CPLB.

See *Cacheability Protection Lookaside Buffer*

DAB.

See *DMA Access Bus*

DAG.

See *Data Address Generator*

Data Address Generator (DAG).

Processing component that provides memory addresses when data is transferred between memory and registers.

Data Register File.

A set of data registers that is used to transfer data between computation units and memory while providing local storage for operands.

data registers (Dreg).

Registers located in the data arithmetic unit that hold operands and results for multiplier, ALU, or shifter operations.

DCB.

See *DMA Core Bus*

DEB.

See *DMA External Bus*

descriptor block, DMA.

A set of parameters used by the direct memory access (DMA) controller to describe a set of DMA sequences.

descriptor loading, DMA.

The process in which the direct memory access (DMA) controller downloads a DMA descriptor from data memory and autoinitializes the DMA parameter registers.

DFT (Design For Testability).

A set of techniques that helps designers of digital systems ensure that those systems will be testable.

Digital Signal Processor (DSP).

An integrated circuit designated for high-speed manipulation of analog information that has been converted into digital form.

direct branches.

Jump or call/return instructions that use absolute addresses that do not change at runtime (such as a program label), or they use a PC-relative address.

direct-mapped.

Cache architecture where each line has only one place that it can appear in the cache. Also described as 1-Way associative.

Direct Memory Access (DMA).

A way of moving data between system devices and memory in which the data is transferred through a DMA port without involving the processor.

dirty, modified.

A state bit, stored along with the tag, indicating whether the data in the data cache line has been changed since it was copied from the source memory and, therefore, needs to be updated in that source memory.

DMA.

See *Direct Memory Access*

DMA Access Bus (DAB).

A bus that provides a means for DMA channels to be accessed by the peripherals.

DMA chaining.

The linking or chaining of multiple direct memory access (DMA) sequences. In chained DMA, the I/O processor loads the next DMA descriptor into the DMA parameter registers when the current DMA finishes and autoinitializes the next DMA sequence.

DMA Core Bus (DCB).

A bus that provides a means for DMA channels to gain access to on-chip memory.

DMA descriptor registers.

Registers that hold the initialization information for a direct memory access (DMA) process.

DMA External Bus (DEB).

A bus that provides a means for DMA channels to gain access to off-chip memory.

DPMC (Dynamic Power Management Controller).

A processor's control block that allows the user to dynamically control the processor's performance characteristics and power dissipation.

DQM Data I/O Mask Function.

The `SDQM[1:0]` pins provide a byte-masking capability on 8-bit writes to SDRAM.

DRAM (Dynamic Random Access Memory).

A type of semiconductor memory in which the data is stored as electrical charges in an array of cells, each consisting of a capacitor and a transistor. The cells are arranged on a chip in a grid of rows and columns. Since the capacitors discharge gradually—and the cells lose their information—the array of cells has to be refreshed periodically.

DSP.

See *Digital Signal Processor*

EAB.

See *External Access Bus*

EBC.

See *External Bus Controller*

EBIU.

See *External Bus Interface Unit*

edge-sensitive interrupt.

A signal or interrupt the processor detects 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.

The ordering of bytes in a multibyte number.

EPB.

See *External Port Bus*

EPROM (Erasable Programmable Read-Only Memory).

A type of semiconductor memory in which the data is stored as electrical charges in isolated ("floating") transistor gates that retain their charges almost indefinitely without an external power supply. An EPROM is programmed by "injecting" charge into the floating gates—a process that requires relatively high voltage (usually 12V – 25V). Ultraviolet light, applied to the chip's surface through a quartz window in the package, will discharge the floating gates, allowing the chip to be reprogrammed.

EVT (Event Vector Table).

A table stored in memory that contains sixteen 32-bit entries; each entry contains a vector address for an interrupt service routine (ISR). When an event occurs, instruction fetch starts at the address location in the corresponding EVT entry. See *ISR*.

exclusive, clean.

The state of a data cache line indicating the line is valid and the data contained in the line matches that in the source memory. The data in a clean cache line does not need to be written to source memory before it is replaced.

External Access Bus (EAB).

A bus mastered by the core memory management unit to access external memory.

External Bus Controller (EBC).

A component that provides arbitration between the External Access Bus (EAB) and the DMA External Bus (DEB), granting at most one requester per cycle.

External Bus Interface Unit (EBIU).

A component that provides glueless interfaces to external memories. It services requests for external memory from the core or from a DMA channel.

external port.

A channel or port that extends the processor's internal address and data buses off-chip, providing the processor's interface to off-chip memory and peripherals.

External Port Bus (EPB).

A bus that connects the output of the EBIU to external devices.

FFT (Fast Fourier Transform).

An algorithm for computing the Fourier transform of a set of discrete data values. The FFT expresses a finite set of data points, for example a periodic sampling of a real-world signal, in terms of its component frequencies. Or conversely, the FFT reconstructs a signal from the frequency data. The FFT can also be used to multiply two polynomials.

FIFO (First In, First Out).

A hardware buffer or data structure from which items are taken out in the same order they were put in.

flash memory.

A type of single transistor cell, erasable memory in which erasing can only be done in blocks or for the entire chip.

fully associative.

Cache architecture where each line can be placed anywhere in the cache.

glueless.

No external hardware is required.

Harvard architecture.

A processor memory architecture that uses separate buses for program and data storage. The two buses let the processor fetch a data word and an instruction word simultaneously.

HLL (High Level Language).

A programming language that provides some level of abstraction above assembly language, often using English-like statements, where each command or statement corresponds to several machine instructions.

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.

Address portion that is used to select an array element (for example, line index).

Index registers.

A Data Address Generator (DAG) register that holds an address and acts as a pointer to memory.

indirect branches.

Jump or call/return instructions that use a dynamic address from the data address generator, evaluated at runtime.

input clock.

Device that generates a steady stream of timing signals to provide the frequency, duty cycle, and stability to allow accurate internal clock multiplication via the phase locked loop (PLL) module.

internal memory bank.

There are several internal memory banks within a given SDRAM. Each of the internal banks can be active (open) simultaneously.

The SDC assumes that all SDRAMs to which it interfaces have four internal banks.

interrupt.

An event that suspends normal processing and temporarily diverts the flow of control through an interrupt service routine (ISR). See *ISR*.

invalid.

Describes the state of a cache line. When a cache line is invalid, a cache line match cannot occur.

IrDA (Infrared Data Association).

A nonprofit trade association that established standards for ensuring the quality and interoperability of devices using the infrared spectrum.

isochronous.

Processes where data must be delivered within certain time constraints.

ISR (Interrupt Service Routine).

Software that is executed when a specific interrupt occurs. A table stored in low memory contains pointers, also called vectors, that direct the processor to the corresponding ISR. See *EVT*.

JTAG (Joint Test Action Group).

An IEEE Standards working group that defines the IEEE 1149.1 standard for a test access port for testing electronic devices.

JTAG port.

A channel or port that supports the IEEE standard 1149.1 JTAG standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system.

jump.

A permanent transfer of the program flow to another part of program memory.

latency.

The overhead time used to find the correct place for memory access and preparing to access it.

Least Recently Used algorithm.

Replacement algorithm used by cache that first replaces lines that have been unused for the longest time.

Least Significant Bit (LSB).

The last or rightmost bit in the normal representation of a binary number—the bit of a binary number giving the number of ones.

Length registers.

A Data Address Generator (DAG) register that specifies the range of addresses in a circular buffer.

Level 1 (L1) memory.

Memory that is directly accessed by the core with no intervening memory subsystems between it and the core.

Level 2 (L2) memory.

Memory that is at least one level removed from the core. L2 memory has a larger capacity than L1 memory, but it requires additional latency to access.

level-sensitive interrupts.

A signal or interrupt that the processor detects if the input signal is low (active) when sampled on the rising edge of `CLKIN`.

LIFO (Last In, First Out).

A data structure from which the next item taken out is the most recent item put in.

little endian.

The native data store format of the processor. Words and half words are stored in memory (and registers) with the least significant byte at the lowest byte address and the most significant byte at the highest byte address of the data storage location.

loop.

A sequence of instructions that executes several times.

LRU.

See *Least Recently Used algorithm*.

LSB.

See *Least Significant Bit*.

MAC (Multiply/Accumulate).

A mathematical operation that multiplies two numbers and then adds a third to get the result (see *Multiply Accumulator*).

Memory Management Unit (MMU).

A component of the processor that supports protection and selective caching of memory by using Cacheability Protection Lookaside Buffers (CPLBs).

Mode Register.

Internal configuration registers within SDRAM devices which allow specification of the SDRAM device's functionality.

modified addressing.

The process whereby the Data Address Generator (DAG) produces an address that is incremented by a value or the contents of a register.

Modify register.

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.

MMR (Memory-Mapped Register).

A specific location in main memory used by the processor as if it were a register.

MMU.

See *Memory Management Unit*

MSB (Most Significant Bit).

The first or leftmost bit in the normal representation of a binary number—the bit of a binary number with the greatest weight (2(n-1)).

multifunction computations.

The parallel execution of multiple computational instructions. These instructions complete in a single cycle, and they combine parallel operation of the computational units and memory accesses. The multiple operations perform the same as if they were in corresponding single function computations.

multiplier.

A computational unit that performs fixed-point multiplication and executes fixed-point multiply/add and multiply/subtract operations.

NMI (Nonmaskable Interrupt).

A high priority interrupt that cannot be disabled by another interrupt.

NRZ (Non-return-to-Zero).

A binary encoding scheme in which a 1 is represented by a change in the signal and a 0 by no change—there is no return to a reference (0) voltage between encoded bits. This method eliminates the need for a clock signal.

NRZI (Non-return-to-Zero Inverted).

A binary encoding scheme in which a 0 is represented by a change in the signal and a 1 is represented by no change—there is no return to a reference (0) voltage between encoded bits. This method eliminates the need for a clock signal.

orthogonal.

The characteristic of being independent. An orthogonal instruction set allows any register to be used in an instruction that references a register.

PAB.

See *Peripheral Access Bus*

page size.

The amount of memory which has the same row address and can be accessed with successive read or write commands without needing to activate another row.

PC (Program Counter).

A register that contains the address of the next instruction to be executed.

peripheral.

Functional blocks not included as part of the core, and typically used to support system level operations.

Peripheral Access Bus (PAB).

A bus used to provide access to EBIU memory-mapped registers.

PF (Programmable Flag).

General-purpose I/O pins. Each PF pin can be individually configured as either an input or an output pin, and each PF pin can be further configured to generate an interrupt.

Phase Locked Loop (PLL).

An on-chip frequency synthesizer that produces a full speed master clock from a lower frequency input clock signal.

PLL.

See *Phase Locked Loop*.

precision.

The number of bits after the binary point in the storage format for the number.

post-modify addressing.

The process in which the Data Address Generator (DAG) provides an address during a data move and auto-increments after the instruction is executed.

Precharge command.

The Precharge command closes a specific internal bank in the active page or all internal banks in the page.

pre-modify addressing.

The process in which the Data Address Generator (DAG) provides an address during a data move and auto-increments before the instruction is executed.

PWM (Pulse Width Modulation).

Also called Pulse Duration Modulation (PDM), PWM is a pulse modulation technique in which the duration of the pulses is varied by the modulating voltage.

RAS (Row Address Strobe).

A signal sent from the SDC to a DRAM device to indicate validity of row address lines.

Real-Time Clock (RTC).

A component that generates timing pulses for the digital watch features of the processor, including time of day, alarm, and stopwatch countdown features.

ROM (Read-Only Memory).

A data storage device manufactured with fixed contents. This term is most often used to refer to non-volatile semiconductor memory.

RTC.

See *Real-Time Clock*

RZ (Return-to-Zero modulation).

A binary encoding scheme in which two signal pulses are used for every bit. A 0 is represented by a change from the low voltage level to the high voltage level; a 1 is represented by a change from the high voltage level to the low voltage level. A return to a reference (0) voltage is made between encoded bits.

RZI (Return-to-Zero-Inverted modulation).

A binary encoding scheme in which two signal pulses are used for every bit. A 1 is represented by a change from the low voltage level to the high voltage level; a 0 is represented by a change from the high voltage level to the low voltage level. A return to a reference (0) voltage is made between encoded bits.

saturation (ALU saturation mode).

A state in which all positive fixed-point overflows return the maximum positive fixed-point number, and all negative overflows return the maximum negative number.

SDC (SDRAM Controller).

A configurable memory controller supporting a bank of synchronous memory consisting of SDRAM.

SDRAM (Synchronous Dynamic Random Access Memory).

A form of DRAM that includes a clock signal with its other control signals. This clock signal allows SDRAM devices to support “burst” access modes that clock out a series of successive bits.

SDRAM bank.

Region of external memory that can be configured to be 16M bytes, 32M bytes, 64M bytes, or 128M bytes and is selected by the $\overline{\text{SMS}}$ pin.

Self-Refresh.

When the SDRAM is in Self-Refresh mode, the SDRAM's internal timer initiates Auto-Refresh cycles periodically, without external control input. The SDRAM Controller (SDC) must issue a series of commands including the Self-Refresh command to put SDRAM into low power mode, and it must issue another series of commands to exit Self-Refresh mode. Entering Self-Refresh mode is programmed in the SDRAM Memory Global Control register (EBIU_SDGCTL) and any access to the SDRAM address space causes the SDC to exit SDRAM from Self-Refresh mode. See "Entering and Exiting Self-Refresh Mode (SRFS)" on page 17-37.

Serial Peripheral Interface (SPI).

A synchronous serial protocol used to connect integrated circuits.

serial ports (SPORTs).

A high speed synchronous input/output device on the processor. The processor uses two synchronous serial ports that provide inexpensive interfaces to a wide variety of digital and mixed-signal peripheral devices.

set.

A group of *N*-line storage locations in the Ways of an *N*-Way cache, selected by the Index field of the address.

set associative.

Cache architecture that limits line placement to a number of sets (or Ways).

shifter.

A computational unit that completes logical and arithmetic shifts.

SIC (System Interrupt Controller).

Part of the processor's two-level event control mechanism. The SIC works with the Core Event Controller (CEC) to prioritize and control all system interrupts. The SIC provides mapping between the peripheral interrupt sources and the prioritized general-purpose interrupt inputs of the core.

SIMD (Single Instruction, Multiple Data).

A parallel computer architecture in which multiple data operands are processed simultaneously using one instruction.

SP (Stack Pointer).

A register that points to the top of the stack.

SPI.

See *Serial Peripheral Interface*

SRAM.

See *Static Random Access Memory*

stack.

A data structure for storing items that are to be accessed in last in, first out (LIFO) order. When a data item is added to the stack, it is "pushed"; when a data item is removed from the stack, it is "popped."

Static Random Access Memory (SRAM).

Very fast read/write memory that does not require periodic refreshing.

system.

The system includes the peripheral set (Timers, Real-Time Clock, programmable flags, UART, SPORTs, PPI, and SPIs), the external memory controller (EBIU), the Memory DMA controller, as well as the interfaces between these peripherals, and the optional, external (off-chip) resources.

System clock (SCLK).

A component that delivers clock pulses at a frequency determined by a programmable divider ratio within the PLL.

System Interrupt Controller (SIC).

Component that maps and routes events from peripheral interrupt sources to the prioritized, general-purpose interrupt inputs of the Core Event Controller (CEC).

TAP (Test Access Port).

See *JTAG port*

TDM.

See *Time Division Multiplexing*

Time Division Multiplexing (TDM).

A method used for transmitting separate signals over a single channel. Transmission time is broken into segments, each of which carries one element. Each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of the 24 channels.

UART.

See *Universal Asynchronous Receiver Transmitter*

Universal Asynchronous Receiver Transmitter (UART).

A module that contains both the receiving and transmitting circuits required for asynchronous serial communication.

Valid.

A state bit (stored along with the tag) that indicates the corresponding tag and data are current and correct and can be used to satisfy memory access requests.

victim.

A dirty cache line that must be written to memory before it can be replaced to free space for a cache line allocation.

Von Neumann architecture.

The architecture used by most non-DSP microprocessors. This architecture uses a single address and data bus for memory access.

Way.

An array of line storage elements in an *N*-Way cache.

W1C.

See *Write-1-to-Clear*

W1S.

See *Write-1-to-Set*

Write-1-to-Clear (W1C) bit.

A control or status bit that can be cleared (= 0) by being written to with 1.

Write-1-to-Set (W1S) bit.

A control or status bit that is set by writing 1 to it. It cannot be cleared by writing 0 to it.

write back.

A cache write policy (also known as copyback). The write data is written only to the cache line. The modified cache line is written to source memory only when it is replaced.

write through.

A cache write policy (also known as store through). The write data is written to both the cache line and to source memory. The modified cache line is *not* written to the source memory when it is replaced.

I INDEX

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V