ADSP-BF535 Blackfin® Processor Hardware Reference

Revision 3.3, February 2013

Part Number
82-000410-13

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

DATA ADDRESS GENERATORS

MEMORY

DIRECT MEMORY ACCESS

SPI COMPATIBLE PORT CONTROLLERS

UART PORT CONTROLLER

PCI BUS INTERFACE

USB DEVICE

PROGRAMMABLE FLAGS

TIMERS

SYSTEM DESIGN

BLACKFIN PROCESSOR'S DEBUG

SYSTEM MMR ASSIGNMENTS

PREFACE

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

Purpose of This Manual

ADSP-BF535 Blackfin Processor Hardware Reference provides architectural information about the Blackfin processors. The architectural descriptions cover functional blocks, buses, and ports, including all features and processes that they support.

For programming information, see *Blackfin Processor Programming Reference*. For timing, electrical, and package specifications, see *ADSP-BF535 Blackfin Embedded Processor Data Sheet*.

Intended Audience

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

Manual Contents

This manual contains:

- Chapter 1, Introduction
  Provides a high level overview of the Blackfin processor. Architectural descriptions include functional blocks, buses, and ports, including all features and processes that they support.
- Chapter 2, Computational Units
  Describes the arithmetic/logic units (ALUs), multiplier/accumulator units, 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 ADSP-BF535 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, exceptions, and the `IDLE` instruction.
- 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 and L2 memories. In particular, details their memory architecture, memory model, memory transaction model, and memory-mapped registers (MMRs). The L1 section

discusses the instruction, data, and scratchpad memory, which are part of the Blackfin processor core. The L2 section discusses on-chip and off-chip L2 memories.

- 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 channel DMA and Memory DMA controllers. The channel 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 ADSP-BF535 processor memory spaces and the L1, L2, external synchronous, and asynchronous memories.
- Chapter 10, SPI Compatible Port Controllers
  Describes the two independent Serial Peripheral Interface (SPI) ports, SPI0 and SPI1, that provide an I/O interface to a variety of SPI compatible peripheral devices.
- Chapter 11, 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 12, UART Port Controller

  Describes the Universal Asynchronous Receiver/ Transmitter (UART) ports (UART0 and UART1), which convert data between serial and parallel formats and include modem control and interrupt handling hardware.

- Chapter 13, PCI Bus Interface
  Describes the Peripheral Component Interconnect (PCI) peripheral, which supports PCI device and Host-to-PCI bridge functions.

- Chapter 14, USB Device
  Discusses the USB Device module in the ADSP- 21535 processor. The chapter includes a block diagram of the USBD as well as descriptions of the USBD registers, interrupts, and programming model.

- Chapter 15, Programmable Flags
  Describes the programmable flags, or general-purpose I/O pins, in the ADSP-BF535 processor, including how to configure the pins as inputs and outputs and how to generate interrupts.

- Chapter 16, 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 17, Real-Time Clock (RTC)
  Describes a set of digital watch features of the ADSP-BF535 processor, including time of day, alarm, and stopwatch countdown.

- Chapter 18, External Bus Interface Unit
  Describes the External Bus Interface Unit of the ADSP-BF535 processor. The chapter also discusses the asynchronous memory interface, the SDRAM controller (SDC), related registers, and SDC configuration and commands.
- Chapter 19, System Design
  Describes how to use the processor as part of an overall system. It includes information about interfacing the ADSP-BF535 processor to external memory chips, bus timing and latency numbers, semaphores, and a discussion of the treatment of unused processor pins.
- Chapter 20, Blackfin Processor's Debug
  Describes the Blackfin processor debug functionality, which can be used for software debugging and complements some services often found in an operating system.
- Appendix A, Blackfin Processor Core MMR Assignments
  Lists the core memory-mapped registers and their addresses.
- Appendix B, System MMR Assignments
  Lists the system memory-mapped registers and their addresses.
- Appendix C, Test Features
  Describes test features for the ADSP-BF535 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 3.3 of *ADSP-BF535 Blackfin Processor Hardware Reference*. This revision corrects minor typographical errors and the following issues:

- `RETI` instructions need not be first in nested interrupts and complete table of hardware conditions causing hardware interrupts in Chapter 4, "Program Sequencer"
- Note on programming the `STOPCK` bit in Chapter 8, "Dynamic Power Management"
- Description of multichannel mode operation in Chapter 11, "Serial Port Controllers"
- Note on timing dependencies for the `TRP` and `TRAS` settings in the `EBIU_SDGCTL` register in Chapter 18, "External Bus Interface Unit"
- Clarification of watchpoint ranges in Chapter 20, "Blackfin Processor's Debug"

Technical Support

You can reach Analog Devices processors and DSP technical support in the following ways:

- Post your questions in the processors and DSP support community at EngineerZone®:
  `http://ez.analog.com/community/dsp`
- Submit your questions to technical support directly at:
  `http://www.analog.com/support`

- E-mail your questions about processors, DSPs, and tools development software from **CrossCore® Embedded Studio** or **VisualDSP++®**:

  Choose **Help > Email Support**. This creates an e-mail to `processor.tools.support@analog.com` and automatically attaches your **CrossCore Embedded Studio** or **VisualDSP++** version information and `license.dat` file.

- E-mail your questions about processors and processor applications to:
  `processor.support@analog.com` or
  `processor.china@analog.com` (Greater China support)

- In the **USA only**, call **1-800-ANALOGD** (1-800-262-5643)

- Contact your Analog Devices sales office or authorized distributor. Locate one at:
  `www.analog.com/adi-sales`

- Send questions by mail to:
  Processors and DSP Technical Support
  Analog Devices, Inc.
  Three 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. Refer to the CCES or VisualDSP++ online help for a complete list of supported processors.

Product Information

Product information can be obtained from the Analog Devices Web site and the CCES or VisualDSP++ online help.

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 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 provides access to books, application notes, data sheets, code examples, and more.

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

EngineerZone

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

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

Notation Conventions

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

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

Register Diagram Conventions

Register diagrams use the following conventions:

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

The following figure shows an example of these conventions.

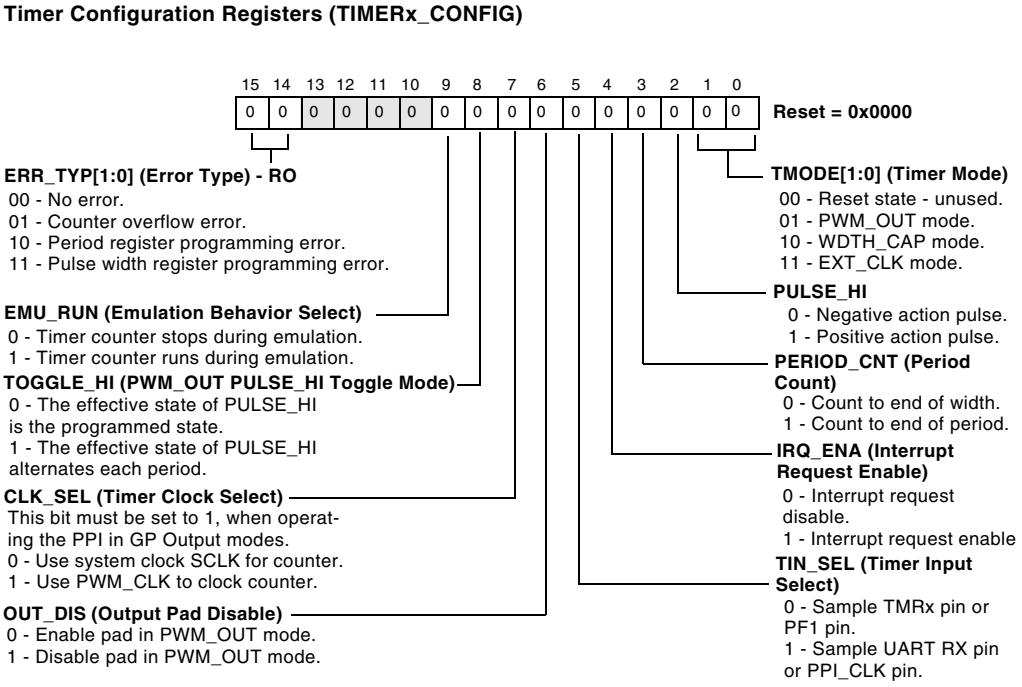


Figure 1. Register Diagram Example

1 INTRODUCTION

The ADSP-BF535 processor is a member of the Blackfin family of products. The core architecture combines a dual-MAC signal processing engine, an orthogonal RISC-like microprocessor instruction set, and single instruction multiple data (SIMD), multimedia capabilities into a single instruction set architecture.

Blackfin processors feature dynamic power management, the ability to vary both the voltage and frequency of operation to provide lower overall power consumption than other DSPs.

Throughout this manual, the words *core* and *system* are used to describe sections of the ADSP-BF535 processor:

- The word *core* denotes the processor, L1 memory, Event Controller, core timer, and Performance Monitoring registers.
- The word *system* denotes the peripheral set described in the following section plus the Peripheral Component Interconnect (PCI) controller, the external memory controller (EBIU), the Direct Memory Access (DMA) controller, and the interfaces between these, the system, and the optional, external (off chip) resources.

ADSP-BF535 Peripherals

The ADSP-BF535 processor system peripherals include:

- Universal Asynchronous Receiver Transmitters (UARTs)
- Serial Peripheral Interfaces (SPIs)

- Serial Ports (SPORTs)
- General-purpose timers
- Real-Time Clock (RTC)
- General-purpose I/O (Programmable Flags)
- Watchdog Timer
- Universal Serial Bus (USB)
- Peripheral Component Interconnect (PCI) Bus

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

All DMA capable peripherals benefit from individual DMA channels, which are integrated into the peripheral. There is also a separate memory DMA channel dedicated to data transfers among the various chip memory spaces, including external, synchronous dynamic random access memory (SDRAM) and asynchronous memory, and internal Level 2 SRAM and PCI memory spaces. Multiple on-chip 32-bit buses run at up to 133 MHz to provide bandwidth for the processor core and for on-chip and external peripherals.

ADSP-BF535 Core Architecture

The ADSP-BF535 processor core contains two multiplier/accumulators (MACs), two 40-bit arithmetic logic units (ALUs), four video ALUs, and a single shifter, as shown in Figure 1-2. The computational units process 8-, 16-, or 32-bit data from the register file. Each MAC performs a 16- by 16-bit multiply per cycle, with accumulation to a 40-bit result. This provides eight bits of extended precision.

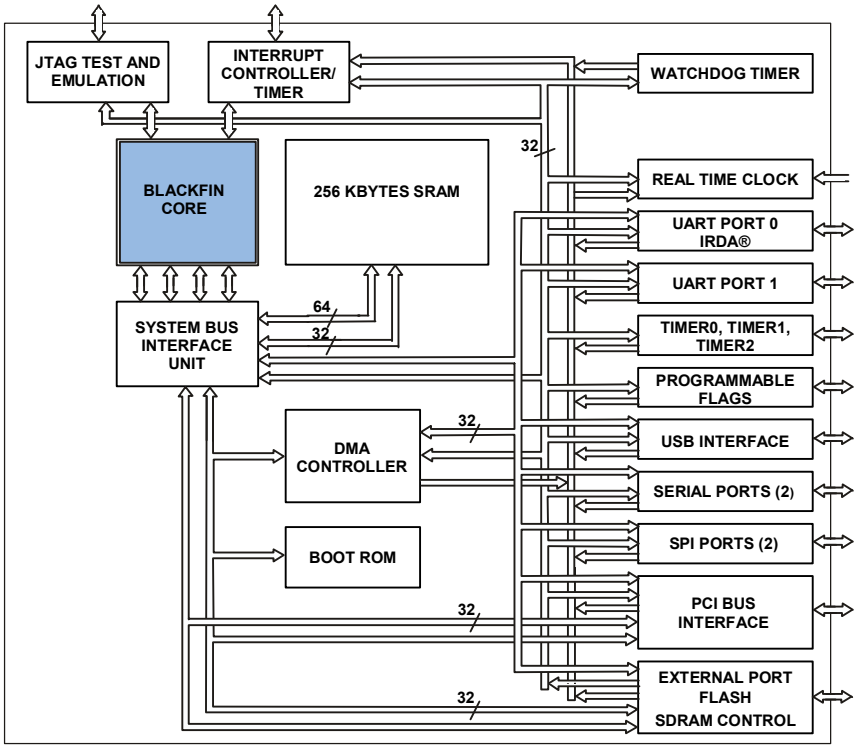


Figure 1-1. ADSP-BF535 Block Diagram

Two ALUs perform all standard arithmetic and logical operations on 16- or 32-bit data. Each of the two 32-bit input registers can be regarded as two 16-bit halves. By using the registers as pairs of 16-bit operands, dual 16-bit or single 32-bit operations can occur in a single cycle.

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

Data for the computational units is located in a multiported register file of sixteen 16-bit entries or eight 32-bit entries.

A program sequencer controls the instruction execution flow, including instruction alignment and decoding. The sequencer supports conditional jumps and subroutine calls, as well as zero-overhead looping. A loop buffer stores instructions locally.

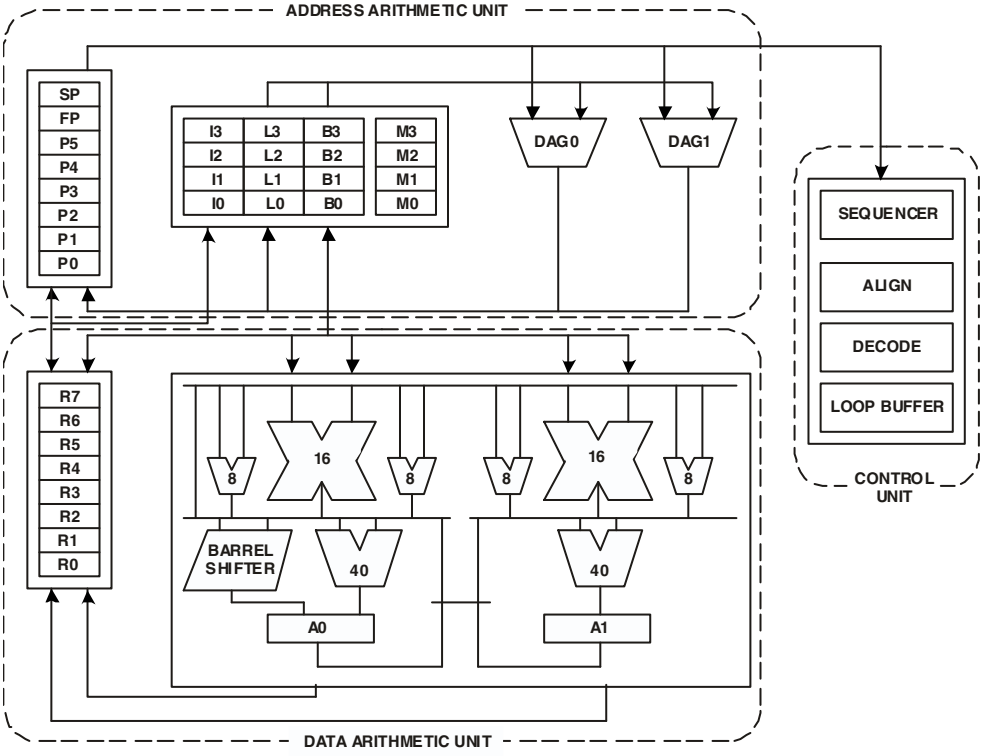


Figure 1-2. ADSP-BF535 Core Architecture

Two data address generators (DAGs) provide addresses for simultaneous dual operand fetches from memory. The DAGs share a register file containing four sets of 32-bit Index, Modify, Length, and Base registers. Eight additional 32-bit registers provide pointers for general indexing of variables and stack locations.

Blackfins support a modified Harvard architecture in combination with a hierarchical memory structure. Level 1 (L1) memories operate at the full processor speed. On-chip and off-chip Level 2 (L2) memories can take multiple processor cycles to access. 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. At the L2 level, there is a single unified memory space, holding both instructions and data.

In addition, the L1 instruction memory and L1 data memories may be configured as either static RAMs (SRAMs) or caches. 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.

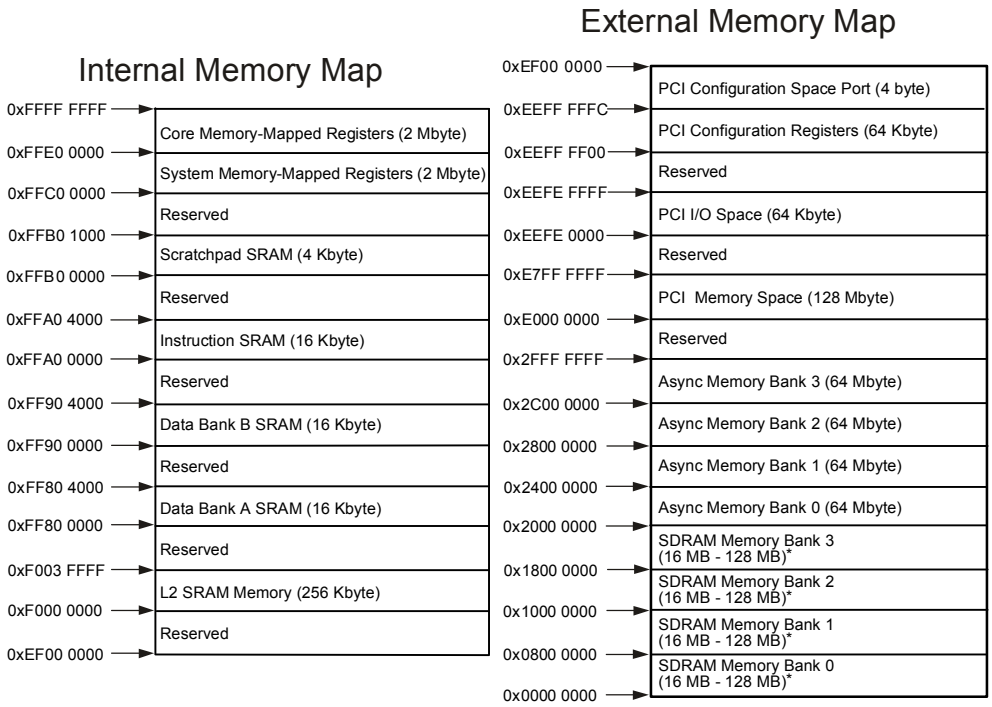
The Blackfin instruction set is optimized so that 16-bit opcodes represent the most frequently used instructions. Complex DSP instructions are encoded into 32-bit opcodes as multifunction instructions. Blackfins support a limited multi-issue capability, where a 32-bit instruction can be issued in parallel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

The Blackfin assembly language uses an algebraic syntax. The architecture is optimized for use with a C/C++ compiler.

Memory Architecture

The Blackfin architecture structures memory as a single, unified 4 Gbyte address space using 32-bit addresses. All resources, including internal memory, external memory, PCI address spaces, 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. Fast, low latency memory systems for cache or SRAM are located close to the processor. Figure 1-3 shows the memory map for the ADSP-BF535 processor.



* The addresses shown for the SDRAM banks reflect a fully populated SDRAM array with 512 Mbytes of memory. If any bank contains less than 128 Mbytes of memory, it would only extend to the length of the real memory systems and the end address would become the start address of the next bank. This would continue for all four banks, with any remaining space between the end of Memory Bank 3 and the beginning of Async Memory Bank 0 at address 0x2000 0000 treated as reserved address space.

Figure 1-3. ADSP-BF535 Processor Internal/External Memory Map

The L1 memory system is the primary highest performance memory available to the Blackfin core. The L2 memory system provides additional capacity with slightly lower performance. The off-chip memory system,

accessed through the External Bus Interface Unit (EBIU), provides expansion with SDRAM, flash memory, and SRAM, optionally accessing more than 768 Mbytes 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 L1/L2 memories and the external memory spaces, including PCI memory space.

Internal (On-Chip) Memory

The ADSP-BF535 processor has four blocks of on-chip memory that provide high bandwidth access to the core:

- L1 instruction memory consisting of 16 Kbytes of 4-way set associative cache memory. In addition, the memory may be configured as an SRAM. This memory is accessed at full processor speed.
- L1 data memory, consisting of two banks of 16 Kbytes each. Each L1 data memory bank may be configured as a 1- or 2-way set associative cache or as an SRAM, and it is accessed at full speed by the core.
- 4 Kbyte 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.
- L2 SRAM memory array, which provides 256 Kbytes of high speed SRAM at the full bandwidth of the core and slightly longer latency than the L1 memory banks. The L2 memory is a unified instruction and data memory and can hold any mixture of code and data required by the system design.

The ADSP-BF535 processor core has a dedicated low latency 64-bit wide datapath port into the L2 SRAM memory. For example, at a core frequency of 300 MHz, the peak data transfer rate across this interface is up to 2.4 Gbytes per second.

External (Off-Chip) Memory

External memory is accessed via the External Bus Interface Unit. This interface provides a glueless connection to up to four banks of synchronous DRAM (SDRAM) as well as up to 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 with up to four banks of SDRAM. Each bank contains between 16 Mbytes and 128 Mbytes, providing access to up to 512 Mbytes of RAM. Each bank can be programmed independently and is contiguous with adjacent banks, regardless of the size or placement of the banks. This permits configuration and upgrade of the system memory, and allows the core to view all SDRAM as a single, contiguous, physical address space.

The asynchronous memory controller can also be programmed to control up to four banks of devices. Each bank occupies a 64 Mbyte segment regardless of the size of the devices used, so that these banks are only contiguous if fully populated with 64 Mbytes of memory.

PCI

The PCI bus defines three separate address spaces, which are accessed through windows in the ADSP-BF535 processor memory space:

- PCI memory
- PCI I/O
- PCI configuration space

In addition, the PCI interface can either be used as a bridge from the processor core as the controlling CPU in the system or as a host port where another CPU in the system is the host and the ADSP-BF535 processor is functioning as an intelligent I/O device on the PCI bus.

When the ADSP-BF535 processor acts as the system controller, it views the PCI address spaces through its mapped windows. It can initialize all devices in the system and maintain a map of the topology of the environment.

The PCI memory region is a 4 Gbyte space that appears on the PCI bus and can be used to map memory on I/O devices on the bus. The ADSP-BF535 processor uses a 128 Mbyte window in memory space to see a portion of the PCI memory space. A base address register positions this window anywhere in the 4 Gbyte PCI memory space while its position with respect to the processor addresses remains fixed.

The PCI I/O region is also a 4 Gbyte space; however, most systems and I/O devices use only a 64 Kbyte subset of this space for I/O mapped addresses. The ADSP-BF535 processor implements a 64 Kbyte window in this space, along with a base address register that positions the window anywhere in PCI I/O address space, while the window remains at the same address in the processor address space.

PCI configuration space is a limited address space that is used for system enumeration and initialization. It is a very low performance communication mode between the processor and PCI devices. The ADSP-BF535 processor provides a 1-value window to access a single data value at any address in PCI configuration space. This window is fixed and receives the address of the value, and the value if the operation is a write. Otherwise the device returns the value into the same address on a read operation.

I/O Memory Space

Blackfins do not define a separate I/O space. All resources are mapped through the flat 32-bit address space. On-chip I/O devices have their control registers mapped into memory-mapped registers (MMRs) at addresses near the top of the 4 Gbyte address space. These are separated into two smaller blocks: one that contains the control MMRs for all CPU core functions and the other contains the registers needed for setup and control of the on-chip peripherals outside of the CPU core. The core MMRs are accessible only by the core and only in Supervisor mode. They appear as reserved space by on-chip peripherals, as well as external devices accessing resources through the PCI bus. The system MMRs are accessible by the core in Supervisor mode and can be mapped as either visible or reserved to other devices, depending on the system protection model.

Event Handling

The event controller on the ADSP-BF535 processor handles all asynchronous and synchronous events to the processor. The ADSP-BF535 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 shut down 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, undefined instructions, etc. cause exceptions.
- Interrupts
- Asynchronous to program flow. These are caused by timers, peripherals, input pins, etc.

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 kernel stack.

The ADSP-BF535 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.

DMA Support

The ADSP-BF535 processor has independent DMA channels for each DMA capable peripheral that supports automated data transfers with minimal processor core overhead. DMA transfers can occur between the ADSP-BF535 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. These include the SDRAM controller, asynchronous

memory controller, and the PCI bus interface. DMA capable peripherals include the SPORTs, SPIs, UARTs, USBs, and memory DMA controller. Each individual DMA capable peripheral has at least one dedicated DMA channel. DMA to and from PCI is accomplished via the memory DMA channel.

To describe each DMA sequence, the DMA controller uses a set of parameters, called a *descriptor block*. When successive DMA sequences are needed, these descriptor blocks can be linked or chained together so the completion of one DMA sequence autoinitiates and starts the next sequence. The descriptor blocks include full 32-bit addresses for the base pointers for source and destination, enabling access to the entire Blackfin address space.

In addition to the dedicated peripheral DMA channels, there is a separate memory DMA channel provided for transfers between the various ADSP-BF535 system memories. This enables transfers of blocks of data between any of the memories, including on-chip Level 2 memory, external SDRAM, ROM, SRAM and flash memory, and PCI address spaces with little processor intervention.

External Bus Interface Unit

The External Bus Interface Unit on the ADSP-BF535 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 up to four separate banks of industry-standard SDRAM devices or DIMMs. Fully compliant with the PC133 SDRAM standard, each bank can be configured to contain between 16 and 128 Mbytes of memory.

The controller maintains all banks as a contiguous address space. The processor sees this as a single address space, even if different size devices are used in different banks. This allows a system to be upgraded with either similar or different memories.

A set of programmable timing parameters is available to configure SDRAM banks to support slower memory devices. The memory banks can be configured as either 32 bits wide for maximum performance and bandwidth or 16 bits wide for minimum device count and lower system cost.

All four banks share common SDRAM control signals and have their own bank select lines, providing a glueless interface for most system configurations.

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 64 Mbyte window in the processor address space, but if not fully populated, these are not made contiguous by the memory controller. The banks can also be configured as 16- or 32-bit wide buses for interfacing to a range of memories and I/O devices for high performance or low cost and power.

PCI Interface

The ADSP-BF535 processor provides a 33 MHz, 32-bit, revision 2.2 compliant Peripheral Component Interconnect (PCI) interface. The PCI provides a bus bridge function between the processor core and on-chip peripherals and an external PCI bus. The PCI interface of the ADSP-BF535 processor supports two PCI functions:

- A host to PCI Bridge function, in which the ADSP-BF535 processor resources (the processor core, internal and external memory, and the memory DMA controller) provide the necessary hardware components to emulate a host PC-PCI interface from the perspective of a PCI target device.
- A PCI target function, in which an ADSP-BF535 processor-based intelligent peripheral can be designed to easily interface to a revision 2.2 compliant PCI bus.

PCI Host Function

The ADSP-BF535 processor provides the necessary PCI host (platform) functions to support and control a variety of off-the-shelf PCI I/O devices, such as Ethernet controllers, bus bridges, etc. in systems where the ADSP-BF535 processor is the host.

The three PCI address spaces (memory, I/O, and configuration space) are mapped into the ADSP-BF535 flat 32-bit memory space. Since the PCI memory space is as large as the ADSP-BF535 memory address space, a segmented, or windowed, approach is employed. This uses separate windows in the ADSP-BF535 address space for accessing the three PCI address spaces. Base address registers are provided so that these windows can be positioned to view any range in the PCI address spaces while they remain fixed in position in the ADSP-BF535 processor address range.

For devices on the PCI bus that view the ADSP-BF535 resources, several mapping registers are provided to allow resources to be viewed in the PCI address space. The ADSP-BF535 external memory space, internal L2, and some I/O MMRs can be selectively enabled as target memory spaces. Devices on the PCI bus can use these as PCI memory transaction targets.

PCI Target Function

As a PCI target device, the PCI host processor can configure the ADSP-BF535 subsystem during enumeration of the PCI bus system. The ADSP-BF535 subsystem then acts as an intelligent I/O device. As a target device, the PCI controller uses the memory DMA controller to perform DMA transfers required by the PCI host.

USB Port

The ADSP-BF535 processor provides a USB 1.1-compliant device type interface to support direct connection to a host system. The USB core interface provides a programmable environment with up to eight end-points. Each endpoint can support all of the USB data types, including Control, Bulk, Interrupt, and Isochronous. Each endpoint provides a memory-mapped buffer for transferring data to the application. The ADSP-BF535 USB port has a dedicated DMA controller and interrupt input to minimize processor polling overhead and to enable asynchronous requests for CPU attention when only transfer management is required.

Real-Time Clock

The ADSP-BF535 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 ADSP-BF535 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, or day clock ticks, or interrupt on programmable stopwatch countdown.

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 6-bit second counter, a 6-bit minute counter, a 5-bit hours counter, and an 8-bit 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.

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

Like the other peripherals, the RTC can wake up the ADSP-BF535 processor from a low power state upon generation of any interrupt.

Watchdog Timer

The ADSP-BF535 processor includes a 32-bit timer, which can be used to implement a software watchdog function. A software watchdog can improve system availability by forcing the processor to a known state, via generation of a hardware reset, non-maskable 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, which 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 timer can be programmed to reset only the ADSP-BF535 CPU, or both the CPU and the ADSP-BF535 peripherals. After a reset, software can determine if the watchdog was the source of the hardware reset by interrogating a status bit in the timer control register, which is set only upon a watchdog generated reset.

Timers

There are three general-purpose programmable timer units in the ADSP-BF535 processor. Each timer has one external pin that can be configured either as a Pulse Width Modulator (PWM) or timer output, as an input to clock the timer, or for measuring pulse widths of external events. Each of the three timer units can be independently programmed as a PWM, internally or externally clocked timer, or pulse width counter.

The timer units can be used in conjunction with either of the UARTs to measure the width of the pulses in the data stream 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.

The ADSP-BF535 processor uses a programmable interval timer to generate periodic interrupts. An 8-bit prescale register sets the number of clock cycles for decrementing a 32-bit count register. The number of clock cycles per timer decrement can be from 1 to 256. An interrupt is generated when this count register reaches 0. The count register can be automatically reloaded from a 16-bit period register and the count resumed.

Serial Ports (SPORTs)

The ADSP-BF535 processor incorporates two complete synchronous serial ports (SPORT0 and SPORT1) for serial and multiprocessor communications. The SPORTs support these features:

- Bidirectional operation
    - Each SPORT has independent transmit and receive pins.
- Buffered (8 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 either uses an external serial clock or generates a wide range of frequencies.
- Word length
    - Each SPORT supports serial data words from 3 to 16 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. The chained DMA can be dynamically allocated and updated through the descriptor blocks or DMA parameters that set up the chain.
- 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 and is compatible with the H.100, H.110, MVIP-90, and HMVIP standards.

Serial Peripheral Interface (SPI) Ports

The ADSP-BF535 processor has two SPI-compatible ports that enable 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. Two SPI chip select input pins let other SPI devices select the processor, and fourteen 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 ports provide a full duplex, synchronous serial interface, which supports both master and slave modes and multimaster environments.

Each SPI port baud rate and clock phase/polarities are programmable, and each has an integrated DMA controller, configurable to support either transmit or receive data streams.

During transfers, the SPI ports simultaneously transmit and receive by serially shifting data in and out on their two serial data lines. The serial clock line synchronizes the shifting and sampling of data on the two serial data lines.

UART Ports

The ADSP-BF535 processor provides two full-duplex independent Universal Asynchronous Receiver/Transmitter (UART) ports: UART0 and UART1. UART0 enhances the standard functionality and supports the half duplex IrDA® SIR (9.6/115.2 Kbps rate) protocol. The UART ports provide a simplified UART interface to other peripherals or hosts. They provide full duplex, DMA-supported, asynchronous transfers of serial data. Each UART port includes support for 5 to 8 data bits; 1 or 2 stop bits; and none, even, or odd parity. The UART ports support two modes of operation:

- Programmed I/O (PIO)
- The processor sends or receives data by writing or reading I/O-mapped UART transmit and receive registers, respectively. 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. Each 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 baud rate, serial data format, error code generation and status, and interrupts can be programmed to support the following:

- 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

Programmable Flags

The ADSP-BF535 processor supports 16 bidirectional programmable flag (PF) or general-purpose I/O pins, PF[15:0]. Each pin can be individually configured as either an input or an output.

Each PF pin can be further configured to generate an interrupt. When a PF pin is configured as an input, an interrupt can be generated according to the state of the pin (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.

When a PF pin is configured as an output, enabling interrupts for the pin provides a software interrupt mechanism. Output sensitivity is defined on a per-bit basis.

The ADSP-BF535 processor provides two independent interrupt channels for the PF pins.

The PF pins are also used by the Serial Port Interface (SPI) as a chip select and by the phase locked loop (PLL) circuitry to determine the multiplication factor applied. For more information, see "Serial Port Controllers" on page 11-1 and "Dynamic Power Management" on page 8-1.

Low Power Operation

The ADSP-BF535 processor provides four operating modes, each with a different performance/power dissipation profile. In addition, Dynamic Power Management provides the control functions with the appropriate external power regulation capability to dynamically alter the processor core supply voltage to further reduce power dissipation. Control of clocking to each of the ADSP-BF535 peripherals also reduces power dissipation.

Full On Operating Mode (Maximum Performance)

In the Full On mode, the phase-locked loop (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 Operating Mode (Low Power Savings)

In the Active mode, the PLL is enabled, but bypassed. The input clock is used to directly generate the clocks for the processor core and peripherals. When running off the input clock rather than the PLL, significant power savings can be achieved. Before transitioning back to the Full On Operating mode, software can dynamically change the PLL multiplication ratio by writing the appropriate values in the PLL Control register, choosing either to increase performance or reduce power dissipation.

Sleep Operating Mode (High Power Savings)

The Sleep mode reduces power consumption by disabling the clock to the processor core. The system clock however, continues to operate in this mode. Any interrupt, typically via some external event or Real-Time clock (RTC) activity, will wake up the processor. In this mode, the core proces-

sor is effectively shut down while all peripherals continue to operate. Without clocking to the processor core, significant power savings are achieved.

Deep Sleep Operating Mode (Maximum Power Savings)

The Deep Sleep mode maximizes power savings by disabling the clocks to the processor core and to all synchronous systems. Asynchronous systems, such as the RTC, continue to operate but access to processor resources is limited. This powered-down mode can only be exited by assertion of the reset interrupt or by an interrupt generated by the RTC.

Clock Signals

The ADSP-BF535 processor can be clocked by a sine wave input or a buffered, shaped clock derived from an external clock oscillator.

The processor provides a user-programmable 1 to 31 multiplication of the input clock to support external to internal (processor core) clock ratios. At runtime, the multiplication factor can be controlled in software.

All on-chip peripherals operate at the rate set by the system clock. The system clock frequency is programmable by the PLL Control register. The programmable values define a divide ratio between the core clock (CCLK) and the system clock (SCLK).

Boot Modes

The ADSP-BF535 processor has three mechanisms for automatically loading internal L2 memory after a reset. A fourth mode is provided to execute from external memory, bypassing the boot sequence:

- Execute from 16-bit external memory (Bypass boot ROM)
- Boot from 8-bit flash memory
- Boot from SPI0 serial ROM (8-bit address range)
- Boot from SPI0 serial ROM (16-bit address range)

Instruction Set Description

The Blackfin family assembly language instruction set uses an algebraic syntax that compiles to a small final memory size. The set also provides multifunction instructions that allow the programmer to use many of the processor core resources in a single instruction. The architecture supports both user (algorithm/application code) and supervisor (O/S kernel, device drivers, debuggers, ISRs) modes of operations.

Development Tools

The processor is supported by a complete set of software and hardware development tools, including Analog Devices' emulators and the CrossCore Embedded Studio or VisualDSP++ development environment. (The emulator hardware that supports other Analog Devices processors also emulates the processor.)

The development environments support advanced application code development and debug with features such as:

- Create, compile, assemble, and link application programs written in C++, C, and assembly
- Load, run, step, halt, and set breakpoints in application programs
- Read and write data and program memory
- Read and write core and peripheral registers
- Plot memory

Analog Devices DSP emulators use the IEEE 1149.1 JTAG test access port 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 JTAG interface—the emulator does not affect target system loading or timing.

Software tools also include Board Support Packages (BSPs). Hardware tools also include standalone evaluation systems (boards and extenders). In addition to the software and hardware development tools available from Analog Devices, third parties provide a wide range of tools supporting the Blackfin processors. Third party software tools include DSP libraries, real-time operating systems, and block diagram design tools.

2 COMPUTATIONAL UNITS

The ADSP-BF535 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. Tracing inputs to the computational units and considering details about

register files and data buses shows how to set up the 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 ADSP-BF535 Data Register File and computational units: multipliers, ALUs, and shifter.

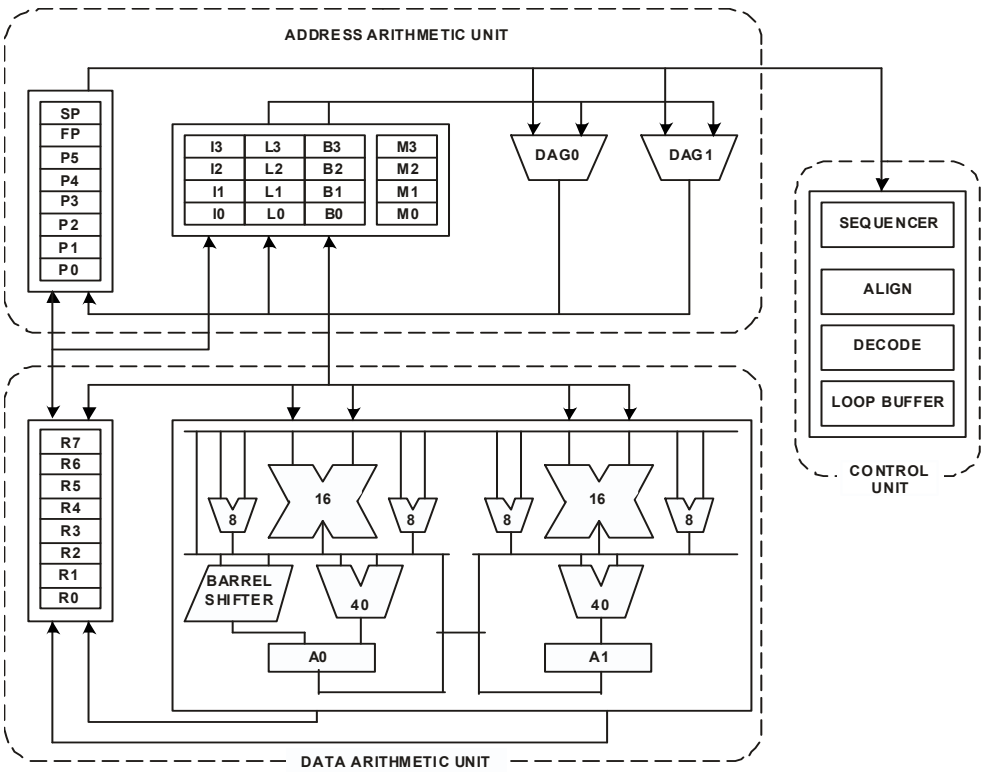


Figure 2-1. ADSP-BF535 Core Architecture

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 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/conditions for the results of the computational operations.

Using Data Formats

Blackfin 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 "Numeric Formats" on page D-1.

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

Binary String

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

Unsigned

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

Signed Numbers: Two's-Complement

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

Fractional Representation: 1.15

Blackfin processor arithmetic is optimized for numerical values in a fractional binary format denoted by 1.15 ("one dot fifteen"). In the 1.15 format, one sign bit (the MSB) and fifteen fractional bits represent values from –1 up to one LSB less than +1.

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

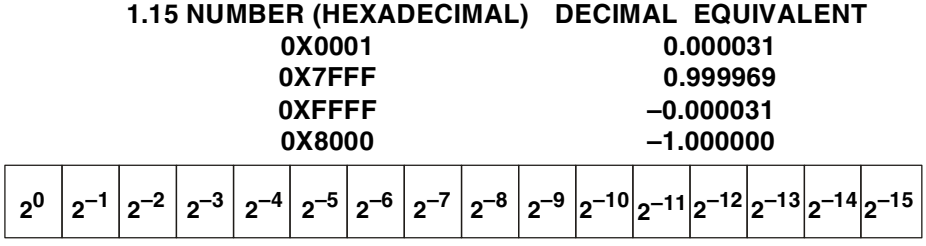


Figure 2-2. Bit Weighting for 1.15 Numbers

Register Files

The ADSP-BF535 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 "Data Address Generators" on page 5-1.

The ADSP-BF535 processor register files appear in Figure 2-3.

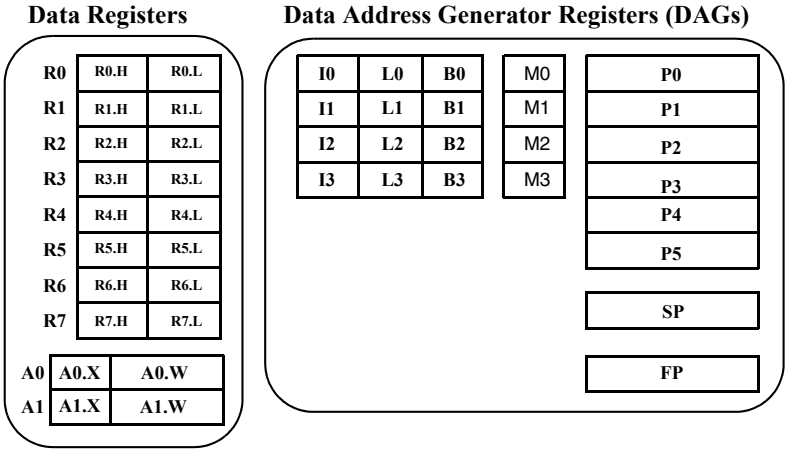


Figure 2-3. ADSP-BF535 Processor Register Files

In the ADSP-BF535 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 of which are 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`

Two separate buses 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 ADSP-BF535 processor has two dedicated, 40-bit accumulator registers. Each may be referred to as its 16-bit low half (`An.L`) or high half (`An.H`) plus its 8-bit extension (`An.X`), or 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 the P-registers, are organized as:

- A 6-entry, P-register file `P[5:0]`
- A Frame Pointer (`FP`) used to point to the current procedure's activation record

- A Stack Pointer register (SP) used to point to the last used location on the runtime stack. See mode dependent registers in "Operating Modes and States" on page 3-1.

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 (Base) and L (Length) registers define circular buffers. B contains the starting address of a buffer, and L 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 "Data Address Generators" on page 5-1.

Register File Instruction Summary

Table 2-1 lists the register file instructions. For more information about assembly language syntax, see *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`, `EMUDAT`, `CYCLES`, and `CYCLES2`.
- A*n* denotes `A0` or `A1`.
- Dreg denotes any Data Register File register.
- Sysreg denotes `ASTAT`, `SEQSTAT`, `SYSCFG`, `RETI`, `RETX`, `RETN`, `RETE`, or `RETS`, `LC[1:0]`, `LT[1:0]`, `LB[1:0]`, `EMUDAT`, `CYCLES`, and `CYCLES2`.
- Preg denotes any Pointer register, FP or SP register.
- Dreg_even denotes `R0`, `R2`, `R4`, or `R6`.
- Dreg_odd denotes `R1`, `R3`, `R5`, or `R7`.
- DPreg denotes any Data Register File register or any Pointer register, `FP`, or `SP` register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- A*n*.L denotes the lower 16 bits of Accumulator `A`*n*`.W`.
- A*n*.H denotes the upper 16 bits of Accumulator `A`*n*`.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 AC1	AV0 AV0S	AV1 AV1S	CC	V VS
allreg = allreg ; [1]	*	*	–	–	–	–	*
An = An ;	–	–	–	–	–	–	–
An = Dreg ;	–	–	–	–	–	–	–
Sysreg = Preg ;	–	–	–	–	–	–	–
Dreg_even = A0 ;	*	*	–	–	–	–	*
Dreg_odd = A1 ;	*	*	–	–	–	–	*
Dreg_even = A0, Dreg_odd = A1 ;	*	*	–	–	–	–	*
Dreg_odd = A1, Dreg_even = A0 ;	*	*	–	–	–	–	*
Dreg_even = A0.X ;	*	*	–	–	–	–	*
Dreg_odd = A1.X ;	*	*	–	–	–	–	*
IF CC DPreg = DPreg ;	–	–	–	–	–	–	–
IF ! CC DPreg = DPreg ;	–	–	–	–	–	–	–
Dreg = Dreg_lo (Z) ;	*	**	**	–	–	–	**/–
Dreg = Dreg_lo (X) ;	*	*	**	–	–	–	**/–
An.X = Dreg_lo ;	*	*	–	–	–	–	*
Dreg_lo = An.X ;	*	*	–	–	–	–	*

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

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC1	AV0 AV0S	AV1 AV1S	CC	V VS
An.L = Dreg_lo ;	*	*	–	–	–	–	*
An.H = Dreg_hi ;	*	*	–	–	–	–	*
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 ADSP-BF535 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 2-2 illustrates the data 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 *Blackfin Processor Programming Reference*.

Data Formats

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

Table 2-2. Data Formats (Cont'd)

Format	Representation in Memory	Representation in 32-Bit Register
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 6-76.

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 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 may result in either a negative or positive result, but cannot 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 2-28.

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 ADSP-BF535 processors support 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 on page 2-17.

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 on page 2-17.

Multiplier results generate status information when they are transferred to a destination register in the register file. For more information, see "Multiplier Instruction Summary" on page 2-35.

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 2-48.

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 ADSP-BF535 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 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.

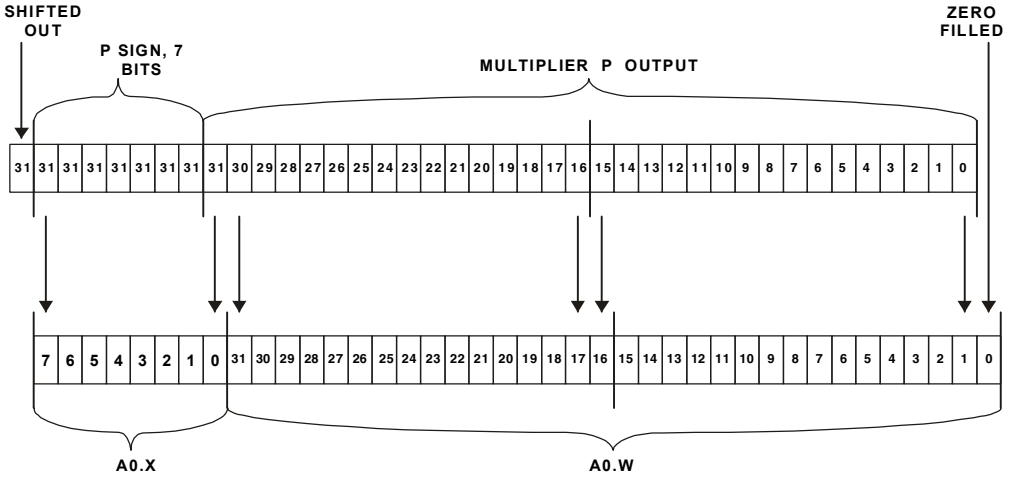


Figure 2-4. Fractional Multiplier Results Format

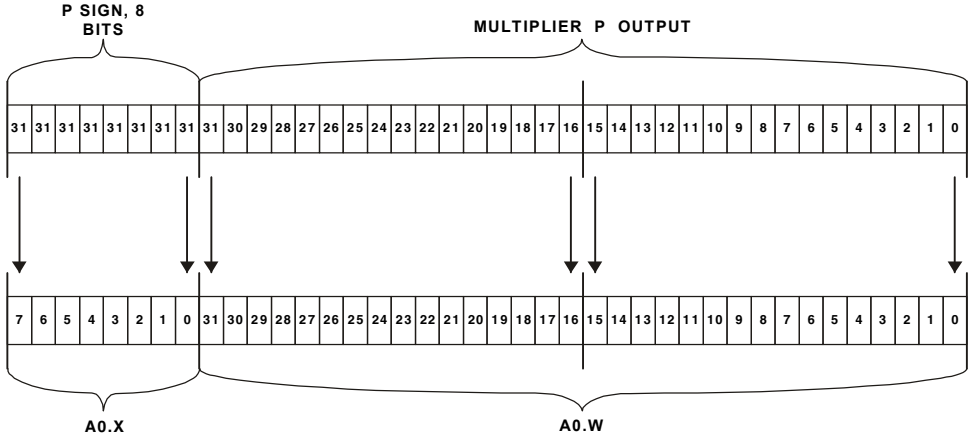


Figure 2-5. Integer Multiplier Results Format

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 multiplier'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, A0.H/A1.H contain the rounded 16-bit result; the rounding effect in A0.H/A1.H affects A0.X/A1.X as well. The A0.X/A0.H and A1.X/A1.H registers represent the rounded 24-bit result, including sign extension and overflow.

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 zero when it detects this midway point. Forcing bit 16 to zero has the effect of rounding odd `A0.L/A1.L` values upward and even values downward, yielding a zero large sample bias, 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`.

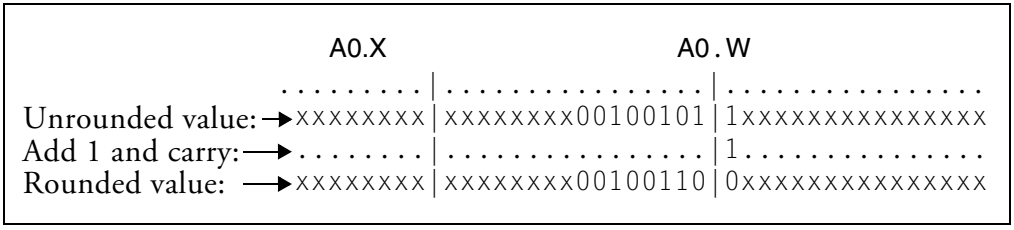


Figure 2-6. Typical Unbiased Multiplier Rounding

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

In Figure 2-7, A0 bit 16 is forced to zero. 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.

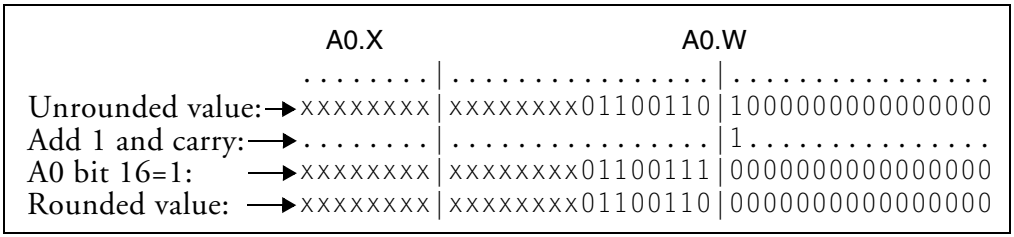


Figure 2-7. Avoiding Net Bias in Unbiased Multiplier Rounding

Biased Rounding

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

The RND_MOD bit in the ASTAT register enables biased rounding. When the RND_MOD bit is cleared, the RND option in multiplier instructions uses the normal, unbiased rounding operation, as discussed in "Unbiased Rounding" on page 2-18.

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 Figure 2-8.

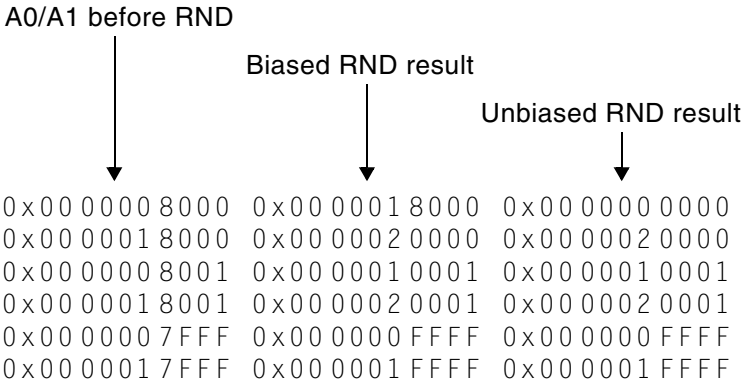


Figure 2-8. Biased Rounding in Multiplier Operation

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 (bit 5, 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.

Arithmetic Status Register (ASTAT)

Figure 2-9 describes the ASTAT register. The processor updates the status bits in ASTAT, indicating the status of the most recent ALU, multiplier, or shifter operation.

See Appendix A, "ADSP-BF535 Considerations", in the *Blackfin Processor Programming Reference* for information regarding how ALU operations affect this register.

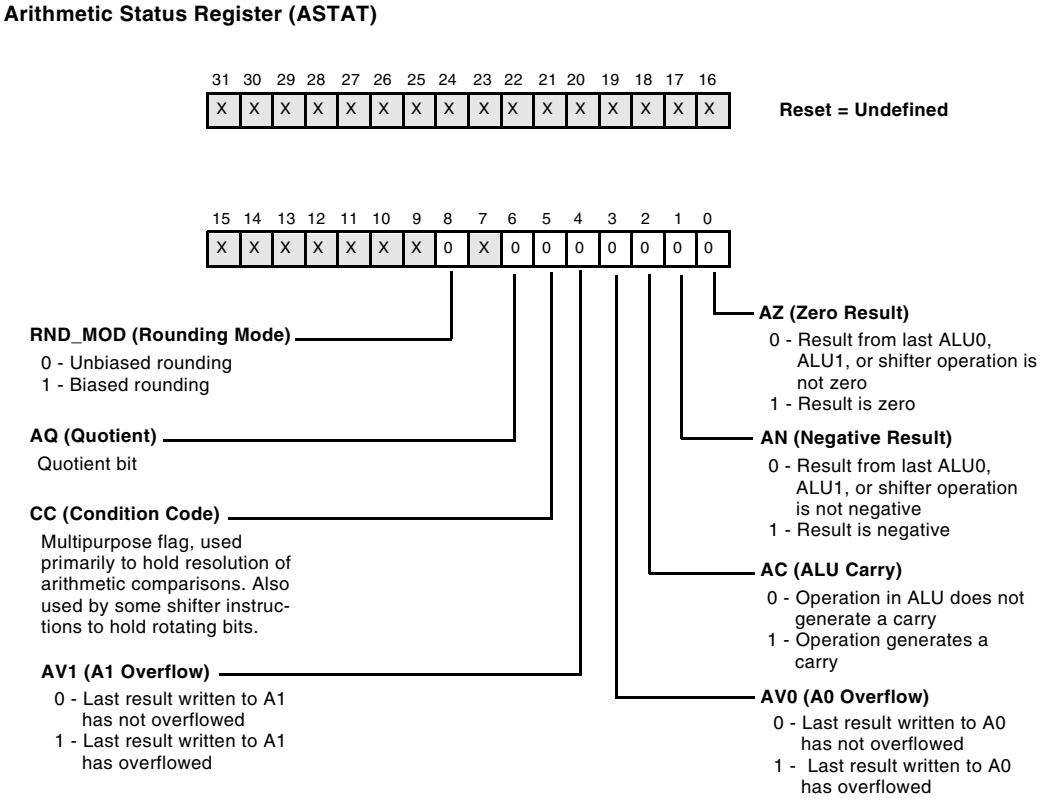


Figure 2-9. Arithmetic Status Register

Arithmetic Logic Unit (ALU)

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

- Fixed-point addition and subtraction of registers
- Addition and subtraction of immediate values
- Accumulator 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 2-7 describes the possible inputs and outputs of each ALU.

Table 2-7. 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 2-11 on page 2-30).

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 the `R0.H (R0` high half) and deposits the result in `R3.H`, with saturation.
- Adds `R1.L + R0.L` and deposits the result in `R3.L`, with saturation.
- Subtracts 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 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.

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

For information about assembly language syntax and the effect of ALU instructions on the status flags, see Appendix A, "ADSP-BF535 Considerations", in the *Blackfin Processor Programming Reference*.

ALU Data Flow Details

Figure 2-10 shows a more detailed diagram of the Arithmetic Units and Data Register File, which appears in Figure 2-1 on page 2-2.

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.

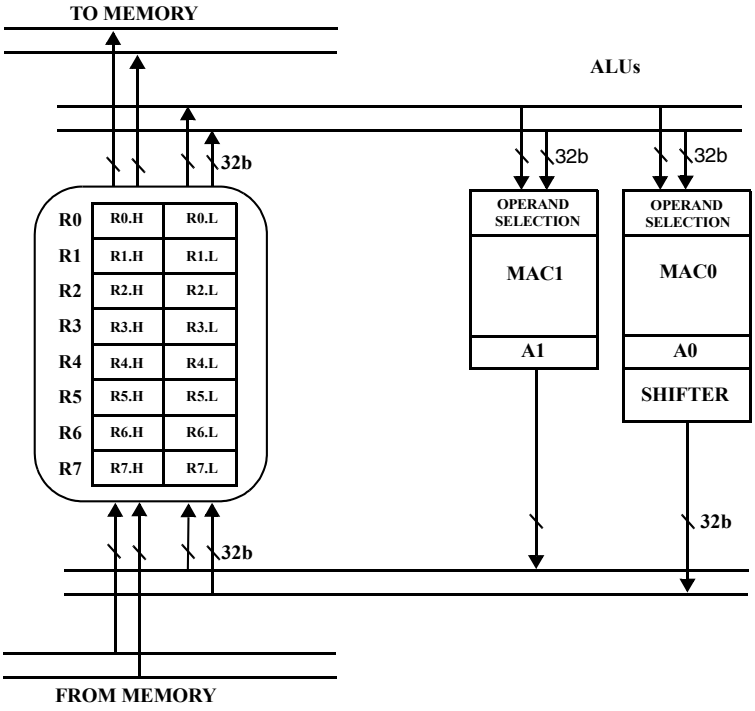


Figure 2-10. Register Files and ALUs

As shown in Figure 2-11, 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
```

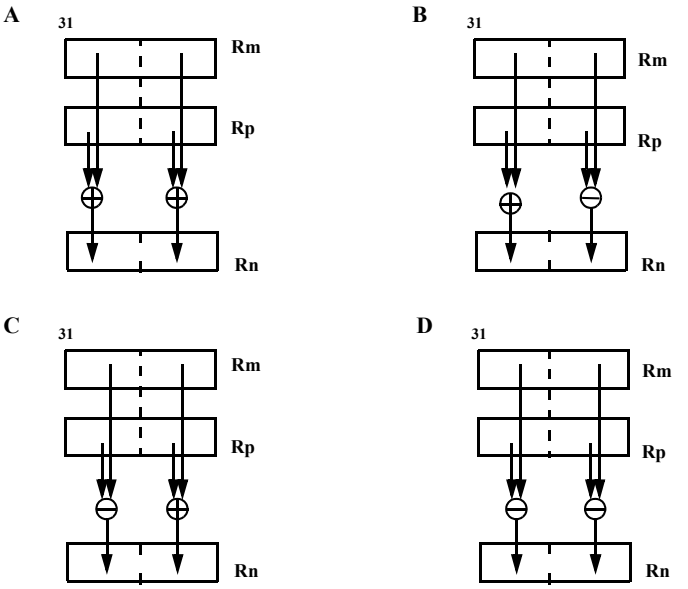


Figure 2-11. Dual 16-Bit ALU Operations

Dual 16-Bit Cross Options

For dual 16-bit operations, the results may be *crossed*. Crossing the results changes where in the result register an operation places 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 placed in the high half of the destination register (see Figure 2-12). This is particularly useful when dealing with complex math and portions of the Fast Fourier Transform (FFT).

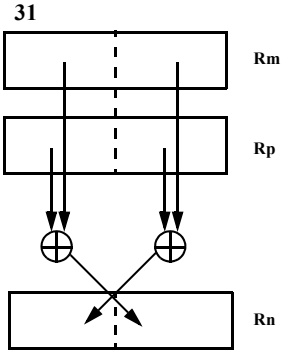


Figure 2-12. Cross Options for Dual 16-Bit ALU Operations

ALU Status Signals

Each ALU generates five status signals: the zero (AZ) status, the negative (AN) status, the carry (AC) status, 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 Appendix A, "ADSP-BF535 Considerations", in the *Blackfin Processor Programming Reference*.

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), add-subtract-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 *Blackfin Processor Programming Reference*.

Special SIMD Video ALU Operations

Four 8-bit Video ALUs enable the ADSP-BF535 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:

- Byte alignment
- Quad-byte-sum absolute differences
- Quad-byte averaging
- Quad-byte pack and unpack
- Addition with trimming

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

Multiply Accumulators (Multipliers)

The two ADSP-BF535 processor 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. Multiplier instruction options specify the data format of the inputs:

- `FU` for Fractional Unsigned
- `IS` for Integer Signed
- `M` for Mixed signed and unsigned operands

Also available is the `W32` option, which specifies 32-bit saturation of the accumulation result.

Placing Multiplier Results in Multiplier Accumulator Registers

As shown in Figure 2-10 on page 2-29, 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.
- 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

These bits in ASTAT indicate multiplier overflow status:

- Bit 3 (AV0) and bit 4 (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.

Multiplier Instruction Summary

For information about assembly language syntax and the effect of multiplier instructions on the status flags, see Appendix A, "ADSP-BF535 Considerations", in the *Blackfin Processor Programming Reference*.

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 *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.
(S2RND)	If multiplying and accumulating to a register: Input data operands are signed fraction. When copying to the destination register, scales Accumulator contents (multiply x2 by a one-place shift-left) and rounds. If scaling and rounding produce a signed value larger than 32 bits, the number is saturated to its maximum positive or negative value.
(S2RND)	If multiplying and accumulating to a half register: Input data operands are signed fraction. When copying to the destination register, scales Accumulator contents (multiply x2 by a one-place shift-left) and rounds the upper 16 bits before truncating the lower 16 bits of the Accumulator. If scaling and rounding produce a signed value larger than 16 bits, the number is saturated to its maximum positive or negative value.
(ISS2)	If multiplying and accumulating to a register: Input data operands are signed integer. When copying to the destination register, scales Accumulator contents (multiply 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.

(ISS2)	If multiplying and accumulating to a half register: When copying the lower 16 bits to the destination half-register, scales the Accumulator contents. 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, 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.
(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. The (M) option can be used alone or in conjunction with one other format option.

Multiplier Data Flow Details

Figure 2-13 shows the multiplier/accumulators.

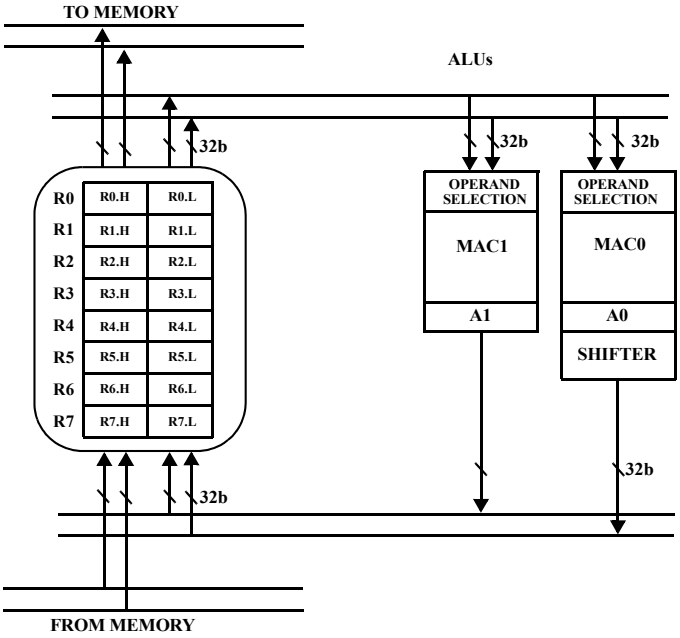


Figure 2-13. Register Files and ALUs

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 contain the same register information giving the options for squaring and multiplying the high half and low half of the same register. Figure 2-14 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 register. For results, the `A0` and `A1` registers are 40 bits wide. Each of these registers consists of smaller 32-bit and 8-bit registers: `A0.W`, `A1.W`, `A0.X`, and `A1.X`.

Some example instructions follow:

```
A0 = R3.L * R4.H (S) ;
```

The `MAC0` multiplier/accumulator performs a multiply and puts the result in the accumulator register.

```
A1 += R3.H * R4.H (S) ;
```

The `MAC1` multiplier/accumulator performs a multiply and accumulates the result with the previous results in the `A1` accumulator.

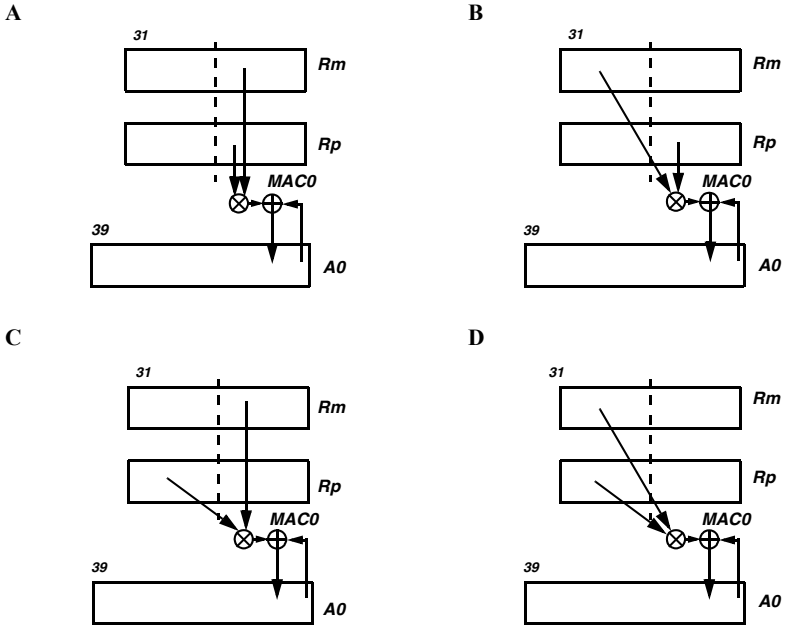


Figure 2-14. Four Possible Combinations of MAC Operations

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 2-15).
- 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 2-16).

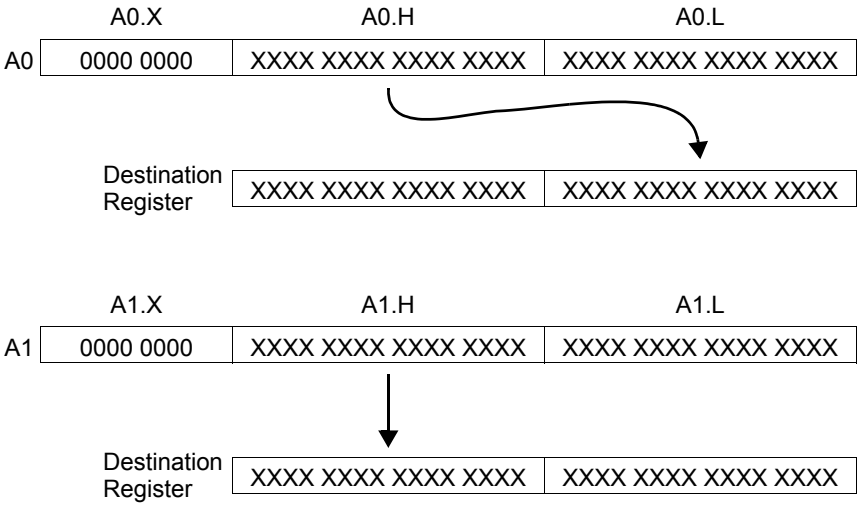


Figure 2-15. Multiplication of Fractional Operands

This example 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 example 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 (S) ;
```

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

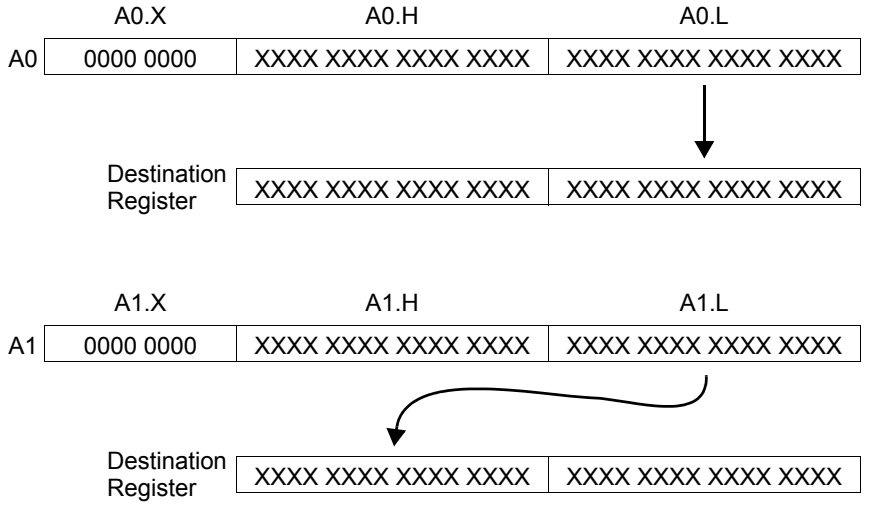


Figure 2-16. Multiplication of Integer Operands

Special 32-Bit Integer MAC Instruction

The ADSP-BF535 processor supports a multicycle 32-bit MAC instruction, that is,

```
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 *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.

Dual MAC Operations

The ADSP-BF535 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, in 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, in 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 2-40.

```
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- or 32-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. `ASHIFT` and `>>>` represents the arithmetic shift. `LSHIFT` and `>>` 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 2-48.

Two Operand Shifts

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

Immediate Shifts

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

The following example shows the input value down-shifted.

```
R0 = 0x0000 B6A3 ;
R0 >>= 0x04 ;
```

results in

```
R0.H = 0x0000 ;
R0.L = 0xB6A3 ;
```

The following example shows the input value up-shifted.

```
R0 = 0x0000 B6A3 ;
R0 <<= 0x04 ;
```

results in

```
R0.H = 0x000B ;
R0.L = 0x6A30 ;
```

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 32, then the result is either 0 or –1.

The following example shows the input value up-shifted.

```
R0 = 0x0000 B6A3 ;
R2 = 0x0000 0004 ;
R0 <<= R2 ;
```

results in

```
R0 = 0x000B 6A30 ;
```

Three Operand Shifts

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

Immediate Shifts

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

The following example shows the input value down-shifted.

```
R0 = 0x0000 B6A3 ;
R1 = R0 >> 0x04 ;
```

results in

```
R1 = 0x0000 0B6A ;
```

The following example shows the input value up-shifted.

```
R0.L = 0xB6A3 ;
R1.H = R0.L << 0x04 ;
```

results in

```
R1.H = 0x6A30 ;
```

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 6 bits of `Rn.L` are masked off and used as the shift value.

The following example shows the input value up-shifted.

```
R0 = 0x0000 B6A3 ;
R2.L = 0x0004 ;
R1 = R0 ASHIFT by R2.L ;
```

results in

```
R1 = 0x000B 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 4-12.

```
R0 = 0xABCD EF12 ;
R2.L = 0x0004 ;
R1 = R0 ROT by R2.L ;
```

results in

```
R1 = 0xBCDE F125 ;
```

Note that 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

For information about assembly language syntax and the effect of shifter instructions on the status flags, see Appendix A, “ADSP-BF535 Considerations”, in *Blackfin Processor Programming Reference*.

3 OPERATING MODES AND STATES

The ADSP-BF535 processor supports 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, non-maskable interrupt (NMI), or 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.

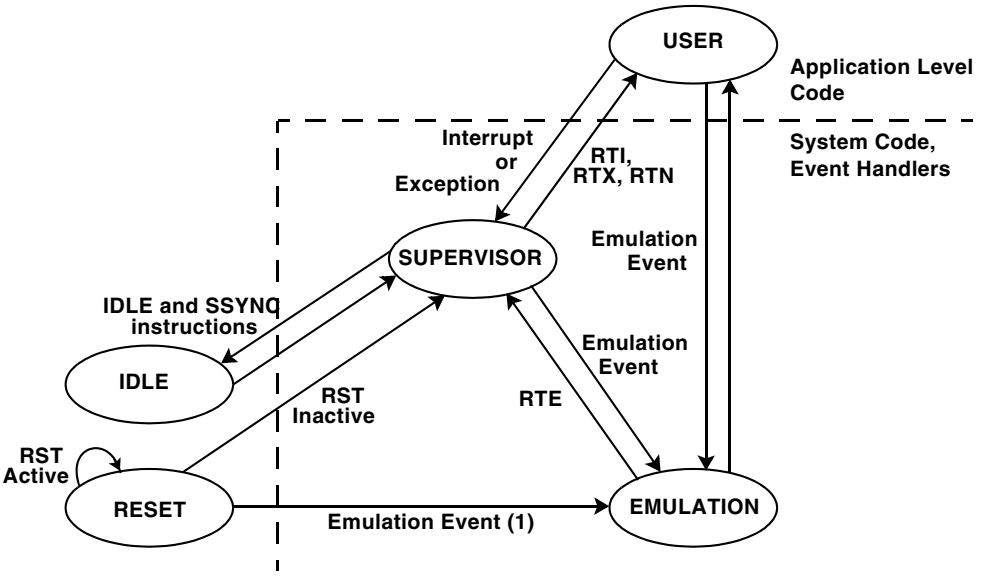
Table 3-1. Identifying the Current Processor Mode

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.
Emulation	Emulator	= 0x01 The processor is in Emulation mode if IPEND[0] = 1, and the state of the remaining bits IPEND[15:1] are undefined. Can be 0's or 1's.
None	User	= 0x00

In addition, the ADSP-BF535 processor supports two nonprocessing states:

- Idle state
- Reset state

Figure 3-1 illustrates the processor modes and states and 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 or 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 the 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. Attempts to access MMR space in User mode cause an exception.

A list of protected instructions appears in Table 3-3. Attempts 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
IDLE	Idle
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, as described in "Memory Management Unit" on page 6-56.

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.

Example Code to Enter User Mode Upon Reset

Listing 3-1 provides code for entering User mode from the Reset state.

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. 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. Once RETx has been loaded, the RTx instruction can be issued.

Note the stack pop is optional. The 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, as described in "Memory Management Unit" on page 6-56. Only Supervisor mode can use the register alias USP, which locates the User Stack Pointer in memory.

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 preempt this activity. Lower priority events cannot be processed.

One method to keep the processor in Supervisor mode and still allow lower priority events to be processed can be implemented by setting up and forcing the lowest priority interrupt (IVG15). Events and interrupts are described further in "Events and Sequencing" on page 4-17. Once 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. Once 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 to Stay in Supervisor Mode Coming Out of Reset

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

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

```
P0.L = IVG15_EVT & 0xFFFF ;   /* Point to IVG15 in Event Vector
Table */
P0.H = (IVG15_EVT >> 16) & 0xFFFF;
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 = IMASK & 0xFFFF ;
R0 = W[P0] ;
R1.L = IVG15 & 0xFFFF ;
R0 = R0 | R1 ;
W[P0] = R0 ;   /* Set (enable) IVG15 bit in SIC 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 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 combination of the `IDLE` and `SSYNC` instructions in Supervisor mode. The `IDLE` instruction notifies the processor hardware that the Idle state is requested. The `SSYNC` instruction purges all speculative and transient states in the core and external system.

Without the `SSYNC` instruction, the `IDLE` instruction does *not* place the processor into an Idle state.

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.

Idle state can be entered only in Supervisor mode, and the processor returns to Supervisor mode when transitioning from the Idle state. Application programs in User mode cannot invoke the Idle state, except through a system call provided by an operating system kernel.

In the following code example, core interrupts are disabled and the IDLE instruction is executed. When all the pending processes have completed, the core disables its clocks. Idle state can be terminated only by asserting a WAKEUP signal. For more information, see "System Interrupt Wakeup-Enable Register (SIC_IWR)" on page 4-24. While not required, an interrupt could also be enabled in conjunction with the WAKEUP signal.

When the WAKEUP signal is asserted, the processor finishes executing the SSYNC instruction. The next instruction in this sequence should be the STI. Interrupts are enabled after this instruction is executed.

Example Code for Transition to Idle State

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

Listing 3-3. Transitioning to Idle State

```
CLI R0 ;   /* disable interrupts */
IDLE ;   /* source NOPs into pipeline and assert IDLE output on
SSYNC */
SSYNC ;   /* drain the pipeline, IDLE asserts after SSYNC_ACK
back from system */
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.

The processor remains in the Reset state as long as external logic asserts the external 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 identified in the Event Vector Table (EVT).

Software in Supervisor or Emulation mode can invoke the Reset state without involving the external RESET signal by issuing the Reset version of the RAISE instruction. After the reset sequence, the processor returns to the Supervisor or Emulation mode that issued 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 on Reset

Item	Description of Reset State
Core	
Operating mode	Supervisor mode in reset event
Rounding mode	Unbiased rounding
Cycle counters	Disabled
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	Random (must be set to invalid prior to cache initialization)
System	
Booting methods	Determined by the values of BMODE pins at reset
MSEL clock frequency	Determined by sampling MSEL pins at reset

Table 3-5. Processor State on Reset (Cont'd)

Item	Description of Reset State
PLL Bypass mode	Determined by sampling BYPASS pin at reset
Core clock/system clock ratio	Determined by sampling SSEL pins at reset
Peripheral clocks	Enabled

System Reset and Power-up Configuration

Table 3-6 describes the five types of ADSP-BF535 processor resets. Note all resets, except System Software, reset the core.

Table 3-6. ADSP-BF535 Processor Resets

Reset	Source	Result
Hardware reset	The ADSP-BF535 processor $\overline{\texttt{RESET}}$ 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 "System Reset Configuration Register (SYSCR)" on page 3-14.
System Software reset	Writing b#111 to bits [2:0] in the system MMR SWRST at address 0xFFC0 0410 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 Peripheral Clock Enable register (PLL_IOCK) and the No Boot on Software Reset bit in SYSCR. Does not reset the core.

Table 3-6. ADSP-BF535 Processor Resets (Cont'd)

Reset	Source	Result
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 DPMC resets only PLL_IOCK. The Software Reset register (SWRST) can be read to determine whether the reset source was the watchdog timer.
Core Double-Fault reset	If the core enters a double-fault state, a reset can be caused by unmasking the Core Double-Fault Reset Mask bit in the SIC Interrupt Mask register (SIC_IMASK).	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The DPMC resets only PLL_IOCK. SWRST can be read to determine whether the reset source was Core Double-Fault.
Core-Only Software reset	Executing a RAISE1 instruction or writing to an MMR through the emulator causes a Core-Only Software reset.	Resets only the core. The peripherals have no knowledge of this reset.

Hardware Reset

The ADSP-BF535 processor chip reset is an asynchronous reset event. The $\overline{\text{RESET}}$ input pin must be deasserted to perform a Hardware reset. See *ADSP-21535 Blackfin Embedded Processor Data Sheet* for more information.

A hardware initiated reset results in a system wide reset that includes both core and peripherals. After the $\overline{\text{RESET}}$ pin is deasserted, the ADSP-BF535 processor ensures that all asynchronous peripherals, such as the Universal Serial Bus (USB), have recognized and completed a reset. After the reset, the ADSP-BF535 processor transitions into the boot mode sequence configured by the BMODE state.

The BMODE[2:0] pins are dedicated mode control pins. No other functions are shared with these pins, and they may be permanently strapped by tieing them directly to either VDD or VSS. The pins and the corresponding bits in SYSCR configure the boot mode that is employed after Hardware reset or System Software reset. See also "Reset" on page 4-36, and Table 4-11 on page 4-40.

The MSEL[6:0], and DF pins are shared with the programmable flag pins PF[7:0]. During Hardware reset, these pins are sensed for configuration state. Configuration state may either be directly driven by off-chip hardware or strapped high or low by resistor. The sensed state is used to configure the phase locked loop. For more information, see "Phase Locked Loop and Clock Control" on page 8-2.

System Reset Configuration Register (SYSCR)

The values sensed from the BMODE[2:0] pins are latched into the System Reset Configuration register (SYSCR) upon the deassertion of the $\overline{\text{RESET}}$ pin. They 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).

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

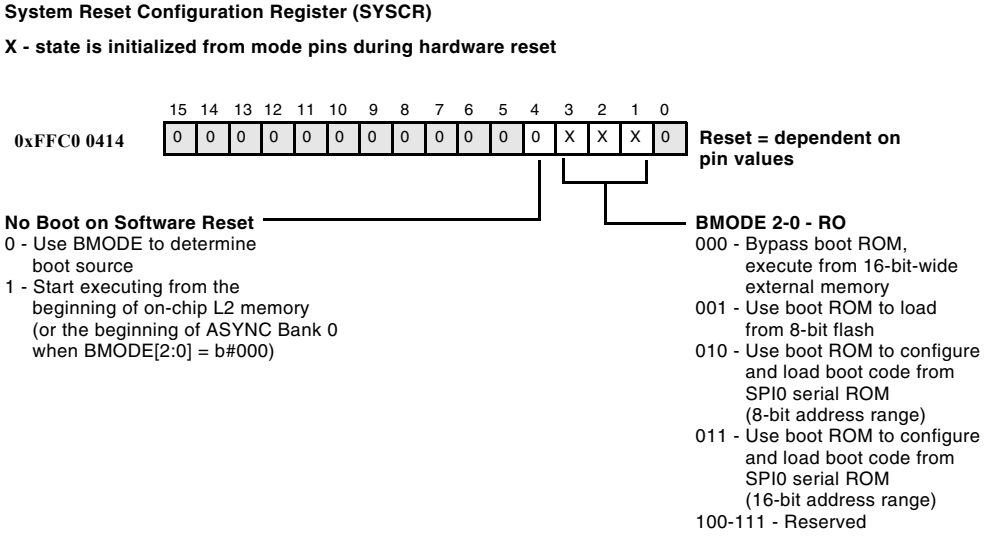


Figure 3-2. System Reset Configuration Register

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.

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 ADSP-BF535 processor ensures that all asynchronous peripherals, such as USB, have recognized and completed a reset.

For a reset generated by the watchdog timer, the ADSP-BF535 processor 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[2:0] control bits.

Software Reset Register (SWRST)

A software reset can be initiated by setting the System Software Reset field in the Software Reset register (SWRST), which is shown in Figure 3-3. 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 [2:0] are read/write.

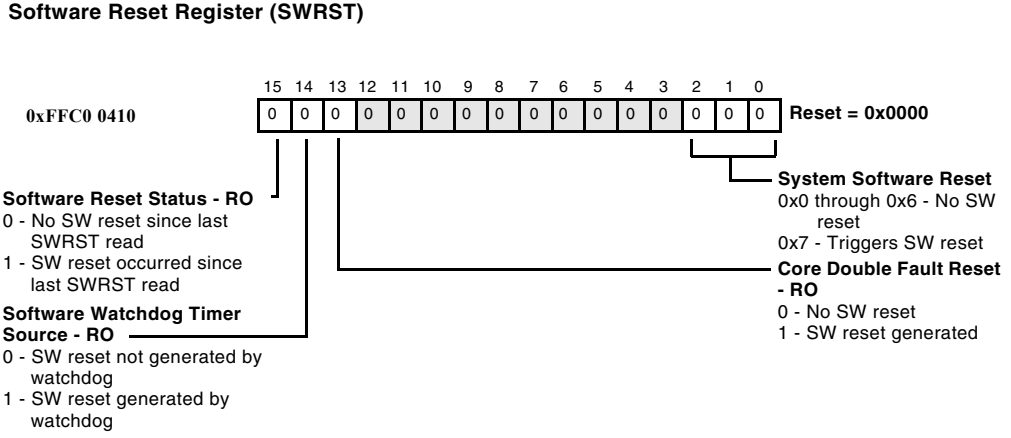


Figure 3-3. Software Reset Register

When the BMODE pins are not set to b#000 and the No Boot on Software Reset bit in SYSCR is set, the ADSP-BF535 processor starts executing from the start of on-chip L2 memory. In this configuration, the core begins fetching instructions from address 0xF000 0000 (the beginning of on-chip L2 memory).

When the BMODE pins are set to b#000 and the No Boot on Software Reset bit is set, the core begins fetching instructions from address 0x2000 0000 (the beginning of ASYNC Bank 0).

Core Only Software Reset

A 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.

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, as defined in Table 4-10 on page 4-37. The boot kernel reads the BMODE[2:0] pin state at reset to identify the download source (see Table 4-7 on page 4-22). When in Bypass mode, 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. For these modes, the boot kernel sets up the selected peripheral based on the BMODE[0:2] pin settings.

For each boot mode, user code read in from the memory device is placed at the starting location of L2 memory (0xF000 0000). The boot kernel terminates the boot process with a jump to the start of the L2 memory space. The processor then begins execution from this address.

If booting from SPI0, general-purpose flag pin 10 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. The Core Only Software reset resets only the core; it does not affect the rest of the ADSP-BF535 processor 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 L2 memory and begins execution.

The boot kernel assumes these conditions for the flash boot mode:

- 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 AOE deasserted) = 3 cycles
- Bank 0 read/write access times = 15 cycles

The boot kernel assumes that the SPI baud rate is 500 kHz. Both 8-bit and 16-bit addressable SPI serial PROMs are supported. A second stage loader is available to include the SPI baud rate up to 2 MHz.

4 PROGRAM SEQUENCER

In the ADSP-BF535 processor, the Program Sequencer controls program flow, constantly providing the address of the next instruction to be executed by other parts of the ADSP-BF535 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 ADSP-BF535 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.

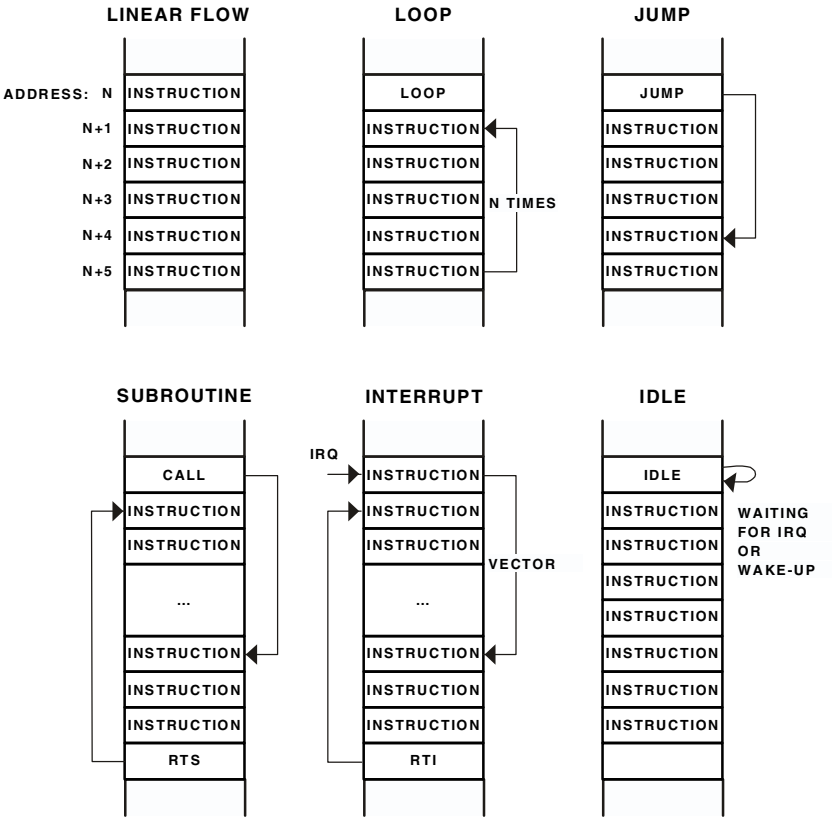


Figure 4-1. Program Flow Variations

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 ADSP-BF535 processor that are related to the Sequencer. Except for the `PC` register, 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) or `Rn = [SP++]` (for pop).

Table 4-1. Sequencer Related Registers

Register Name	Description
SEQSTAT	Sequencer Status register
	Return Address registers: See "Events and Sequencing" on page 4-17.
RETX RETN RETI RETE RETS	Exception Return NMI Return Interrupt Return Emulation Return Subroutine Return
	Zero-Overhead Loop registers:
LC0, LC1 LT0, LT1 LB0, LB1	Loop Counters Loop Tops Loop Bottoms
FP, SP	Frame Pointer and Stack Pointer: See "Data Address Generators" on page 5-1.
SYSCFG	System Configuration Register
CYCLES, CYCLES2	Cycle Counters: See "Blackfin Processor's Debug" on page 20-1.

Sequencer Status Register (SEQSTAT)

The Sequencer Status register (SEQSTAT), shown in Figure 4-2, 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.

Sequencer Status Register (SEQSTAT)

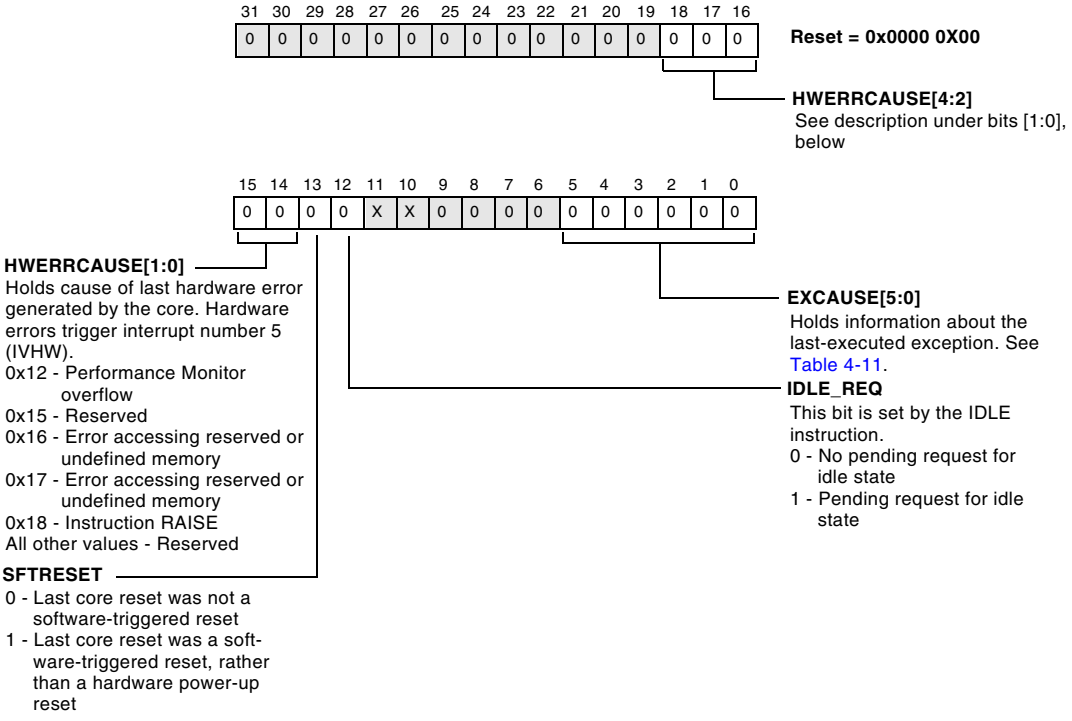


Figure 4-2. Sequencer Status Register

Zero-Overhead Loop Registers (LC, LT, 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.

The 32-bit loop register sets are described in Table 4-2.

Table 4-2. Loop Registers

Registers	Description	Function
LC[1:0]	Loop Counters	Maintain a count of the remaining iterations of the loop
LT[1:0]	Loop Tops	Hold the address of the first statement within a loop
LB[1:0]	Loop Bottoms	Hold the address of the last statement of the loop

System Configuration Register (SYSCFG)

The System Configuration Register (SYSCFG), shown in Figure 4-3, controls the configuration of the processor. This register is accessible only from the Supervisor mode.

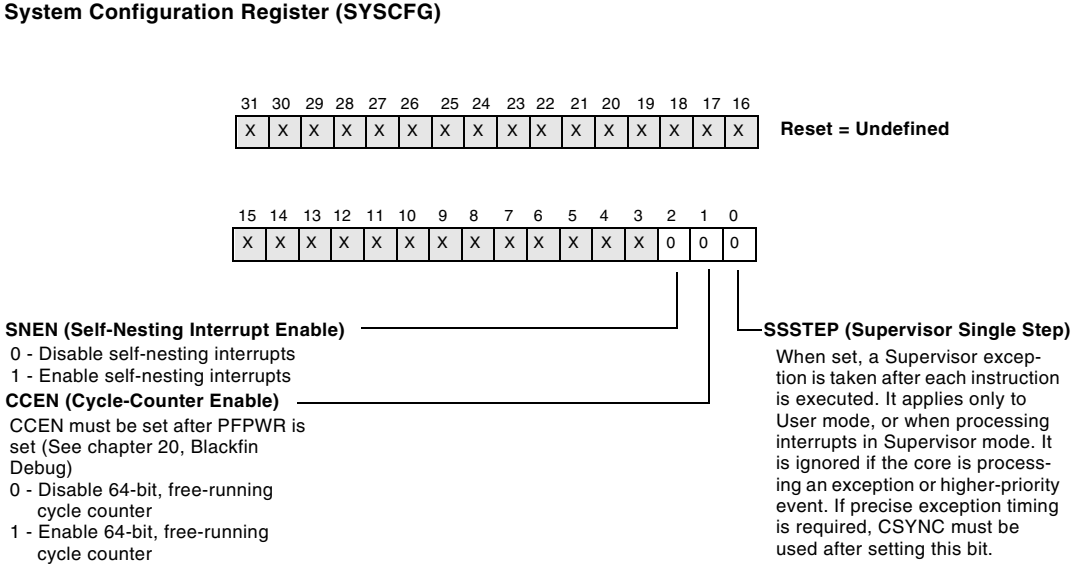


Figure 4-3. System Configuration Register

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 ADSP-BF535 processor has an eight-stage instruction pipeline, shown in Figure 4-4.

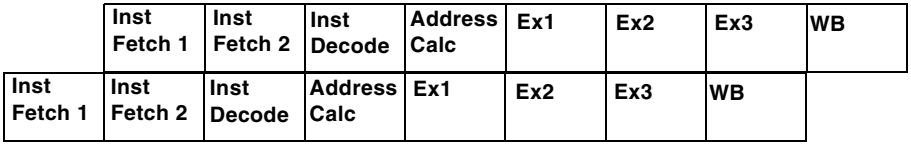


Figure 4-4. ADSP-BF535 Processor Pipeline

Table 4-3. Stages of Instruction Pipeline

Pipeline Stage	Description
Instruction Fetch 1 (IF1)	Start instruction memory access.
Instruction Fetch 2 (IF2)	Finish L1 instruction memory access and align instruction.
Instruction Decode (DEC)	Start instruction decode and access Pointer register file.
Address Calculation (AC)	Calculate data addresses and branch target address.
Execute 1 (EX1)	Read data and start access of data memory.
Execute 2 (EX2)	Finish accesses of data memory and start execution of dual cycle instructions.
Execute 3 (EX3)	Execute single cycle instructions.
Write Back (WB)	Write states to Data and Pointer register files and process events.

The Sequencer decodes and distributes operations to the Instruction Memory Unit and Instruction Alignment Unit. It also controls stalling and invalidating the instructions in the pipeline. The Sequencer ensures that the pipeline is fully interlocked and that the programmer does not need to manage the 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 IF2 stage.

For each instruction type (16-, 32-, or 64-bit), the Alignment Unit ensures that the alignment buffers have enough valid data to be able to provide an instruction every cycle. Since the instructions can be 16, 32, or 64 bits wide, the Alignment Unit may not need to fetch data from the cache every cycle. For example, for a series of 16-bit instructions, the Alignment Unit gets data from the Instruction Memory Unit once in 4 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. It 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, the Sequencer communicates it to the Instruction Memory Unit, which invalidates the data in the Alignment Unit. In addition to the change-of-flow indication, the Sequencer can also kill instructions in IF1 and IF2 stages or stall the Instruction Memory Unit.

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 EX1 pipeline stage (for operands) and the EX3 pipeline stage (for stores). Writes occur in the WB stage. The multipliers and the video unit are active in the EX2 stage, and the ALUs and shifter are active in the EX3 stage. The accumulators are written at the end of the EX3 stage.

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

- Program memory data accesses that conflict with instruction fetches
- 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 *Blackfin Processor Programming Reference*. Briefly, these instructions operate as follows:

- A `JUMP` or a `CALL` instruction transfers program flow to another memory location. The difference between a `JUMP` and a `CALL` is that a `CALL` automatically 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 are return from subroutine (RTS), return from interrupt (RTI), return from exception (RTX), return from emulation (RTE), and return from non-maskable interrupt (RTN). Each return type has its own register for holding the return address.
- JUMP instructions can be conditional, depending on the status of the CC bit of the ASTAT register. They 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 because of the effects of the pipeline.

Branches can be direct or indirect. The difference is that the Sequencer generates the address for a direct branch, for example JUMP 0x30, and the Data Address Generator produces the address for an indirect branch, for example JUMP (P3).

Direct branches are JUMP or CALL instructions that use a PC-relative address.

Indirect branches are JUMP or CALL instructions that use a dynamic address—an address that changes at runtime—that comes from a Data Address Generator. For more information, see "Data Address Generators" on page 5-1.

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 zero must be a zero). 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 zero must be a zero). 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 0xnnnn` instruction must be used. If the destination offset is unknown and development tools must evaluate the offset, then use the instruction `JUMP 0xnnnn`. 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 location of the instruction after the call into the `RETS` register. The `CALL` instruction has a 25-bit PC-relative offset that must be a multiple of two (bit zero must be a zero). 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 use a dynamic address—an address that changes at runtime—that comes from the Data Address Generator. The effective address is stored in a P-register. For the `CALL` instruction, the `RETS` register is loaded with the address of the instruction following the `CALL` instruction.

For example:

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

PC-Relative Indirect Branch and Call

The PC-relative indirect JUMP and CALL instructions use 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 following the CALL instruction.

For example:

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

Condition Code Flag

The ADSP-BF535 supports a condition code (CC) flag bit, which is used to resolve the direction of a branch. This flag may be accessed five 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.
- A status flag may be copied into CC, and the value in CC may be copied to a status flag.
- CC may be set to the result of a Pointer register comparison.
- CC may be set to the result of a Data register comparison.

These five 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 branch on true or 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 register is set.

Conditional Branches

The Sequencer supports conditional branches. These are JUMP instructions whose execution is based on testing an IF condition that is based on the status 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 zero must be a zero). 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

For condition handling where the value of a register is set based on the condition code, a conditional move instruction can be used instead of a branch statement. A register move 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.

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.

The branch target address calculation takes place in the AC stage of the instruction pipeline. In the EX3 stage, the Sequencer compares the actual CC bit value to the predicted value. If the value was mis-predicted, 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: 6 CCLK cycles.
- If prediction was “to take branch,” and branch was actually taken: 3 CCLK cycles.
- If prediction was “to take branch,” and branch was actually not taken: 6 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 EX1 stage. All unconditional branches have a latency of 3 CCLK cycles.

Consider the two examples in Table 4-4.

Table 4-4. Branch Prediction

Instruction	Description
`If CC JUMP back_dest (bp)`	This instruction tests the `CC` flag, and if it is set, jumps to a location, identified by the label, `back_dest`. If `back_dest` is a prior address and the `CC` flag is set, then the branch is correctly predicted and the branch latency is reduced.
`If CC JUMP sub_dest (bp)`	This instruction tests the `CC` flag, and if it is set, jumps to a location, identified by the label, `sub_dest`. If `sub_dest` is a subsequent address and the `CC` flag is set, then the branch is incorrectly predicted and the branch latency increases.

Loops and Sequencing

Another type of nonsequential program flow that the Sequencer supports is looping. A loop occurs when the values in a pair of loop top and loop bottom registers define an address range that includes the currently executing instruction. The corresponding loop counter must contain a nonzero value. One way to load these registers is by using the Loop Setup (`LSETUP`) instruction; however, it is also possible to load these registers directly. The ADSP-BF535 processor provides two sets of dedicated registers to support two nested loops.

The condition for terminating a loop is that the counter decreases to zero. This condition tests whether the loop has completed the number of iterations loaded from the Loop Count register (`LC[1]` or `LC[0]`).

The zero-overhead loop hardware provides efficient loops without the overhead of additional instructions to branch, test a condition, or decrement a counter. If the effective range of the loop needs to be larger than that shown in Table 4-5, the loop can be set up manually by loading the LCx, LTx, and LBx registers manually.

The code example in Listing 4-1 shows a loop that contains two instructions and iterates 32 times.

Listing 4-1. Loop

```
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

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.

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

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

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).

The ADSP-BF535 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 shortens 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.

Events and Sequencing

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

- Emulation
- Reset
- Non-maskable interrupts (NMI)
- Exceptions
- Interrupts

Note that the word *event* describes all five types. The Event Controller manages fifteen 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 is preempted by one of higher priority.

The ADSP-BF535 processor employs a two-level event control mechanism. The ADSP-BF535 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 lowest two priority interrupts (`IVG14` and `IVG15`) be reserved for software interrupt handlers, leaving seven prioritized interrupt inputs (`IVG7 - IVG13`) to support the ADSP-BF535 processor system. Refer to Table 4-6.

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

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
System Interrupts	RTC USB PCI	IVG7
	SPORT0 RX/TX SPORT1 RX/TX	IVG8
	SPI0 SPI1	IVG9
	UART0 RX/TX UART1 RX/TX	IVG10
	Timer0, Timer1, Timer2	IVG11
	Programmable Flags Interrupt A/B	IVG12
	Memory DMA Watchdog Timer	IVG13
	Software Interrupt 1	IVG14
	Software Interrupt 2 (lowest priority)	IVG15

System Interrupt Processing

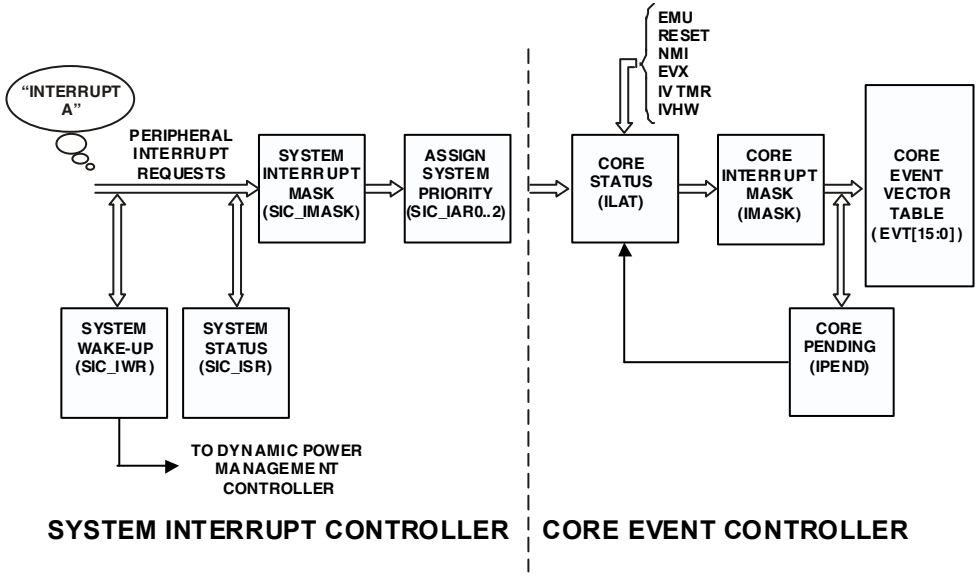
Referring to Figure 4-5, note that when an interrupt (Interrupt A) is generated by an interrupt-enabled peripheral:

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 ADSP-BF535 processor 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 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 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.



Note: Names in parentheses are memory-mapped registers.

Figure 4-5. Interrupt Processing Block Diagram

System Peripheral Interrupts

The ADSP-BF535 processor system has numerous peripherals, which therefore require many supporting interrupts. Table 4-7 lists:

- Peripheral interrupt source
- Peripheral interrupt ID used in the System Interrupt Assignment registers (`SIC_IARx`). See "System Interrupt Assignment Registers (SIC_IARx)" on page 4-29.
- 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 4-29.

Table 4-7. Peripheral Interrupt Source Reset State

Peripheral Interrupt Source	Peripheral Interrupt ID	General-purpose Interrupt (Assignment at Reset)	Core Interrupt ID
Real-Time Clock interrupts (alarm, second, minute, hour, countdown)	0	IVG7	0
Reserved	1	IVG7	0
USB interrupt	2	IVG7	0
PCI interrupts	3	IVG7	0
SPORT0 RX DMA interrupt	4	IVG8	1
SPORT0 TX DMA interrupt	5	IVG8	1
SPORT1 RX DMA interrupt	6	IVG8	1
SPORT1 TX DMA interrupt	7	IVG8	1
SPI0 DMA interrupt	8	IVG9	2
SPI1 DMA interrupt	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
UART0 RX interrupt	10	IVG10	3
UART0 TX interrupt	11	IVG10	3
UART1 RX interrupt	12	IVG10	3
UART1 TX interrupt	13	IVG10	3
Timer0 interrupt	14	IVG11	4
Timer1 interrupt	15	IVG11	4
Timer2 interrupt	16	IVG11	4
PF interrupt A	17	IVG12	5
PF interrupt B	18	IVG12	5
Memory DMA interrupt	19	IVG13	6
Software Watchdog Timer interrupt	20	IVG13	6

The peripheral interrupt structure of the ADSP-BF535 processor is flexible. By default upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core, as shown in the preceding table.

An interrupt service routine that supports multiple interrupt sources must interrogate the appropriate system 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 EVT vector address entries and `IMASK` register, and unmasking the specific peripheral interrupts in `SIC_IMASK` that the system requires.

System Interrupt Wakeup-Enable Register (SIC_IWR)

The SIC provides the mapping between the peripheral interrupt source and the Dynamic Power Management Controller (DPMC). Any of the ADSP-BF535 processor 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 (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 wake-up sequence in order to process the interrupt. Note that 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 "Dynamic Power Management" on page 8-1.

By default, all interrupts generate a wake-up request to the core. However, for some applications it may be desirable to disable this function for some peripherals, such as 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 wake-up 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 only when all peripheral interrupts are disabled.

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

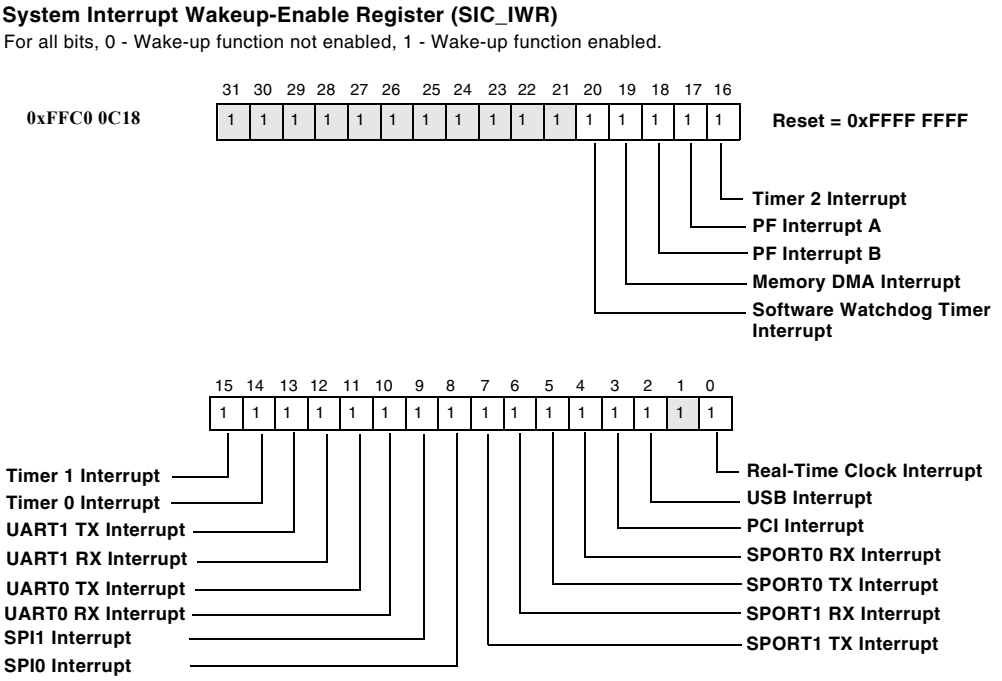


Figure 4-6. System Interrupt Wakeup-Enable Register

System Interrupt Status Register (SIC_ISR)

The SIC includes a read-only status register, the System Interrupt Status register, 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 that for some peripherals, such as programmable flag asynchronous input interrupts, many cycles of latency may pass from the time that an interrupt service routine initiates the clearing of the interrupt (usually by writing a system MMR) to the time that 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 Interrupt Mask register and can be read at any time. Writes to SIC_ISR have no effect on its contents.

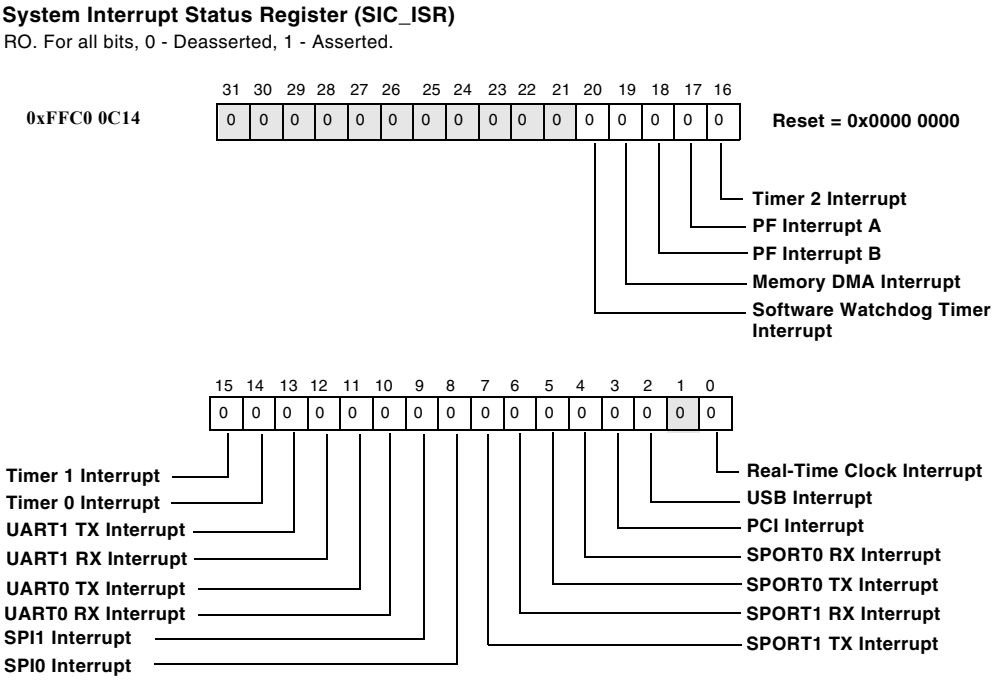


Figure 4-7. System Interrupt Status Register

System Interrupt Mask Register (SIC_IMASK)

The System Interrupt Mask register, shown in Figure 4-8, allows masking of any peripheral interrupt source at the 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.

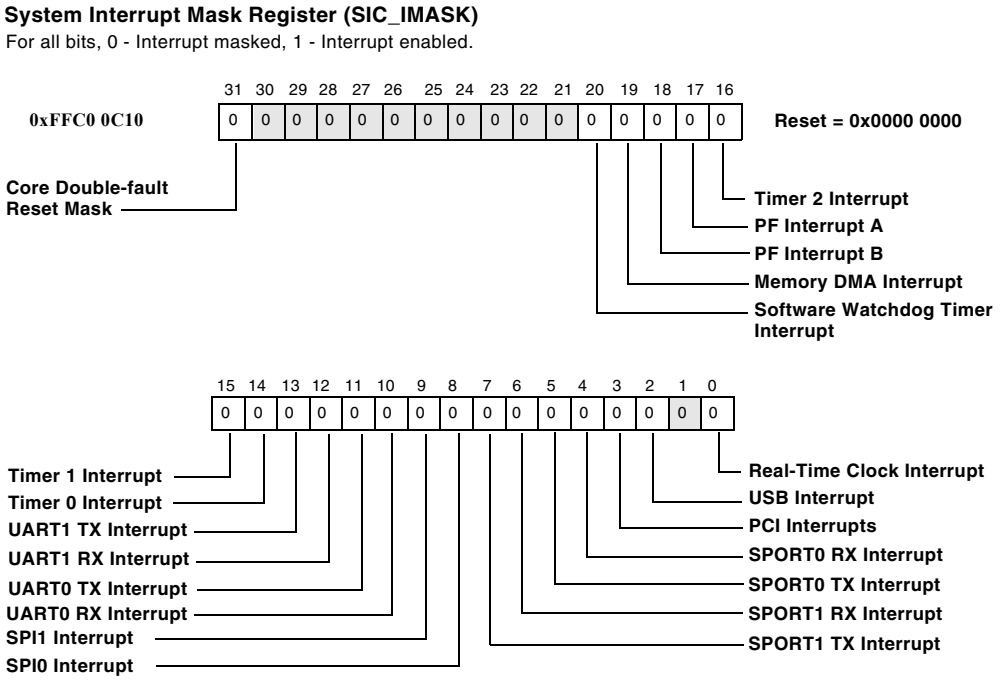


Figure 4-8. System Interrupt Mask Register

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.

`SIC_IMASK[31]` provides a mask for the core double-fault condition:

- If the condition is unmasked and the core detects a double-fault condition, a hardware reset is generated.
- If the condition is masked and the core detects a double-fault condition, core behavior may be unreliable.

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.

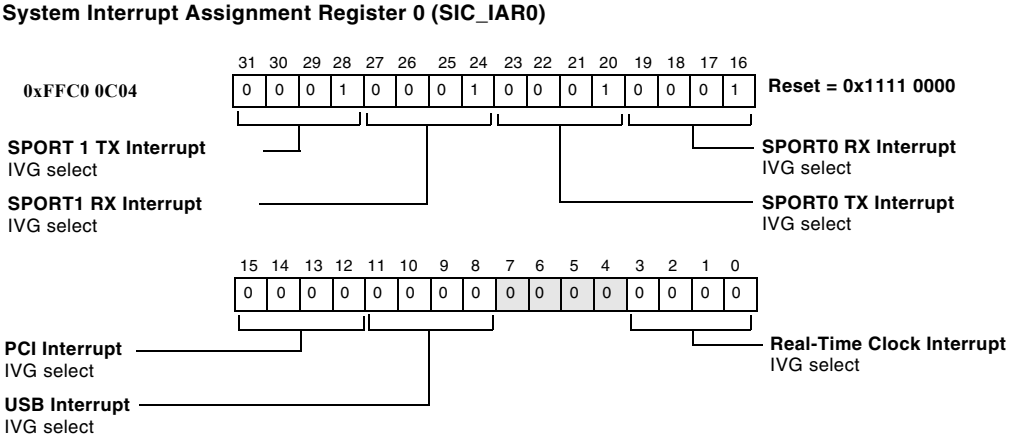


Figure 4-9. System Interrupt Assignment Register 0

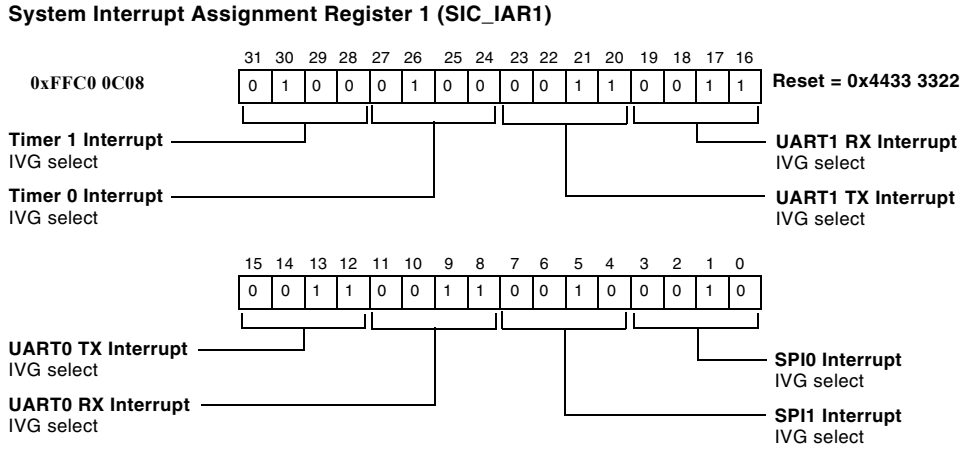


Figure 4-10. System Interrupt Assignment Register 1

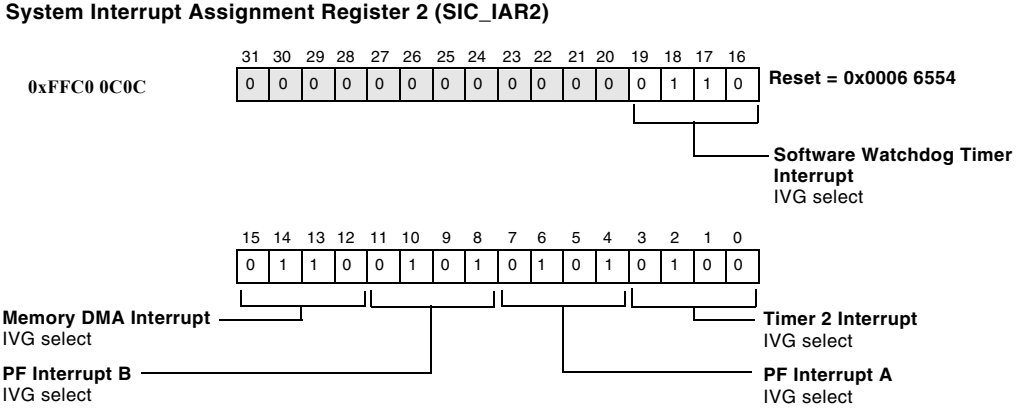


Figure 4-11. System Interrupt Assignment Register 2

These registers can be read or written 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

Event Controller Registers

The Event Controller uses three MMRs to coordinate pending event requests. Each register is 16 bits wide, and each bit, N, corresponds to the EVT[N] event in the Event Vector Table. The registers are:

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

The Event Controller updates ILAT and IPEND. The IPEND register is read-only in Supervisor mode. The ILAT and IMASK registers may be both read and written in Supervisor mode, with the exception of ILAT[0], which is read-only. None of the three registers can be accessed in User mode.

Core Interrupt Mask Register (IMASK)

Each bit in IMASK indicates when the corresponding interrupt is enabled (see Figure 4-12). When an interrupt bit is set in the ILAT register, the corresponding interrupt is accepted only if the corresponding bit is also set in the IMASK register. The IMASK register may be read and written in Supervisor mode.

Only the emulator may mask emulation events. Refer to "Blackfin Processor's Debug" on page 20-1.

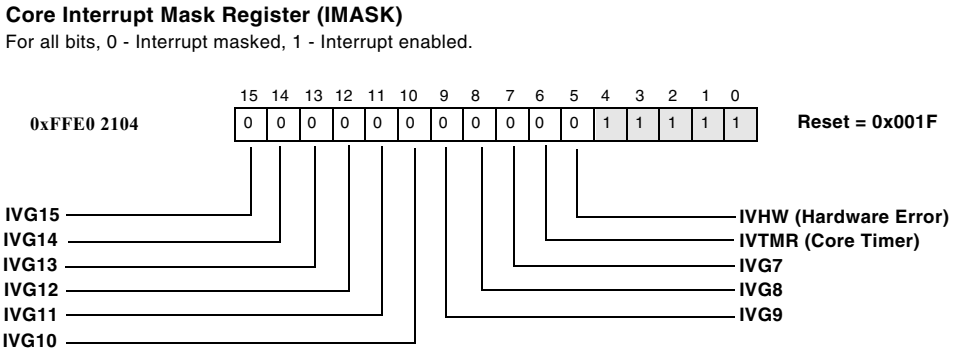


Figure 4-12. Core Interrupt Mask Register

Core Interrupt Latch Register (ILAT)

Each bit in ILAT indicates when the corresponding event is latched (see Figure 4-13). The bit is reset before the first instruction in the corresponding ISR is executed. Writes to any ILAT bit can only occur when the corresponding IMASK bit is set. Otherwise, the write is ignored. This write functionality to ILAT is provided for cases where latched interrupt requests need to be cleared (cancelled) instead of servicing them.

When an event is serviced, its corresponding bit in ILAT is cleared. The RAISE N instruction causes specific bits in ILAT to be set and can trigger only events IVG15-IVG7, IVTMR, IVHW, NMI and RST.

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

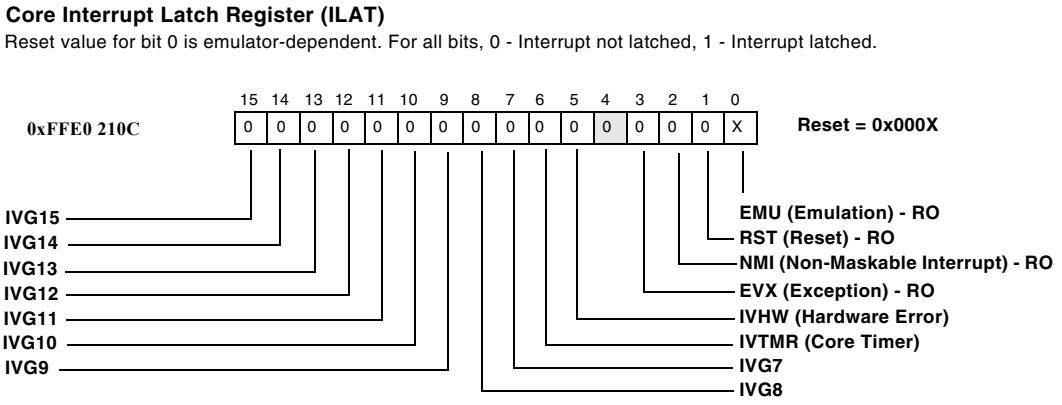


Figure 4-13. Core Interrupt Latch Register

Core Interrupts Pending Register (IPEND)

The IPEND register 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.

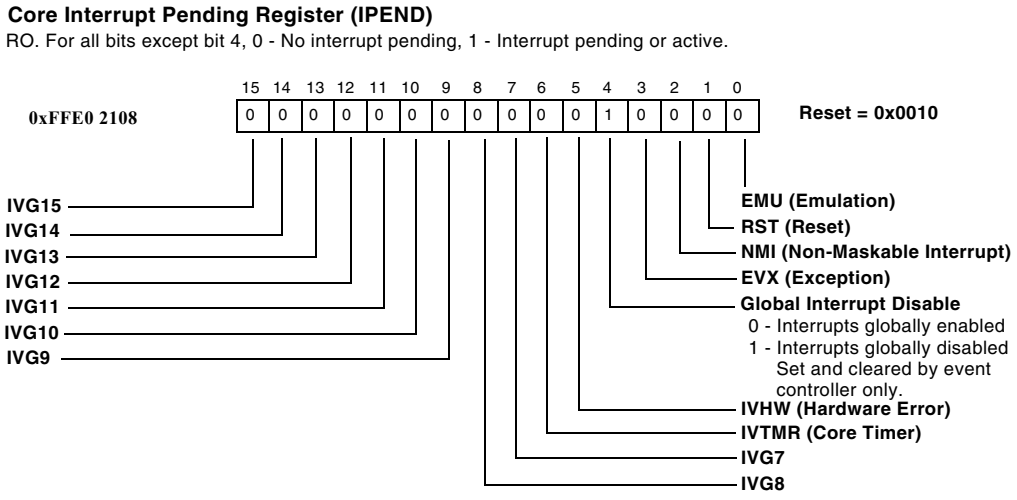


Figure 4-14. Core Interrupt Pending Register

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 in *Blackfin Processor Programming Reference*.

When program code is too time critical to be delayed by an interrupt, disable 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 ADSP-BF535 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	Read-only.
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, in which instructions are read from the JTAG interface. It is the highest priority interrupt to the core.

For detailed information about emulation, see "Blackfin Processor's Debug" on page 20-1.

Reset

The reset interrupt (`RST`) can be initiated via the `RESET` pin or through expiration of the Watchdog timer. The `EVT[1]` register holds the address at which the processor begins execution after reset. This location differs from that of other interrupts in that its content is read-only. Writes to this address have no effect.

The core has an output that indicates that a double-fault has occurred. This is a non-recoverable state. The SIC (via `SIC_IMASK`) 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 ADSP-BF535 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[2:0] pins. Refer to Table 4-10.

Table 4-10. Reset Vector Addresses

Boot Source	BMODE[2:0]	Execution Start Address
Bypass boot ROM; execute from 16-bit-wide external memory (Async Bank 0)	000	0x2000 0000
Use boot ROM to boot from 8-bit flash	001	0xF000 0000
Use boot ROM to configure and load boot code from SPI0 serial ROM (8-bit address range)	010	0xF000 0000
Use boot ROM to configure and load boot code from SPI0 serial ROM (16-bit address range)	011	0xF000 0000
Reserved	100-111	N/A

If the BMODE[2:0] pins indicate either booting from flash or serial ROM, 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[2:0] pins, which determine the appropriate boot sequence. For information about the ADSP-BF535 processor boot ROM, see "Booting Methods" on page 3-17.

If the BMODE[2:0] 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 (Non-Maskable Interrupt)

The NMI entry is reserved for a non-maskable interrupt, which can be generated by the Watchdog timer or by the NMI input signal to the ADSP-BF535 processor. An example of an event that requires immediate processor attention, and thus is appropriate as an NMI, is a power-down warning. During the execution of the NMI service routine, exceptions are suspended, as they have a lower priority than the NMI interrupt.

If an exception occurs during handling of an NMI, servicing the exception is delayed until completion of the NMI.

Exceptions

Exception events are synchronous to the instruction that generates the exception; that is, the exception is taken before the instruction is allowed to complete. Note, however, the architecture can generate exceptions for memory references that meet any of these criteria:

- Misalign
- Miss the ICPLB (for an instruction fetch), or DCPLB (for a load or store)
- Violate the protection specification for the CPLB entry
- Generate multiple hits on the respective CPLB
- Receive a bus error response on an instruction fetch

For more information about alignment and CPLBs, see "Memory" on page 6-1.

As shown in Table 4-11, exceptions can be services or error conditions. The value in the Type column indicates whether the exception for that row is a service or an error condition.

- For services (S), the return address is the address of the instruction that follows the exception, because a service is never re-executed.
- To allow the program to resume execution on return from a service routine, the processor state must be preserved before the excepting instruction.
- For error conditions (E), the return address is the address of the instruction that caused the exception, because some types of errors, such as a page fault, require that the offending instruction be re-executed.
- Determining whether the instruction is re-executed depends on the excepting instruction and error condition and is determined by the exception handler. When the instruction should be re-executed, the return address is used. When the instruction should not be re-executed, the exception handler must advance the return address by the instruction length.
- `EXCAUSE[5:0]` is placed in the Sequencer Status register (`SEQSTAT`), depending on the type of event.

Table 4-11. Events That Cause Exceptions

Exception	EXCAUSE[5:0]	Type: (E) error (S) service See note 1.	Notes/Examples
Force Exception instruction EXCPT with 4-bit field m	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 an emulation trace buffer overflow	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).

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

Exception	EXCAUSE[5:0]	Type: (E) error (S) service See note 1.	Notes/Examples
Data access misaligned address violation	0x24	E	Attempted misaligned data memory or data cache access.
Unrecoverable event	0x25	E	For example, an exception generated while processing a previous exception.
Data access CPLB miss	0x26	E	Used by the MMU to signal a CPLB miss on a data access.
Data access multiple CPLB hits	0x27	E	More than one CPLB entry matches data fetch address.
Exception caused by an emulation watchpoint match	0x28	E	There is a watchpoint match, and one of the EMUSW bits in the Watchpoint Instruction Address Control register (WPIACTL) is set.
Instruction fetch access exception	0x29	E	Error from instruction fetch, for example, instruction bus parity error.
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 that 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).

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

Exception	EXCAUSE[5:0]	Type: (E) error (S) service See note 1.	Notes/Examples
Instruction fetch CPLB miss	0x2C	E	CPLB miss on an instruction fetch.
Instruction fetch multiple CPLB hits	0x2D	E	More than one CPLB entry matches instruction fetch address.
Illegal use of supervisor resource	0x2E	E	Attempted to use a Supervisor register or instruction from User mode. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions.

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

If an instruction causes multiple 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

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

Priority	Exception	EXCAUSE
8	Undefined Instruction	0x21
9	Illegal Combination	0x22
10	Illegal use protected resource	0x2E
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. Generating an exception before the exception handler finishes results in these actions:

- 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.

To determine whether an exception occurred while an exception handler was executing, check SEQSTAT at the end of the exception handler. If an exception has occurred, register RETX holds the address of the instruction that caused the exception.

At the end of the exception handler, the return address is used to return normally to lower interrupt level code. Whether an exception occurs during the execution of an NMI, emulation, or reset event, the same set of actions occurs.

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:

- Attempted access to a reserved memory location
- Attempted access to uninitialized external memory space
- Bus parity errors
- Internal error conditions within the core, such as Performance Monitor overflow
- The DMA Access Bus Comparator interrupt (attempted write to an active DMA register)

- 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
DMA Bus Comparator Source	0b00001	0x01	The Compare Hit output is routed directly to the Hardware Error interrupt input. The Compare Hit interrupt is maskable by writing to the DMA Bus Control Comparator register (DB_CCOMP). See "DMA Bus Debug Registers" on page 20-27.
Performance Monitor Overflow	0b10010	0x12	Refer to "Performance Monitoring Unit" on page 20-19.
Error accessing reserved or undefined memory	0b10110	0x16	An access to reserved or uninitialized memory was attempted
Error accessing reserved or undefined memory	0b10111	0x17	An access to reserved or uninitialized memory was attempted
RAISE instruction	0b11000	0x18	Software issued a RAISE instruction to invoke the hardware error interrupt (IVHW).
Reserved	All other bit combinations.	All other bit combinations.	

Core Timer

The Core Timer Interrupt (IVTMR) is triggered when the core timer value reaches zero. See "Timers" on page 16-1.

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 can force 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 CEC has a single interrupt queueing element per event, as a bit in the ILAT register. The appropriate ILAT bit is set when an interrupt rising edge is detected (which takes 2 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 asserted 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 EVT becomes the next fetch address.

   On an interrupt, all 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-executed instruction 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.

Interrupts With and Without Nesting

Interrupts are handled either with or without nesting. 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.

If nested interrupts are desired, the return address to the interrupted point in the original interrupt service routine (ISR) must be explicitly saved and subsequently restored when execution of the nested ISR has completed. Nesting is enabled by pushing the return address currently held in RETI to the Supervisor stack (`[--SP] = RETI`), which is typically done early in the ISR prolog of the lower priority interrupt. 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.

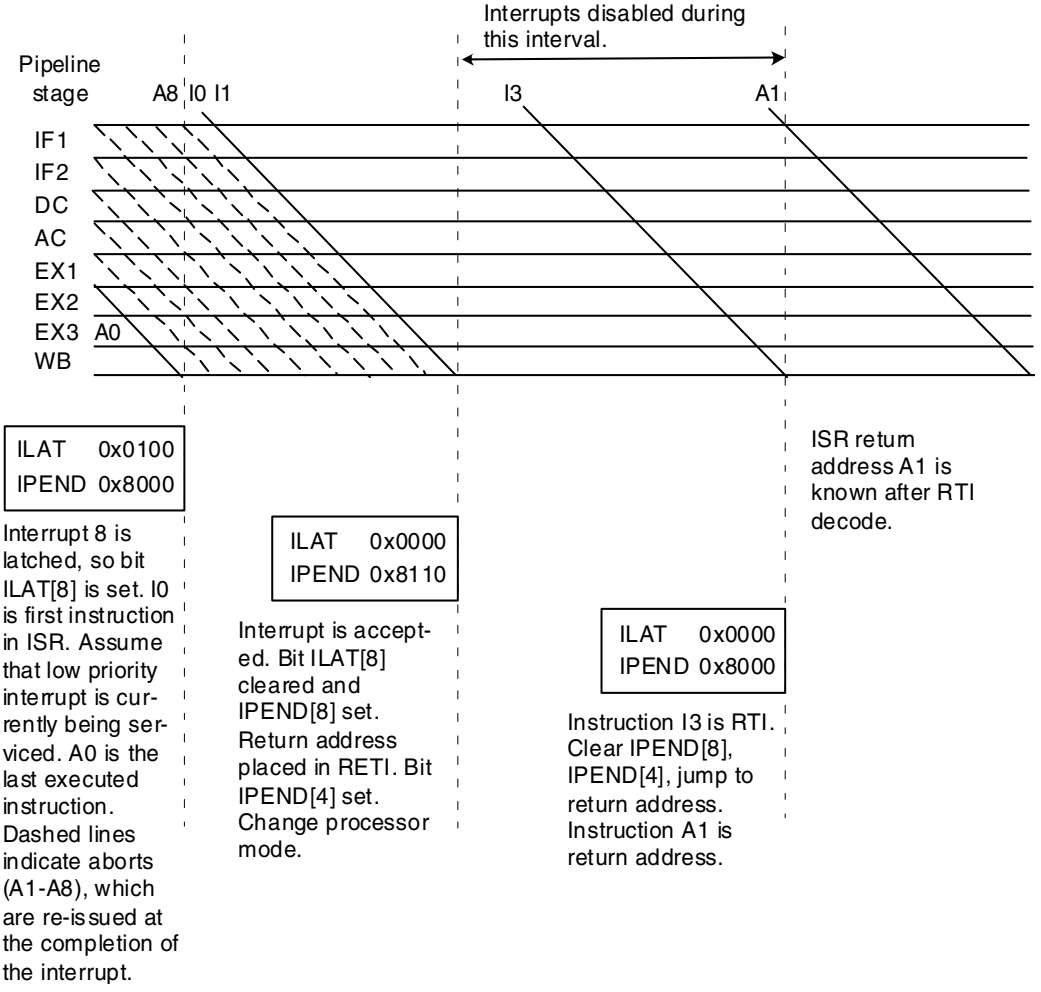


Figure 4-15. Non-Nested Interrupt Handling

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

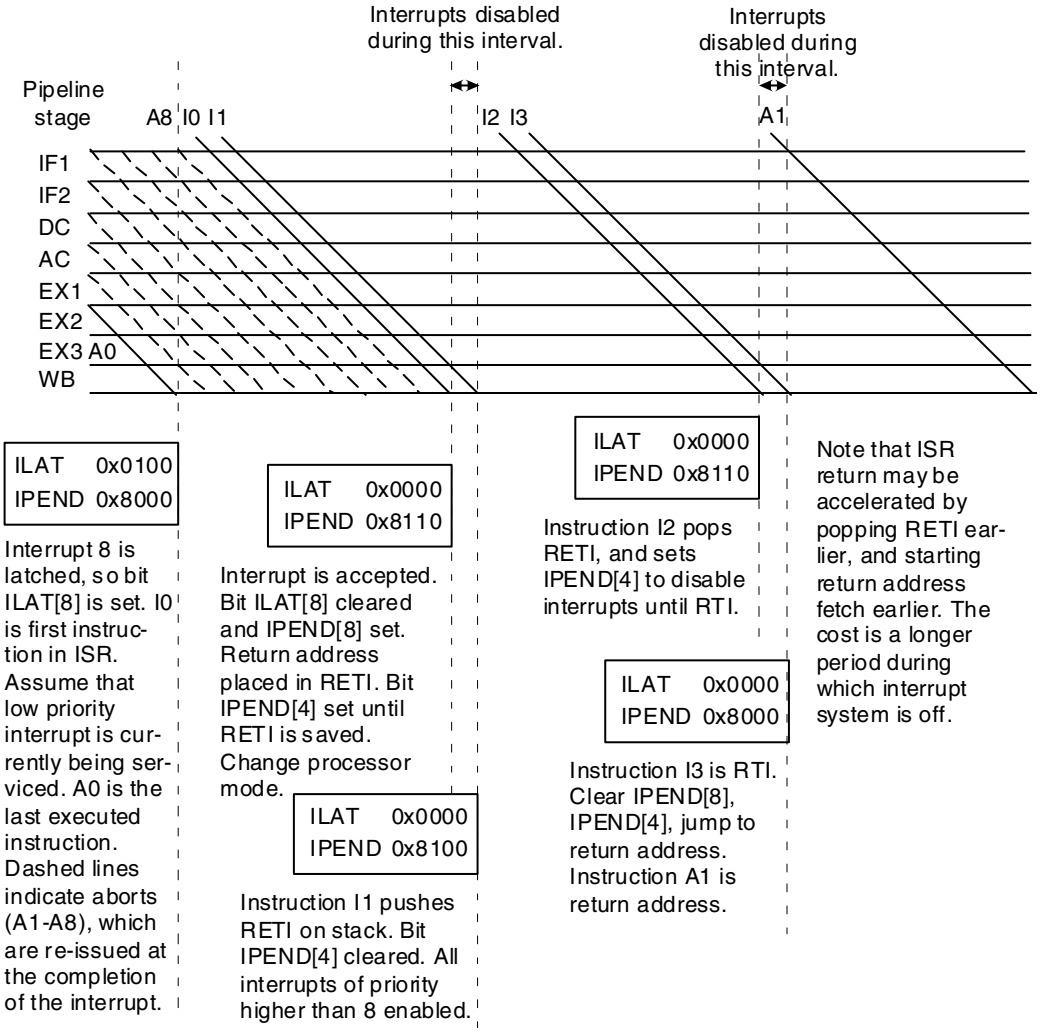


Figure 4-16. Nested Interrupt Handling

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 that none of the processor
resources (accumulators, DAGs, loop counters and bounds) have
been saved. It's assumed that this interrupt service routine does
not use them. */
```

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 the interrupts by clearing IPEND[4].

Logging of Nested Interrupt Requests

The SIC detects level-sensitive interrupt requests from the peripherals. The 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.

Because multiple interrupt sources can map to a single core processor general-purpose interrupt, 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.

Self-Nesting Mode

The nesting method described in the previous section allows an interrupt of higher priority to preempt interrupts of lower priority. However, it does not allow one interrupt to preempt another interrupt of the same priority, nor does it allow nesting at the same priority level.

When self-nested interrupts are not enabled, general-purpose interrupts can only preempt general-purpose interrupts of a lower priority (higher index). For example, when processing interrupt 7, another interrupt 7 request remains pending (`ILAT[7] = 1`), but the interrupt is not serviced until the current interrupt 7 service routine executes an RTI.

On the other hand, when self-nesting interrupts are enabled, an event interrupts processing at the same interrupt service level, provided `RETI` is pushed to the stack and interrupts are enabled. For example, when processing interrupt 7 (`IPEND[7:0] = 0x80`), another interrupt 7 request causes software to again vector to the interrupt 7 service routine.

Self-nesting of interrupts applies only to core interrupts or interrupts generated using the `RAISE` instruction; it is not allowed with peripheral interrupts.

For the interrupt system to allow self-nesting, software must set bit `SNEN` (self-nesting enable) in `SYSCFG`. See Figure 4-3 on page 4-6. In self-nesting mode, the system sets the LSB of the address in the return register `RETI` (`RETI[0]`) to indicate an incoming interrupt has the same priority as an interrupt that is currently being serviced. The bit `RETI[0]` is automatically set on entry into an interrupt service routine, and it is simply used as a status bit to flag the processor that the current ISR is self-nesting.

The return-from-interrupt instruction, RTI, is sensitive to the state of RETI[0] and SNEN. When both RETI[0] and SNEN are set, RTI clears only the global disable bit IPEND[4]. However, when self-nesting mode is disabled (SNEN = 0), RTI clears both IPEND[4] and the IPEND bit that corresponds to the current interrupt level.

The SNEN bit should be set once in the reset service routine and not changed again during normal interrupt processing.

Since the LSB of the RETI register is used to store the self-nesting state, avoid changing the contents of the RETI register when self-nesting is enabled, except for saving and restoring the register to the stack.

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 service 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 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

Listing 4-4. Exception Routine Handler With Deferred Processing

```
/* Determine exception cause by examining EXCAUSE field in SEQ-
STAT (first save contents of R0, P0, P1 and ASTAT in Supervisor
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 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, parame-
ter-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

Listing 4-5 provides an example framework for an exception routine that would be jumped to from an exception handler such as that described above.

Listing 4-5. Exception Routine

```
[--SP] = RETX ;   /* Push return address on stack. No change to
ILAT or IPEND. */

/* Put body of exception routine here. */

RETX = [SP++] ;   /* To return, pop return address and jump. No
change to ILAT or IPEND. */

RTX ;   /* Return from exception. Clear IPEND[3] (Exception Pend-
ing bit). */
```

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 produces the following results.

- If a program mistakenly uses RTX, RTN, or RTE to return from an interrupt, the core branches to the address in the corresponding return register (RETX, RETN, RETE) and leaves IPEND unaffected.
- If a program mistakenly uses RTI or RTX to return from an NMI routine, the core branches to the address in the corresponding return register (RETI, RETX), and clears the bit in IPEND that corresponds to the return instruction.
- 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.

Recommendation for 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 DSP 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 (ISR) may be held off for around 30 instruction clock cycles. When cache line fill operations are taken into account, the ISR could be held off for many hundreds of cycles.

In order for high-priority interrupts to be serviced with the least latency possible, the ADSP-BF535 processor allows any high latency fill operation to be completed at the system level, while an ISR executes from L1 memory. Figure 4-17 illustrates this concept.

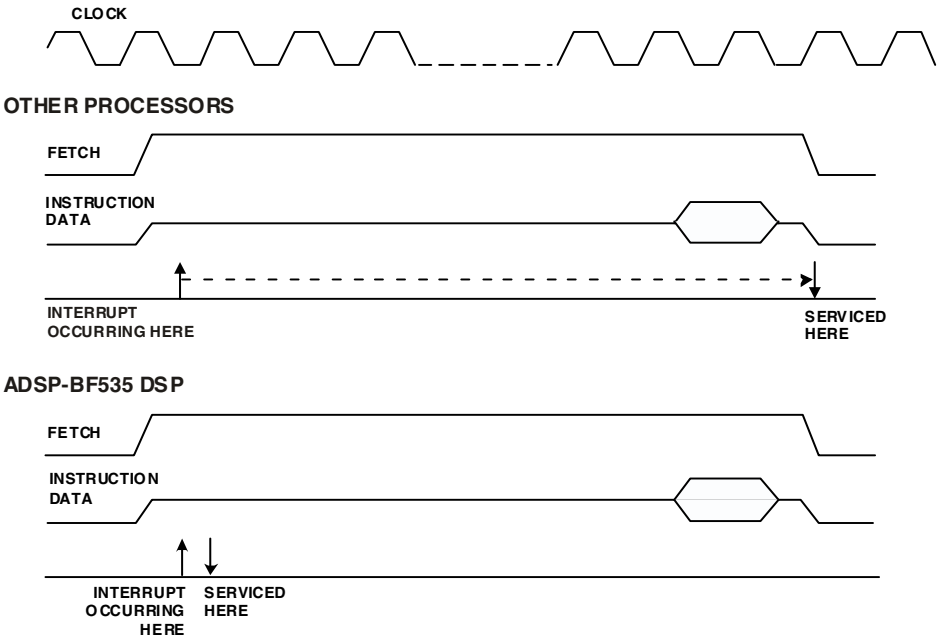


Figure 4-17. Minimizing Latency in Servicing an ISR

If an ADSP-BF535 processor instruction load operation misses the L1 Instruction Cache and generates a high latency line fill operation on the SBIU, then when an interrupt occurs, it is not held off until the fill has completed. Instead, the processor executes the ISR in its new context, and the cache fill operation completes in the background.

Note the ISR 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 SBIU is already busy completing the original cache line fill operation. If a

load or store operation is executed in the ISR requiring the SBIU, then the ISR is held off while the original external access is completed, before initiating the new load or store.

If the ISR finishes execution before the load operation has completed, then the ADSP-BF535 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 "Memory" on page 6-1.

5 DATA ADDRESS GENERATORS

The Data Address Generators (DAGs) of the ADSP-BF535 processor 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, which appears in Figure 5-1, supports several functions that minimize overhead in data access routines. These functions include:

- **Supply address and post-modify**—provides an address during a data move and auto-increments/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, eight 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].** 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].** 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`
    - Then modify the contents of `I0` by the value contained in the `M1` register.
- **Base and Length registers, B[3:0] and L[3:0].** 32-bit Base and Length registers set up the range of addresses and the starting address of a circular buffer. For more information on circular buffers, see "Addressing Circular Buffers" on page 5-6.
- **Pointer registers, P[5:0], FP, and SP.** 32-bit Pointer registers hold an address pointer to memory. The `P[5:0]` field, `FP` (Frame Pointer), and `SP` (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.

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.

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

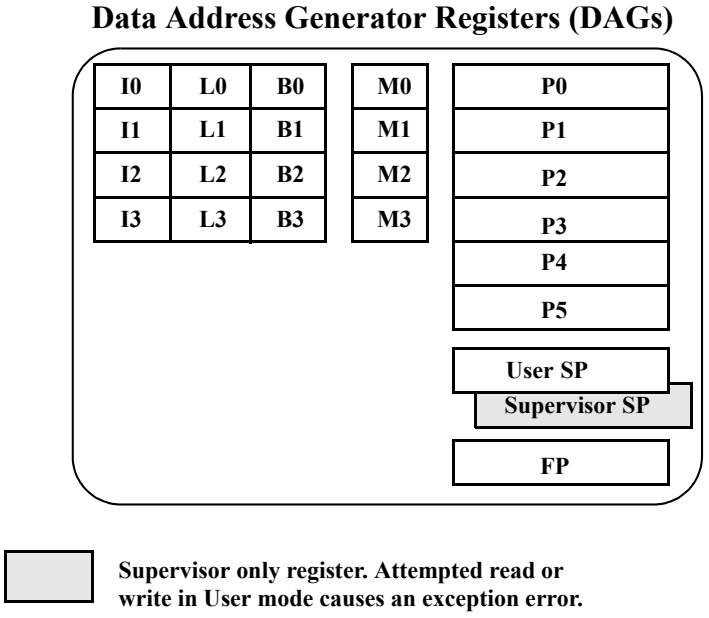


Figure 5-1. ADSP-BF535 Processor DAG Registers

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 ADSP-BF535 processor addressing is always byte aligned. 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-bit, 16-bit, or 32-bit accesses.

For example, consider this 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++ ] (X) ;
```

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 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 instruction summary details, see Table 5-3 on page 5-17.

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` exclusively, for example:

- `FP`-indexed load/store, which extends the addressing range for 16-bit encoded load/stores
- Stack push/pop instructions

Addressing Circular Buffers

The DAGs support addressing circular buffers—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 $2^{31} - 1$. 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. Circular buffering uses post-modify addressing.

- 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.

- 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)

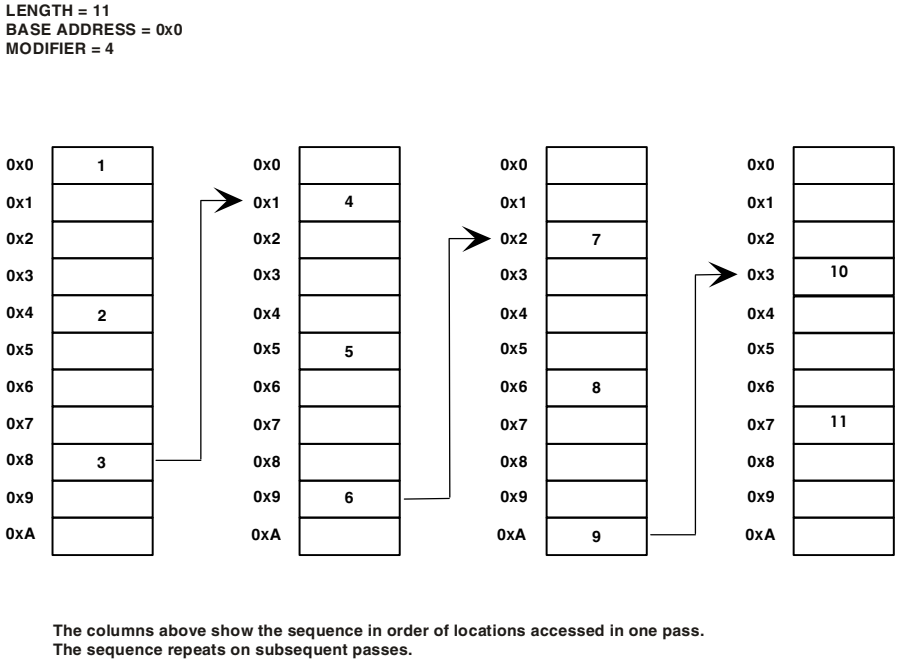


Figure 5-2. Circular Data Buffers

- 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 *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-bit 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-bit 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 ADSP-BF535 processor uses the Stack Pointer register, `SP`. The address in `SP` is decremented by four 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.

For example:

```
P5 = [ P1 + 0x10 ] ;
```

is an acceptable offset, but

```
P5 = [ P1 + 0x11 ] ;
```

causes an alignment exception.

Be sure the offset is divisible by the word-transfer size, measured in bytes; for example, for a 32-bit transfer, the offset should be a multiple of 4; for a 16-bit transfer, the offset should be a multiple of 2. Correct offsets ensure memory alignment.

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 another Pointer register. Index registers are modified by a Modify register. This instruction does not support the Pointer registers as a destination register, 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. It adds the value the pointer P4 by the value of the pointer P5.

For example:

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

loads a 32-bit word into the destination register R2. It updates the value in the Index register I2 by the value in the Modify register M1.

Modifying DAG and Pointer Registers

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

The instruction 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 instruction 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 ADSP-BF535 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 that the addressing modes support.

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

Be careful when using the `DISALGNEXPT` instruction, because it disables automatic detection of memory alignment errors.

Table 5-2 summarizes the addressing modes. In the table, an asterisk (*) indicates that the ADSP-BF535 processor supports the addressing mode.

Table 5-2. Addressing Modes

	P Auto-inc	P Auto-dec	P Indirect	P Indexed	FP-Indexed	P Post-inc	I Auto-inc	I Auto-dec	I Indirect	I Post-inc
	[P0++]	[P0--]	[P0]	[P0+im]	[FP+im]	[P0++P1]	[I0++]	[I0--]	[I0]	[I0++M0]
32-bit Word	*	*	*	*	*	*	*	*	*	*
16-bit Half Word	*	*	*	*		*	*	*	*	
8-bit Byte	*	*	*	*						
Sign/ Zero Extend	*	*	*	*		*				
Data Register	*	*	*	*	*	*	*	*	*	*
Pointer Register	*	*	*	*	*					
Data Register Half			*			*	*	*	*	

DAG Instruction Summary

Table 5-3 lists the DAG instructions. For more information on assembly language syntax, see *Blackfin Processor Programming Reference*. In the table, note the meaning of these symbols:

- **Dreg** denotes any Data Register File register.
- **Dreg_lo** denotes the lower 16 bits of any Data Register File register.
- **Dreg_hi** denotes the upper 16 bits of any Data Register File register.
- **Preg** denotes any Pointer register, `FP` or `SP` register.
- **Ireg** denotes any 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.

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) ;
Dreg =W [Preg + uimm16m2] (Z) ;

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
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) ;
Dreg = B [Preg – uimm15] (Z) ;
Dreg = B [Preg] (X) ;
Dreg = B [Preg ++] (X) ;

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
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 ;
[Ireg ++ Mreg] = Dreg ;
W [Ireg] = Dreg_hi ;
W [Ireg ++] = Dreg_hi ;
W [Ireg --] = Dreg_hi ;
W [Preg] = Dreg_hi ;

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
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) ;
Preg += Preg (BREV) ;
Ireg += Mreg (BREV) ;
Preg –= Preg ;
Ireg –= Mreg ;

6 MEMORY

The ADSP-21535 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 faster memories located closer to the processor core, whereas Level 2 (L2) memories are memory systems farther from the core, typically with longer access latencies. The faster L1 memories, which include an instruction, data, and scratchpad memory as part of the Blackfin core, are accessed in a single cycle. The L2 memories, which include an on-chip SRAM and off-chip mapping to synchronous, asynchronous and PCI devices, provide much larger memory spaces with higher latencies.

The focus of this chapter is on-chip L1 and L2 memory. The off-chip L2 memory is explained in more detail in "External Bus Interface Unit" on page 18-1. The PCI memory space, which is also part of the off-chip L2 memory, is further described in "PCI Bus Interface" on page 13-1.

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.

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.

little endian.

The native data store format of the ADSP-21535 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 ADSP-BF535 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 6-6 on page 6-18).

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 6-6 on page 6-18).

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 the source memory. The modified cache line is *not* written to the source memory when it is replaced.

Memory Architecture

The ADSP-BF535 processor has a unified 4GB address range that spans a combination of on-chip and off-chip memory and memory-mapped I/O resources. Of this range, 285 MB of address space are dedicated to internal, on-chip resources. The ADSP-BF535 processor populates portions of this internal memory space with:

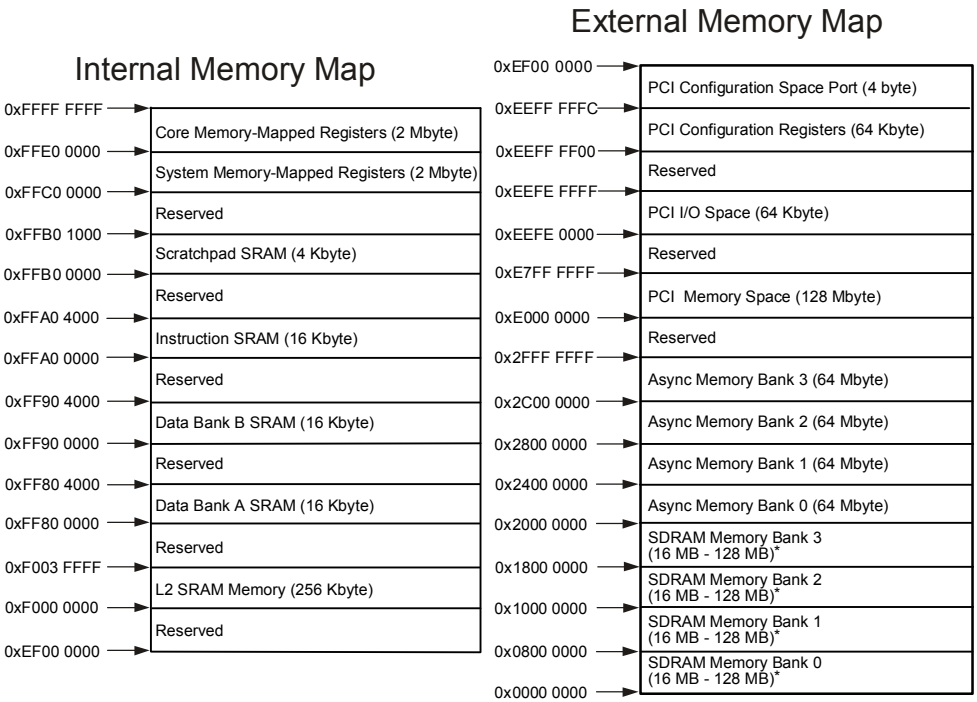
- A set of L1 and L2 Static Random Access Memories (SRAM)
- A set of memory-mapped registers (MMRs) and
- A boot Read Only Memory (ROM)

A portion of the internal L1 SRAM can also be configured to run as cache. The ADSP-BF535 processor also provides support for an external memory space that includes PCI space, asynchronous memory space, and synchronous DRAM (SDRAM) space. See "PCI Bus Interface" on page 13-1 and "External Bus Interface Unit" on page 18-1, for a detailed discussion of each of these memory regions and the controllers that support them.

The diagram in Figure 6-1 provides an overview of the ADSP-BF535 processor system memory map. Note that 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 Figure 6-1, L1 instruction and data memories occupy 52 KB of internal memory space:

- 16 KB of instruction SRAM/cache
- 32 KB of data SRAM/cache (two 16 KB banks)
- 4 KB of data scratchpad SRAM



* The addresses shown for the SDRAM banks reflect a fully populated SDRAM array with 512 Mbytes of memory. If any bank contains less than 128 Mbytes of memory, it would only extend to the length of the real memory systems and the end address would become the start address of the next bank. This would continue for all four banks, with any remaining space between the end of Memory Bank 3 and the beginning of Async Memory Bank 0 at address 0x2000 0000 treated as reserved address space.

Figure 6-1. ADSP-BF535 Processor Memory Map

An on-chip SRAM provides 256 KB of L2 memory space. For systems using some or all ADSP-BF535 processor L1 memory as cache, the on-chip L2 SRAM memory system can help provide deterministic, bounded memory access times.

The upper portion of internal memory space is allocated to the core and system MMRs of the ADSP-BF535 processor. Accesses to this area are allowed only when the processor is in Supervisor mode or Emulation mode (see “Operating Modes and States” on page 3-1).

The lowest 1 KB of internal memory space is occupied by the boot ROM of the ADSP-BF535 processor. Depending on the booting option selected, the appropriate boot program is executed from this memory space when the ADSP-BF535 processor is reset (see "Booting Methods" on page 3-17).

Within the external memory map, the PCI address range consists of PCI memory, PCI I/O space and PCI configuration space. In addition, four banks are available for SDRAM. Each bank can vary in size from 16 MB to 128 MB. An additional four banks of asynchronous memory space are also available. Each of the asynchronous banks is 64 MB.

Figure 6-2 shows the overall memory architecture for the ADSP-BF535 processor.

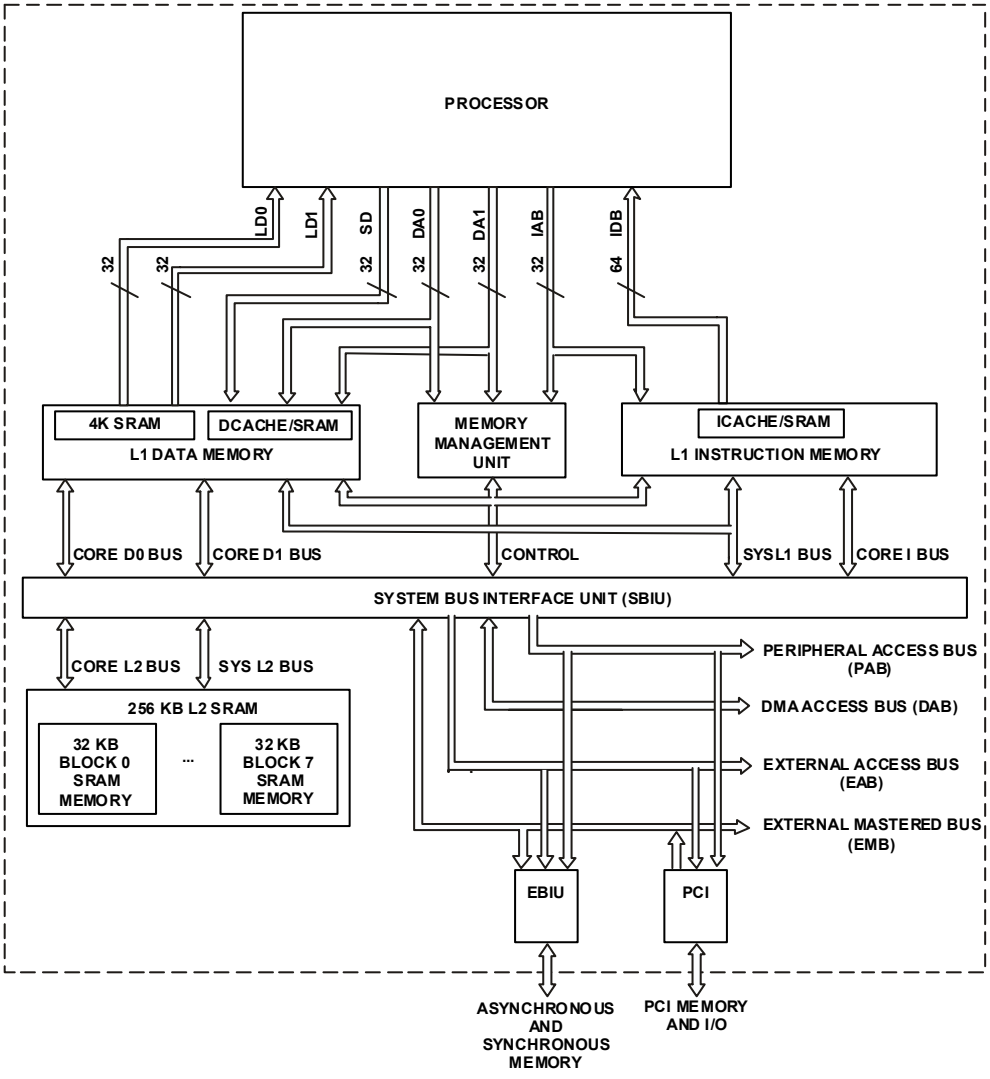


Figure 6-2. ADSP-BF535 Memory Architecture

Internal Memory

The ADSP-BF535 processor 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 chip. The ADSP-BF535 processor supports this memory architecture for applications that require direct control over access time.

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 the L1 memories. Code can be ported to or developed for the ADSP-BF535 processor quickly without requiring performance optimization for the memory organization.

The ADSP-BF535 processor L1 memory provides:

- A modified Harvard architecture, allowing up to three core memory accesses per clock cycle (one 64-bit instruction fetch, and two 32-bit data references)
- Simultaneous system DMA and core accesses
- Fast SRAM access for critical processor 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

Overview of L1 Instruction SRAM

The 16 KB L1 instruction SRAM consists of four 4 KB sub-banks. When configured as cache, the L1 Instruction Memory is 4-Way set associative. Consequently, instructions can be brought into four different sub-banks of cache, decreasing the frequency of cache line replacements and increasing overall performance. Although the entire 16 KB L1 memory must be configured as an SRAM or a cache, individual sub-banks of L1 instruction cache can be locked down, allowing further control over the location of time critical code. The cache locking concept is explained further in "Instruction Cache Locking" on page 6-24. For more information about L1 instruction SRAM, see "L1 Instruction SRAM" on page 6-14.

Overview of L1 Data SRAM

The ADSP-BF535 processor provides two 16 KB L1 data SRAM banks (Data Bank A and Data Bank B). Each 16 KB L1 data bank consists of four 4 KB sub-banks. This organization—like the L1 Instruction memory—provides an effective dual port capability that gives the system DMA controller and the core simultaneous access to the SRAMs, provided that collisions to the same sub-bank do not occur.

Each 16 KB L1 data bank can be configured to operate either as a cache or an SRAM. Consequently, the ADSP-BF535 processor can be configured to run in these configurations:

- 32 KB L1 data SRAM
- 32 KB L1 data cache
- 16 KB L1 data SRAM and 16 KB L1 data cache

Each L1 data bank is a 2-Way set associative structure. Even when one 16KB L1 Data Memory bank is configured as SRAM, the other still functions as a 2-Way cache rather than as a 16 KB direct mapped cache. This also provides two separate locations that can hold cached data, decreasing the rate of cache line replacements and increasing overall performance.

When both banks are configured as cache, they operate as two independent, 16 KB, 2-Way set associative caches that can be independently mapped into the Blackfin address space. For more information about L1 data SRAM, see "L1 Data SRAM" on page 6-38.

Overview of Scratchpad Data SRAM

The ADSP-BF535 processor provides a dedicated 4 KB bank of scratchpad data SRAM. The scratchpad is independent of the configuration of the other L1 memory banks and cannot be configured as cache. 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`).

Overview of On-Chip L2 Memory

The on-chip Level 2 (L2) memory provides 256 KB of low latency, high bandwidth capacity. This memory system is referred to as *on-chip L2*, because it forms an on-chip memory hierarchy with L1 memory. On-chip L2 memory provides much more capacity than L1 memory, but the latency is higher. The on-chip L2 memory of the ADSP-BF535 processor is SRAM and cannot be configured as cache.

For more information about on-chip L2 Memory, see "On-Chip Level 2 (L2) Memory" on page 6-52.

On-chip and off-chip L2 memories are capable of storing both instructions and data.

Level 1 Memory

Figure 6-3 and Figure 6-4 show the Data Memory Control register (DMEM_CONTROL) and Instruction Memory Control register (IMEM_CONTROL), respectively. The bits in these registers can be used to configure L1 Data Memory and L1 Instruction Memory.

The sections after the figures describe the instruction and data memories in more detail—including SRAM and cache configurations for each type of L1 memory.

L1 Instruction Memory can be used only to store instructions, and L1 Data Memory can be used only to store data.

Data Memory Control Register (DMEM_CONTROL)

The Data Memory Control register (DMEM_CONTROL), shown in Figure 6-3, contains control bits for the L1 Data Memory. The reset values of the ENDM and DMC bits indicate that the L1 Instruction Memory is enabled and configured as SRAM after reset. The ENDCPLB bit is used to enable/disable the sixteen Cacheability Protection Lookaside Buffers (CPLBs) used for data (see "L1 Data Cache" on page 6-40).

Instruction Memory Control Register (IMEM_CONTROL)

The Instruction Memory Control register (IMEM_CONTROL), shown in Figure 6-4, contains control bits for the L1 Instruction Memory. The reset values of the ENIM and IMC bits indicate that the L1 Instruction Memory is enabled and configured as SRAM after reset. The ENICPLB bit is used to enable/disable the sixteen CPLBs used for instructions (see "L1 Instruction Cache" on page 6-17).

Data Memory Control Register (DMEM_CONTROL)

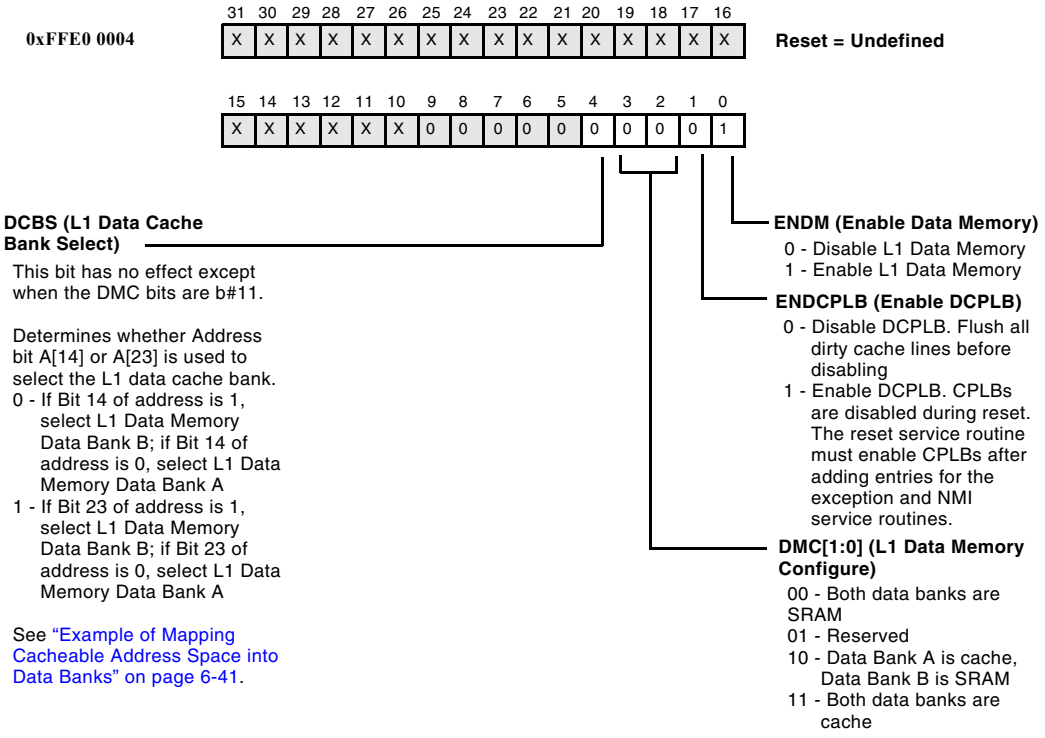


Figure 6-3. L1 Data Memory Control Register

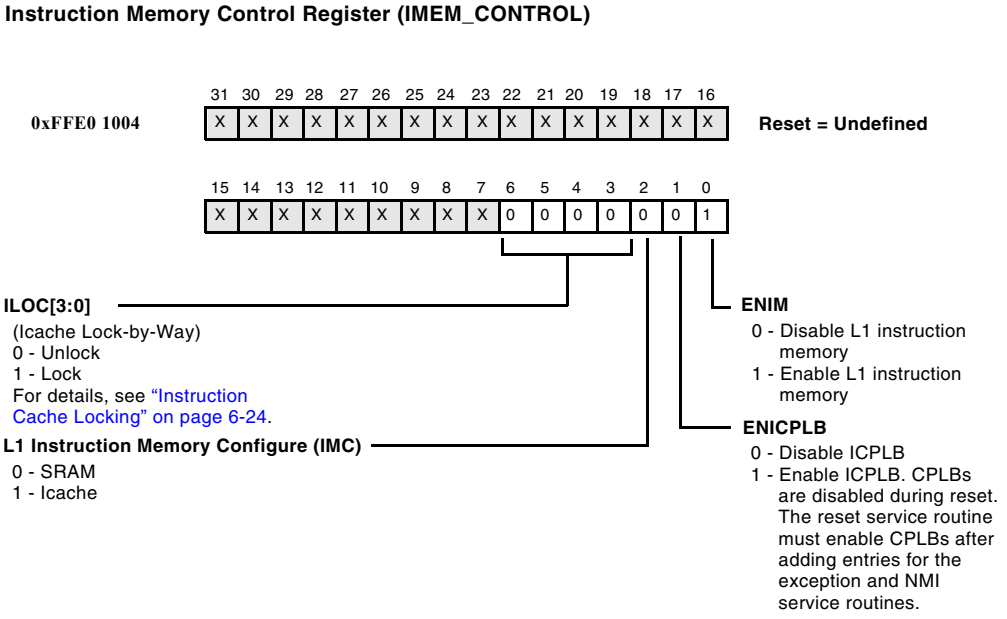


Figure 6-4. L1 Instruction Memory Control Register

L1 Instruction Memory

Control bits in the IMEM_CONTROL register can be used to organize all four sub-banks 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 SRAM

L1 Instruction Memory can be configured as a 16 KB SRAM.

The ADSP-BF535 core reads the 16 KB 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. If instruction space must be accessed as data, L2 memory must be used, because it serves as a unified space for both instructions and data. A DAG reference to instruction memory SRAM space generates an exception (see "Exceptions" on page 4-38).

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 four single ported sub-banks, the instruction memory is effectively dual ported. Provided that system and core accesses do not collide on the same sub-bank, effective dual porting of the instruction memory is achieved. If both system and core attempt to access the same bank, the system DMA controller has priority over the core instruction fetch.

Alternatively, both the `IMC` and `ENIM` bits in the `IMEM_CONTROL` register can disable these sub-banks entirely. The configuration state in the control register can be modified only while in Supervisor mode or Emulation mode.

Before changing the configuration state, be sure to flush the cache or move all modified data from the SRAM, if so configured.

Table 6-1 lists the memory start locations of the L1 Instruction Memory sub-banks.

Table 6-1. L1 Instruction Memory Sub-Banks

Memory Sub-Bank	Memory Start Location
0	0xFFA0 0000
1	0xFFA0 1000
2	0xFFA0 2000
3	0xFFA0 3000

Figure 6-5 describes the bank architecture of the L1 Instruction Memory.

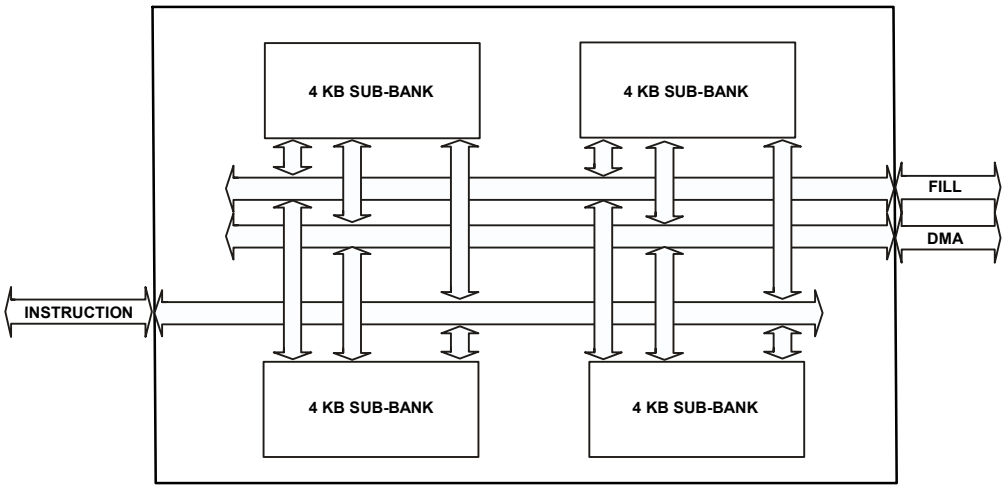


Figure 6-5. L1 Instruction Memory Bank Architecture

L1 Instruction Cache

The L1 Instruction Memory may also be configured as a flexible, 4-Way set associative instruction cache. To improve the average access latency for critical code sections, each Way of the cache can be locked independently. When the memory is configured as cache, it cannot be accessed directly.

When cache is enabled, memory pages must be defined with Cacheability Protection Lookaside Buffers (CPLBs). When CPLBs are enabled, any memory location that is accessed must have an associated page definition available, or an exception will be generated. CPLBs are described in "Memory Protection and Properties" on page 6-56.

Figure 6-6 shows the overall Blackfin processor instruction cache organization.

Cache Lines

As shown in Figure 6-6, 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, and a Valid bit.
- The data component is made up of four 64-bit words of instruction data.

The tag and data components of cache lines are stored in the tag and data memory arrays, respectively.

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 4 KB memory sub-bank 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.

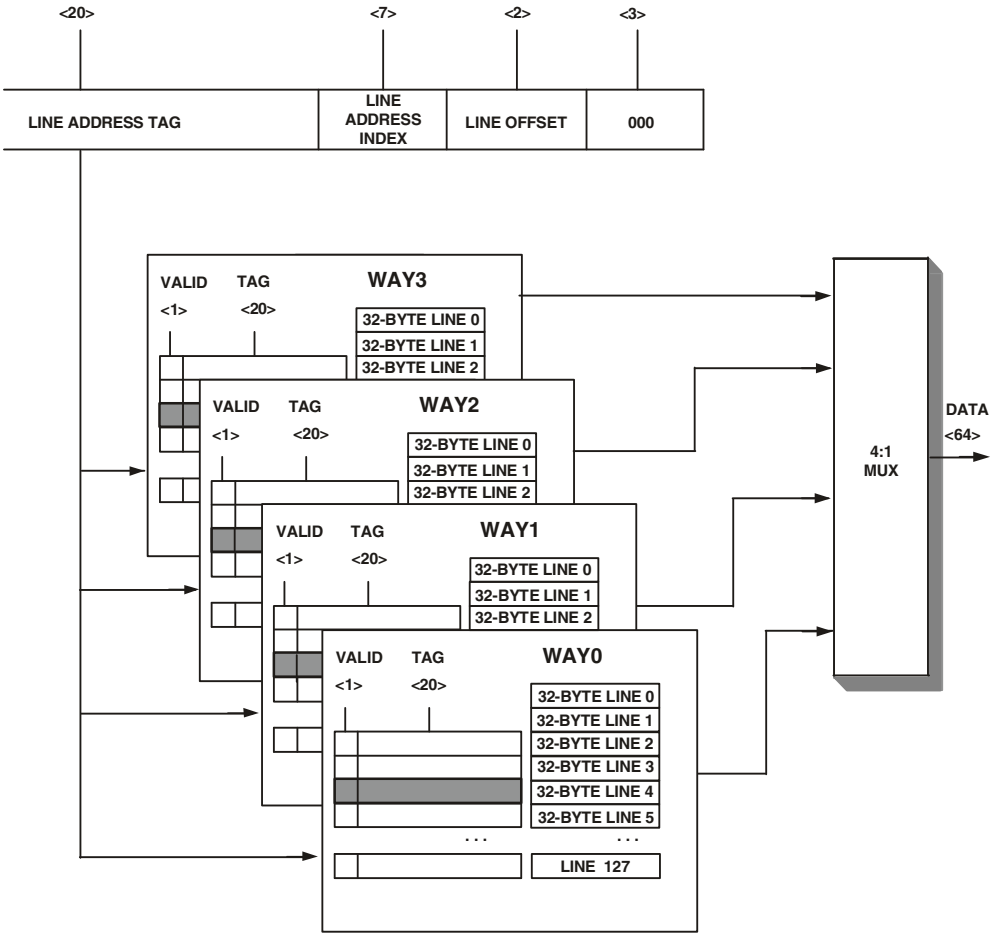


Figure 6-6. Blackfin Instruction Cache Organization

The Valid bit indicates the state of a cache line. A cache line is always valid or invalid:

- Invalid cache lines have their Valid bit cleared, indicating the line will be ignored during an address tag compare operation.
- Valid cache lines have their Valid bit set, indicating the line contains valid instruction/data that is consistent with the source memory.

The tag and data components of a cache line are illustrated in Figure 6-7.

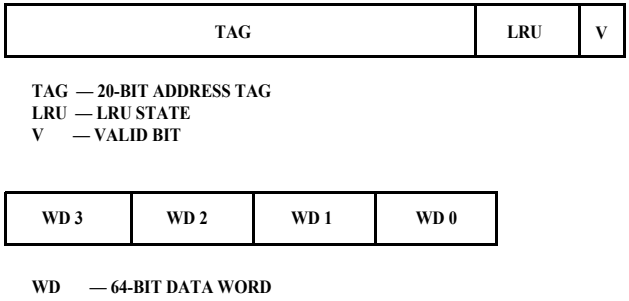


Figure 6-7. Cache Line – Tag and Data Portions

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.

Note that only valid cache lines (cache lines with their Valid bit set) are included in the address tag compare operation. For cache hits to occur:

- The instruction memory unit must be enabled and configured as cache by setting bit 0 and bit 2 of the `IMEM_CONTROL` register.
- The CPLB must be enabled, and the page corresponding to the instruction fetch address must be mapped as cacheable.

When a cache hit occurs, the target 64-bit instruction word is first sent to the Instruction Alignment Unit where it is stored in one of two 64-bit instruction buffers.

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 L2 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 will wrap around if necessary, as shown in Table 6-2.

Table 6-2. 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 L2 memory, each 64-bit word is buffered in a four-entry line fill buffer before it is written to a 4 KB 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.

The line fill buffer organization is shown in Figure 6-8.

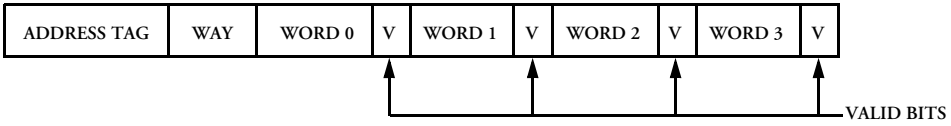


Figure 6-8. Line Fill Buffer Organization

During a line fill operation, the line fill buffer's address tag contains the upper 18 bits plus bits 11 and 10 of the instruction fetch address. The contents of the Way field identify the cache Way selected for replacement (see Table 6-3). Both fields are loaded at the end of a tag-address compare operation that results in a cache miss.

Table 6-3. Encoded Cache Ways

Line-Buffer Way [1:0]	Cache Way
00	Way 0
01	Way 1
10	Way 2
11	Way 3

Non-Cacheable Accesses

The line fill buffer is also used to support non-cacheable accesses. A non-cacheable access consists of a single 64-bit word data transfer on the instruction memory unit's external read port. Non-cacheable accesses include:

- External memory accesses performed while the instruction memory unit is disabled
- External memory accesses performed while the instruction memory unit is enabled and configured as SRAM
- Accesses to non-cacheable pages

A page is considered non-cacheable if the `CPLB_L1_CHBL` bit of the associated CPLB descriptor for the matching address is cleared.

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 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 the cache organization in Figure 6-6 on page 6-18).

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 this 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.

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.

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 (See the Instruction Test Command Register in Figure 6-9 on page 6-27 and Data Test Command Register in Figure 6-15 on page 6-49.)

Instruction Cache Locking

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, each sub-bank of L1 Instruction Memory is a Way. Setting the lock bit for a specific Way prevents state transitions for all the lines in that Way; that is, lines cannot change state from valid to invalid, or *vice versa*. Thus, setting the lock bit for a Way effectively prevents the Way from participating in the replacement policy.

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 Invalidation" on page 6-25).
- Disable interrupts, if needed, 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]`), are 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 `IFLUSH` instruction can explicitly invalidate cache lines based on their tag addresses. The target address of the instruction is generated from the P-registers. Because the instruction cache never contains 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 ADSP-BF535 processor memory map, it is impractical to use this instruction to invalidate an entire bank of cache. A second, faster technique can be used to invalidate an entire cache bank directly. This second technique directly invalidates Valid bits, which are in a random state when the ADSP-BF535 processor comes out of reset; it sets the Valid bit of each cache line to the invalid state. To implement this technique, additional MMRs are available to allow arbitrary read/write of all cache entries directly.

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 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 the access is to read from. 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. Graphic representations of the ITEST registers begin with Figure 6-9 on page 6-27.

Before the cache entries are accessed through the ITEST registers, L1 Instruction Memory should be enabled by setting the ENIM and IMS bits and clearing the ENICPLB bit in the Instruction Memory Control register (IMEM_CONTROL).

These figures describe the ITEST registers:

- Instruction Test Command Register in Figure 6-9 on page 6-27
- Instruction Test Data 1 Register in Figure 6-10 on page 6-28
- Instruction Test Data 0 Register in Figure 6-11 on page 6-29

Access to these registers is possible only in Supervisor mode 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, write the ITEST_COMMAND register first, then read the ITEST_DATA registers.

Instruction Test Command Register (ITEST_COMMAND)

When the Instruction Test Command register (ITEST_COMMAND)s, shown in Figure 6-9, 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]).

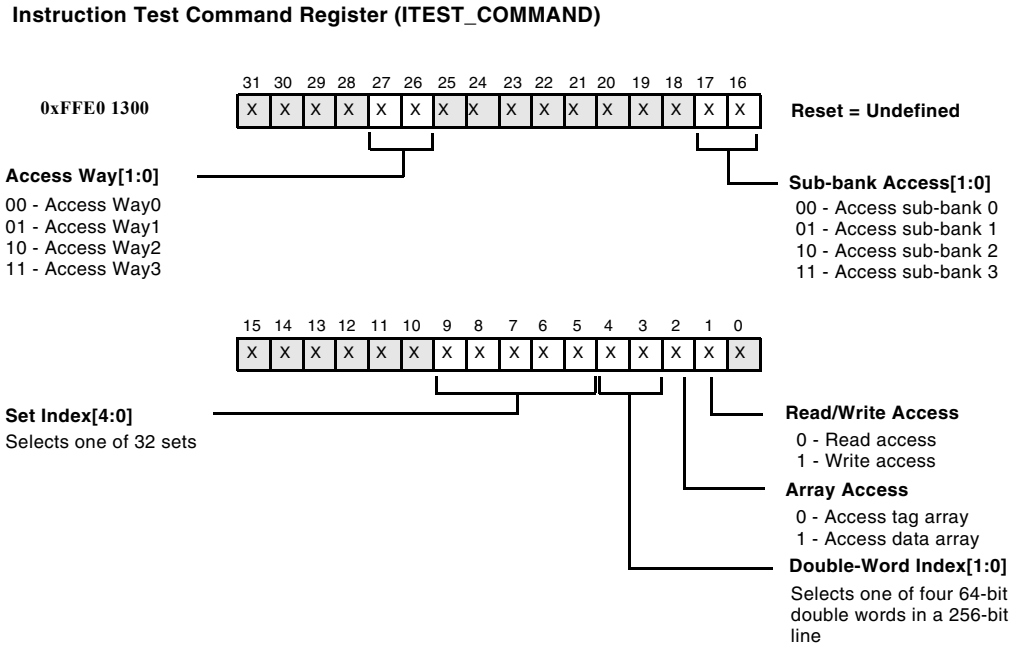


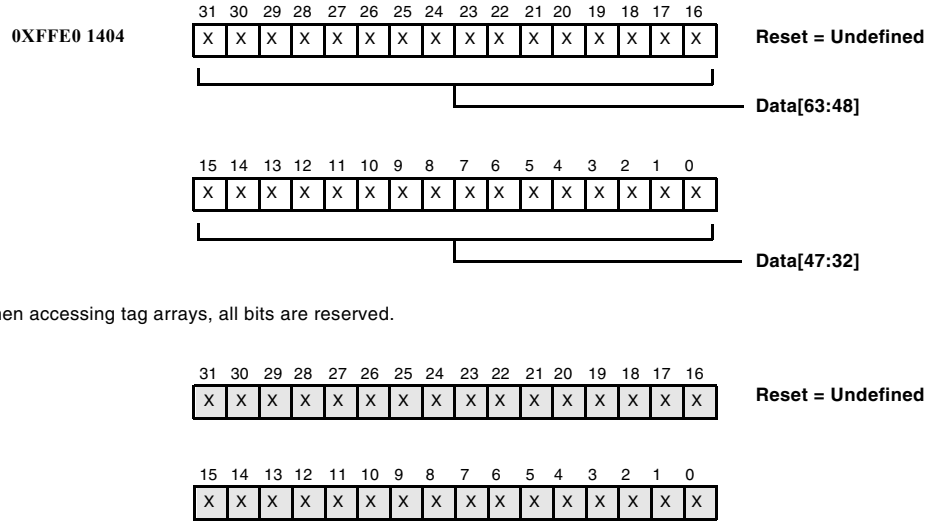
Figure 6-9. Instruction Test Command Register

Instruction Test Data 1 Register (ITEST_DATA1)

Instruction Test Data registers (ITEST_DATA[1:0]), shown in Figure 6-10, 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.

Instruction Test Data 1 Register (ITEST_DATA1)

Used to access L1 cache data arrays and tag arrays. When accessing a data array, stores the upper 32 bits of 64-bit words of instruction data to be written to or read from by the access. See "Cache Lines" on page 6-17.



When accessing tag arrays, all bits are reserved.

Figure 6-10. Instruction Test Data 1 Register

Instruction Test Data 0 Register (ITEST_DATA0)

The Instruction Test Data 0 register (ITEST_DATA0), shown in Figure 6-11, 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 the tag arrays and contains the Valid and Dirty bits, which indicate the state of the cache line.

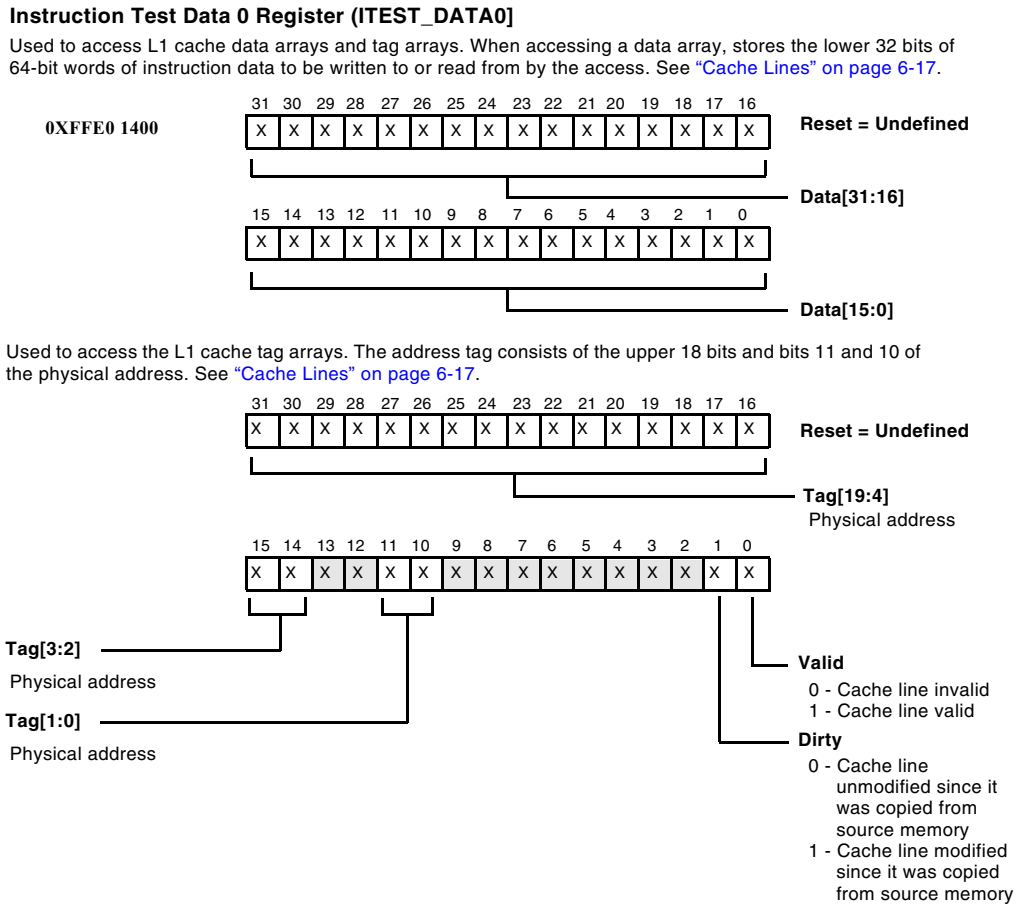


Figure 6-11. Instruction Test Data 0 Register

Example Code for Direct Invalidation

The following four listings show how to invalidate instruction cache (Listing 6-1), how to invalidate data cache (Listing 6-2), how to invalidate Superbank A (Listing 6-3), and how to invalidate Superbank B (Listing 6-4).

Listing 6-1. Invalidating Instruction Cache Lines Directly

```
/* Setting the icache to cache for preloading */
   P0.L = (IMEM_CONTROL & 0xFFFF);
   P0.H = (IMEM_CONTROL >> 16);
   R0.L = 0X5;   /* Enable instruction memory as cache. */
   R0.H = 0X0;
   [P0] = R0;
   P1.L = (DMEM_CONTROL & 0xFFFF);
   P1.H = (DMEM_CONTROL >> 16);
   R0.L = 0XD;   /* Enable data bank A and B as caches. */
   R0.H = 0X0;
   [P1] = R0;
   csync;
/* Preloading the Instruction Cache */
/* Registers used: */
/* r0 - contains the command value. written to ITEST COMMAND */
/* r1 - value written to ITEST DATA0 and ITEST DATA1 */
/*    * DATA write */
/*    - contains the instruction to be written */
/* r2 - contains intermediate values during computation */
/* r3 - contains mask for the double word index (bit[3:4]=00) */
/* r4 - contains mask for the set index (bit[5:9]=00000) */
/* r5 - contains mask for the ways (bit[26:27]=00) */
/* r6 - value written to ITEST DATA0 */
/*    * TAG write */
/*    - contains the original address */
```

```
/*    - used to determine sub-banks */
/* i0 - pointer to ITEST_COMMAND */
/* i1 - pointer to ITEST_DATA0 */
/* i2 - pointer to ITEST_DATA1 */
/* i3 - pointer to relocated (instruction) data */
/* p4 - loop counter to perform the writes to DATA and TAGs */
/* m0 - save command for sub-bank 0 */
/* m1 - save command for sub-bank 1 */
/* m2 - save command for sub-bank 2 */
/* m3 - save command for sub-bank 3 */
/* */

   R7.L = 0;
   R7.H = 0;
      L0 = R7;
      L1 = R7;
      L2 = R7;
      L3 = R7;
      M0 = R7;
      M1 = R7;
      M2 = R7;
      M3 = R7;

   I0.L = (ITEST_COMMAND & 0xFFFF);
   I0.H = (ITEST_COMMAND >> 16);
   I1.L = (ITEST_DATA0 & 0xFFFF);
   I1.H = (ITEST_DATA0 >> 16);

/* Writing zero to the ITEST_DATA0 */
/* Whenever we write to Cache TAG/ARRAY, the value in the
ITEST_DATA0 gets written to the caches. The ITEST_DATA0/1 should
be written to before the write to ITEST_COMMAND register */
```

```
R0.L = 0;
R0.H = 0;

[I1] = R0;
/* The cache lines come out of reset in a random state. The valid
bits of each line must be set to zero at initialization. */
/* Explanation: The invalidation routine is as follows: */
/*  Cache setup: */
/*  I-CACHE: */
/*    4 sub-banks, */
/*    each sub-bank has 4 ways, */
/*    each way has 32 lines, */
/*    each line (or set) has 4 double words. */
/*  Routine: */
/*  - Take way 0, and way 1 */
/*  - Inner loop: */
/*      Invalidate all sets (cache lines) in sub- bank for way 0
and way 1 */
/*      Instruction Cache has 32 sets (cache lines), hence loop
count is 32 */
/*  - Outer loop: */
/*      Increment sub-banks */
/*      repeat inner loop */
/*      do it 4 times because of 4 sub-banks. */
/*  - Repeat again for way 2, way 3 */

R2 = 32;   /* Need to increment the set index every loop (Should
be 64 for D-cache) */
R3.L = 0;   /* sub-bank increment, at the end of inner loop */
R3.H = 1;
P4 = 4;   /* Number of sub-bank - also outer loop counter (Should
be 2 for D-cache) */
P3 = R2;   /* Inner loop counter (Number of set index) */
```

```
R4 = 2;   /* Initial value for ITEST_COMMAND for way 0 - should be
WRITE to TAG */
R5.H = 0x400;   /* Initial value for ITEST_COMMAND for way 1 -
should be WRITE to TAG */
R5.L = 2;

/* Way 0,1 invalidation */
lsetup(LBL0A,LBL3A) lc1 = p4;
LBL0A: r0 = r4;
       r1 = r5;
       lsetup(LBL1A,LBL2A) lc0 = p3;
       LBL1A:   r0 = r0+|+r2 || [i0]=r0;
       LBL2A:   r1 = r1+|+r2 || [i0]=r1;
       r4 = r4+r3;
LBL3A: r5 = r5+r3;

R4.H = 0x800;   /* Initial value for ITEST_COMMAND for way 2 -
should be WRITE to TAG */
R4.L = 2;
R5.H = 0xc00;   /* Initial value for ITEST_COMMAND for way 3 -
should be WRITE to TAG */
R5.L = 2;

/* Way 2,3 invalidation */
lsetup(LBL0B,LBL3B) lc1 = p4;
LBL0B: r0 = r4;
       r1 = r5;
       lsetup(LBL1B,LBL2B) lc0 = p3;
       LBL1B:   r0 = r0+|+r2 || [i0]=r0;
       LBL2B:   r1 = r1+|+r2 || [i0]=r1;
       r4 = r4+r3;
LBL3B: r5 = r5+r3;
```

Listing 6-2. Invalidating Data Cache

```
   I0.L = (DTEST_COMMAND & 0xFFFF);
   I0.H = (DTEST_COMMAND >> 16);
   I1.L = (DTEST_DATA0 & 0xFFFF);
   I1.H = (DTEST_DATA0 >> 16);

R0.L = 0;
R0.H = 0;

[I1] = R0;

/* Repeat for data cache */
/* Explanation: The invalidation routine is as follows: */
/*  Cache construction: */
/*  D-CACHE: */
/*    2 data banks, */
/*    each data bank has 4 sub-banks, */
/*    each sub-bank has 2 ways, */
/*    each way has 64 lines, */
/*    each line (or set) has 4 double words. */
/*  Routine: */
/*  - Take way 0, and way1 */
/*  - Inner loop: */
/*      Invalidate all sets (cache lines) in sub-bank for way 0
and way1 */
/*      Instruction Cache has 64 sets (cache lines), hence loop
count is 64 */
/*  - Outer loop: */
/*      Increment sub-banks */
/*      repeat inner loop */
/*      do it 4 times because of 4 sub-banks. */
/*  - Repeat again for superbank B */
```

```
R2 = 64;   /* Need to increment the set index every loop (Should
be 64 for D-cache) */
R3.L = 0;   /* sub-bank increment, at the end of inner loop */
R3.H = 1;
P4 = 4;   /* Number of sub-bank - also outer loop counter (Should
be 2 for D-cache) */
P3 = R2;   /* Inner loop counter (Number of set index) */
```

Listing 6-3. Invalidating Bank A

```
R4.H = 0x00;   /* Initial value for DTEST_COMMAND for way 0 -
should be WRITE to TAG */
R4.L = 2;
R5.H = 0x400;   /* Initial value for DTEST_COMMAND for way 1 -
should be WRITE to TAG */
R5.L = 2;

/* Way 0,1 invalidation */
lsetup(LBL0C,LBL3C) lc1 = p4;
LBL0C: r0 = r4;
       r1 = r5;
       lsetup(LBL1C,LBL2C) lc0 = p3;
       LBL1C:   r0 = r0+|+r2 || [i0]=r0;
       LBL2C:   r1 = r1+|+r2 || [i0]=r1;
       r4 = r4+r3;
LBL3C: r5 = r5+r3;
```

Listing 6-4. Invalidating Bank B

```
R4.H = 0x080;   /* Initial value for DTEST_COMMAND for way 0 -
should be WRITE to TAG */
R4.L = 2;
R5.H = 0x480;   /* Initial value for DTEST_COMMAND for way 1 -
should be WRITE to TAG */
R5.L = 2;

/* Way 0,1 invalidation */
lsetup(LBL0D,LBL3D) lc1 = p4;
LBL0D: r0 = r4;
       r1 = r5;
       lsetup(LBL1D,LBL2D) lc0 = p3;
       LBL1D:   r0 = r0 +|+ r2 || [i0]=r0;
       LBL2D:   r1 = r1 +|+ r2 || [i0]=r1;
       r4 = r4+r3;
LBL3D: r5 = r5+r3;

/* Configure L1 SRAM data banks as SRAM */
/* - Default to DCBS==0, so LOWBIT (bit14) selects bank A or B
(because that splits L2 across A and B) */
/* - set ENDM==1, so L1 Data Memory is enabled. */
P0.L = (DMEM_CONTROL & 0xFFFF);
P0.H = (DMEM_CONTROL >> 16);
R0 = (ENDM);
[P0] = R0;
/* Configure L1 SRAM code bank as SRAM */
/* - Default to ILOC==0000 */
/* - Set ENIM==1, so Code memory is enabled. */
P0.L = (IMEM_CONTROL & 0xFFFF);
P0.H = (IMEM_CONTROL >> 16);
R0 = (ENIM);
```

```
[P0] = R0;
/* L1CACHE */
```

L1 Data Memory

In addition to the 4 KB scratchpad memory, the ADSP-BF535 processor has two 16 KB, L1 data SRAM banks. Each 16 KB, L1 data bank consists of four 4 KB sub-banks.

Each 16 KB L1 Data bank can be configured to operate as either a cache or an SRAM. As shown in Table 6-4, the data memory banks of the ADSP-BF535 processor can be configured several ways.

Table 6-4. Primary Data Memory Banks

Data Memory Bank	Description
Data Scratchpad SRAM	A local memory space particularly suited for Supervisor/User stack allocation.
Data Bank A	Can be configured as SRAM, data cache, or disabled. If Data Bank A is configured as SRAM, the system DMA can also access it directly.
Data Bank B	Can be configured as SRAM, data cache, or disabled. If Data Bank B is configured as SRAM, the system DMA can also access it directly.

The scratchpad SRAM bank is always enabled. However, DMA access is not available to the scratchpad memory.

The interaction of the Data Memory Configure bits (DMC[1:0]) and the Enable Data Memory (ENDM) control bit in the DMEM_CONTROL register determines how the data banks are configured (see Table 6-5 and Figure 6-3 on page 6-13). The ENDM bit is used to enable or disable both L1 data banks.

If ENDM is cleared, L1 memory is disabled, and all data memory references generate exceptions.

Table 6-5. Data Memory Configure Bits

ENDM	DMC[1:0]	Configuration
0	xx	Data Bank A is disabled; Data Bank B is disabled
1	00	Data Bank A is SRAM; Data Bank B is SRAM. This is the default state at reset.
1	01	Reserved
1	10	Data Bank A is cache; Data Bank B is SRAM
1	11	Data Bank A is cache; Data Bank B is cache

Critical processor applications can take advantage of this memory organization so that one data bank can be configured as data cache to support other software functions.

L1 Data SRAM

Each of the 16 KB SRAM regions implemented for Data Bank A and Data Bank B is further divided into four sub-banks. Like the organization of L1 Instruction Memory, this organization provides an effective dual port capability that gives the system DMA simultaneous access to the SRAMs—provided that collisions to the same sub-bank do not occur. Each of these regions is single ported, and if address collision is detected, access is granted first to system DMA, then to the DAGs.

The division of each data bank into four sub-banks allows code optimization that permits two simultaneous, parallel DAG references to the same data bank. The division into sub-banks and subsequent code optimization also permit system DMA access—for example, to preload or postunload data buffers. For more information, see"Data Address Generators" on page 5-1.

Table 6-6 shows how the sub-bank organization is mapped into memory.

Table 6-6. L1 Data Memory SRAM Sub-Bank Start Addresses

Memory Sub-Bank	Data Bank A	Data Bank B
0	0xFF80 0000	0xFF90 0000
1	0xFF80 1000	0xFF90 1000
2	0xFF80 2000	0xFF90 2000
3	0xFF80 3000	0xFF90 3000

Figure 6-12 describes the L1 Data Memory architecture.

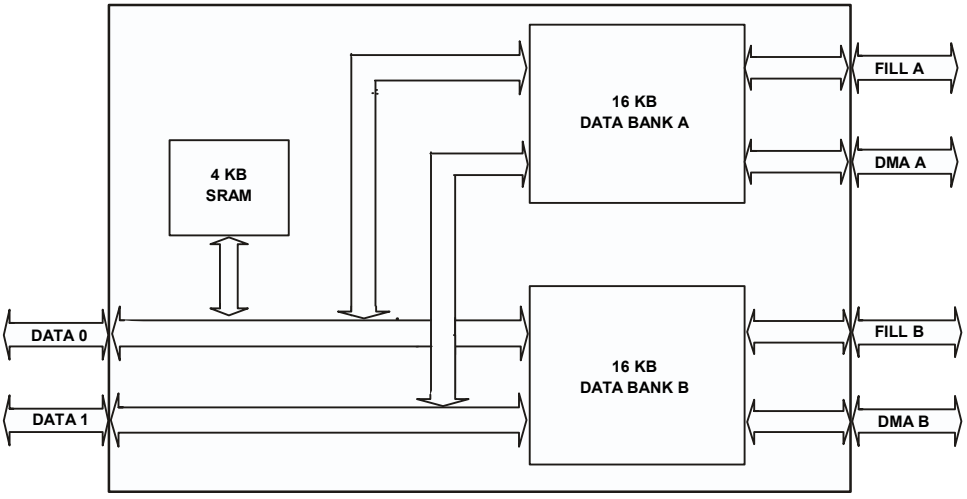


Figure 6-12. L1 Data Memory Architecture

Figure 6-13 describes the L1 data bank architecture.

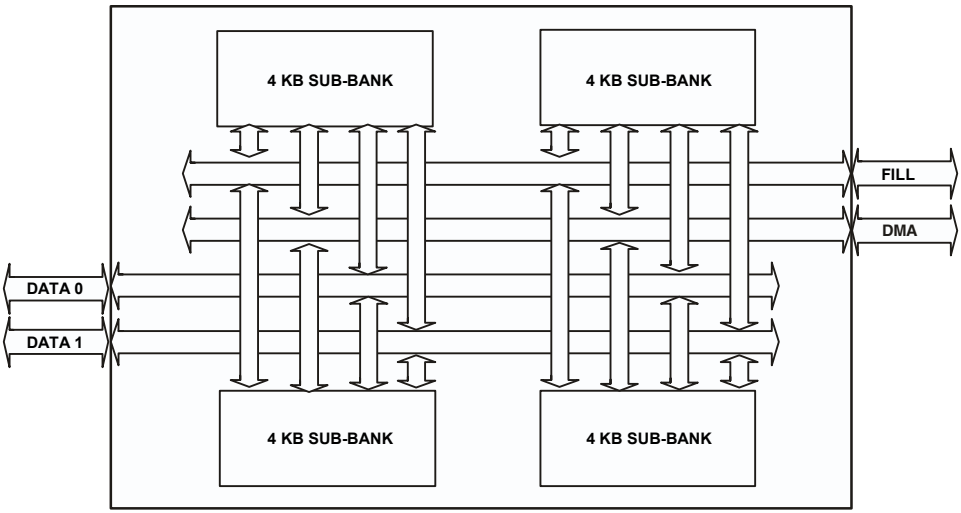


Figure 6-13. L1 Data Bank Architecture

L1 Data Cache

The L1 Data Memory architecture provides two 16 KB banks that can be configured as cache. In general, L1 data cache operates like the L1 Instruction cache described in the previous section. In addition to operating as a 2-Way set associative cache, however, L1 data cache has some unique features that are described below.

The DCPLB descriptors control data cacheability. Through these descriptors, the data cache can contain data from any address region that is defined as cacheable and has a valid CPLB entry. The data cache cannot, however, contain data from the MMR address space, which is always cache inhibited. Software must maintain the CPLBs in a manner consistent with the hardware configuration.

If one or more L1 data banks are configured as cache, CPLB descriptors must be present and must describe these things as cache inhibited:

- The scratchpad SRAM
- Any of the data or instruction banks that are enabled as SRAM
- PCI space
- Peripherals that cannot support burst accesses and that are mapped into memory outside the system MMR space

If only Data Bank A is configured as cache, then all cacheable memory references access that bank. If both data banks are configured as cache, the bank-select mechanism is programmable. The Data Cache Bank Select (DCBS) bit in the DMEM_CONTROL register selects either the large bank size or the small bank size (see Table 6-7).

Table 6-7. Data Cache Bank-Select Address (DBCS) Bit

DCBS	Description
0	Use Address bit A[14] to select Data Cache Bank A or B
1	Use Address bit A[23] to select Data Cache Bank A or B

Example of Mapping Cacheable Address Space into Data Banks

An example of how the cacheable address space maps into the two banks follows.

When both banks are configured as cache, they operate as two independent, 16 KB, 2-Way set associative caches that can be independently mapped into the Blackfin address space.

If both data banks are configured as cache, DCBS 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 A. All addresses in which A[14]=1 use Data Bank B.
- In this case, A[23] is treated as just 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 A. All addresses where A[23]=1 use Data Bank B.
- In this case, A[14] is treated as just 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 16 KB pages of memory map into each of the two 16 KB caches implemented by the two data banks. Consequently, any data in the first 16 KB of memory could be stored *only* in Data Bank A. Any data in the next address range (16 KB through 32 KB)–1 could be stored *only* in Data Bank B. Any data in the next range (32 KB through 48 KB)–1 would be stored in Data Bank A. Alternate mapping would continue.
- As a result, the cache operates as if it were a single, contiguous, 2-Way set-associative 32 KB cache. Each Way is 16 KB long and all data elements having the same first 14 bits of address compete for two entries that may be used to store them.

- 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 16 KB, 2-Way set associative cache. Each Bank serves an alternating set of 8 MB blocks of memory. For example, Data Bank A caches all data accesses for the first 8 MB of memory address range. That is, every 8 MB of range vies for the two line entries (rather than every 16 KB repeat). Likewise, Data Bank B caches data located above 8MB and below 16 MB.
- For example, if the application is working from a data set that is 1 MB long and located entirely in the first 8 MB of memory, it is effectively served by only half the cache, that is, by Data Bank A, which is a 16 KB, 2-Way set associative cache. In this instance, the application never derives any benefit from Data Bank B.

For most applications, it is best to operate with DCBS=0. When DCBS=1, on-chip L2 memory is cached only in Data Bank A.

However, if the application is working from *two* data sets, located in *two* memory spaces at least 8 MB 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.

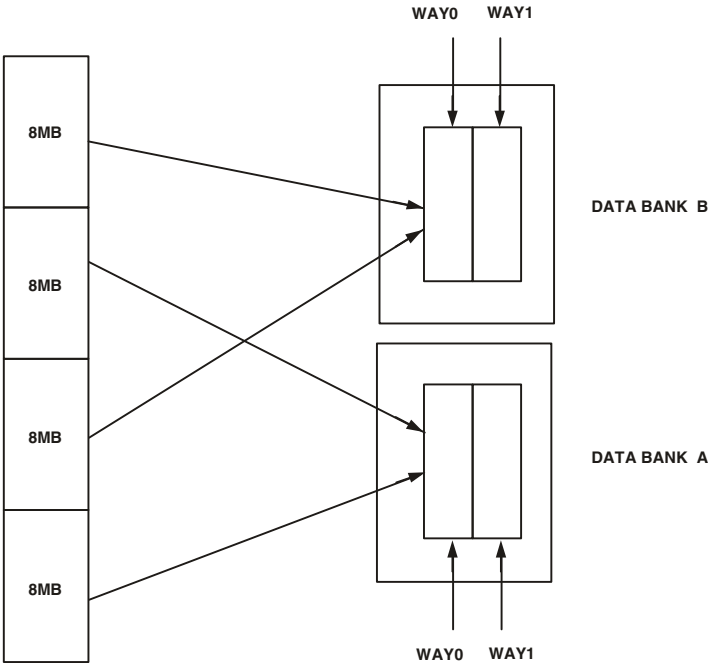


Figure 6-14. Data Cache Mapping When DCBS=1

The DCBS selection can be changed dynamically; however, to ensure that no data is lost, first flush and invalidate the entire cache.

The ADSP-BF535 processor implements each data cache as a multibank organization of subcaches. Provided that the dual access DSP instructions address different subcaches and the cache line fill is not writing to the same bank, address collision detection does not halt the processor. If a collision is detected, the access priority is:

- System DMA access
- Cache Line Fill
- DAGs

Data Cache Access

The cache controller tests the address from either DAG 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 L2 memory via the System Bus Interface Unit (SBIU).

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 space 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 L2 memory before the new data is written to the cache.

The ADSP-BF535 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 the L2 memory via the SBIU before the victim copyback operation. 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 ADSP-BF535 processor 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 does not stall on the miss. 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 4-38.

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 L2 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 L2 memory.
- A write back cache does not write to L2 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 four-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.

Data Cache Control Instructions

The ADSP-BF535 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 L2 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 L2 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 Test Registers

Like L1 Instruction Memory, L1 Data Memory contains additional MMRs to allow arbitrary read/write of all cache entries directly. They make 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] and the upper 32-bits are stored in the DTEST_DATA[1] register. When the tag arrays are being accessed, then DTEST_DATA[0] is used.

Before accessing the cache entries through the DTEST registers, enable the L1 Data memories by setting the ENDM and DMC[1:0] bits and clearing the ENDCPLB bit in DMEM_CONTROL register.

These figures describe the DTEST registers:

- Data Test Command Register in Figure 6-15 on page 6-49
- Data Test Data 1 Register in Figure 6-16 on page 6-50
- Data Test Data 0 Register in Figure 6-17 on page 6-51

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. When reading from DTEST registers, reverse the sequence. Always read the DTEST_COMMAND register first, then the DTEST_DATA registers.

Data Test Command Register (DTEST_COMMAND)

When the Data Test Command register (DTEST_COMMAND), shown in Figure 6-15, 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]).

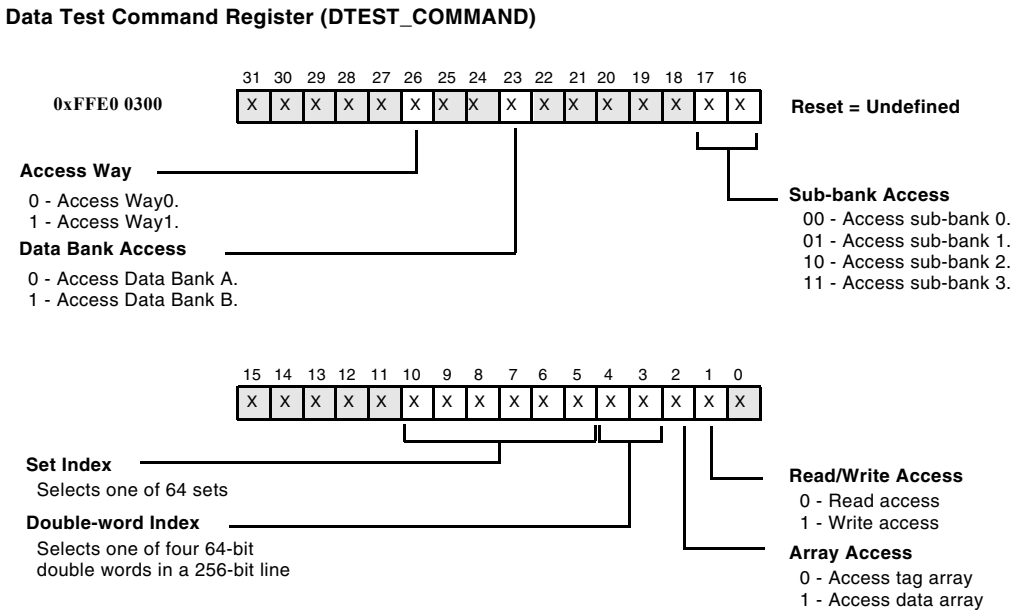


Figure 6-15. Data Test Command Register

Data Test Data 1 Register (DTEST_DATA1)

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), shown in Figure 6-16, stores the upper 32 bits.

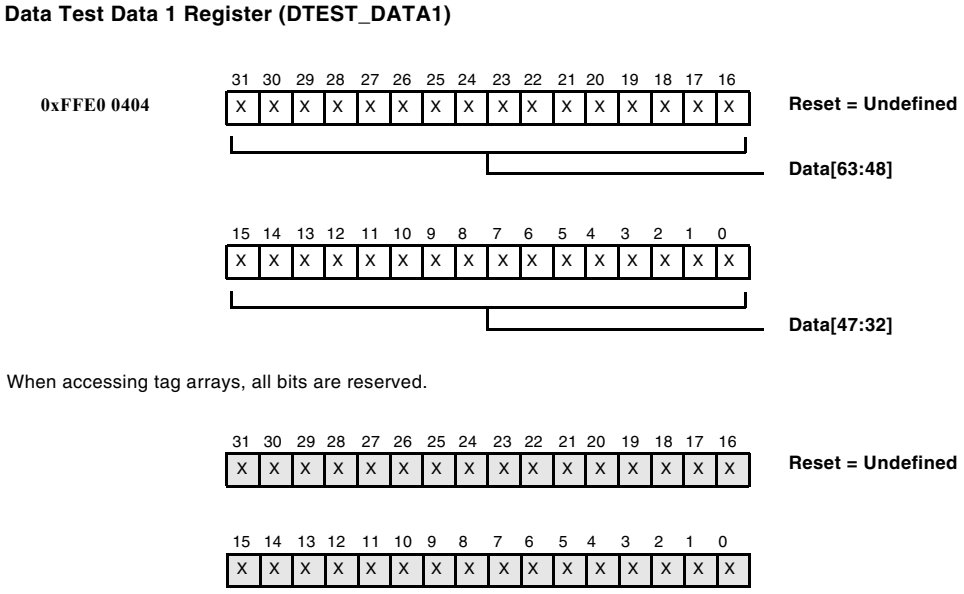


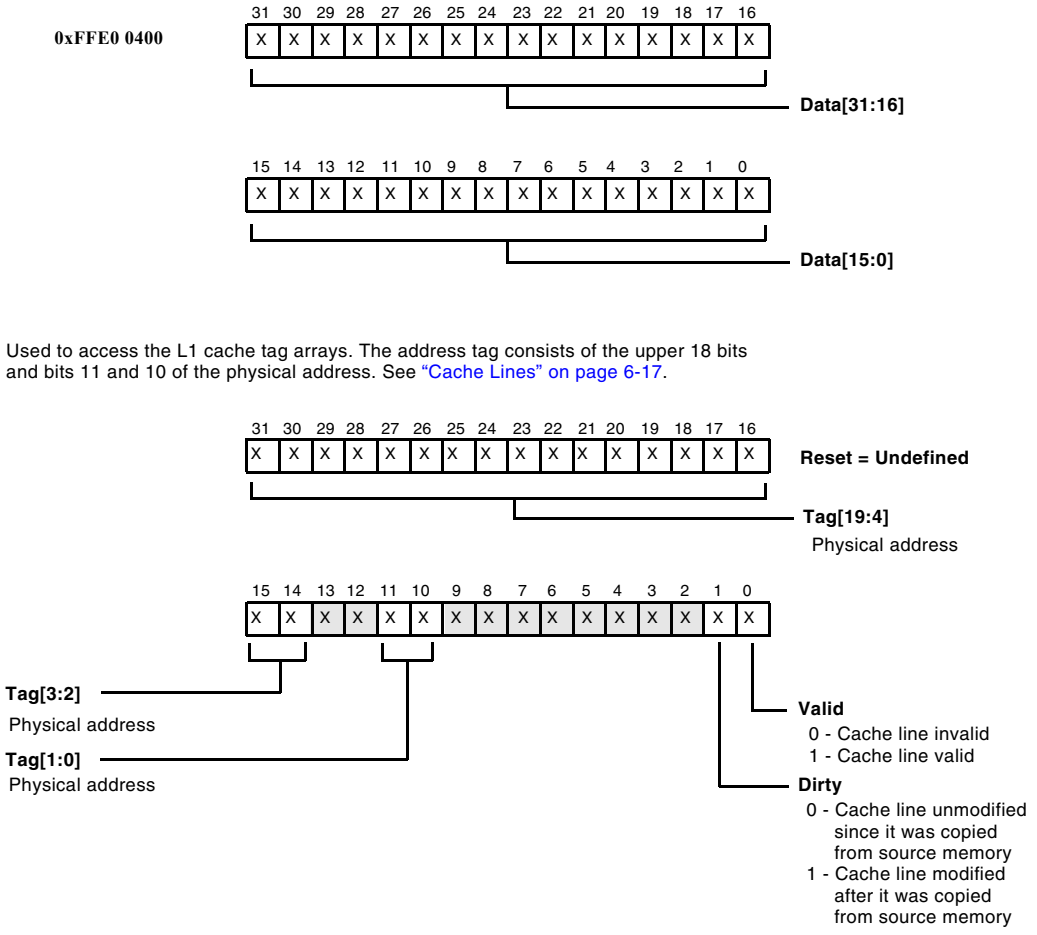
Figure 6-16. Data Test Data 1 Register

Data Test Data 0 Register (DTEST_DATA0)

The Data Test Data 0 register (DTEST_DATA0), Figure 6-17, stores the lower 32 bits of the 64-bit data to be written or 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.

Data Test Data 0 Register (DTEST_DATA0)



Used to access the L1 cache tag arrays. The address tag consists of the upper 18 bits and bits 11 and 10 of the physical address. See "Cache Lines" on page 6-17.

Figure 6-17. Data Test Data 0 Register

On-Chip Level 2 (L2) Memory

Configured as SRAM, the on-chip Level 2 (L2) memory of the ADSP-BF535 processor provides 256 KB of low latency, high bandwidth storage capacity. For systems that use some or all ADSP-BF535 processor L1 memory as cache, the on-chip L2 SRAM memory system can help provide deterministic, bounded memory access times.

Simultaneous access to the multibanked, on-chip L2 memory architecture from the core processor and peripheral bus can occur in parallel, provided that they access different banks. A fixed-priority arbitration scheme resolves conflicts. The on-chip DMA controllers and the PCI controller share a dedicated 32-bit data path into the L2 memory system. This interface operates at the SCLK frequency. Dedicated L2 access from the processor core is also supported.

The ADSP-BF535 processor has a dedicated, low latency, 64-bit data path into the L2 SRAM memory. At a core clock frequency of 300 MHz, the peak data transfer rate across this interface is 2.4 GB/second.

On-Chip L2 Bank Access

The on-chip L2 memory employs a multibank architecture like that of the L1 memory (see Figure 6-18). The ADSP-BF535 processor has 256 KB of SRAM organized as eight SRAM banks of 32 KB each. Figure 6-18 shows the size and starting location of each L2 bank.

Two L2 access ports, a processor core port and an I/O port, are provided to allow concurrent access to the L2 memory, provided that the two ports access different memory sub-banks. If simultaneous access to the same memory sub-bank is attempted, collision detection logic in the L2 provides arbitration. This is a fixed priority arbiter; the DMA/PCI port always has the highest priority, unless the core is granted access to the sub-bank for a burst transfer. In this case, the L2 finishes the burst transfer before the system bus is granted access.

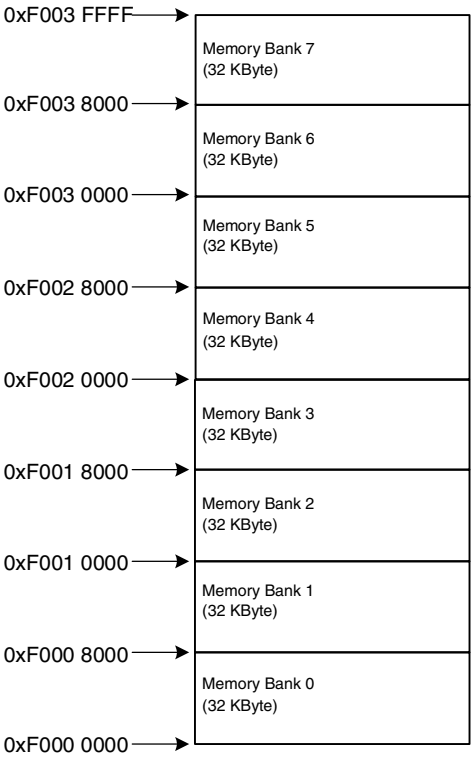


Figure 6-18. L2 Memory Map

Latency

When cache is enabled, the bus between the core and L2 memory is fully pipelined for contiguous burst transfers. The cache line fill from on-chip memory behaves the same for instruction and data fetches. Operations that miss the cache trigger a cache line replacement. This replacement fills one 256-bit (32-byte) line with four 64-bit reads. Under this condition, the L1 cache line fills from the L2 SRAM in `7+1+1+1=10` cycles. In other words, after seven core cycles, the first 64-bit (8-byte) fill is available for the processor. Figure 6-19 shows an example of L2 latency with cache on.

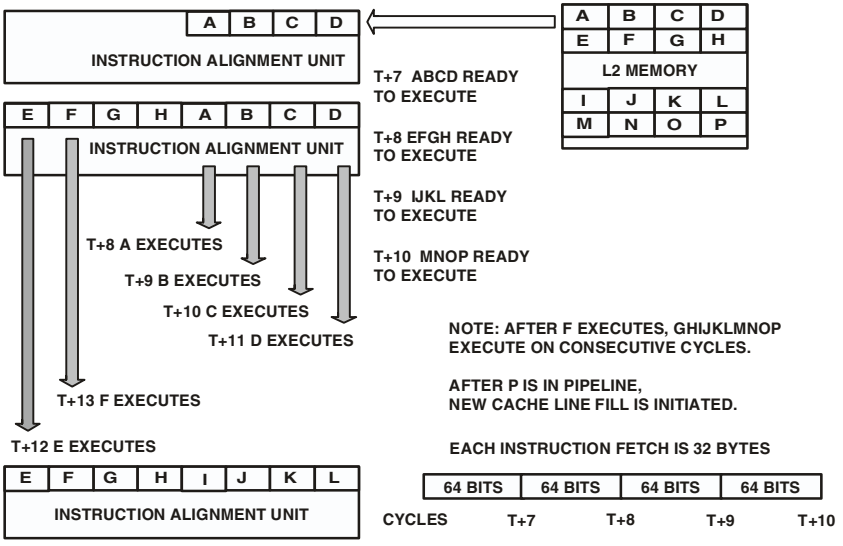


Figure 6-19. L2 Latency With Cache On

In this example, at the end of ten core cycles, 32 bytes of instructions or data have been brought into cache and are available to the sequencer. If all the instructions contain 16 bits, sixteen instructions are brought into cache at the end of ten cycles. In addition, the first instruction that is part of the cache-line fill executes on the eighth cycle; the second instruction executes on the ninth cycle, and the third instruction executes on the tenth cycle—all of them in parallel with the cache line fill.

Each cache line fill is aligned on a 32-byte boundary. When the requested instruction or data is not 32-byte aligned, the requested item is always loaded in the first read; each read is forwarded to the core as the line is filled. Sequential memory accesses miss the cache only when they reach the end of a cache line.

When on-chip L2 memory is configured as non-cacheable, instruction fetches and data fetches occur in 64-bit fills. In this case, each fill takes seven core cycles to complete. As shown in Figure 6-20, on-chip L2

memory is configured as non-cacheable. To illustrate the concept of L2 latency with cache off, simple instructions are used that do not require additional external data fetches. In this case, consecutive instructions are issued on consecutive cycles if multiple instructions are brought into the core in a given fetch. Note also that two of the seven cycles are hidden by the Blackfin processor pipeline. This is because the sequencer fetches the next 64-bit fill before executing all available instructions obtained from the previous fill.

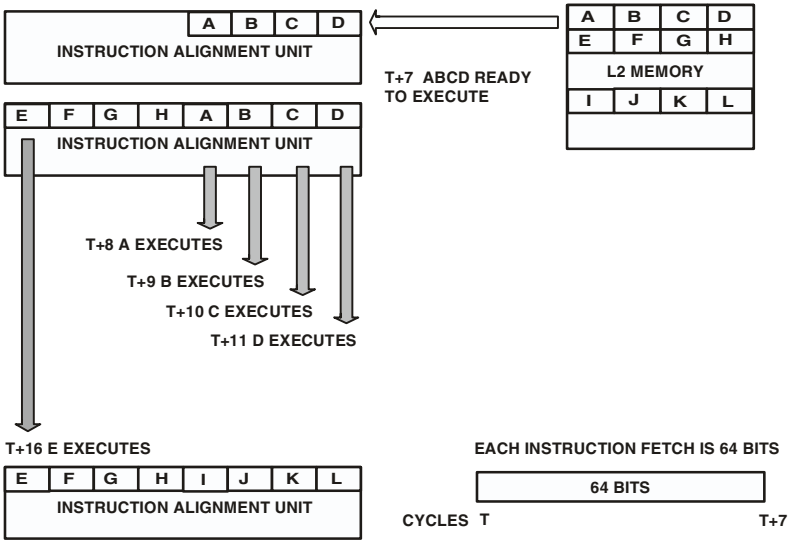


Figure 6-20. L2 Latency With Cache Off

Off-Chip L2 Memory

The external memory space is shown in Figure 20-1 on page 20-5. Four of the memory regions are dedicated to SDRAM support. The size of each SDRAM bank is programmable. Each SDRAM bank can range in size from 16 MB to 128 MB. The start address of bank 0 is 0x0000 0000. The start addresses of banks 1, 2, and 3 follow contiguously from the previous

bank. If all four SDRAM banks are not fully populated with 128 MB of SDRAM, the area from the end of bank 3 to address 0x2000 0000 is reserved.

Each of the next four banks contains 64 MB and is dedicated to support asynchronous memories.

The next region is reserved off-chip 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 that generates a hardware exception for a core access or, optionally, generates an interrupt from PCI or a DMA channel, depending on where the access originated.

Because the PCI memory space is as large as the full memory address space of the ADSP-BF535 processor, a segmented, or windowed, approach must be employed, so that only a portion of one of the PCI address spaces is visible to the core at one time.

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 ADSP-BF535 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). 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 Level 1 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 Instruction Memory Control (`IMEM_CONTROL`) and Data Memory Control (`DMEM_CONTROL`) registers, respectively. These registers are shown in Figure 6-4 on page 6-14 and Figure 6-3 on page 6-13, 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.

Memory Pages

The 4 GB address space of the ADSP-BF535 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 ADSP-BF535 architecture supports four different page sizes:

- 1 KB
- 4 KB
- 1 MB
- 4 MB

Different page sizes provide a flexible mechanism for matching the mapping of attributes to different kinds of memory and I/O. For example, a large data structure in memory, such as a video frame buffer, would typically span a significant address range, as large as 16 MB or more. The entire address range would typically share the same attributes, such as access protection and cacheability; therefore, ideally only one descriptor for the entire range is needed. In the case of the ADSP-BF535 processor, such a video frame buffer could be mapped using four 4 MB pages.

Conversely, when protecting access to different parts of memory between programs or tasks running on a system, a small memory granularity is desirable. If a task uses a data structure that is only 1 KB long and needs to be protected from other tasks, storing it in a 1 MB page would waste 1 MB minus 1 KB of memory, or 1023 KB, to keep this structure isolated from other tasks.

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 4 MB page must start on an address divisible by 4 MB; whereas a 1 KB page can start on any 1 KB boundary. The second word in the descriptor specifies the other properties or attributes of the page. These properties include:

- Page size

  1 KB, 4 KB, 1 MB, 4 MB

- 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.

- 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
  - Data pages only
- User write access permission
  - Enables or disables writes to this page when in User mode
  - Data pages only
- User read access permission

  Enables or disables reads from this page when in User mode

- 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. As in many RISC architectures, 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 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 4-38 for more information). The handler is typically part of the operating system 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 bit of these CPLB descriptors are commonly set so that they are not inadvertently replaced.

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 by clearing 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 restarted. It should then find a valid CPLB descriptor for the requested address, and the operation should proceed.

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:

1. Instruction page miss
2. A miss on DAG0
3. A miss on DAG1

MMU Application

Memory management is an optional feature in the Blackfin architecture. Its use is predicated on the system requirements of a given application. Upon reset, all CPLBs are disabled, and the 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 may contain 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 6-21, a starting point is provided for basic CPLB allocation for Instruction and Data CPLBs. Note that some ICPLBs and DCLBs have common descriptors for the same address space. For example, on-chip L2 memory would typically be configured as cacheable with an instruction and data CPLB. In such a case, external memory would take advantage of the page replacement mechanism described in the section above.

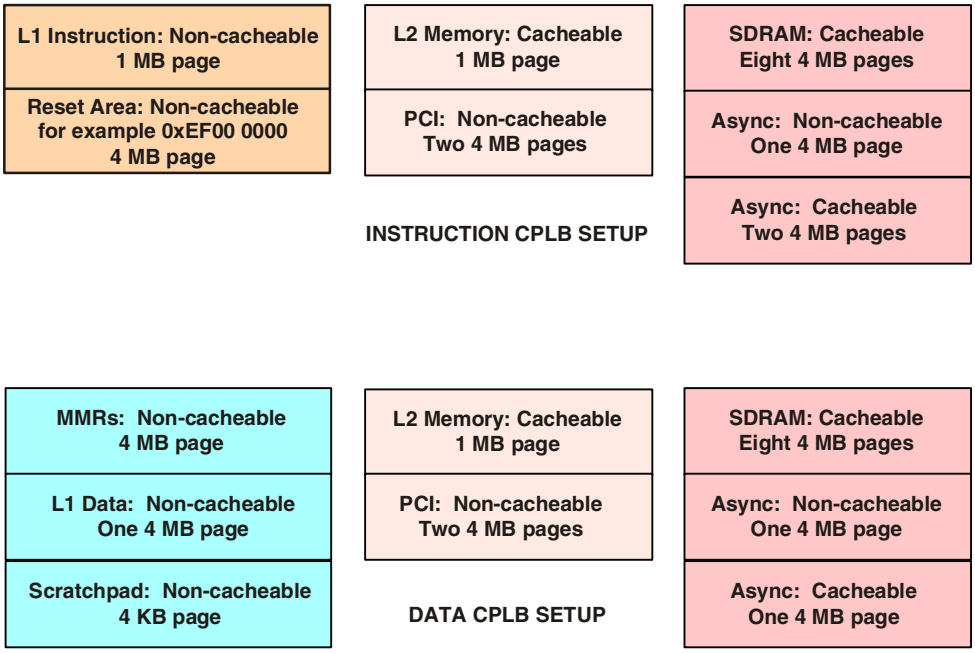


Figure 6-21. Examples of Protected Memory Regions

DCPLB Data Registers (DCPLB_DATAx)

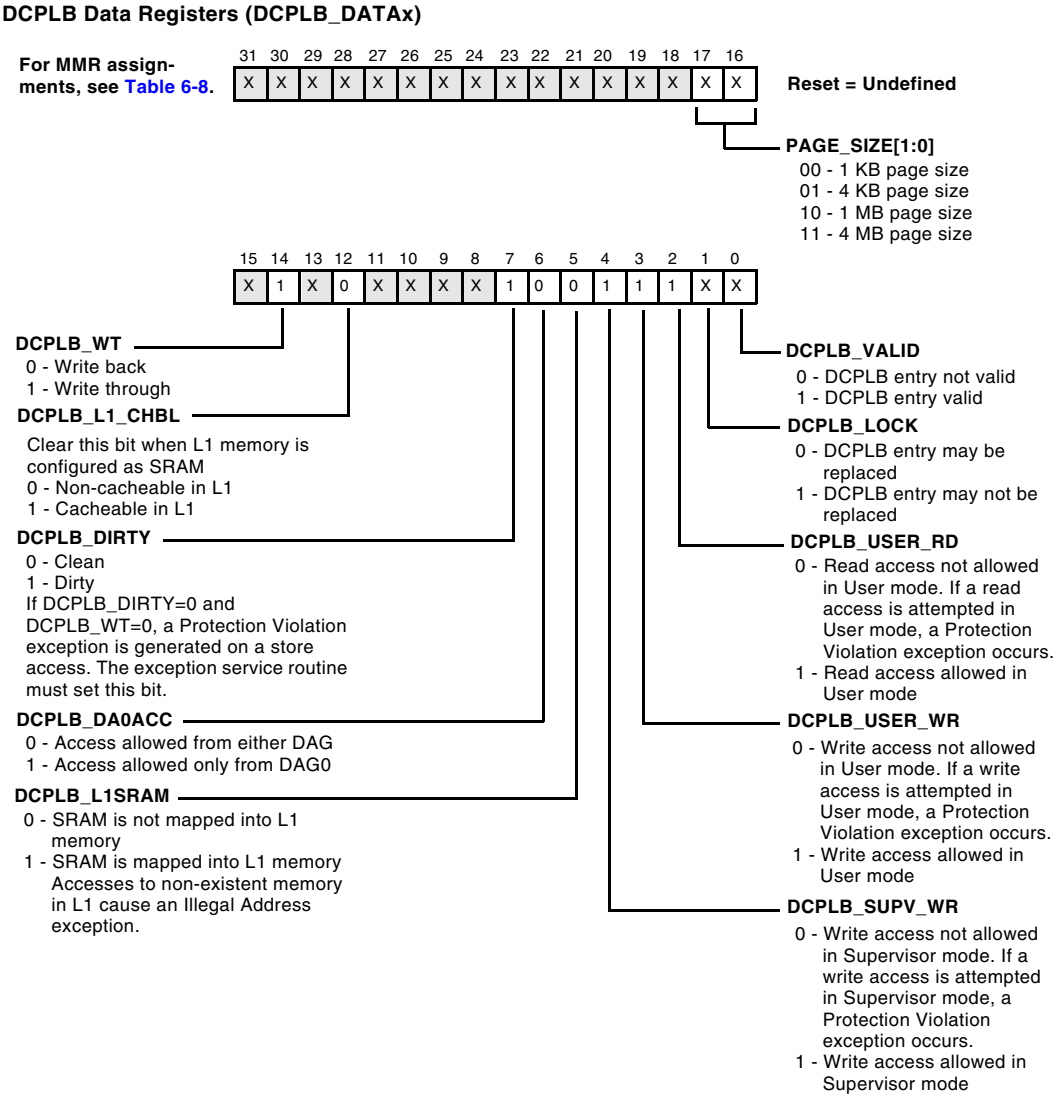


Figure 6-22. DCPLB Data Registers

Table 6-8. DCPLB Data Register MMR Assignments

Register Name	Memory-Mapped Address
DCPLB_DATA0	0xFFE0 0200
DCPLB_DATA1	0xFFE0 0204
DCPLB_DATA2	0 xFFE0 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

ICPLB Data Registers (ICPLB_DATAx)

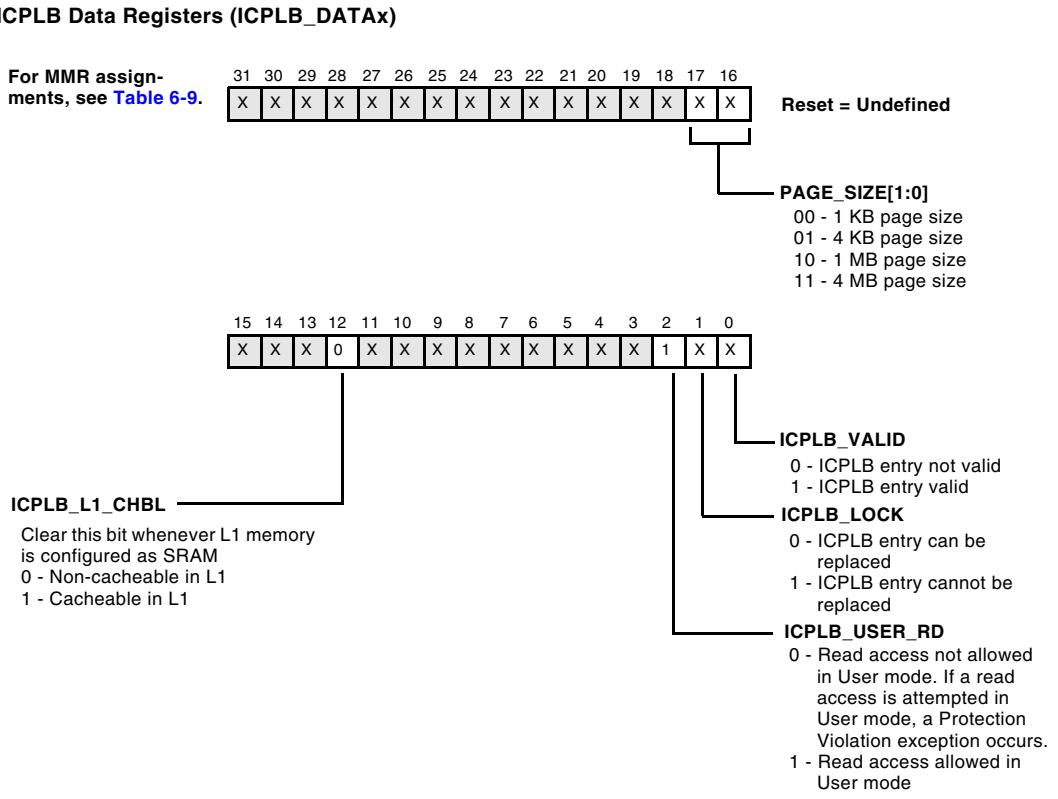


Figure 6-23. ICPLB Data Registers

Table 6-9. ICPLB Data Register MMR Assignments

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 Address Registers (DCPLB_ADDRx)

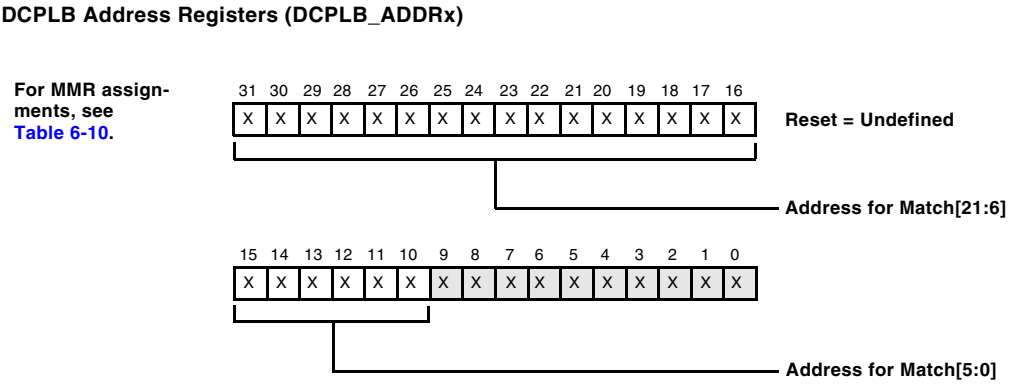


Figure 6-24. DCPLB Address Registers

Table 6-10. DCPLB Address Register MMR Assignments

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
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 Address Registers (ICPLB_ADDRx)

ICPLB Address Registers (ICPLB_ADDRx)

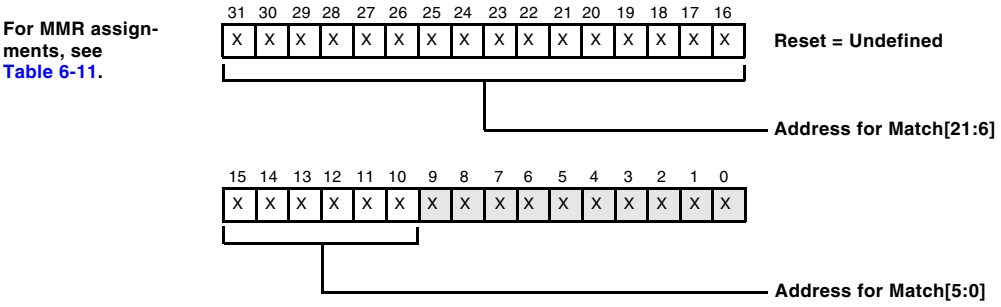


Figure 6-25. ICPLB Address Registers

Table 6-11. ICPLB Address Register MMR Assignments

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 and ICPLB Status Registers (DCPLB_STATUS, ICPLB_STATUS)

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.

DCPLB Status Register (DCPLB_STATUS)

Bits FAULT_DAG, FAULT_USERSUPV and FAULT_READWRITE in the DCPLB Status register (DCPLB_STATUS) are used to identify the DCPLB entry that has triggered the DCPLB related exception (see Figure 6-26).

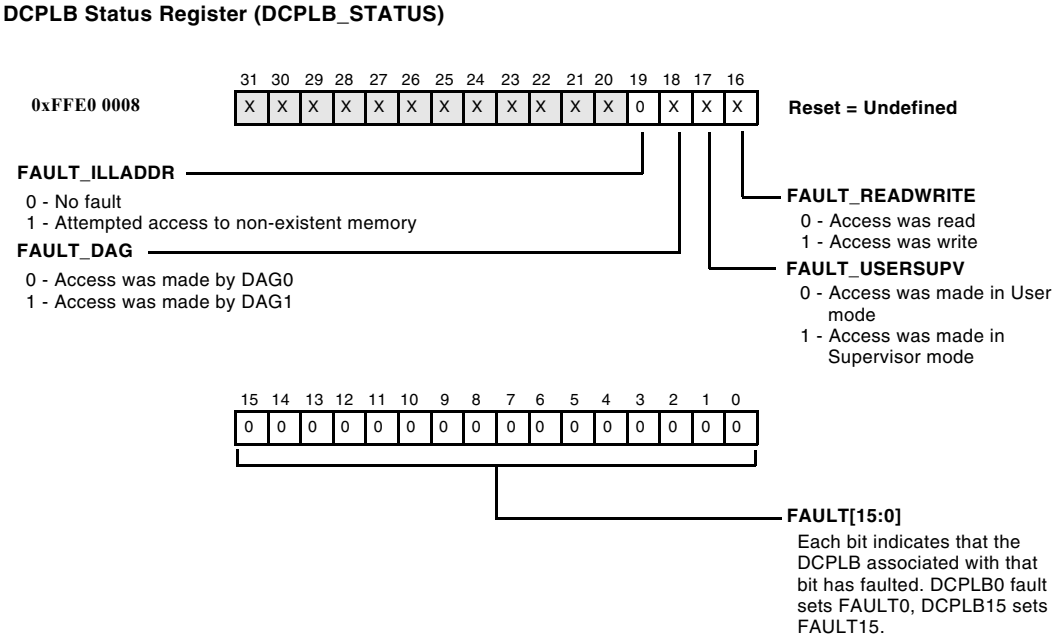


Figure 6-26. DCPLB Status Register

ICPLB Status Register (ICPLB_STATUS)

Bit FAULT_USERSUPV in the ICPLB Status register (ICPLB_STATUS) is used to identify the ICPLB entry that has triggered the ICPLB related exception (see Figure 6-27)

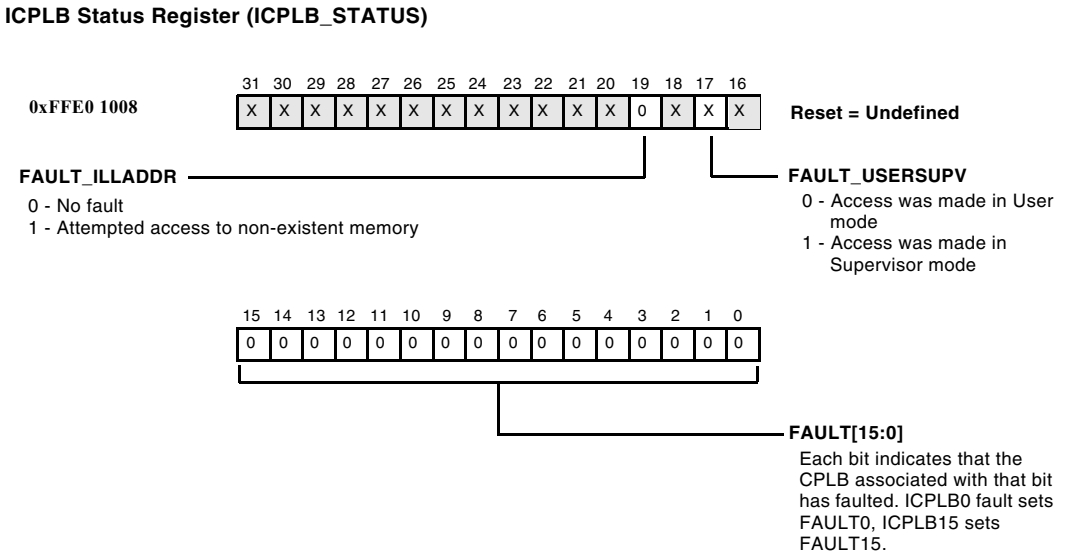


Figure 6-27. ICPLB Status Register

DCPLB and ICPLB Fault Address Registers (DCPLB_FAULT_ADDR, ICPLB_FAULT-ADDR)

The DCPLB Fault Address register (DCPLB_FAULT_ADDR), shown in Figure 6-28, and ICPLB Fault Address register (ICPLB_FAULT_ADDR), shown in Figure 6-29, hold the address that has caused a fault in the L1 Data Memory or L1 Instruction Memory, respectively.

DCPLB Fault Address Register (DCPLB_FAULT_ADDR)

The DCPLB Fault Address register holds the address that has caused a fault in the L1 Data Memory.

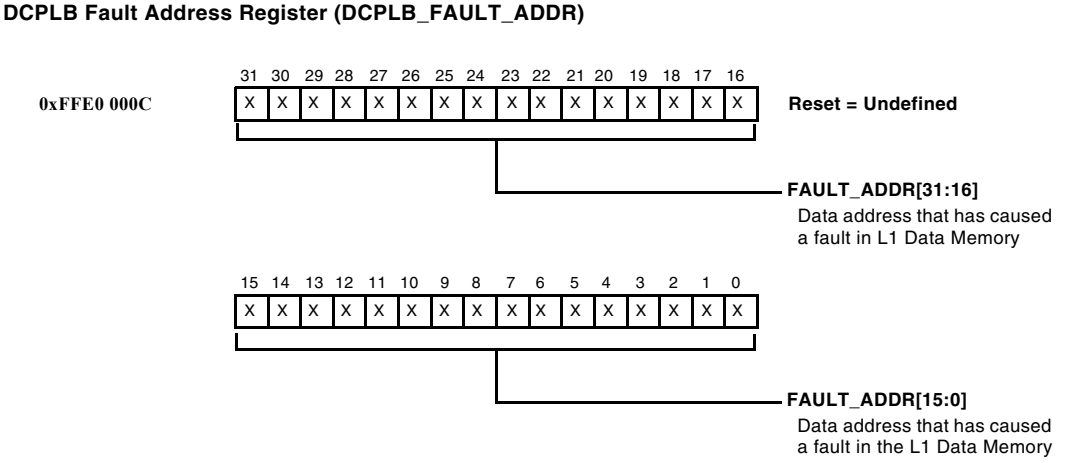


Figure 6-28. DCPLB Fault Address Register

ICPLB Fault Address Register (ICPLB_FAULT_ADDR)

The Instruction Fault Address Register, shown in Figure 6-29, holds the address that has caused a fault in the L1 Instruction Memory.

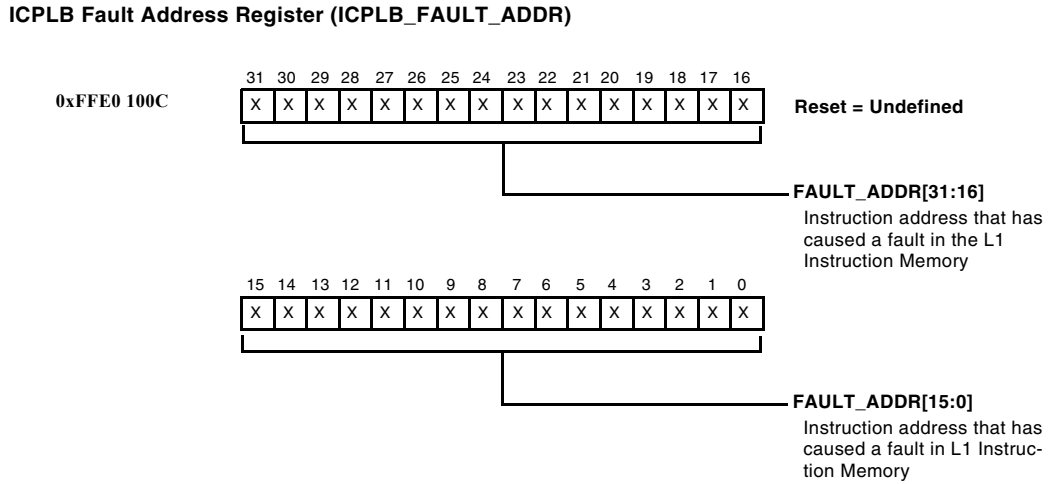


Figure 6-29. ICPLB Fault Address Register

Memory Transaction Model

Both internal and external memory locations are accessed in little endian byte order. Figure 6-30 shows a data word as it is stored in register `R0` and in memory at address location *addr*. B0 refers to the least significant byte of the 32-bit word.

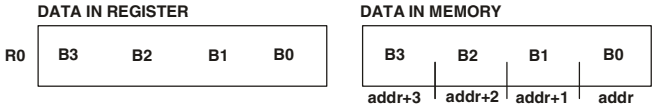


Figure 6-30. Data Stored in Little Endian Order

Figure 6-31 shows 16-bit and 32-bit instructions stored in memory. The diagram on the left shows how 16-bit instructions are 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*).

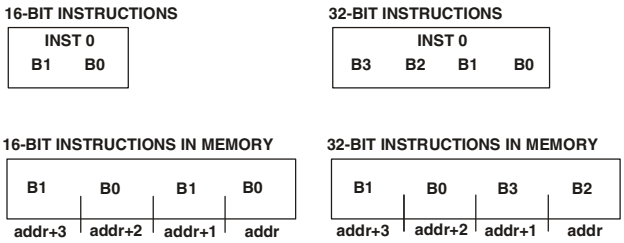


Figure 6-31. Instructions Stored in Little Endian Order

The diagram on the right shows how 32-bit instructions are stored in memory. 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. The unrelated instructions execute in parallel while the processor waits for the memory system to return the data. 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 the execution of subsequent instructions is not guaranteed and 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 instruction(s) 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 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 non-memory 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, or *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 that 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 this example code sequence:

```
IF CC JUMP away_from_here
csync;
r0 = [p0];
away_from_here:
```

In the preceding example code, the CSYNC instruction ensures that

- 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, SSYNC flushes any write buffers between the L1 memory and the SBIU and generates a sync request to the SBIU. The sync request requires acknowledgement by the SBIU 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 re-executed. 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 re-executed. 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 or write operations on a given address location, disable interrupts before performing the load or store 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

Unaligned memory operations are not directly supported. An unaligned memory reference generates a Misaligned Access exception event (see"Exceptions" on page 4-38). However, because some data streams, such as 8-bit video data, can properly be unaligned 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.

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 6-47), and/or explicit line invalidation through the core MMRs (see "ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71).

Atomic Operations

Atomic operations are used to provide non-interruptible memory operations in support of semaphores between tasks.

On the ADSP-BF535 processor, atomic operations use a test and set instruction (TESTSET). The TESTSET instruction is a two-step process that no other memory transaction can interrupt. The test portion of the instruction is a load. The set portion sets the CC bit and stores a 1 to the MSB at the same memory address. The set portion of the instruction is conditional; that is, set occurs only if the value tested is 0.

The atomic operation can access the entire logical memory space except the core MMR address region. However, atomicity may not be guaranteed for all memory regions. The memory architecture treats atomic operations as cache inhibited accesses even if the CPLB descriptor for the address indicates a cache enabled access.

If a cache hit is detected, the line is flushed and invalidated before the TESTSET is allowed to proceed.

The ADSP-BF535 processor does not guarantee atomic access to L1 memory space configured as SRAM. Consequently, semaphores must not reside in L1 memory.

To ensure that all previous exceptions and interrupts have been processed before the atomic operation begins, a CSYNC or SSYNC instruction may precede the TESTSET atomic access instruction.

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 6-77.

All MMRs are accessible only in Supervisor mode. Access to MMRs in User mode generates an Illegal Access 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.

Accesses to nonexistent MMRs generate an 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.

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

Core MMR Programming Code Example

Core MMRs may be accessed only as aligned 32-bit words. Non-aligned access to MMRs generates an exception event. Listing 6-5 shows the instructions required to manipulate a generic core MMR.

Listing 6-5. Core MMR Programming

```
CLI R0;   /* stop interrupts and save IMASK */
P0 = MMR_BASE;   /* 32-bit instruction to loadbase 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 */
```

In the code listing, CLI saves the contents of the `IMASK` register and disables interrupts by clearing `IMASK`, while `STI` restores the contents of the `IMASK` register, thus enabling interrupts. The instructions between `CLI` and `STI` are not interruptible.

7 CHIP BUS HIERARCHY

This chapter discusses the on-chip buses, including how data moves through the system and factors that determine the system organization. The chapter describes the system internal chip interfaces and discusses the system interconnects, including the System Bus Interface Unit (SBIU) and the associated system buses.

Internal Interfaces

Figure 7-1 shows the core processor and system boundaries and the interfaces between them. In the figure, a box receiving an arrow is a slave on that bus and a box sourcing an arrow is a master on that bus. The System Bus Interface Unit (SBIU) functions as the major crossbar switch between the various buses.

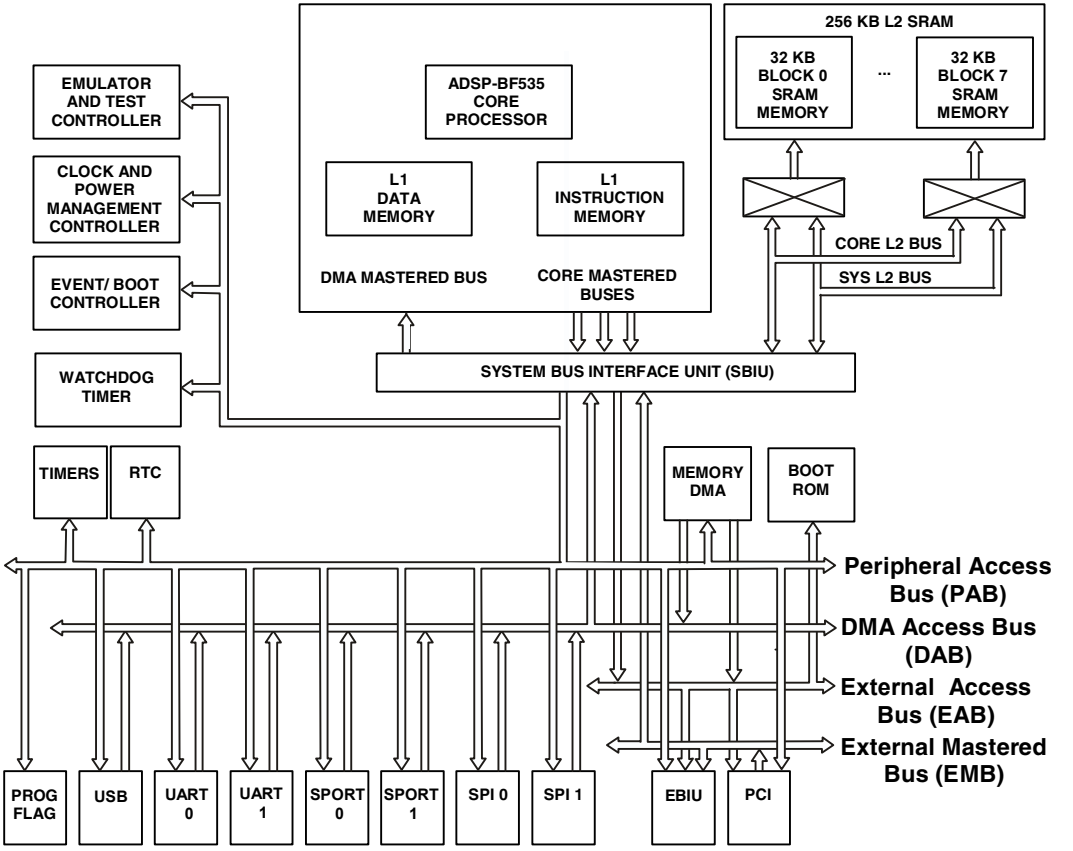


Figure 7-1. ADSP-BF535 Processor Bus Hierarchy

ADSP-BF535 Internal Clocks

The Peripheral Access Bus (PAB), the DMA Access Bus (DAB), the External Access Bus (EAB), the External Mastered Bus (EMB), and the External Bus Interface Unit (EBIU) run at the core clock frequency (`CCLK` domain) divided by 2, 2.5, 3, or 4. This divided frequency is the frequency of the

SCLK domain. This divider ratio is set using the SSEL parameter of the PLL Control register. For example, for a 300 MHz core frequency, dividing by 2.5 yields 120 MHz.

These buses can also be cycled at a lower frequency to reduce power consumption. Note that all synchronous peripherals derive their timing from the SCLK. For example, the UART clock rate is determined by further dividing this clock frequency.

Core Overview

For the purposes of this discussion, Level 1 memories (L1) are included in the description of the core; they enjoy full bandwidth access from the processor core. The core includes the Level 1 (L1) memory subsystem, with 16 KB Instruction SRAM/cache, a dedicated 4 KB Scratchpad SRAM block, and 32 KB of data SRAM/cache configured as two independent banks. Except for the Scratchpad, each independent bank can be configured as either SRAM or cache.

The block diagram in Figure 7-2 shows the core processor and its interfaces to the SBIU. The core processor is master to these three off-core interfaces:

- Instruction Fetch (Core I bus), used to fetch 64 bits of instruction from memory
- Bank0 Load/Store (Core D0 bus), used to write or store 32 bits of data to or from memory
- Bank1 Load/Store (Core D1 bus), used to write or store 32 bits of data to or from memory

The core can generate up to three simultaneous off-core accesses per cycle.

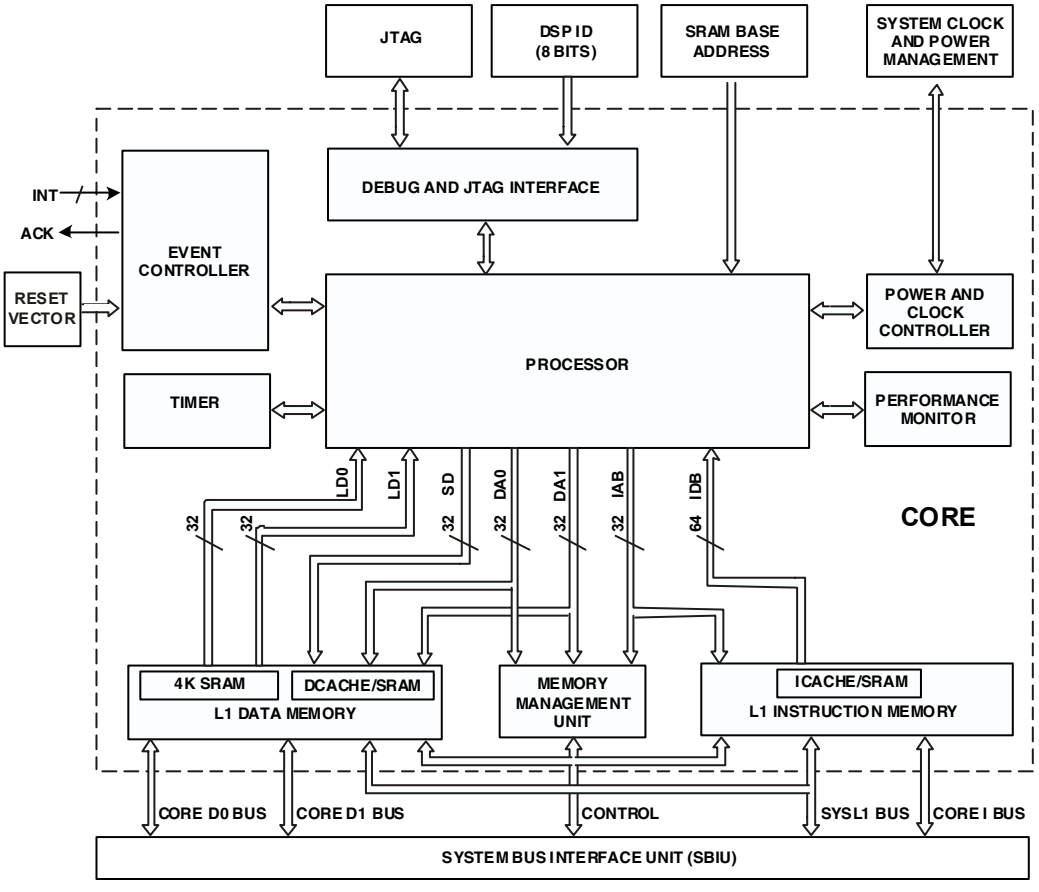


Figure 7-2. Blackfin Processor Core Block Diagram

The off-core system is master to a dedicated L1 interface (System L1 bus). This bus provides direct memory access to L1 memory configured as SRAM, except for the Scratchpad SRAM. L1 memory configured as cache is not accessible by the off-core system.

The Core I bus, the Core D0 bus, the Core D1 bus, and the System L1 bus run at the full core frequency, have data paths up to 64 bits, and support burst transfers. All four ports have 32-bit address buses. The Core D0

and Core D1 ports can perform single-byte, 2-byte (16-bit), or 4-byte (32-bit) aligned read/writes on behalf of the core load/store instructions, or four 64-bit wide, 32-byte burst transfers on behalf of the cache line fill/write back hardware.

The Core I bus makes only 64-bit read requests (either single, or as part of a 32-byte line-fill burst operation).

When the request is filled, the 64-bit read can contain a single 64-bit instruction or any combination of 16-, 32-, or partial 64-bit 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. The Core I bus handles both instruction and data cache fills.

System Overview

The system includes the SBIU, the on-chip L2 SRAM memory, the peripheral set, and the controllers for system interrupts, test/emulation, and clock and power management. The SBIU intercepts all bus transfers to or from the core, the L2 memory, and the ADSP-BF535 processor system. The SBIU performs all protocol conversions, including any synchronous clock domain conversion required to support the transaction.

System Bus Interface Unit (SBIU)

The SBIU functions as a high-speed parallel switch, or router. It provides crossbar switches between the core buses, operating at the full core frequency, and the system buses, running at the SCLK frequency.

Table 7-1 describes the interconnect routing supported by the SBIU. The relative priority of the requesters is shown in the target resource columns. A value of 1 indicates the highest priority. Empty table cells represent unsupported interconnects.

Table 7-1. SBIU Internal Routing Priority

	Target Resource				
Requestor	**SysL1**	**CoreL2**	**SysL2**	**PAB**	**EAB**
CoreD0		1		2	1
CoreD1		2			2
CoreI		3			3
PCI (EMB)			1	1	
DAB	1		2		4
MemDMA (EAB)					5

Up to five parallel, concurrent bus operations can be in progress in any one cycle.

For example:

- A peripheral DMA channel is accessing L1 memory.
- PCI is accessing L2 memory.
- The core is fetching instructions from L2 memory.
- Core D0 is accessing a system MMR on the PAB.
- Core D1 is accessing external memory.

On-Chip L2 SRAM Memory Interface

The on-chip L2 SRAM memory block consists of 256 KB of fast, deterministic access time memory. The L2 memory is unified; that is, it is directly accessible by both the instruction core port and the data core ports of the architecture. The L2 is organized as a multibank architecture of single ported SRAMs, so that multiple accesses to different banks can occur in parallel. The SBIU has two ports into the L2 memory, one dedicated to core requests (Core L2), the other dedicated to system DMA and PCI requests (Sys L2). For additional information about L2 memory, see "Memory" on page 6-1.

System Interfaces

The ADSP-BF535 processor system includes the peripheral set (Timers, Real Time Clock, USB, programmable flags, UARTs, SPORTs, and SPIs), the PCI controller, the external memory controller (EBIU), the Memory DMA controller, and the interfaces between these, the system, and the optional external (off-chip) resources. See Figure 7-1.

These sections describe the four on-chip interfaces between the system and the peripherals:

- Peripheral Access Bus (PAB)
- DMA Access Bus (DAB)
- External Access Bus (EAB)
- External Mastered Bus (EMB)

There are also two primary chip pin buses, the PCI Bus and the External Bus Interface Unit (EBIU). The PCI Bus is discussed in "PCI Bus Interface" on page 13-1. The External Bus Interface Unit is discussed in "External Bus Interface Unit" on page 18-1.

The system interfaces operate at a divided frequency from that of the core.

Peripheral Bus (PAB)

The ADSP-BF535 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 ADSP-BF535 processor memory map. Both the core and the PCI Controller can access system MMR space through the PAB bus.

Both the core processor and the PCI masters have byte addressability, but the programming model is restricted to only 16- or 32-bit (aligned) access to the system MMRs. Byte access to this region is not supported.

PAB Arbitration

The core, through the Core D0 bus, and the PCI port, through the EMB, are the only masters on this bus. A fixed priority arbitration policy is supported. The PCI port interface unit has highest priority. Maximum arbitration latency for either master is 2 SCLK cycles.

PAB Performance

For PAB, the primary performance criteria is latency. Transfer latencies for both read and write transfers on the PAB are 2 SCLK cycles.

The core or the PCI controller can transfer up to 32 bits per access to the PAB slaves. Configured with a 1:2 core clock ratio, the first and subsequent system MMR (single cycle) read or write accesses take 4 core clocks (CCLK) of latency. Configured with a 1:2.5 or 1:3 core clock ratio, the first and subsequent system MMR (single cycle) read or write accesses take 6 core clocks (CCLK) of latency.

With a 300 MHz core clock and a 1:2.5 bus clock ratio, the peak peripheral bus throughput is 240 MBytes per second.

PAB Agents (Masters, Slaves)

The processor core and the PCI controller can master bus operations on the PAB, through the SBIU. All peripherals have a peripheral bus slave interface which allows the processor core to access control and status state. These registers are mapped into the system MMR space of the memory map. System MMR addresses are listed in Appendix A, "Blackfin Processor Core MMR Assignments" and Appendix B, "System MMR Assignments".

The slaves on the PAB bus are as follows:

- Event Controller
- Emulation/Test Control
- Clock and Power Management Controller
- Watchdog Timer
- Real Time Clock
- Timer 0, 1, and 2
- SPORT0
- SPORT1
- SPI0
- SPI1
- Programmable Flags
- UART0
- UART1
- USB

- PCI Port
- Asynchronous Memory Controller (AMC)
- SDRAM Controller (SDC)
- Memory DMA Controller
- SBIU

DMA Bus (DAB)

The DAB provides 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 8 DMA capable peripherals in the ADSP-BF535 processor system. Twelve DMA channels and bus masters support these devices. The peripheral DMA controllers can transfer data between peripherals and one of:

- Internal memory (L1 or L2 memory, through SBIU)
- External memory (EAB, through SBIU)

Both the read and write channels of the Memory DMA controller access their descriptor lists through the DAB.

The DAB has priority over the core processor on arbitration into L1 configured as SRAM and L2. For off-chip memory, the core has priority over the DAB on the EAB bus. The ADSP-BF535 processor uses a fixed priority arbitration policy on the DAB. Table 7-2 shows the arbitration priority.

Table 7-2. DAB Arbitration Priority

DAB Master	Arbitration Priority
SPORT0 RCV DMA Controller	0 - highest
SPORT1 RCV DMA Controller	1
SPORT0 XMT DMA Controller	2
SPORT1 XMT DMA Controller	3
USB DMA Controller	4
SPI0 DMA Controller	5
SPI1 DMA Controller	6
UART0 RCV Controller	7
UART1 RCV Controller	8
UART0 XMT Controller	9
UART1 XMT Controller	10
Memory DMA Controller	11 - lowest

DAB Performance

The ADSP-BF535 processor DAB supports single word accesses of 8-bit, 16-bit, or 32-bit data types as well as 4- and 8-cycle bursts (32-byte transfers on behalf of Memory DMA controller only). The data bus has a 32-bit width. Therefore, maximum throughput on this bus is (SCLK × 4) MBs per second. However, arbitration latencies and memory read latencies can degrade this peak throughput, particularly to off-chip memory.

The DAB has a dedicated port into L1 memory. The DAB and PCI share a port to L2 on-chip memories. No stalls occur as long as the core access, PCI direct access, and the DMA access are not to the same memory bank (4 KB size for L1, 32 KB size for L2). If there is a conflict, PCI is the highest priority requester, followed by the DMA access, then 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 L2 memory can only be stalled by:

- An access already in progress from another DMA channel.
- A PCI request (1 cycle stall on L1 or L2 access).
- A core processor access already in progress to the same 32 KB super bank.

DMA access to L1 memory can only be stalled by:

- An access already in progress from another DMA channel.

DMA read and write latency through the DAB to both on-chip and off-chip memory is shown in Table 7-3. The latencies shown are the total number of cycles per memory access including cycles required for arbitration.

Table 7-3. DAB Latencies

Use of DAB	Core Clock to System Clock Ratio (CCLK/SCLK)	Latency in SCLKs Burst of 4	Latency in SCLKs Burst of 8
Burst Write to L1 Memory	Best cases (4:1, 3:1, 2.5:1)	5-2-2-2	5-2-2-2-2-2-2-2
Burst Read from L1 Memory	Best case (4:1)	6-2-2-2	6-2-2-2-2-2-2-2
Burst Write to L2 Memory (16-byte and 32-byte)	All ratios	4-1-1-1	4-1-1-1-1-1-1-1
Burst Read from L2 Memory (16-byte and 32-byte)	4:1, 3:1	5-1-1-1	5-1-1-1-1-1-1-1
	2.5:1, 2:1	6-1-1-1	6-1-1-1-1-1-1-1

Table 7-3. DAB Latencies (Cont'd)

Use of DAB	Core Clock to System Clock Ratio (CCLK/SCLK)	Latency in SCLKs Burst of 4	Latency in SCLKs Burst of 8
Burst Read from External Memory (SDRAM) (16-byte and 32-byte)	All ratios	9-1-1-1	9-1-1-1-1-1-1-1
Write to SDRAM (burst of 8)	All ratios	4-2-2-2	4-2-2-2-2-2-2-2
Burst Read from Async Memory	All ratios	12-4-4-4	12-4-4-4-4-4-4-4
Burst Write to Async Memory	All ratios	12-4-4-4	12-4-4-4-4-4-4-4

The PCI controller, 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 to ensure that those isochronous peripherals targeting internal memory have enough allocated bandwidth and the appropriate maximum arbitration latencies.

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.

Memory DMA transfers typically 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. An additional 1 to 2 cycles per burst access is inherent in the design.

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 7-4 shows an example of two types of memory DMA transfers.

Table 7-4. Memory DMA Transfer Rates

Description of Transfer	Primary Burst of 8 Access in SCLKs	Secondary Burst of 8 Access in SCLKs	Total SCLKs
Burst of 8 Transfer from SDRAM to L1 (Off-chip to On-chip Memory Transfer)	19 Cycles (L1)	16 Cycles (SDRAM)	21 (19+2)
Burst of 8 Transfer from L2 to L1 (On-chip to On-chip Memory Transfer)	19 Cycles (L1)	12 Cycles (L2)	33 (19+12+2)

DAB Bus Agents (Masters)

All peripherals capable of sourcing a DMA access are masters on this bus, as shown in the DAB arbitration priorities in Table 7-2. A single arbiter supports a fixed priority arbitration policy for access to the DAB.

External Access Bus (EAB)

The EAB provides a way for the processor core and the Memory DMA controller to directly access off-chip memory and the PCI memory space to perform instruction fetches, data loads, data stores, and high throughput memory-to-memory DMA transfers.

EAB Arbitration

The EAB is mastered by the SBIU (on behalf of core processor requests, or DAB requests), and by the Memory DMA controller. Arbitration for use of the External Bus Interface Unit (EBIU) resources or the PCI bus resources is required because of possible contention between the potential masters of this bus. A fixed priority arbitration scheme is used, with the SBIU internal routing priorities shown in Table 7-1.

EAB Performance

The EAB supports single word accesses of either 8-bit, 16-bit or 32-bit data types, as well as 4- and 8-cycle bursts. The EAB operates at the same frequency as the PAB and the DAB.

Table 7-5 provides EAB latency and bandwidth estimates for PCI, very slow (100 ns) flash memory, and SDRAM accesses. Because of the wide range of access parameters supported by the EAB slaves, and the many possible uses of these resources for different applications, this analysis may not apply to a particular application. It is presented here for comparative purposes only.

Table 7-5. EAB Performance

EAB Performance	SCLK/ (PCI_CLK) cycles	Core cycles at 2.5:1	Peak BW MB/s at 133 MHz	Notes
133/33 SCLK to PCI clock ratio	5	13		4 SCLK cycles/PCI cycle (slight rounding)
PCI read, non-burst	36/ (9)	90		4 PCI cycles for core delay and synchronization + 1 PCI cycle for arbitration + 1 PCI cycle for address + 1 PCI cycle for data + 2 PCI cycles for core delay and synchronization
PCI read, 16-byte burst	48/ (12)	120	44	4 PCI cycles for core delay and synchronization + 1 PCI cycle for arbitration + 1 PCI cycle for address + 4 PCI cycles for data + 2 PCI cycles for core delay and synchronization
PCI read, 32-byte burst	64/ (16)	160	66	4 PCI cycles for core delay and synchronization + 1 PCI cycle for arbitration + 1 PCI cycle for address + 8 PCI cycles for data + 2 PCI cycles for core delay and synchronization
PCI write, non-burst, FIFO empty	2	5	2128	
PCI write, 16-byte burst, FIFO empty	5	13	851	
PCI write, 32-byte burst, FIFO empty	9	23	472	
PCI write, non-burst, FIFO full	12/ (3)	30	44	1 PCI cycle for arbitration + 1 PCI cycle for address + 1 PCI cycle for data (4 bytes)

Table 7-5. EAB Performance (Cont'd)

EAB Performance	SCLK/ (PCI_CLK) cycles	Core cycles at 2.5:1	Peak BW MB/s at 133 MHz	Notes
PCI write, 16-byte burst, FIFO full	24/ (6)	60	88	1 PCI cycle for arbitration + 1 PCI cycle for address + 4 PCI cycles for data (4 bytes each)
PCI write, 32-byte burst, FIFO full	40/ (10)	100	106	1 PCI cycle for arbitration + 1 PCI cycle for address + 8 PCI cycles for data (4 bytes each)
DMA or cache-line read, 32-byte burst access to 16-bit 100 ns flash	258	645	16	Worst-case read latency
DMA read, 32-byte burst, from SDRAM	13.4		318	10% page misses
DMA read, 16-byte burst, from SDRAM	9.4		227	10% page misses
DMA write, 32-byte burst, to SDRAM	16.4		260	10% page misses
DMA write, 16-byte burst, to SDRAM	8.4		254	10% page misses
Cache line read, 32-byte burst, from SDRAM	8		533	Continuous Read Buffer full hits
Cache line read, 32-byte burst, from SDRAM	13.4		318	Read Buffer disabled, 10% page misses
Cache line write, 32-byte burst, to SDRAM	16.4		260	10% page misses

EAB Bus Agents (Masters, Slaves)

The SBIU and the Memory DMA controller devices are masters on this bus. The PCI, boot ROM, and EBIU are slaves on this bus.

External Mastered Bus (EMB)

The EMB provides a means for the PCI to directly access internal L2 memory, off-chip memory, and the system MMRs.

EMB Arbitration

For ADSP-BF535 processors, only the PCI controller can master this bus.

EMB Performance

The EMB supports single word accesses of either 8-bit, 16-bit or 32-bit data types. The EMB operates at the same frequency as the PAB and the DAB.

The EMB supports access to the external memory space. PCI-mastered accesses to internal L2 memory have higher priority than core accesses to the same bank, unless the core has the memory lock asserted, or the core is in the middle of a burst.

The EMB provides access to memory-mapped registers from the PCI. Access to the system MMRs through the SBIU to the PAB is deterministic and always requires 3 SCLK cycles for the first read, requiring 1 wait state. Writes require no wait states and are fully pipelined. Burst access is not supported.

EMB Bus Agents (Masters, Slaves, Bridges)

The PCI is the only master on this bus. The SBIU and EBIU are the slaves on this bus. The PCI controller is a bridge between the EMB and the PCI buses.

Resources Accessible From EMB

The EMB provides access to:

- L2 SRAM
- External memory
- System MMR space

The EMB does *not* have access to:

- L1 memory
- Boot ROM
- PCI space (the PCI EAB slave cannot be accessed by the EMB bus, but the system MMRs of the PCI can be accessed)

8 DYNAMIC POWER MANAGEMENT

This chapter describes the Dynamic Power Management functionality of the ADSP-BF535 Blackfin processor. This functionality includes:

- Clocking
- Phase Locked Loop (PLL)
- Dynamic Power Management Controller
- Operating Modes
- Voltage Control

Clocking

The input clock into the ADSP-BF535 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. In normal operation, the user programs the PLL with a multiplication factor for `CLKIN`. The resulting multiplied signal is the core clock (`CCLK`). The `CCLK` signal clocks the Blackfin core processor.

A user-programmable value then divides the `CCLK` signal to generate the system clock (`SCLK`). The `SCLK` signal clocks the Peripheral Access Bus (PAB), DMA Bus (DAB), External Address Bus (EAB), External Mastered Bus (EMB) and the External Bus Interface Unit (EBIU). To optimize performance and power dissipation, the ADSP-BF535 processor allows the core and system clock frequencies to be changed dynamically.

All synchronous peripherals derive their timing from SCLK. For example, the Universal Asynchronous Receiver/Transmitter (UART) clock rate is determined by further dividing the SCLK signal. Several of the ADSP-BF535 peripherals, such as Universal Serial Bus (USB), Peripheral Control Interface (PCI), and Real-Time Clock (RTC), have their own clock inputs that operate asynchronously to SCLK.

Phase Locked Loop and Clock Control

To provide the clock generation for the core and system, the ADSP-BF535 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 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 PLL interacts with the Dynamic Power Management Controller (DPMC) block to provide power management functions for the ADSP-BF535 processor.

PLL Overview

The PLL supports a wide range of multiplier ratios, achieving 1–31x multiplication of the input clock, CLKIN. To achieve this wide multiplication range, the ADSP-BF535 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.

The input clock, CLKIN, is a square wave derived from a crystal oscillator or external reference clock. The Voltage Controlled Oscillator (VCO) is an intermediate clock from which the core clock (CCLK) and system clock (SCLK) are derived.

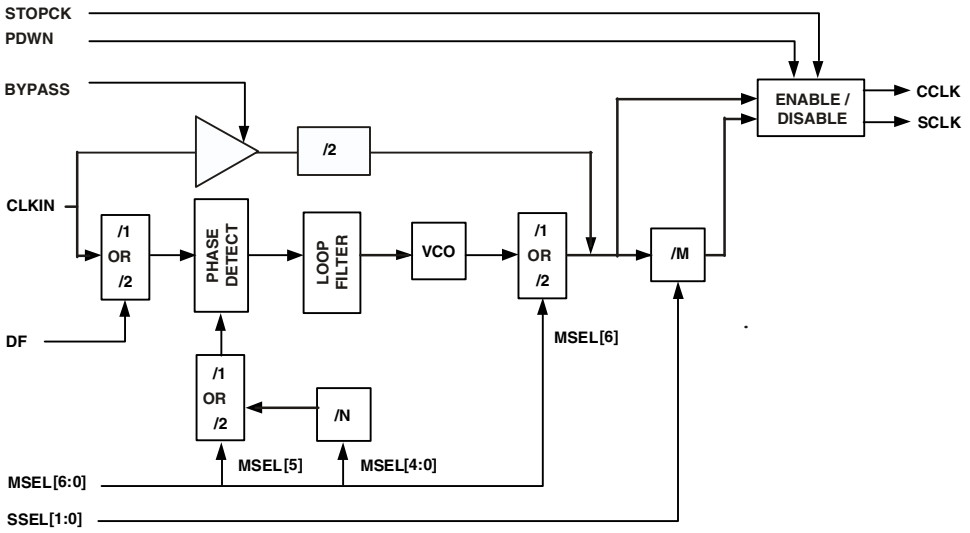


Figure 8-1. PLL Conceptual Block Diagram

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 Control Register (PLL_CTL)" on page 8-7.

The Divide Frequency (DF) bit and Multiplier Select (MSEL[6:0]) field configure the various PLL clock dividers:

- The DF bit enables the input divider
- The MSEL[6] bit enables the output divider
- The MSEL[5:0] bits control the feedback dividers
- The feedback divider is composed in two stages, a divide by *N* (1:32), selected by MSEL[4:0], and a divide by 2, selected by MSEL[5].

Table 8-1 illustrates the CCLK and VCO multiplication factors for the various MSEL and DF settings. Blank entries in the table indicate MSEL/DF combinations that are not supported.

As shown in the table, different combinations of MSEL[6:0] and DF can generate the same CCLK and 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. Note that when MSEL[5] is set to 1, DF must also be set to 1. Whether MSEL[5] and DF are set to 1 or both are set to 0, multiplication factors are the same. See *ADSP-BF535 Blackfin Embedded Processor Data Sheet* for maximum and minimum frequencies for CLKIN, CCLK, and VCO.

Table 8-1. CCLK/VCO Multiplication Factors

MSEL[4:0]	MSEL[6:5]:DF							
	000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
00000	–	16x/16x	–	-	16x/32x	8x/16x	–	16x/32x
00001	1x/1x	–	–	1x/1x	–	–	–	–
00010	2x/2x	1x/1x	–	2x/2x	1x/2x	–	–	1x/2x
00011	3x/3x	–	–	3x/3x	–	–	–	–
00100	4x/4x	2x/2x	–	4x/4x	2x/4x	1x/2x	–	2x/4x
00101	5x/5x	–	–	5x/5x	–	–	–	–
00110	6x/6x	3x/3x	–	6x/6x	3x/6x	–	–	3x/6x
00111	7x/7x	–	–	7x/7x	–	–	–	–
01000	8x/8x	4x/4x	–	8x/8x	4x/8x	2x/4x	–	4x/8x
01001	9x/9x	–	–	9x/9x	–	–	–	–
01010	10x/10x	5x/5x	–	10x/10x	5x/10x	–	–	5x/10x
01011	11x/11x	–	–	11x/11x	–	–	–	–
01100	12x/12x	6x/6x	–	12x/12x	6x/12x	3x/6x	–	6x/12x
01101	13x/13x	–	–	13x/13x	–	–	–	–

Table 8-1. CCLK/VCO Multiplication Factors (Cont'd)

MSEL[4:0]	MSEL[6:5]:DF							
	000	001	010	011	100	101	110	111
01110	14x/14x	7x/7x	–	14x/14x	7x/14x	–	–	7x/14x
01111	15x/15x	–	–	15x/15x	–	–	–	–
10000	16x/16x	8x/8x	–	16x/16x	8x/16x	4x/8x	–	8x/16x
10001	17x/17x	–	–	17x/17x	–	–	–	–
10010	18x/18x	9x/9x	–	18x/18x	9x/18x	–	–	9x/18x
10011	19x/19x	–	–	19x/19x	–	–	–	–
10100	20x/20x	10x/10x	–	20x/20x	10x/20x	5x/10x	–	10x/20x
10101	21x/21x	–	–	21x/21x	–	–	–	–
10110	22x/22x	11x/11x	–	22x/22x	11x/22x	–	–	11x/22x
10111	23x/23x	–	–	23x/23x	–	–	–	–
11000	24x/24x	12x/12x	–	24x/24x	12x/24x	6x/12x	–	12x/24x
11001	25x/25x	–	–	25x/25x	–	–	–	–
11010	26x/26x	13x/13x	–	26x/26x	13x/26x	–	–	13x/26x
11011	27x/27x	–	–	27x/27x	–	–	–	–
11100	28x/28x	14x/14x	–	28x/28x	14x/28x	7x/14x	–	14x/28x
11101	29x/29x	–	–	29x/29x	–	–	–	–
11110	30x/30x	15x/15x	–	30x/30x	15x/30x	–	–	15x/30x
11111	31x/31x	–	–	31x/31x	–	–	–	–

Core Clock/System Clock Ratio Control

The ADSP-BF535 processor divides down the core clock (CCLK) to generate the system clock (SCLK). Table 8-2 shows the allowed divide ratios and various examples of CCLK and SCLK frequencies. Note that the divider ratio must be chosen so that SCLK does not exceed its maximum frequency. See *ADSP-BF535 Blackfin Embedded Processor Data Sheet* for more information about the maximum frequency of SCLK.

The System Select field (SSEL[1:0]) of the PLL Control Register (PLL_CTL) controls the ratio of CCLK to SCLK. Upon reset, the input pins SSEL[1:0] are sampled. (These pins are shared with flag input pins PF[9:8].)

The values on SSEL[1:0] determine the initial system clock ratio.

To change the CCLK-SCLK divider ratio, the SSEL field in the PLL_CTL register should be modified with the new value. This field can be changed at any time, but changes to the divider ratio do not take effect until the PLL programming sequence is executed. See "PLL Programming Sequence" on page 8-17.

The user should ensure that peripheral operation does not suffer adverse effects when the system clock ratio is changed.

Table 8-2 shows the system clock ratio, or frequency ratio of SCLK relative to CCLK.

Table 8-2. System Clock Ratio

Signal Name	Divider Ratio	Example Frequency Ratios (MHz)	
SSEL[1:0]	CCLK/SCLK	CCLK	SCLK
00	2:1	266	133
01	2.5:1	275	110
10	3:1	300	100
11	4:1	300	75

PLL Memory-Mapped Registers (MMRs)

The user interface to the PLL is through three MMRs:

- The PLL Control register (PLL_CTL)
- The PLL Status register (PLL_STAT)
- The PLL Lock Count register (PLL_LOCKCNT)

The PLL_CTL register is a 32-bit MMR and must be accessed with aligned 32-bit reads/writes. The PLL_STAT and PLL_LOCKCNT registers are 16-bit MMRs and must be accessed with aligned 16-bit reads/writes.

PLL Control Register (PLL_CTL)

The PLL Control register (PLL_CTL), shown in Figure 8-2, controls operation of the PLL. Note that 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 8-17.

These fields of the PLL_CTL register are used to control the PLL:

- SSEL[1:0] – The SCLK Select (SSEL) field defines the core clock (CCLK) to system clock (SCLK) divider ratio. Upon reset, the value for this field is sensed from the SSEL pins.
- MSEL[6:0] – The Multiplier Select (MSEL) field defines the input clock (CLKIN) to core clock (CCLK) multiplier. Upon reset, the value for this field is sensed from the MSEL pins.
- BYPASS – This bit is used to bypass the PLL. When the PLL is bypassed, CCLK runs at half the frequency of CLKIN. Upon reset, the value for this field is sensed from the BYPASS pin.

- PDWN – The Power Down (PDWN) bit is used to place the ADSP-BF535 in the Deep Sleep operating mode. For more information on operating modes, see "Operating Modes" on page 8-12.
- STOPCK – The Stop Clock (STOPCK) bit is used to enable/disable 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. Upon reset, the value for this field is sensed from the DF pin.

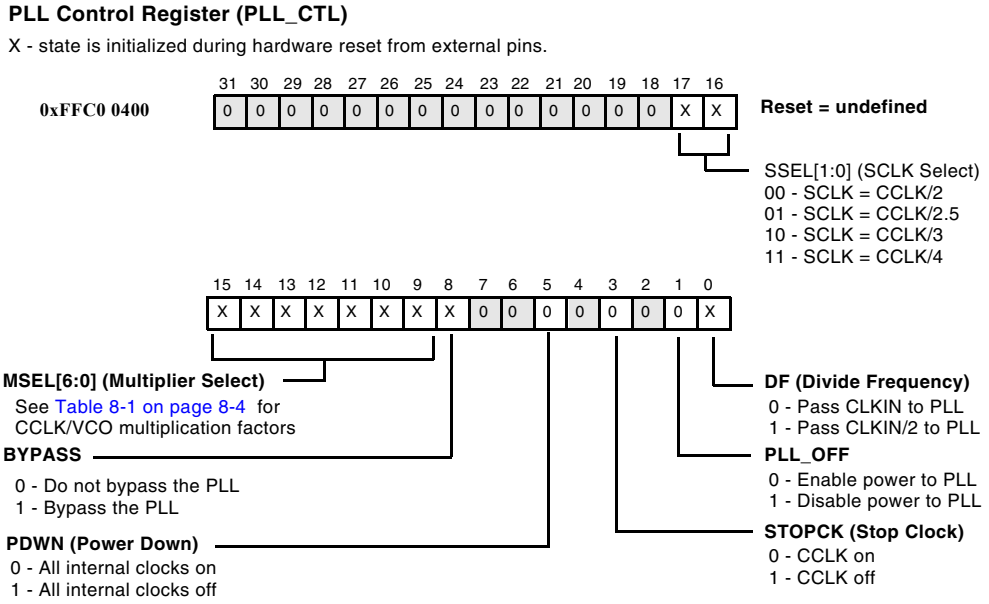


Figure 8-2. The PLL Control Register

Note some fields of the PLL_CTL register cannot be updated with a new value simultaneously. Specifically, the MSEL and DF fields cannot be updated at the same time as the BYPASS field. Should MSEL

and/or DF and BYPASS need to be modified, the PLL_CTL register should be written twice—once with the MSEL and/or DF field changes, then a second time with the BYPASS field change.

PLL Status Register (PLL_STAT)

The PLL Status register (PLL_STAT), shown in Figure 8-3, indicates the operating mode of the PLL and ADSP-BF535 processor. For more information about operating modes, see "Operating Modes" on page 8-12.

PLL Status Register (PLL_STAT)

Read only. 1 - processor operating in this mode. For more information, see "Operating Modes" on page 8-12.

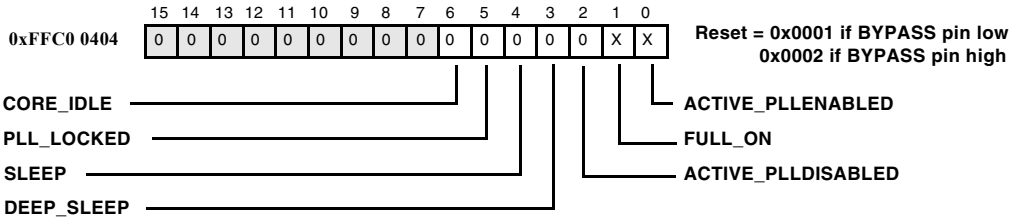


Figure 8-3. PLL Status Register

These fields are used in the PLL_STAT register:

- CORE_IDLE – This field is set to 1 when the Blackfin processor core is idled; that is, an IDLE instruction has executed, and the core awaits a wake-up 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 Lock Count Register (PLL_LOCKCNT)" on page 8-10.
- SLEEP – This field is set to 1 when the ADSP-BF535 processor is in the Sleep operating mode.

- DEEP_SLEEP – This field is set to 1 when the ADSP-BF535 processor is in the Deep Sleep operating mode.
- ACTIVE_PLLDISABLED – This field is set to 1 when the ADSP-BF535 processor is in the Active operating mode with the PLL powered down.
- FULL_ON – This field is set to 1 when the ADSP-BF535 processor is in the Full On operating mode.
- ACTIVE_PLLENABLED – This field is set to 1 when the ADSP-BF535 processor is in the Active operating mode with the PLL powered up.

PLL Lock Count Register (PLL_LOCKCNT)

When changing clock frequencies in the PLL, the PLL requires time to stabilize and lock to the new frequencies.

The PLL Lock Count register (PLL_LOCKCNT), shown in Figure 8-4, defines the number of SCLK cycles that will 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 SCLK cycle. When the lock counter has incremented to the value defined in the PLL_LOCKCNT register, the PLL_LOCKED bit is set.

See *ADSP-BF535 Blackfin Embedded Processor Data Sheet* for more information about PLL stabilization time and values that should be programmed into this register. For more information about operating modes, see "Operating Modes" on page 8-12. For more about the PLL programming sequence, see "PLL Programming Sequence" on page 8-17.

PLL Lock Count Register (PLL_LOCKCNT)

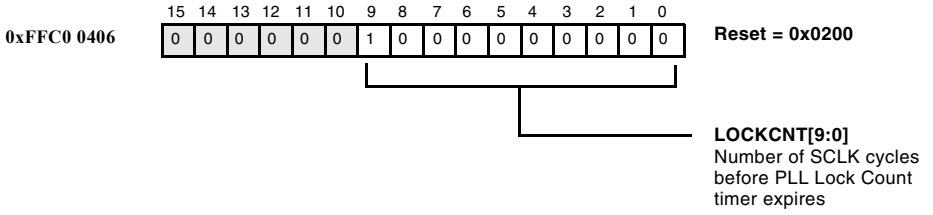


Figure 8-4. PLL Lock Count Register

Dynamic Power Management Controller

The Dynamic Power Management Controller (DPMC) of the ADSP-BF535 processor 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 ADSP-BF535 processor operates in four different operating modes, each with different performance characteristics and power dissipation profiles.
- Peripheral clocks – The user controls which peripherals are clocked and which are not, saving power when a peripheral is idle or not used.
- Voltage control – The ADSP-BF535 processor has five internal power domains, allowing an external power management controller to shut down a power domain if it is not required. Additionally, logic is provided to allow an external power management controller to manipulate the Blackfin processor core's internal voltage, further reducing power.

Operating Modes

The ADSP-BF535 processor operates in four operating modes, each with unique performance and power saving benefits. Table 8-3 summarizes the operational characteristics of each mode.

Table 8-3. Operational Characteristics

Operating Mode	Power Savings	PLL Status	PLL Bypassed	CCLK	SCLK	Allowed DMA Access
Full On	None	Enabled	No	Enabled	Enabled	L1, L2
Active	Medium	Enabled [1]	Yes	Enabled	Enabled	L1, L2
Sleep	High	Enabled	[2]	Disabled	Enabled	L2
Deep Sleep	Maximum	Disabled	–	Disabled	Disabled	–

1 PLL can also be disabled in this mode.
2 Depends on the previous state.

Full On Mode

Full On is the maximum performance mode of the ADSP-BF535 processor. In this mode, the PLL is enabled and not bypassed. This is the normal execution state of the ADSP-BF535 processor, with the processor and all enabled peripherals running at full speed. In this mode, the input clock (CLKIN) to core clock (CCLK) multiplier ratio cannot be changed. DMA access is available to both L1 and L2 memories. From the Full On mode, the processor can transition directly to the Active, Sleep or Deep Sleep modes.

Active Mode

In the Active mode, the PLL is enabled but bypassed. Because the PLL is bypassed, the processor's core clock (CCLK) runs at one-half the input clock (CLKIN) frequency, offering significant power savings. The system clock (SCLK) frequency is also reduced, because it is derived from

CCLK (SCLK = CCLK/(SSEL divider)); this reduction further reduces power. In this mode, the CLKIN to CCLK multiplier ratio can be changed, although the changes are not realized until the Full On mode is entered. DMA access is available to both L1 and L2 memories.

In the 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 the Full On or Sleep modes.

From the Active mode, the processor can transition directly to the Full On, Sleep or Deep Sleep modes.

Sleep Mode

The 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 same frequency as before transitioning to the Sleep mode. If the processor transitions from the Active mode to the Sleep mode, SCLK continues to run at the Active mode SCLK frequency. If the processor transitions from the Full On mode to the Sleep mode, SCLK continues to run at the Full On SCLK frequency. As CCLK is disabled, DMA access is available only to L2 memory in the Sleep mode. From the Sleep mode, a wake-up 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

The processor resumes execution from the program counter value present immediately prior to entering sleep mode.

The STOPCK bit is not a status bit and is therefore unmodified by hardware when the wakeup occurs. Software must explicitly clear STOPCK in the next write to PLL_CTL to avoid going back into sleep mode.

Deep Sleep Mode

The 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.

In the Deep Sleep mode, the DEEP_SLEEP output pin is asserted. The Deep Sleep mode can only be exited by an RTC interrupt or hardware reset event. An RTC interrupt causes the processor to transition to the Active mode; a hardware reset begins the hardware reset sequence. For more information about hardware reset, see "Hardware Reset" on page 3-13.

Note that an RTC interrupt in the Deep Sleep mode automatically resets some fields of the PLL Control register (PLL_CTL), as shown in Table 8-4.

When in the Deep Sleep operating mode, clocking to the SDRAM is turned off. Software should ensure that important information in SDRAM is saved to a non-volatile memory before entering the Deep Sleep mode.

Table 8-4. PLL Control Register Values after RTC Wake-up Interrupt

Field	Value
PLL_OFF	0
STOPCK	0
PDWN	0
BYPASS	1

Operating Mode Transitions

Figure 8-5 graphically illustrates the operating modes and allowed transitions among them. In the diagram, an ellipse represents an operating mode. Thin 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 the 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 8-17.

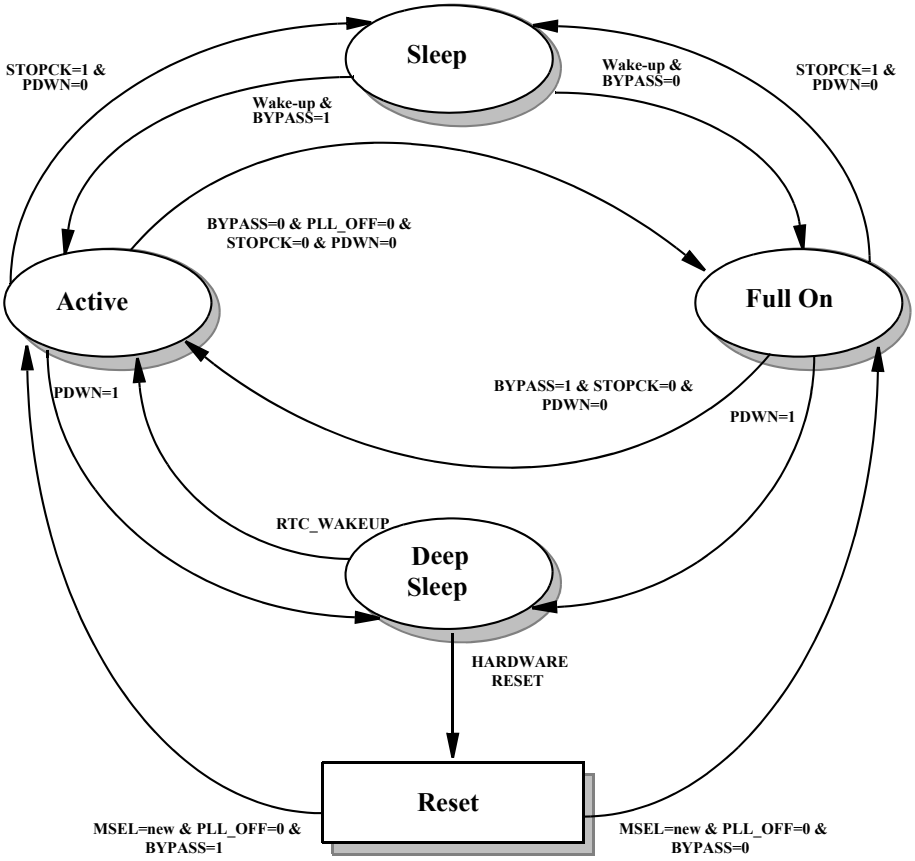


Figure 8-5. Operating Mode Transitions

In addition to the mode transitions shown in Figure 8-5, the PLL can be modified while in the Active operating mode. Power to the PLL can be applied and removed, and new clockin (CLKIN) to core clock (CCLK) 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 8-17:

- 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 the 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: New clockin (CLKIN) to core clock (CCLK) multiplier ratios can be programmed while in the Active mode. Although the CLKIN to CCLK multiplier changes are not realized in the Active mode, forcing the PLL to lock to the new ratio in the Active mode before transitioning to Full On reduces the transition time, because the PLL is already locked to the new ratio. Note that the PLL must be powered up to lock to the new ratio. To program a new CLKIN to CCLK multiplier, write the new MSEL[6:0] and/or DF values to the PLL_CTL register; then execute the PLL programming sequence.

Table 8-5 summarizes the allowed operating mode transitions.

Attempting to cause mode transitions other than those shown in Table 8-5 causes unpredictable behavior.

Table 8-5. Allowed Operating Mode Transitions

New Modes	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 of the ADSP-BF535 processor is defined by the state of the PLL_OFF, BYPASS, STOPCK and PDWN bits of the PLL Control register (PLL_CTL). Simply 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 below. This code sequence first brings the ADSP-BF535 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 ADSP-BF535 processor operates with the new settings, including the new operating mode, if one is programmed.

PLL Programming Sequence

Assuming that the PLL Control register (PLL_CTL) has been modified with the new values, the instruction sequence shown in Listing 8-1 puts those changes into effect.

Listing 8-1. PLL Programming Sequence

```
CLI Rn;   /* disable interrupts, copy IMASK to Rn */
IDLE;   /* source NOPs into pipeline, and enter idled state upon
SSYNC */
SSYNC;   /* drain pipeline, enter idled state */
STI Rn;   /* re-enable interrupts after wakeup, restore IMASK
from Rn */
```

The first three instructions in the sequence (CLI, IDLE, and SSYNC) take the core to an idled state with interrupts disabled; the interrupt mask (IMASK) is saved to the Rn 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 CCLK multiplier or the changes reapply power to the PLL, the PLL needs to relock. To relock, the PLL lock counter is first cleared, then 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).

Depending on how the PLL_CTL register is programmed, the processor proceeds in one of four ways:

- If the PLL_CTL register is programmed to enter either the Active or Full On operating mode, the processor waits for a wake-up signal, then continues with the STI instruction in the sequence, as described in "PLL Programming Sequence Continues" on page 8-19.
- The wake-up signal is an interrupt generated by a peripheral, watchdog or other timer, RTC, or other source. For more information about events that cause the processor to wake-up from being idled, see "System Interrupt Wakeup-Enable Register (SIC_IWR)" on page 4-24.

- 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 wake-up signal before continuing. When the wake-up signal has been asserted, the instruction sequence continues with the STI instruction, as described in the section, "PLL Programming Sequence Continues" on page 8-19 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 the Deep Sleep operating mode, the processor immediately transitions to the Deep Sleep mode and waits for a Real-Time Clock (RTC) interrupt or hardware reset signal:
  - – An RTC interrupt causes the processor to enter the 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 processor waits for a wake-up signal to continue with the STI instruction in the sequence, as described in the section below.

PLL Programming Sequence Continues

The instruction sequence shown in Listing 8-1 on page 8-18 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. These examples use the watchdog timer as the wake-up signal. Some setup code has been removed for clarity, and these assumptions are made:

- P0 points to the PLL Control register (PLL_CTL).
- The watchdog timer has been initialized and enabled as a wake-up signal.
- MSEL[6:0] and DF in PLL_CTL are set to (b#0011111) and (b#0) respectively, signifying a CLKIN to CCLK multiplier of 31x.
- Core clock (CCLK) to system clock (SCLK) divider ratio (SSEL) is set at 2 to 1 (b#00).

Listing 8-2. Transitioning From Active Mode to Full On Mode

```
R1.H = 0x0000;   /* clear BYPASS bit in PLL_CTL */
R1.L = 0x3E00;
[P0] = R1;
SSYNC;
CLI R1;   /* disable interrupts, copy IMASK to R1 */
IDLE;   /* source NOPs into pipeline, prepare to enter idled
state */
SSYNC;   /* drain pipeline, enter idled state, wait for watchdog
*/
STI R1;   /* after watchdog occurs, restore interrupts and IMASK
*/
...   /* processor is now in the Full-On mode */
```

Listing 8-3. Transitioning from Full On Mode to Active Mode

```
R1.H = 0x0000;   /* set BYPASS bit in PLL_CTL */
R1.L = 0x3F00;
[P0] = R1;
SSYNC;
CLI R1;   /* disable interrupts, copy IMASK to R1 */
IDLE;   /* source NOPs into pipeline, prepare to enter idled
state */
SSYNC;   /* drain pipeline, enter idled state, wait for watchdog
*/
STI R1;   /* after watchdog occurs, restore interrupts and IMASK
*/
...   /* processor is now in the Active mode */
```

Listing 8-4. Changing CLKIN to CCLK Multiplier From 31x to 2x in Full On Mode

```
R1.H = 0x0000;   /* set BYPASS bit in PLL_CTL without modifying
MSEL because we must be in BYPASS to change MSEL */
R1.L = 0x3F00;
[P0] = R1;
SSYNC;
CLI R1;   /* disable interrupts, copy IMASK to R1 */
IDLE;   /* source NOPs into pipeline, prepare to enter idled
state */
SSYNC;   /* drain pipeline, enter idled state, wait for watchdog
*/
STI R1;   /* after watchdog occurs, restore interrupts and IMASK
*/
/* processor is now in the Active mode */
R1.H = 0x0000;   /* set CLKIN to CCLK multiplier to 2x in PLL_CTL,
keeping BYPASS bit set as both MSEL and BYPASS cannot be changed
simultaneously */
```

```
R1.L = 0x0500;
[P0] = R1;
SSYNC;
R1.H = 0x0000;   /* clear BYPASS bit in PLL_CTL, satisfying the
requirement of breaking up simultaneous changes to MSEL and
BYPASS */
R1.L = 0x0400;
[P0] = R1;
SSYNC;
CLI R1;   /* disable interrupts, copy IMASK to r1 */
IDLE;   /* source NOPs into pipeline, prepare to enter idled
state */
SSYNC;   /* drain pipeline, enter idled state, wait for watchdog
*/
STI R1;   /* after watchdog occurs, restore interrupts and IMASK
*/
...   /* processor is now in the Full-On mode with the CLKIN to
CCLK multiplier set to 2x */
```

Peripheral Clocking

To further reduce power dissipation, the ADSP-BF535 processor allows software to control the clocking of many peripherals. If a peripheral is not currently needed, clocking to the peripheral can be disabled, resulting in lower power dissipation. Clocking to a peripheral can be re-enabled later when use of the peripheral is required.

Peripheral Clock Enable Register (PLL_IOCK)

As shown in Figure 8-6, the Peripheral Clock Enable register (PLL_IOCK) is a 16-bit MMR that controls clocking to the peripherals. Each peripheral whose clocking can be controlled is represented by a bit in the register. A 1

in the bit enables clocking to that peripheral; a 0 in the bit turns off clocking to that peripheral. This register can be altered at any time in any operating mode.

After a reset, the `PLL_IOCK` register defaults to enable clocking for all peripherals.

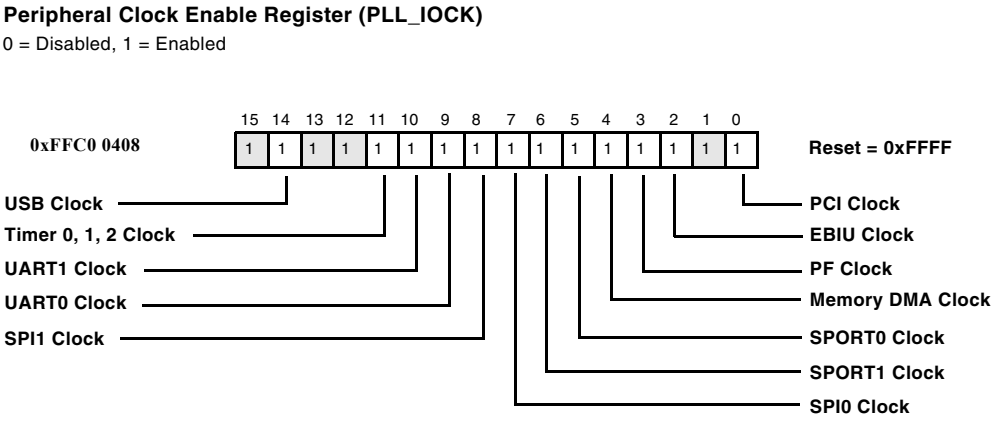


Figure 8-6. Peripheral Clock Enable Register

Dynamic Supply Voltage Control

In addition to clock frequency control, the ADSP-BF535 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 ADSP-BF535 processor uses five power domains. These power domains are shown in Table 8-6. 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 *ADSP-BF535 Blackfin Embedded Processor Data Sheet* for details on the allowed voltage ranges for each power domain and power dissipation data.

Table 8-6. ADSP-BF535 Power Domains

Power Domain	V_{DD} Range
Analog PLL internal logic	Fixed
All internal logic except PLL, RTC	Variable
RTC internal logic, and crystal I/O	Fixed
PCI I/O	Fixed
All other I/O	Variable

PCI Power Savings

If the PCI controller of the ADSP-BF535 processor is not needed for a particular application, it does not need to be clocked. For information about how to disable clocking to the PCI controller, see "Peripheral Clocking" on page 8-22. Refer to *ADSP-BF535 Blackfin Embedded Processor Data Sheet* for the procedure for handling unused pins.

Changing Voltage

Minor changes in operating voltage can be accommodated without requiring special consideration or action by the application program. See *ADSP-BF535 Blackfin Embedded Processor Data Sheet* for more information about voltage tolerances and allowed rates of change.

Reducing the ADSP-BF535 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 ADSP-BF535 processor should be brought to a known and stable state before the operating voltage is modified.

The recommended procedure is to bring the processor to the Sleep operating mode before substantially varying the voltage. For more information about the Sleep operating mode, see "Sleep Mode" on page 8-13. This procedure ensures the integrity and deterministic behavior of the processor.

As soon as the processor is placed in the Sleep operating mode, the operating voltage can be safely modified. 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 *ADSP-BF535 Blackfin Embedded Processor Data Sheet* for the new operating voltage level.

External Voltage Regulator Example

A programmable voltage regulator external to the ADSP-BF535 processor can be used to modify the operating voltage of the processor dynamically. A simple handshaking mechanism allows the ADSP-BF535 processor and the voltage regulator to alter the operating voltage dynamically.

The following examples use a simple signaling mechanism between the ADSP-BF535 processor and the external voltage regulator to toggle between a high performance and power saving mode of operation. The Programmable Flags peripheral of the ADSP-BF535 processor provides a convenient means for such a mechanism. For more information about operating modes, see "Operating Modes" on page 8-12. For more information about the programmable flag peripheral, see "Programmable Flags" on page 15-1.

Power Saving Sequence

In the following example, the Full On operating mode represents the high performance mode of operation, and the Active operating mode represents the power saving mode.

The sequence for toggling between the high performance mode and power saving mode works as follows:

- The application software, running on the ADSP-BF535 processor in the Full On operating mode, determines that the ADSP-BF535 processor should switch to the power saving mode of operation.
- The ADSP-BF535 processor programs the PLL to transition to the Active operating mode after transitioning through the Sleep mode.
- The processor modifies the PLL Control register (`PLL_CTL`), setting `BYPASS` to 1, `STOPCK` to 1 and `PDWN` to 0. Note that the transition does not occur until the PLL programming sequence is executed.
- The ADSP-BF535 processor signals the voltage regulator, via a programmable flag, to lower the operating voltage, then immediately executes the PLL programming sequence to begin the transition to the Active operating mode.
- As described in the section, "PLL Programming Sequence" on page 8-17, the ADSP-BF535 processor waits in the Sleep mode for the wake-up signal before completing the transition to the Active operating mode.
- While the ADSP-BF535 processor is waiting, the voltage regulator lowers the voltage.
- After the voltage has stabilized, the voltage regulator signals the ADSP-BF535 processor, via a programmable flag, that the voltage has been lowered.
- This signal wakes up the ADSP-BF535 processor, allowing it to complete the transition to the Active operating mode.

High Performance Sequence

The same sequence used to toggle to power saving mode can be used to return the ADSP-BF535 into a high performance mode.

- The application software, running on the ADSP-BF535 processor in the Active operating mode, determines that the ADSP-BF535 processor should switch to the high performance mode of operation.
- The ADSP-BF535 processor programs the PLL to transition to the Full On operating mode after transitioning through the Sleep mode.
- The processor modifies PLL_CTL, setting BYPASS to 0, STOPCK to 1 and PDWN to 0. Again, the transition does not occur until the PLL programming sequence is executed.
- The ADSP-BF535 processor signals the voltage regulator, via a programmable flag, to raise the operating voltage, then immediately executes the PLL programming sequence to begin the transition to the Full On operating mode.
- The ADSP-BF535 processor again waits in the Sleep mode for the wake-up signal before completing the transition to the Full On operating mode.
- While the ADSP-BF535 processor is waiting, the voltage regulator raises the voltage.
- When the voltage has stabilized, the voltage regulator signals the ADSP-BF535 processor, via a programmable flag, that the voltage has been raised.
- The signal from the voltage regulator wakes up the ADSP-BF535 processor, allowing it to complete the transition to the Full On operating mode.

9 DIRECT MEMORY ACCESS

The ADSP-BF535 processor uses Direct Memory Access (DMA) to transfer data within memory spaces or between a memory space and a peripheral. The fully integrated DMA controller allows the ADSP-BF535 processor or an external device to specify data transfer operations and return to normal processing while the DMA controller carries out the data transfers independent from processor activity.

The DMA controller can perform several types of data transfers:

- Memory ↔ Memory (MemDMA) (see "Memory DMA (MemDMA)" on page 9-31)
- Memory ↔ Serial Peripheral Interface (SPI) Port (Chapter 10)
- Memory ↔ Serial Port (Chapter 11)
- Memory ↔ UART Port (Chapter 12)
- Memory ↔ USB Device (Chapter 14)

The ADSP-BF535 processor system includes eight DMA capable peripherals, including the Memory DMA controller (MemDMA). Twelve DMA channels and bus masters support these devices:

- SPORT0 RCV DMA Controller
- SPORT1 RCV DMA Controller
- SPORT0 XMT DMA Controller
- SPORT1 XMT DMA Controller

- USB DMA Controller
- SPI0 DMA Controller
- SPI1 DMA Controller
- UART0 RCV Controller
- UART1 RCV Controller
- UART0 XMT Controller
- UART1 XMT Controller
- Memory DMA Controller

This chapter describes the features common to all the DMA channels and 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 Performance" on page 7-11.

DMA transfers on the ADSP-BF535 processor can be descriptor based or autobuffer based. Descriptor based DMA transfers require a set of parameters stored within memory to initiate a DMA sequence. This sort of transfer allows chaining multiple DMA sequences together. In descriptor based DMA operations, a DMA channel can be programmed to set up and start another DMA transfer automatically after the current sequence completes. Autobuffer based DMA allows the processor to program DMA control registers directly to initiate a DMA transfer. Upon completion, the control registers are automatically updated with their original setup values.

Descriptor Based DMA

Descriptor based DMA is the most common method of controlling DMA transfers on the ADSP-BF535 processor. To use this method, the DMA channel requires a set of parameters, called a *DMA descriptor block*, stored within memory. Each descriptor block contains all the information needed for a particular DMA transfer sequence and consists of:

- The 32-bit starting address of the data block to be transferred
- The number of data transfers
- Other miscellaneous control information—for example, the configuration for what the DMA channel does when the transfer is complete
- A pointer to the next descriptor block

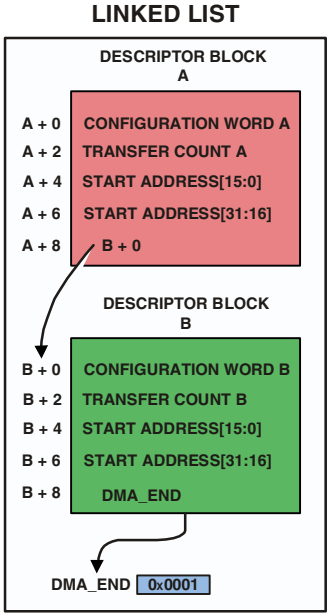


Figure 9-1. Linked List Containing Two Descriptor Blocks

DMA Descriptor Block Structure

The elements of a DMA descriptor block are defined in Table 9-1, where *BASE* refers to the starting address of the descriptor block.

The processor can place multiple sets of descriptor blocks, representing multiple transfers, in memory. A set of descriptor blocks is called a *linked list* (see Figure 9-1). When a linked list has been generated, the DMA channel has all the information needed to perform multiple transfer sequences without processor intervention.

Table 9-1. DMA Descriptor Block Parameters

Address	Location Name	Description
BASE+0	DMA Configuration Word (see Table 9-2 for bit definitions)	Descriptor ownership, DMA configuration, and completion status
BASE+2	DMA Transfer Count	Number of elements (8-bit bytes, 16-bit half words, or 32-bit words) to be transferred
BASE+4	DMA Start Address[15:0]	16 LSBs of the transfer start address
BASE+6	DMA Start Address[31:16]	16 MSBs of the transfer start address
BASE+8	Next Descriptor Pointer[15:0]	16 LSBs of the next descriptor block head address [1]

1 16 MSBs of the next descriptor block base address are determined by the global DMA Descriptor Base Pointer register (DMA_DBP), which is shared across all ADSP-BF535 DMA channels.

To end the sequence of transfers, the Next Descriptor Pointer must point to a memory location containing a 16-bit data value containing a 0 in bit 15 (0xxx xxxx xxxx xxxx). If bit 0 is a 1 (0xxx xxxx xxxx xxx1), the DMA channel is still enabled, but stalled, and the remaining FIFO values are not discarded.

Note the order of the descriptor block elements differs from that of the DMA register set. This order allows the 32-bit address pointers (*BASE*+4 and *BASE*+6) to be 32-bit aligned in little endian memory. The user can configure the DMA Start Address[31:0] as one 32-bit load into the descriptor block rather than as two 16-bit loads. For this reason, it is recommended DMA descriptor blocks be 32-bit aligned within memory. At a minimum, descriptor blocks must be 16-bit aligned.

DMA Configuration Word

Table 9-2 describes the bits in the DMA Configuration Word located in address *BASE*+0 of the descriptor block within memory.

Table 9-2. DMA Configuration Word

Bit	Name	Values
0	DMA Enable	0 - Disabled 1 - Enabled
1	Direction	Peripheral dependent definition of values
2	Interrupt on Completion Enable (IOC)	0 - Interrupt disabled 1 - Interrupt enabled
3	Data Size	Peripheral dependent definition of values (for data size, see Table 9-3)
4	Autobuffer	Peripheral dependent definition of values
5-6	Control State	Peripheral dependent definition of values
7	Buffer Clear	Must be set to 0 in Configuration Word
8	Interrupt on Error Enable	Peripheral dependent definition of values
9-11	Status	Peripheral dependent definition of values
12-13	Buffer Status / Data Size[1]	Peripheral dependent definition of values
14	DMA Completion Status (DCS)	0 - Successful completion 1 - Error
15	Descriptor Block Ownership (DBO)	0 - Processor has ownership 1 - DMA channel has ownership

1 Bit 12 of the DMA Configuration Word is read as a FIFO Status bit and written as a data size bit in some peripherals (see Table 9-3).

Each DMA channel has peripheral dependent functionality. The following peripheral dependent behavior is controlled or monitored through the DMA Configuration Word:

- Data transfer direction (bit 1)
- The peripheral dependent direction bit is dedicated as either read or write. For some peripherals and MemDMA, this bit cannot be modified.
- Data size (bits 3 and 12)
- The peripheral dependent data size can be 8, 16, or 32 bits. Some peripherals support only 16- and 32-bit data sizes. With each transfer, the DMA address (stored in the DMA Start Address registers) increments relative to the data size. The address increments by 1 for 8-bit data transfers, by 2 for 16-bit transfers, or by 4 for 32-bit transfers.

Table 9-3 shows how to configure the DMA Configuration Word for each valid data size.

Table 9-3. DMA Configuration Word Data Size Settings

Bit 12 [1]	Bit 3	Data Size
0	0	16-bit half word
0	1	32-bit word
1	0	Reserved
1	1	8-bit byte

1 Bit 12 of the DMA Configuration Word is read as a FIFO Status bit and written as a data size bit.

The DMA controller does not support data packing. For example, in an 8-bit device, if the data size is set up for 32 bits, the upper 3 bytes of the 32-bit transfer are zero-filled.

- Peripheral control (bits 5 and 6)
- Peripheral dependent control bits provide peripheral control through descriptor blocks. For more information about the use of these bits in a specific peripheral, see the appropriate peripheral chapter.
- Peripheral status (bits 9 – 13)
- The peripheral dependent status bits contain peripheral specific information about the DMA transfer for the current descriptor block, including buffer status. At the completion of the DMA transfer, this information is written back to the DMA Configuration Word of the current DMA descriptor block. For more information about the use of these bits in a specific peripheral, see the appropriate peripheral chapter.

Setting Up Descriptor Based DMA

The following steps illustrate the typical sequence for setting up descriptor based DMA.

Before these steps, the DMA Descriptor Base Pointer register (`DMA_DBP`) must be configured with the upper 16 bits (MSBs) of the descriptor block base address, *BASE*+0.

1. Write the DMA Configuration Word (with bit 15 set to 1), DMA Transfer Count, DMA Start Address[15:0], DMA Start Address[31:16], and Next Descriptor Pointer[15:0] to the descriptor block memory addresses *BASE*+0 through *BASE*+8.

- If the descriptor block is the last one in a linked list or the only one (that is, a standalone transfer sequence), the Next Descriptor Pointer within the descriptor block must point to a memory location that contains a data value with bit 15 set to 0 (0xxx xxxx xxxx xxxx). If bit 0 is a 1 (0xxx xxxx xxxx xxx1), the DMA channel is still enabled, but stalled, and the remaining FIFO values are not discarded. This allows the DMA engine to finish transferring the remaining values in the FIFO.
- Because of this requirement, the Next Descriptor Pointer can point to the base address (*BASE*+0) of the current DMA descriptor block. (After the DMA transfer sequence, bit 15 of the DMA Configuration Word is cleared, returning ownership to the processor. If ownership returns to the processor, the DMA channel is still enabled but stalls after the DMA transfer.)

2. Write the lower 16 bits (LSBs) of the descriptor block base address (the first descriptor block if in a linked list), *BASE*+0, to the Next Descriptor Pointer register of the appropriate peripheral.
3. Set the DMA Enable bit in the appropriate peripheral's DMA Configuration register.
    - This final write is needed only if this DMA descriptor block is for a standalone transfer sequence or the first one in a linked list.

Descriptor-Based DMA Operation

Upon detecting the assertion of the DMA Enable bit in the peripheral's DMA Configuration register, the DMA channel fetches the first element from the descriptor block, the DMA Configuration Word, and copies it to

the DMA Configuration register. Bit 15, the DBO bit, of this register is checked to determine whether the descriptor block is configured and ready for use:

- If bit 15 is not set, the DMA channel stalls and waits until a processor write occurs to the peripheral's DMA Descriptor Ready register (see "Peripheral DMA Descriptor Ready Register" on page 9-25).
- This write triggers the DMA controller to recopy the DMA Configuration Word from the current descriptor block into the DMA Configuration register and to recheck bit 15.
- If bit 15 is set, the DMA channel fetches the remaining four elements (*BASE*+2, *BASE*+4, *BASE*+6, *BASE*+8) from the descriptor block, loads them to the respective DMA control registers, and begins DMA transfers.
- The status of the transfer sequence is updated every cycle in the appropriate peripheral's DMA status registers.

The DMA transfer sequence is completed when the count value in the DMA Transfer Count register has decremented to zero. Upon completion of the transfer sequence, the DMA channel:

- Clears the DBO bit of the DMA Configuration register, returning ownership to the processor
- Copies the contents of the DMA Configuration register to the DMA Configuration Word of the current descriptor block.
- This write-back includes the final status of the transfer sequence.
- Generates an interrupt if interrupts are enabled (see the appropriate peripheral chapter for more information)
- Fetches the Configuration Word of the next descriptor block
- The process continues.

The flow diagram in Figure 9-2 and Figure 9-3 shows how the DMA channel performs two consecutive DMA transfers using descriptor blocks. (The structure of the descriptor blocks is shown in Figure 9-1 on page 9-4.) Figure 9-4 on page 9-14 illustrates the time segments from the flow diagram (Time T1 through Time T5).

In Figure 9-4, the actions in T1 and T2 are the same as those in T3 and T4 except for the name of the descriptor block.

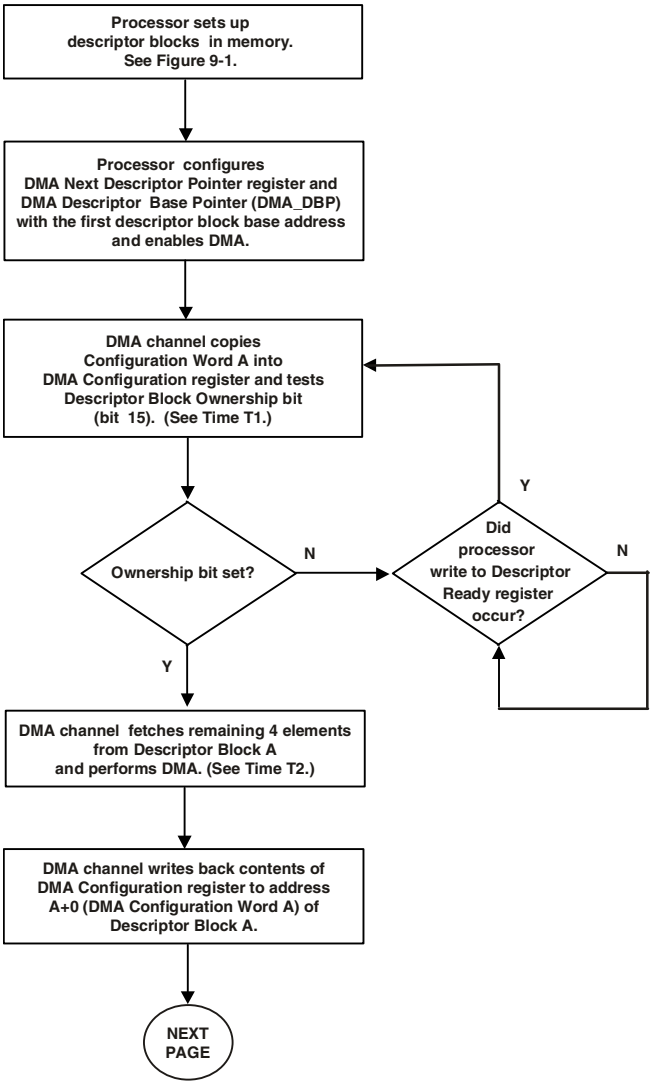


Figure 9-2. Consecutive DMA Sequences With Descriptor Blocks (1 of 2)

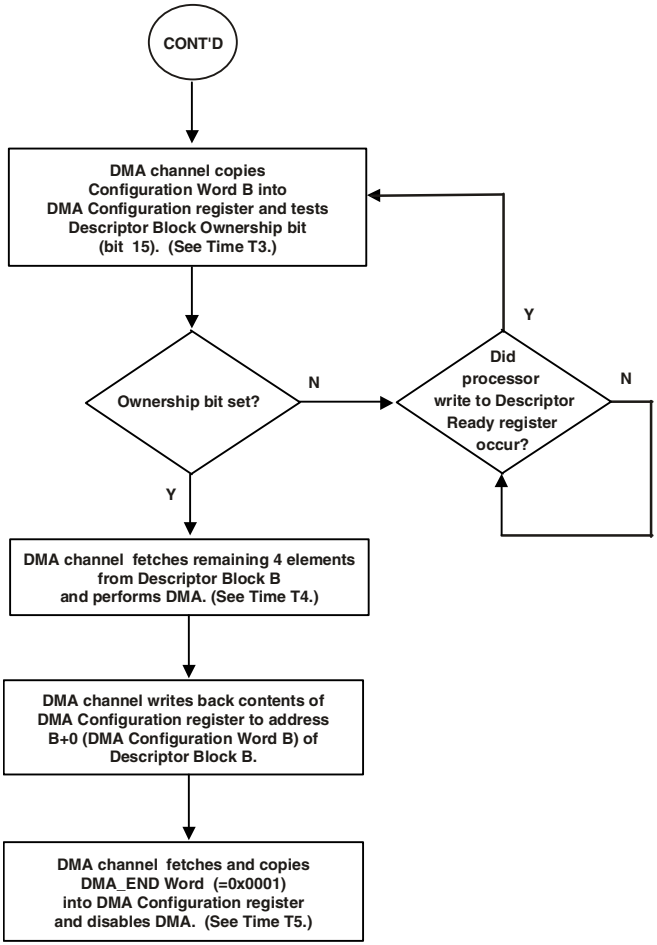


Figure 9-3. Consecutive DMA Sequences With Descriptor Blocks (2 of 2)

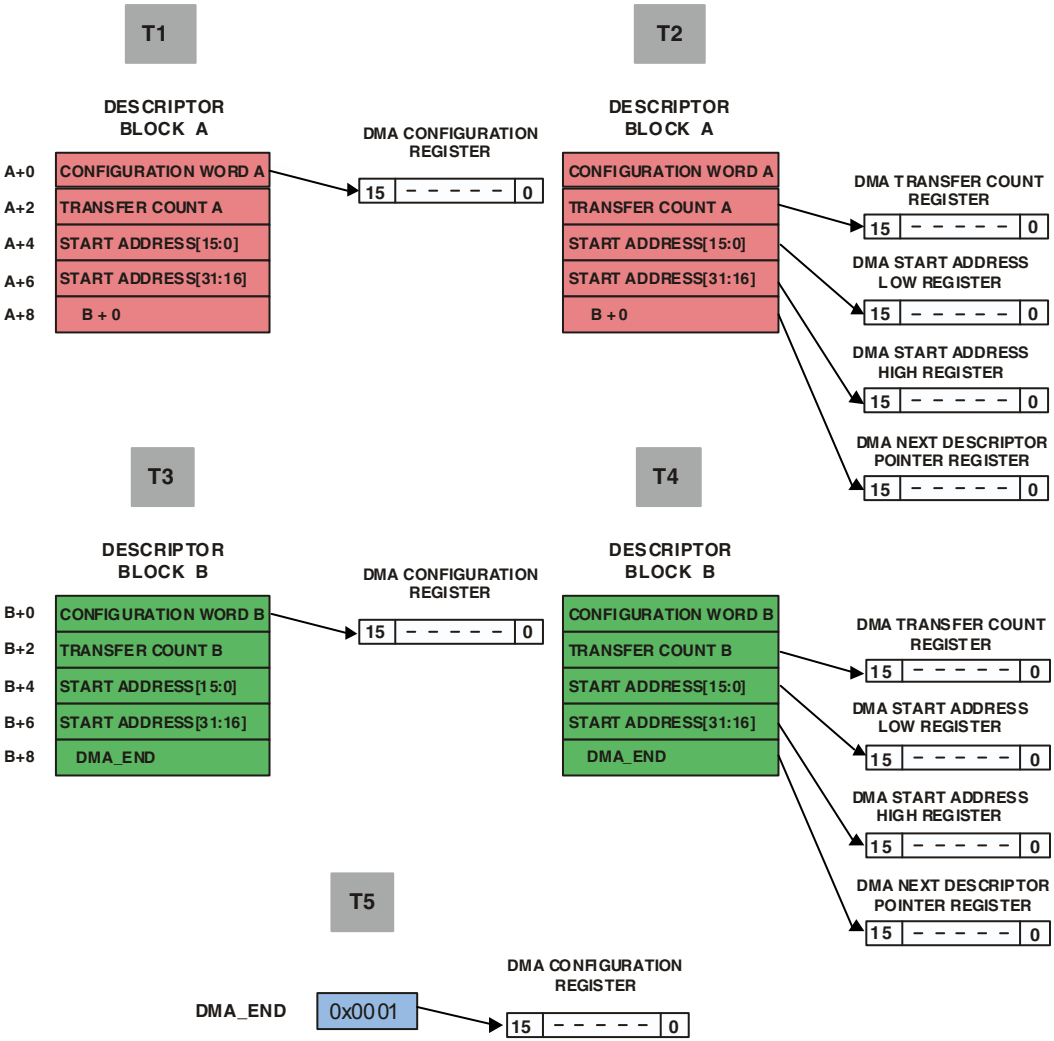


Figure 9-4. Time T1 Through T5 of Consecutive DMA Sequences Using Descriptor Blocks

Autobuffer Based DMA

Autobuffer based DMA operates like descriptor based DMA except that it does not require descriptors in memory. In autobuffer mode, DMA control registers are writable and programmed directly by the processor to initiate a DMA transfer sequence. Upon completion of the transfer sequence, the control registers are reloaded with their original setup values for the next transfer. This effectively creates a circular buffer that continues to transfer data until disabled by clearing the DMA Enable bit in the peripheral's DMA Configuration register. If enabled, interrupts are generated at the halfway and completion points in the transfer sequence. For more information about peripheral interrupts, see the appropriate peripheral chapter.

Setting Up Autobuffer Based DMA

The following steps illustrate the typical sequence for setting up autobuffer based DMA.

1. Set the Autobuffer Enable bit in the DMA Configuration register.
2. Initialize the DMA Transfer Count, DMA Start Address High, and DMA Start Address Low registers.
3. Write to the DMA Configuration register, configuring the DMA transfer and setting the DMA Enable bit.

This final write starts the autobuffer based DMA transfer sequence.

To disable autobuffering, clear the DMA Enable bit in the Configuration register. Clearing the bit during active operation will stop the current single or burst transfer, but it will not cause a DMA error abort as it would for descriptor based DMA. The Autobuffer Enable bit in the DMA Configuration register can then be cleared.

Upon detecting that the DMA Enable bit is cleared, the DMA channel completes the current single or burst transfer and disables DMA. After the bit is cleared, the DMA Start Address and DMA Transfer Count registers can be used to monitor the status of the DMA transfers.

DMA Control Registers

The registers in this section are a generic representation of the actual peripheral DMA control registers, which are listed in Table 9-12 on page 9-30. The word "peripheral" is used in the register name in place of the specific peripheral name.

Peripheral DMA Configuration Register

The peripheral's DMA Configuration register determines whether DMA is enabled and performs other essential DMA functions, including functions that are peripheral dependent (see Figure 9-5).

The DMA Enable bit enables and activates a DMA channel to begin loading a descriptor block from memory. The processor should write the DMA Enable bit only during DMA channel initialization and in activating or terminating autobuffer mode. For descriptor based DMA, if the processor resets this bit during active operation, the DMA channel terminates with an error condition (see "DMA Abort Conditions" on page 9-44).

The Buffer Clear bit is used to clear the DMA buffer and should be accessed only when a DMA channel is not enabled.

The Buffer Clear bit should not be set when using autobuffer mode.

Peripheral DMA Configuration Register

In autobuffer mode, these bits are R/W. In some peripherals, Bit 12 of the DMA Configuration Word is read as a FIFO Status bit and written as a data size bit.

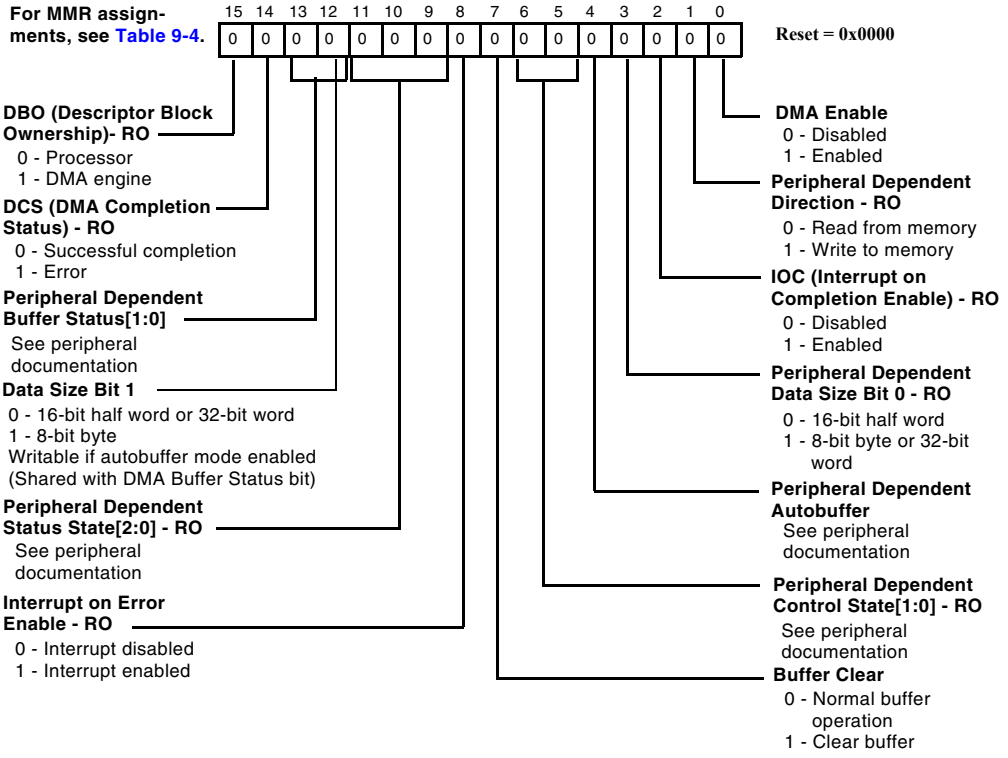


Figure 9-5. Peripheral DMA Configuration Register

Table 9-4 lists the MMR assignments for the peripheral DMA Configuration registers.

Table 9-4. Peripheral DMA Configuration Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_CONFIG	0xFFC0 3202
SPI1_CONFIG	0xFFC0 3602
SPORT0_RX_CONFIG	0xFFC0 2802
SPORT1_RX_CONFIG	0xFFC0 2C02
SPORT0_TX_CONFIG	0xFFC0 2800
SPORT1_TX_CONFIG	0xFFC0 2C00
UART0_CONFIG_RX	0xFFC0 1A02
UART1_CONFIG_RX	0xFFC0 1E02
UART0_CONFIG_TX	0xFFC0 1B02
UART1_CONFIG_TX	0xFFC0 1F02
USBD_DMACFG	0xFFC0 4440

The Descriptor Block Ownership (DBO) bit can be set *only* when bit 15 of a descriptor block's DMA Configuration Word is set and the DMA Configuration Word is loaded into the peripheral's DMA Configuration register.

When the current DMA transfer sequence has completed, the DMA channel clears bit 15 in the peripheral's DMA Configuration register and writes the contents of the register to the current descriptor block's Configuration Word in memory. Then the next descriptor block's Configuration Word is loaded into the peripheral's DMA Configuration register. If bit 15 of the Configuration Word is set (DMA channel has ownership), a full descriptor block download is performed, and bit 15 of the peripheral's DMA Configuration register is set once again.

Peripheral DMA Transfer Count Register

The peripheral's DMA Transfer Count register, shown in Figure 9-6, contains the number of transfers needed to complete the current DMA sequence. For descriptor based DMA, the value of this register is loaded when a descriptor block is read from memory. For autobuffer based DMA, this register is configured directly by the user.

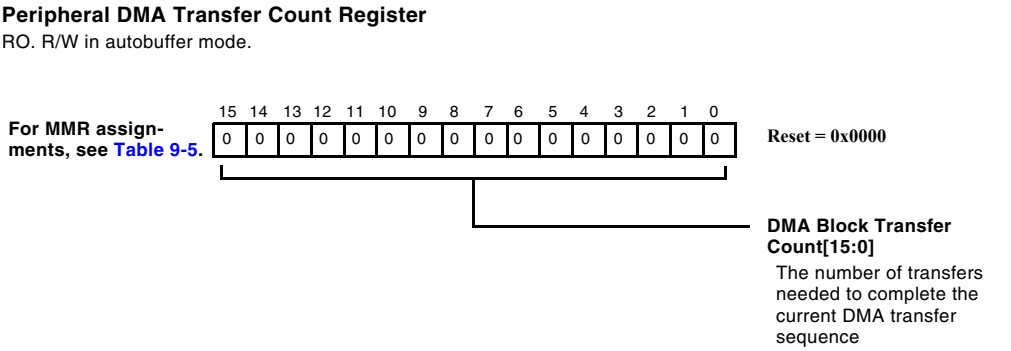


Figure 9-6. Peripheral DMA Transfer Count Register

Table 9-5 lists the MMR assignments for the peripheral DMA Transfer Count registers.

Table 9-5. Peripheral DMA Transfer Count Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_COUNT	0xFFC0 3208
SPI1_COUNT	0xFFC0 3608
SPORT0_COUNT_RX	0xFFC0 2A08
SPORT1_COUNT_RX	0xFFC0 2E08
SPORT0_COUNT_TX	0xFFC0 2B08
SPORT1_COUNT_TX	0xFFC0 2F08

Table 9-5. Peripheral DMA Transfer Count Register MMR Assignments (Cont'd)

Register Name	Memory-Mapped Address
UART0_COUNT_RX	0xFFC0 1A08
UART1_COUNT_RX	0xFFC0 1E08
UART0_COUNT_TX	0xFFC0 1B08
UART1_COUNT_TX	0xFFC0 1F08
USBD_DMACT	xFFC0 4446

The DMA Transfer Count always decrements by 1 for each bus transfer, independent of the transfer size (8, 16, or 32 bits). A DMA transfer sequence is complete when the transfer count reaches 0.

Peripheral DMA Start Address Registers

The peripheral's DMA Start Address range is 32 bits, formed by concatenating its DMA Start Address High [15:0] and its DMA Start Address Low [15:0] registers.

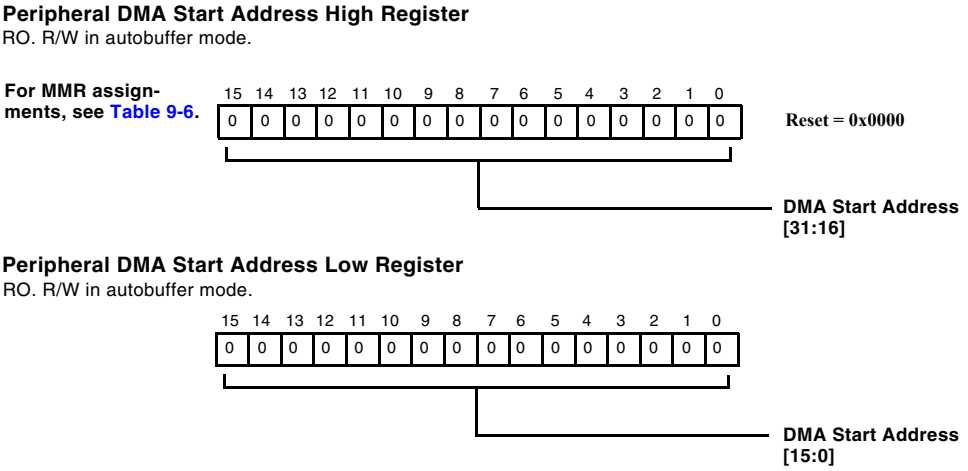


Figure 9-7. Peripheral DMA Start Address Registers

Table 9-6 lists the MMR assignments for the peripheral DMA Start Address registers.

Table 9-6. Peripheral DMA Start Address Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_START_ADDR_HI	0xFFC0 3204
SPI1_START_ADDR_HI	0xFFC0 3604
SPI0_START_ADDR_LO	0xFFC0 3206
SPI1_START_ADDR_LO	0xFFC0 3606
SPORT0_START_ADDR_HI_TX	0xFFC0 2B04

Table 9-6. Peripheral DMA Start Address Register MMR Assignments (Cont'd)

Register Name	Memory-Mapped Address
SPORT1_START_ADDR_HI_TX	0xFFC0 2F04
SPORT0_START_ADDR_HI_RX	0xFFC0 2A04
SPORT1_START_ADDR_HI_RX	0xFFC0 2E04
UART0_START_ADDR_HI_RX	0xFFC0 1A04
UART1_START_ADDR_HI_RX	0xFFC0 1E04
UART0_START_ADDR_LO_RX	0xFFC0 1A06
UART1_START_ADDR_LO_RX	0xFFC0 1E06
USBD_DMABH (High)	0xFFC0 4444
USBD_DMABL (Low)	0xFFC0 4442

The Start Address must be aligned to the transfer size, as shown in Table 9-7.

Table 9-7. Start Address Alignment Requirements

Transfer Size	Start Address Alignment Requirements
8 bits	Start Address has no alignment restriction.
16 bits	Start Address must be 16-bit aligned.
32 bits	Start Address must be 32-bit aligned.

Upon completion of the current access, the DMA address generators increment the start address by 1 for 8-bit transfers, by 2 for 16-bit transfers, or by 4 for 32-bit transfers.

No data packing is done on 8- or 16-bit transfers. These modes exist to support the alignment function, as required. The highest throughput is achieved with 32-bit transfers.

Use extreme caution when programming start addresses and transfer count to make sure that the DMA channel does not access unsupported memory, MMR space, scratchpad, or other critical system resources. Access to illegal memory ranges causes an exception.

Peripheral DMA Next Descriptor Pointer Register

The value of the peripheral's DMA Next Descriptor Pointer register determines the lower 16 bits of the descriptor block base address for the next DMA transfer sequence. After initial configuration of the DMA register set, this register is updated whenever the DMA channel fetches a new descriptor block from memory. Figure 9-8 describes the peripheral DMA Next Descriptor Pointer register.

The LSB of this register must always be 0, because the descriptor list must be 16-bit aligned. This register is not used for autobuffer mode.

Peripheral DMA Next Descriptor Pointer Register

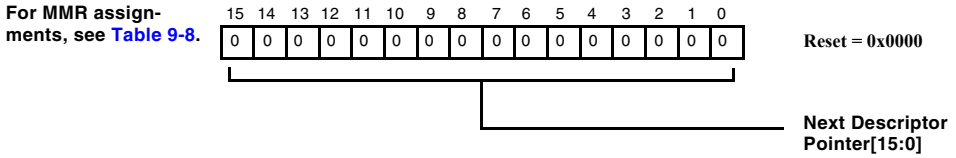


Figure 9-8. Peripheral DMA Next Descriptor Pointer Register

Table 9-8 lists the MMR assignments for the peripheral DMA Next Descriptor Pointer registers.

Table 9-8. Peripheral DMA Next Descriptor Pointer Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_NEXT_DESCR	0xFFC0 320A
SPI1_NEXT_DESCR	0xFFC0 360A
SPORT0_NEXT_DESCR_RX	0xFFC0 2A0A
SPORT1_NEXT_DESCR_RX	0xFFC0 2E0A
SPORT0_NEXT_DESCR_TX	0xFFC0 2B0A
SPORT1_NEXT_DESCR_TX	0xFFC0 2F0A
UART0_NEXT_DESCR_RX	0xFFC0 1A0A
UART1_NEXT_DESCR_RX	0xFFC0 1E0A
UART0_NEXT_DESCR_TX	0xFFC0 1B0A
UART1_NEXT_DESCR_TX	0xFFC0 1F0A
USB (none)	

DMA Descriptor Base Pointer Register (DMA_DBP)

In the ADSP-BF535 processor, the linked list of DMA descriptor blocks is relocatable. By default, the DMA Descriptor Base Pointer register locates the descriptor block linked list in the top 64 KB of on-chip L2 memory. This register, shown in Figure 9-9, is not updated on DMA descriptor block fetches; software controls it directly. The DMA Descriptor Base Pointer must be restricted to L2 memory space.

This register is not used for autobuffer mode and is shared across all peripherals.

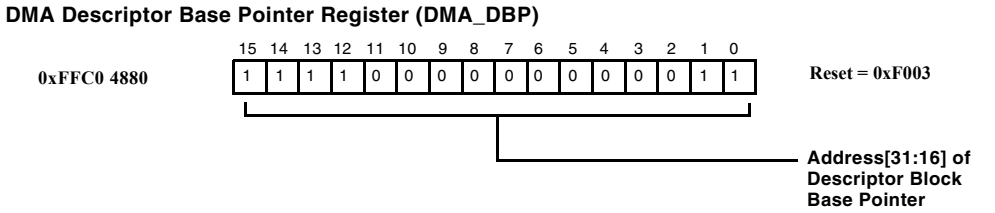


Figure 9-9. DMA Descriptor Base Pointer Register

Peripheral DMA Descriptor Ready Register

If a descriptor block's DBO bit is 0 when the descriptor block Configuration Word is accessed, the DMA channel halts. Writing a 1 to bit 0 of the peripheral's DMA Descriptor Ready register synchronizes the DMA channel, causing the Configuration Word to be reloaded into the DMA Configuration Register and causing bit 15 to be rechecked. If this bit is set, processing of the descriptor block resumes. Bit 0 of the peripheral's DMA Descriptor Ready register remains set until a descriptor block has been successfully fetched. After a successful fetch, bit 0 of the DMA Descriptor Ready register is cleared. Figure 9-10 describes the peripheral DMA Descriptor Ready register.

This register is not used for autobuffer mode.

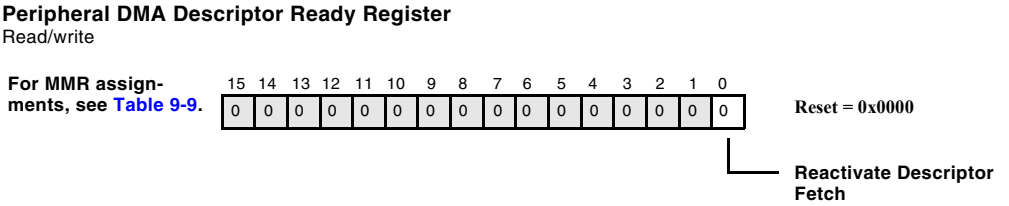


Figure 9-10. Peripheral DMA Descriptor Ready Register

Table 9-9. Peripheral DMA Descriptor Ready Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_DESCR_RDY	0xFFC0 320C
SPI1_DESCR_RDY	0xFFC0 360C
SPORT0_DESCR_RDY_RX	0xFFC0 2A0C
SPORT1_DESCR_RDY_RX	0xFFC0 2E0C
SPORT0_DESCR_RDY_TX	0xFFC0 2B0C
SPORT1_DESCR_RDY_TX	0xFFC0 2F0C
UART0_DESCR_RDY_RX	0xFFC0 1A0C
UART1_DESCR_RDY_RX	0xFFC0 1E0C
UART0_DESCR_RDY_TX	0xFFC0 1B0C
UART1_DESCR_RDY_TX	0xFFC0 1F0C
USBD (N/A)	

Peripheral DMA Current Descriptor Pointer Register

The peripheral's DMA Current Descriptor Pointer register contains the least significant 16 bits of the base address of the descriptor block that the DMA channel is actively processing. This register, shown in Figure 9-11, is loaded with the value from the peripheral's DMA Next Descriptor Pointer register before each DMA work block is fetched. This register is not used for autobuffer mode.

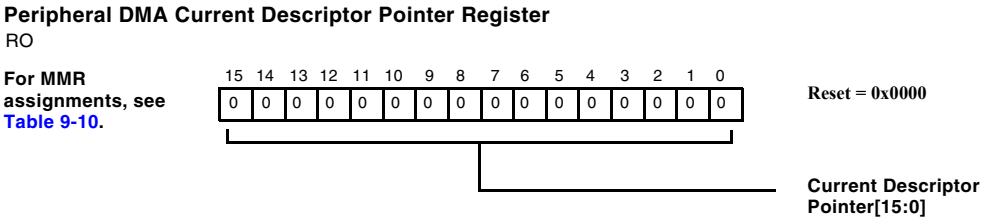


Figure 9-11. Peripheral DMA Current Descriptor Pointer Register

Table 9-10. Peripheral DMA Current Descriptor Pointer Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_CURR_PTR	0xFFC0 3200
SPI1_CURR_PTR	0xFFC0 3600
SPORT0_CURR_PTR_RX	0xFFC0 2A00
SPORT1_CURR_PTR_RX	0xFFC0 2E00
SPORT0_CURR_PTR_TX	0xFFC0 2B00
SPORT1_CURR_PTR_TX	0xFFC0 2F00
UART0_CURR_PTR_RX	0xFFC0 1A00
UART1_CURR_PTR_RX	0xFFC0 1E00
UART0_CURR_PTR_TX	0xFFC0 1B00
UART1_CURR_PTR_TX	0xFFC0 1F00
USBD (N/A)	

Peripheral DMA IRQ Status Register

The peripheral's DMA IRQ Status register, shown in Figure 9-12, is a sticky register that records the occurrence of any of its interrupt sources.

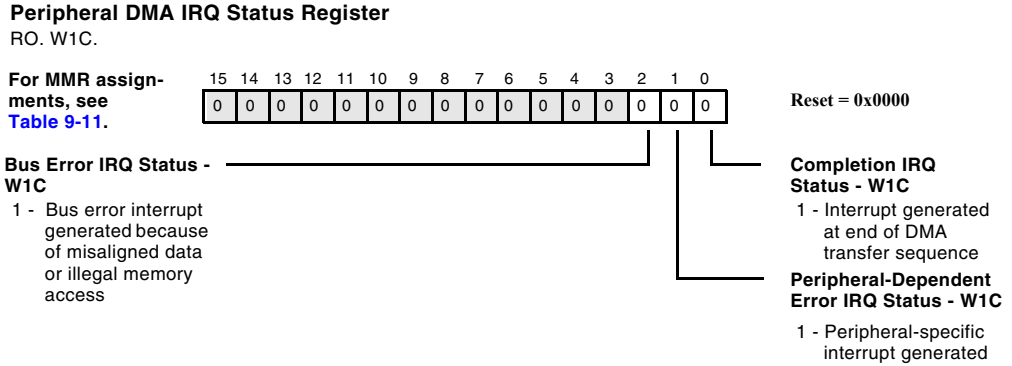


Figure 9-12. Peripheral DMA IRQ Status Register

Table 9-11. Peripheral DMA Interrupt Status Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_DMA_INT	0xFFC0 320E
SPI1_DMA_INT	0xFFC0 360E
SPORT0_IRQSTAT_RX	0xFFC0 2A0E
SPORT1_IRQSTAT_RX	0xFFC0 2E0E
SPORT0_IRQSTAT_TX	0xFFC0 2B0E
SPORT1_IRQSTAT_RX	0xFFC0 2E0E
UART0_IRQSTAT_RX	0xFFC0 1A0E
UART1_IRQSTAT_RX	0xFFC0 1E0E
UART0_IRQSTAT_TX	0xFFC0 1B0E

Table 9-11. Peripheral DMA Interrupt Status Register MMR Assignments (Cont'd)

Register Name	Memory-Mapped Address
UART1_IRQSTAT_TX	0xFFC0 1F0E
USBD_DMAIRQ	0xFFC0 4446

Each ADSP-BF535 processor DMA channel can produce three types of interrupts:

- A completion interrupt, generated when a DMA transfer sequence completes
- A bus error interrupt, generated when one of two events occur:
  - Misalignment of an address to data size (see "Data Misalignment" on page 9-46)
  - A memory access resulting in an error response (see "Illegal Memory Access" on page 9-46)
- A peripheral specific interrupt (see the appropriate chapter for more information)

If the Interrupt on Completion (IOC) bit of the peripheral's DMA Configuration register is set, an interrupt is generated at the end of a DMA transfer sequence, and the Completion IRQ Status bit is set in the peripheral's DMA IRQ Status register:

- The Interrupt on Error Enable bit is associated with peripheral specific interrupts.
- The Bus Error IRQ Status bit is non-maskable because of misalignment or illegal access.

For conditions resulting in a peripheral specific interrupt, see the chapter that describes the peripheral.

Table 9-12 lists the peripheral DMA control registers.

Table 9-12. Peripheral DMA Control Registers

Peripheral	Config Register	Next Descriptor Pointer Register	Descriptor Ready Register	Interrupt Status Register	Current Descriptor Pointer Register	Start Address High Register	Start Address Low Register	Transfer Count Register
MemDMA (Source)	MDS_ DCFG	MDS_ DND	MDS_ DDR	MDS_ DI	MDS_ DCP	MDS_ DSAH	MDS_ DSAL	MDS_ DCT
MemDMA (Destina-tion)	MDD_ DCFG	MDD_ DND	MDD_ DDR	MDD_ DI	MDD_ DCP	MDD_ DSAH	MDD_ DSAL	MDD_ DCT
SPI	SPIx_ CONFIG	SPIx_ NEXT_ DESCR	SPIx_ DESCR_ RDY	SPIx_ DMA_ INT	SPIx_ CURR_ PTR	SPIx_ START_ ADDR_ HI	SPIx_ START_ ADDR_ LO	SPIx_ COUNT
SPORTx RX	SPORTx_ CONFIG_ DMA_ RX	SPORTx_ NEXT_ DESCR_ RX	SPORTx_ DESCR_ RDY_ RX	SPORTx_ IRQSTAT_ RX	SPORTx_ CURR_ PTR_RX	SPORTx_ START_ ADDR_ HI_RX	SPORTx_ START_ ADDR_ LO_RX	SPORTx_ COUNT_ RX
SPORTx TX	SPORTx_ CONFIG_ DMA_ TX	SPORTx_ NEXT_ DESCR_ TX	SPORTx_ DESCR_ RDY_TX	SPORTx_ IRQSTAT_ TX	SPORTx_ CURR_ PTR_TX	SPORTx_ START_ ADDR_ HI_TX	SPORTx_ START_ ADDR_ LO_TX	SPORTx_ COUNT_ TX
UARTx RX	UARTx_ CONFIG_ RX	UARTx_ NEXT_ DESCR_ RX	UARTx_ DESCR_ RDY_RX	UARTx_ IRQSTAT_ RX	UARTx_ CURR_ PTR_RX	UARTx_ START_ ADDR_ HI_RX	UARTx_ START_ ADDR_ LO_RX	UARTx_ COUNT_ RX
UARTx TX	UARTx_ CONFIG_ TX	UARTx_ NEXT_ DESCR_ TX	UARTx_ DESCR_ RDY_TX	UARTx_ IRQSTAT_ TX	UARTx_ CURR_ PTR_TX	UARTx_ START_ ADDR_ HI_TX	UARTx_ START_ ADDR_ LO_TX	UARTx_ COUNT_ TX
USB	USBD_ DMACFG	N/A	N/A	USBD_ DMAIRQ	N/A	USBD_ DMABH	USBD_ DMABL	USBD_ DMACT

Memory DMA (MemDMA)

The Memory DMA (MemDMA) controller provides memory-to-memory DMA transfers among the ADSP-BF535 processor memory spaces. These memory spaces include the Peripheral Component Interconnect (PCI) address spaces, L1, L2, external synchronous, and asynchronous memories.

The MemDMA controller consists of two channels—one for the source, which is used to read from memory, and one for the destination, which is used to write to memory. Both channels share a 16-entry, 32-bit FIFO. The source DMA channel fills the FIFO; the destination DMA channel empties it. The FIFO depth significantly improves throughput on block transfers between internal and external memory. The FIFO supports 8-, 16-, and 32-bit transfers. However, 8- and 16-bit transfers use only part of the 32-bit data bus for each transfer; therefore, the throughput for these transfer sizes is less than for full, 32-bit DMA operations.

The MemDMA controller does not support autobuffer based DMA. Transfers using the MemDMA controller must use descriptor based DMA. Two separate linked lists of descriptor blocks are required—one for the source DMA channel and one for the destination DMA channel. The separation of control allows an off-chip host processor to manage one list through the PCI port bus and the core processor to manage the other list. For a detailed explanation of descriptor based DMA, see "Descriptor Based DMA" on page 9-3.

Because the source and destination DMA channels share a single FIFO buffer, the descriptor blocks must be configured to have the same transfer count and data size.

It is preferable to activate interrupts on only one channel. This eliminates ambiguity when trying to identify the channel (either source or destination) that requested the interrupt.

MemDMA Control Registers

MemDMA control registers are like DMA control registers but have fewer bit assignments. Because MemDMA provides only memory-to-memory DMA transfers, MemDMA control registers contain no peripheral dependent bits.

MemDMA control registers consist of source registers, used to read from memory, and destination registers, used to write to memory. This section shows destination registers first, then source registers. For descriptions of the registers, see the related DMA register.

Destination Memory DMA Configuration Register (MDD_DCFG)

Figure 9-13 shows the Destination Memory DMA Configuration register. The related DMA register is described in "Peripheral DMA Configuration Register" on page 9-16.

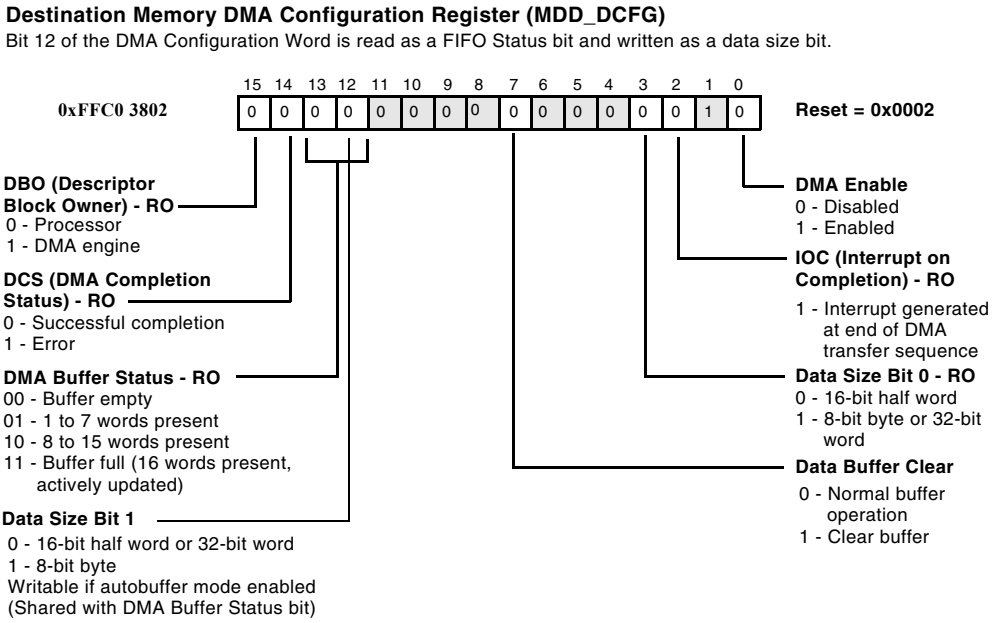


Figure 9-13. Destination Memory DMA Configuration Register

Destination Memory DMA Transfer Count Register (MDD_DCT)

Figure 9-14 shows the Destination Memory DMA Transfer Count register. The related DMA register is described in "Peripheral DMA Transfer Count Register" on page 9-19.

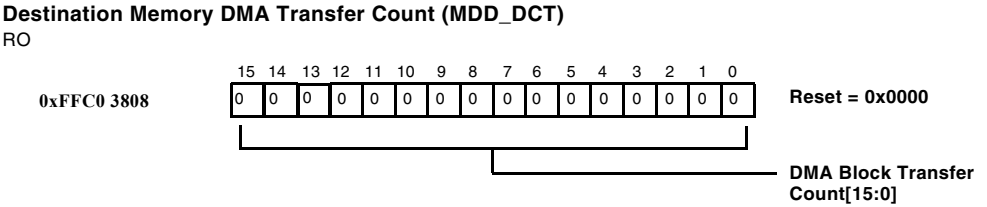


Figure 9-14. Destination Memory DMA Transfer Count Register

Destination Memory DMA Start Address Registers (MDD_DSAH, MDD_DSAL)

Figure 9-15 shows the Destination DMA Start Address registers. The related DMA registers are found in "Peripheral DMA Start Address Registers" on page 9-21.

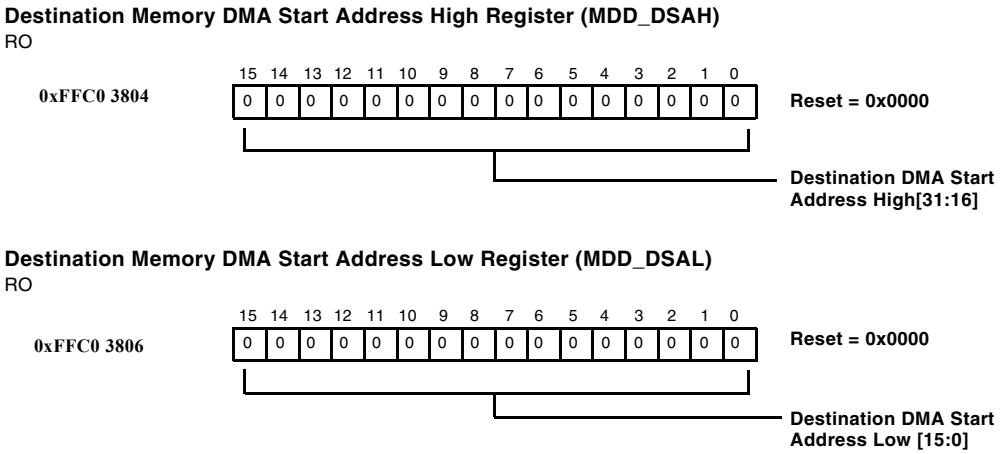


Figure 9-15. Destination Memory DMA Start Address Registers

Destination Memory DMA Next Descriptor Pointer Register (MDD_DND)

Figure 9-16 shows the Destination Memory DMA Next Descriptor Pointer register. The related DMA register is described in "Peripheral DMA Next Descriptor Pointer Register" on page 9-23.

Destination Memory DMA Next Descriptor Pointer Register (MDD_DND)

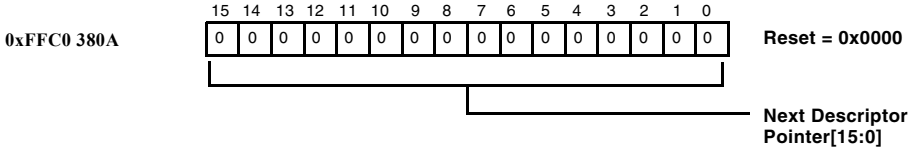


Figure 9-16. Destination Memory DMA Next Descriptor Pointer Register

Destination Memory DMA Descriptor Ready Register (MDD_DDR)

Figure 9-17 shows the Destination Memory DMA Descriptor Ready register. The related DMA register is described in "Peripheral DMA Descriptor Ready Register" on page 9-25.

Destination Memory DMA Descriptor Ready Register (MDD_DDR)

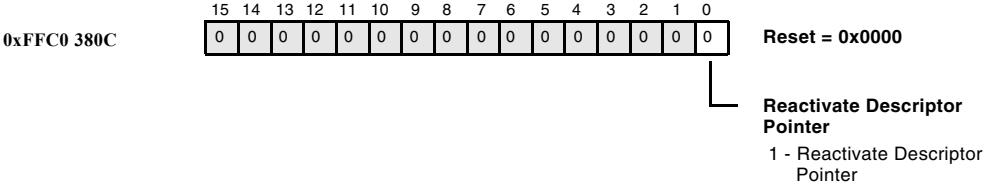


Figure 9-17. Destination Memory DMA Descriptor Ready Register

Destination Memory DMA Current Descriptor Pointer Register (MDD_DCP)

Figure 9-18 shows the Destination Memory DMA Current Descriptor Pointer register. The related DMA register is described in "Peripheral DMA Current Descriptor Pointer Register" on page 9-26.

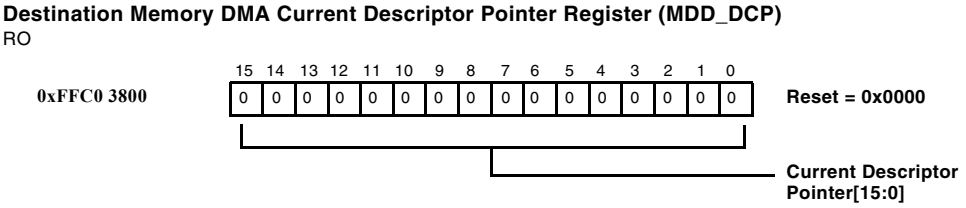


Figure 9-18. Destination Memory DMA Current Descriptor Pointer Register

Destination Memory DMA Interrupt Register (MDD_DI)

Figure 9-19 shows the Destination Memory DMA Interrupt register. The related DMA register is described in "Peripheral DMA IRQ Status Register" on page 9-28.

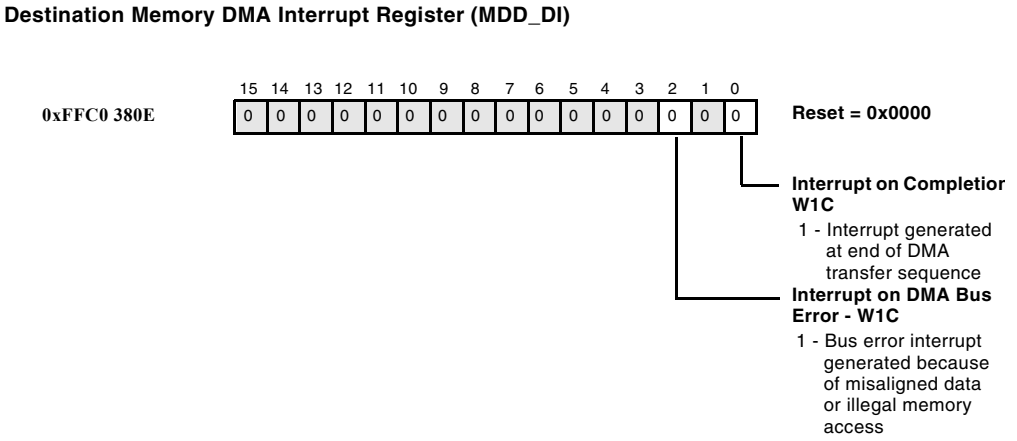


Figure 9-19. Destination Memory DMA Interrupt Register

Source Memory DMA Configuration Register (MDS_DCFG)

Figure 9-20 shows the Source Memory DMA Configuration register. The related DMA register is described in "Peripheral DMA Configuration Register" on page 9-16.

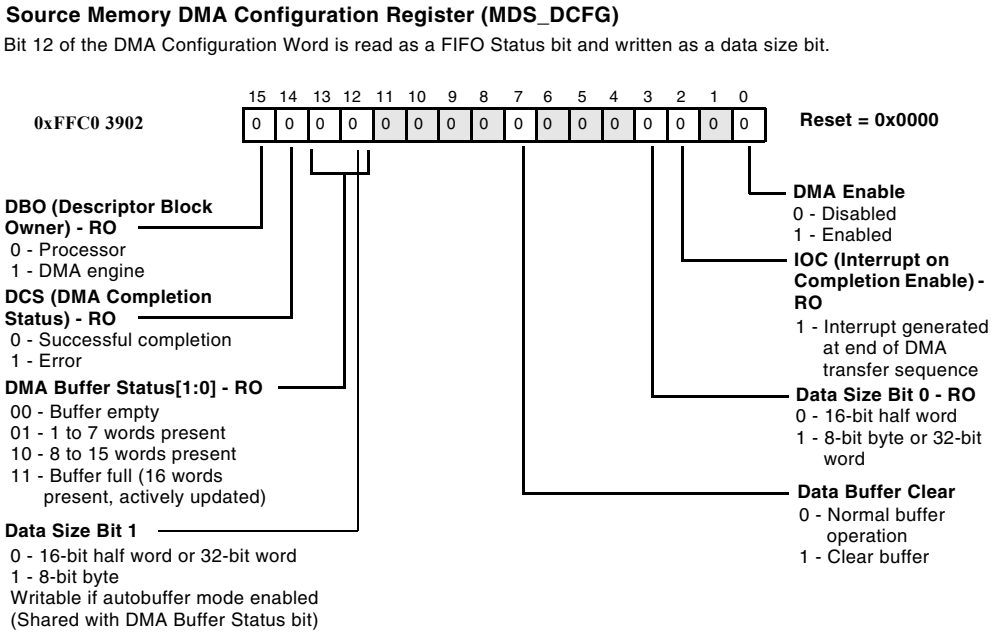


Figure 9-20. Source Memory DMA Configuration Register

Source Memory DMA Transfer Count Register (MDD_DCT)

Figure 9-21 shows the Source Memory DMA Transfer Count register. The related DMA register is described in "Peripheral DMA Transfer Count Register" on page 9-19.

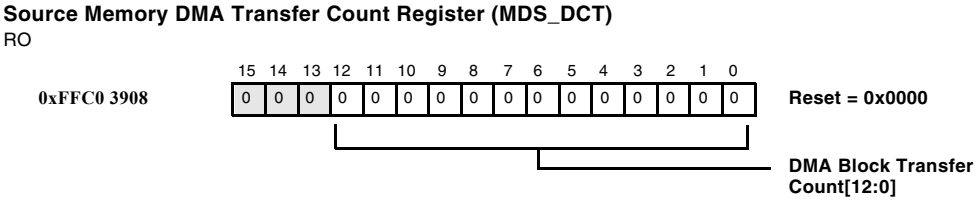


Figure 9-21. Source Memory DMA Transfer Count Register

Source Memory DMA Start Address Registers (MDS_DSAH, MDS_DSAL)

Figure 9-22 shows the Source Memory DMA Start Address registers. The related DMA registers are described in "Peripheral DMA Start Address Registers" on page 9-21.

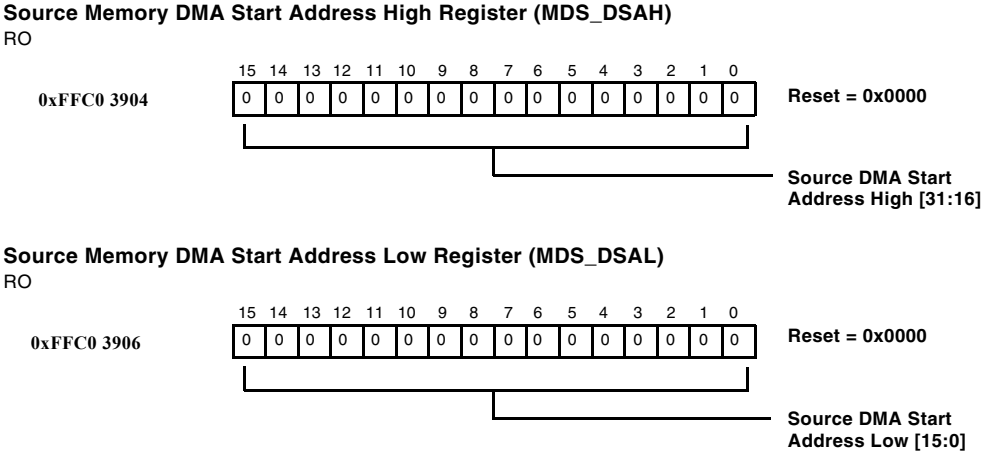


Figure 9-22. Source Memory DMA Start Address Registers

Source Memory DMA Next Descriptor Pointer Register (MDS_DND)

Figure 9-23 shows the Source Memory DMA Next Descriptor Pointer register. The related DMA register is described in "Peripheral DMA Next Descriptor Pointer Register" on page 9-23.

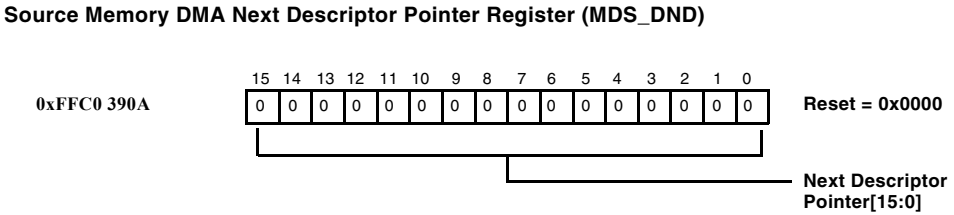


Figure 9-23. Source Memory DMA Next Descriptor Pointer Register

Source Memory DMA Descriptor Ready Register (MDS_DDR)

Figure 9-24 shows the Source DMA Descriptor Ready register. The related DMA register is described in "Peripheral DMA Descriptor Ready Register" on page 9-25.

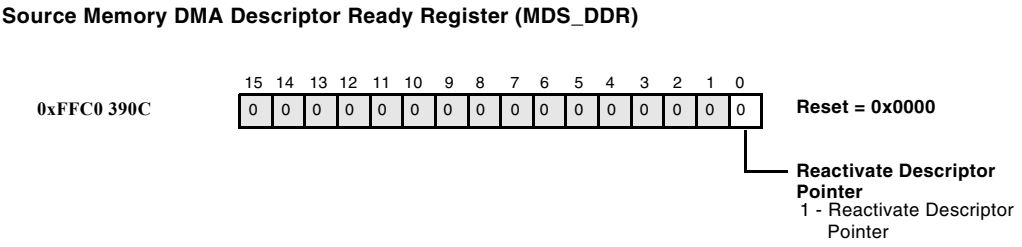


Figure 9-24. Source Memory DMA Descriptor Ready Register

Source Memory DMA Current Descriptor Pointer Register (MDS_DCP)

Figure 9-25 shows the Source Memory DMA Current Descriptor Pointer register. The related DMA register is described in "Peripheral DMA Current Descriptor Pointer Register" on page 9-26.

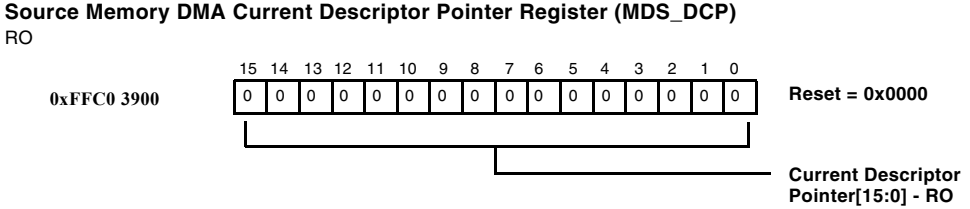


Figure 9-25. Source Memory DMA Current Descriptor Pointer Register

Source Memory DMA Interrupt Register (MDS_DI)

Figure 9-26 shows the Source Memory DMA Interrupt register. The related DMA register is described in "Peripheral DMA IRQ Status Register" on page 9-28.

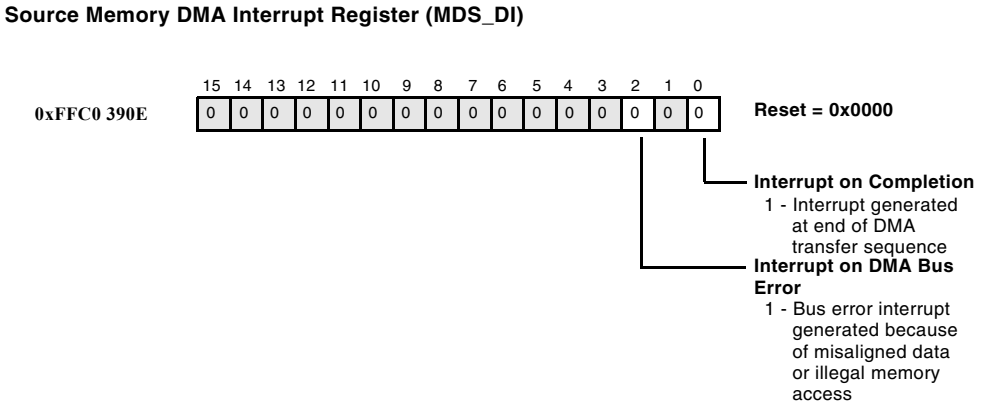


Figure 9-26. Source Memory DMA Interrupt Register

Performance/Throughput for MemDMA

Performance and throughput numbers for the MemDMA controller can be found in "DAB Performance" on page 7-11.

Note that reads from EBIU lower performance because of the higher latency of off-chip memory reads versus writes. Also, throughput is reduced for partial bus-width transfers (that is, specifying 8- or 16-bit DMA transfer widths instead of 32-bit transfer widths). Bus arbitration induces additional latency.

Note the Memory DMA architecture achieves highest throughput on block copies between external SDRAM and internal memory. L1-to-L2 transfer throughput is less, because only the DAB bus is employed.

The Memory DMA controller performs a burst transfer across the DAB bus to fill the FIFO, then a burst transfer across the DAB bus to empty the FIFO in a serialized fashion. Likewise, external-to-external block copies are also serialized across the EAB bus.

Note that DMA descriptor list fetches always occur across the DAB bus.

DMA Abort Conditions

These conditions cause a DMA abort:

- The processor clears the DMA Enable bit during active DMA processing.
- The DMA channel does not relinquish the descriptor block back to the processor, nor does it write back error status information. No interrupt is generated.

- The processor clears the DMA buffer during active DMA processing by setting the Buffer Clear bit (bit 7 of the DMA Configuration register).
- The DMA channel relinquishes the descriptor block to the processor and writes back error status information.
- A system software reset occurs during active DMA processing.
- This action causes loss of state and immediate termination of DMA processing. The DMA channel does not relinquish the descriptor block back to the processor, nor does it write back error status information. No interrupt is generated.
- Any other error condition results in a proper error abort.
- The DMA channel relinquishes the descriptor block to the processor and writes back error status. If enabled, an interrupt is generated.

DMA Bus Error Conditions

Two general conditions cause DMA bus error conditions:

- Data misalignment
- Illegal memory access

Data Misalignment

Data misalignment occurs when the data size to be transferred is not consistent with the current address. Misalignment includes:

- Descriptor fetch on a byte boundary
- 16-bit data access on other than a half word boundary
- 32-bit data access on other than a word boundary

Data misalignment causes these things to occur:

- The DMA channel does not access the bus for data transfers.
- The DMA Enable bit (bit 0 of the Configuration Word) is cleared.
- A Bus Error interrupt is generated, and Bit 2 of the IRQ Status register is set.
- The Configuration Word is not written back to the descriptor block.

Illegal Memory Access

If an illegal memory access occurs, such as access to MMR space, the processor responds with an error, and:

- The DMA channel completes the current single or burst transfer.
- The DMA Enable bit (bit 0 of the Configuration Word) is cleared.
- The Bus Error interrupt is generated, and Bit 2 of the IRQ Status register is set.
- The Configuration Word is not written back to the descriptor block.

10 SPI COMPATIBLE PORT CONTROLLERS

The ADSP-BF535 processor has two independent Serial Peripheral Interface (SPI) ports, SPI0 and SPI1, that provide an I/O interface to a wide variety of SPI compatible peripheral devices. Each SPI port has its own set of control registers and data buffers.

With a range of configurable options, the SPI ports provide a glueless hardware interface with other SPI compatible devices. Typical SPI compatible peripheral devices that can be used to interface to the ADSP-BF535 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 ADSP-BF535 SPI is an industry-standard synchronous serial link that supports communication with multiple SPI compatible devices. The SPI peripheral is a synchronous, 4-wire interface consisting of two data pins (MOSI and MISO), one device select pin ($\overline{\text{SPISS}}$), and a gated clock pin (SCK). The ADSP-BF535 SPI supports these features:

- Full duplex operation
- Master-slave mode multimaster environment
- Open drain outputs
- Programmable baud rates, clock polarities, and phases
- Slave operations with another master SPI device

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 ADSP-BF535 SPI. The interface is essentially a shift register that serially transmits and receives data bits, one bit 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.

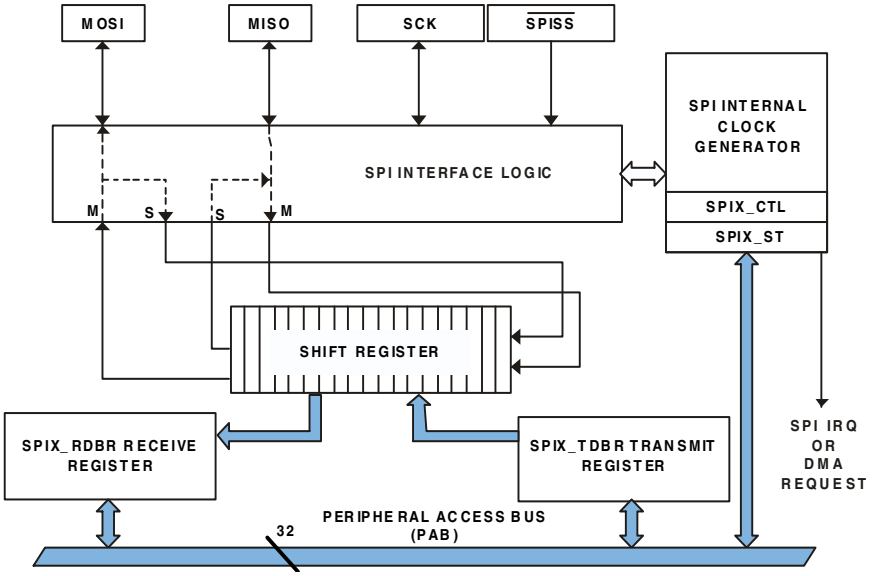


Figure 10-1. ADSP-BF535 SPI Block Diagram

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 ADSP-BF535 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 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 ADSP-BF535 environment where multiple ADSP-BF535 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 ADSP-BF535 processor can make use of 14 programmable flags, PF2–PF15, that are dedicated SPI slave select signals for the SPI slave devices. SPI0 and SPI1 each have seven available slave select signals.

At reset, the SPI is disabled and configured as a slave.

Interface Signals

The following sections discuss 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. SCK 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 signal 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 signal is used to shift out and shift in the data driven on the MISO and MOSI lines. The data is always shifted out on active edges of the clock and sampled on inactive edges of the clock. Clock polarity and clock phase relative to data are programmable in the SPIx Control register (SPIx_CTL) and define the transfer format.

Serial Peripheral Interface Slave Select Input Signal

The $\overline{\texttt{SPISS}}$ signal is the SPI Serial Peripheral Slave Select Input signal. This is an active-low signal used to enable an ADSP-BF535 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 $\overline{\texttt{SPISS}}$ input signal of a master is asserted (driven low), an error has occurred. This means that another device is also trying to be the master device.

Master Out Slave In (MOSI)

The MOSI signal is the Master Out Slave In pin, one of the bidirectional I/O data pins. If the ADSP-BF535 processor is configured as a master, the MOSI pin becomes a data transmit (output) pin, transmitting output data. If the ADSP-BF535 processor is configured as a slave, the MOSI pin becomes a data receive (input) pin, receiving input data. In an ADSP-BF535 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 signal is the Master In Slave Out pin, one of the bidirectional I/O data pins. If the ADSP-BF535 processor is configured as a master, the MISO pin becomes a data receive (input) pin, receiving input data. If the ADSP-BF535 is configured as a slave, the MISO pin becomes a data transmit (output) pin, transmitting output data. In a ADSP-BF535 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 ADSP-BF535 processor can be used as the slave SPI device. The 8-bit host microcontroller is the SPI master.

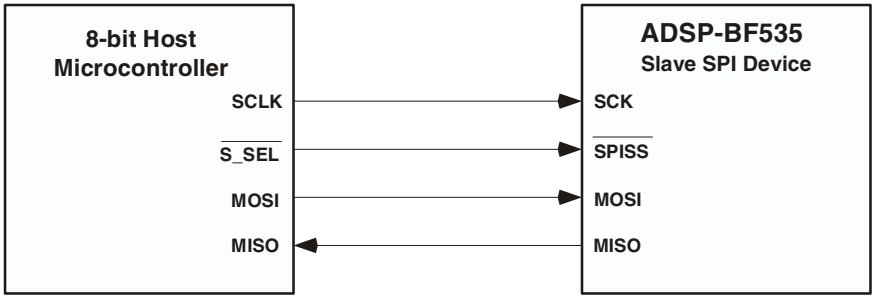


Figure 10-2. ADSP-BF535 Processor as Slave SPI Device

Interrupt Behavior

The behavior of the SPI interrupt signal depends on the transfer initiation mode bit field (TIMOD) in the SPI Control register. In DMA mode, the interrupt can be generated upon completion of a DMA multiword transfer or upon an SPI error condition (MODF, TXE when TRAN = 0, or RBSY when TRAN = 1). When not using DMA mode, an interrupt is generated when the SPI is ready to accept new data for a transfer. The TXE and RBSY error conditions do not generate interrupts in these modes. An interrupt is also generated in a master when the mode fault error occurs.

For more information about this interrupt output, see the discussion of the TIMOD bits in "SPIx Control Register (SPIx_CTL)" on page 10-8.

SPI Registers

The SPI peripheral on the ADSP-BF535 processor 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: SPIx_BAUD, SPIx_CTL, SPIx_FLG, and SPIx_ST. Two registers are used for buffering receive and transmit data: SPIx_RDBR and SPIx_TDBR. Eight registers are related to DMA functionality. The shift register, SFDR, is internal to the SPI module and is not directly accessible.

See "Error Signals and Flags" on page 10-35 for more information about how the bits in these registers are used to signal errors and other conditions. See "Register Functions" on page 10-26 for more information about SPI register and bit functions.

Non-DMA Registers

The following sections describes the SPI non-DMA registers.

SPIx Baud Rate Register (SPIx_BAUD)

The SPIx Baud Rate register (SPIx_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:

- SCKx frequency = (System clock frequency)/(2 × SPIx_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.

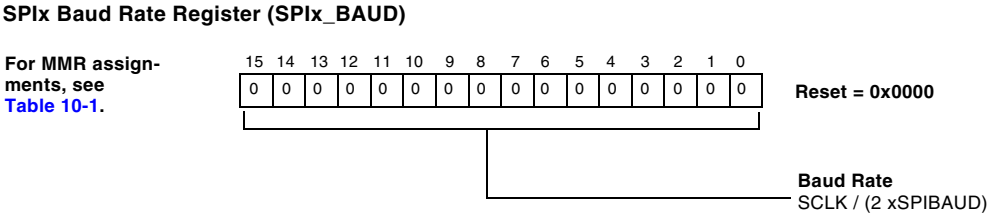


Figure 10-3. SPIx Baud Rate Registers

Table 10-1. SPIx Baud Rate Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_BAUD	0xFFC0 300A
SPI1_BAUD	0xFFC0 340A

Table 10-2 lists several possible baud rate values for `SPIx_BAUD`.

Table 10-2. SPI Master Baud Rate Example

SPIBAUD 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

SPIx Control Register (SPIx_CTL)

The `SPIx_CTL` register 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.

Figure 10-4 provides the bit descriptions for SPIx_CTL.

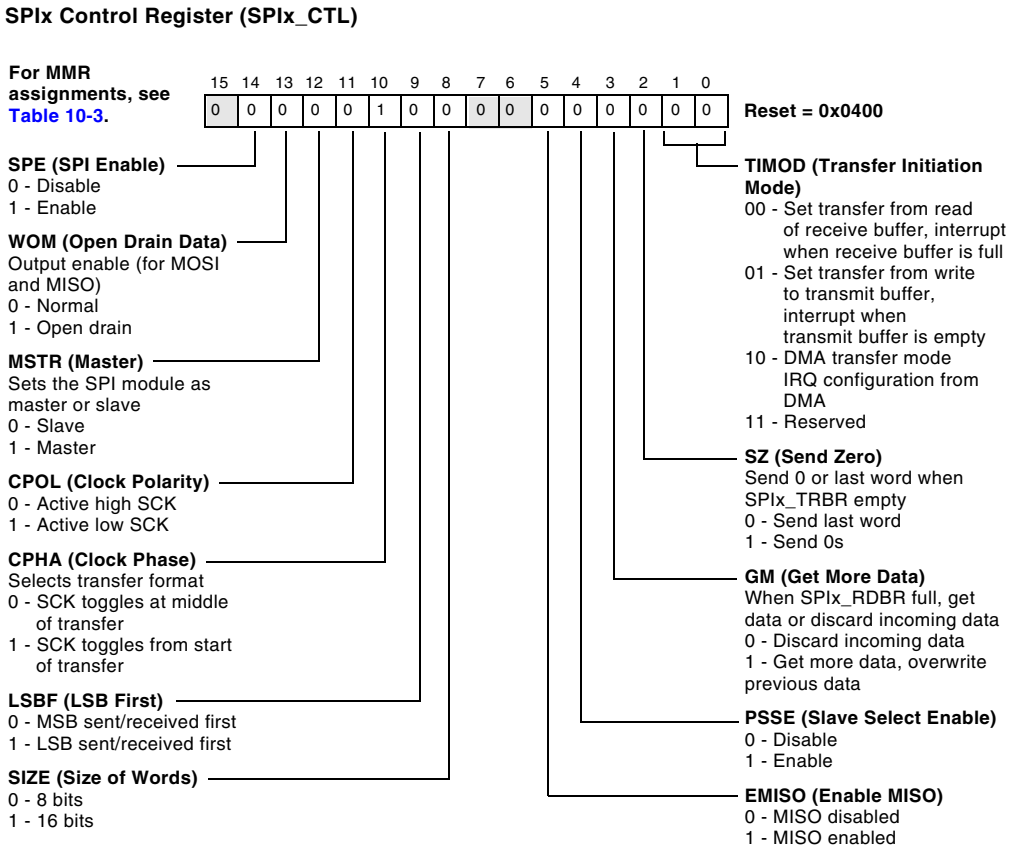


Figure 10-4. SPIx Control Register

Table 10-3. SPIx Control Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_CTL	0xFFC0 3000
SPI1_CTL	0xFFC0 3400

SPIx Flag Register (SPIx_FLG)

If the SPI is enabled as a master, the SPI uses SPIx_FLG to enable up to 14 general-purpose programmable flag pins (7 for each SPI) to be used as individual slave select lines. In slave mode, the SPIx_FLG bits have no effect, and each SPI uses the SPISS input as a slave select. Figure 10-5 shows the SPIx_FLG register diagram.

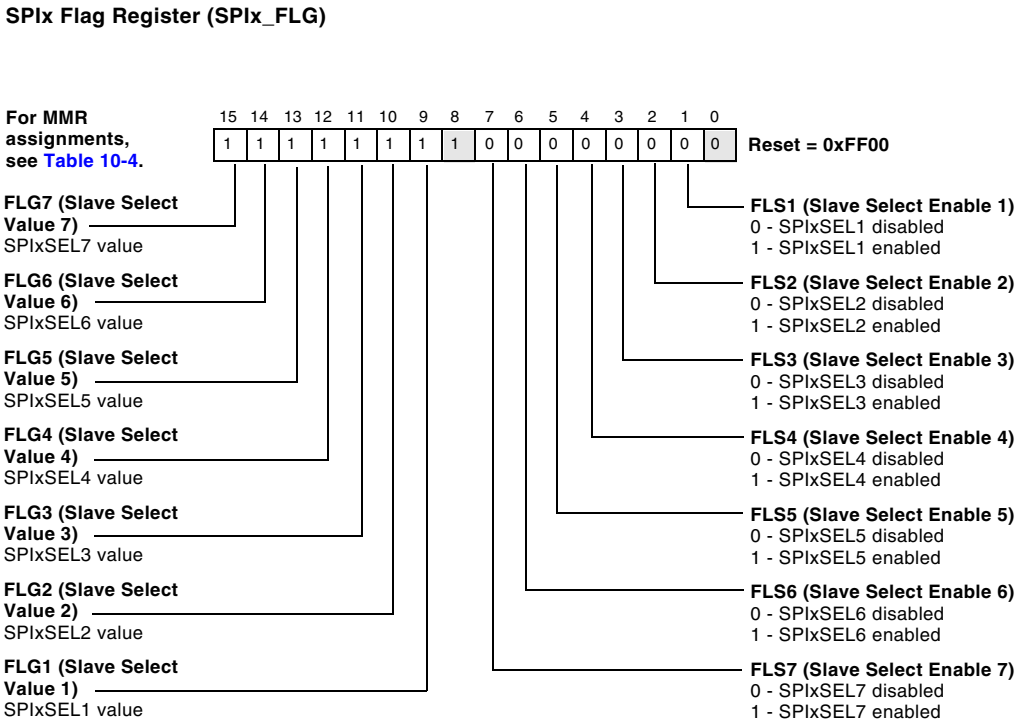


Figure 10-5. SPIx Flag Register

Table 10-4. SPIx Flag Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_FLG	0xFFC0 3002
SPI1_FLG	0xFFC0 3402

The SPIx_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 an FLSx bit is set, the corresponding PFx pin is driven as a slave select. For example, if FLS1 is set in SPI0_FLG, PF2 is driven as a slave select (SPI0SEL1). Table 10-5 and Table 10-6 show 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 SPIx_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, FLS1 in SPI1_FLG must be set. Clearing FLG1 in SPI1_FLG drives PF3 low; setting FLG1 drives PF3 high. The PF3 pin can be cycled high and low between transfers by setting and clearing FLG1. 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 FLS1 bit in SPI1_FLG. It is not necessary to write to the FLG1 bit, because the SPI hardware automatically drives the PF3 pin. Table 10-5 and Table 10-6, respectively, list the SPI0_FLG and SPI1_FLG bit mapping to the PFx pins.

Table 10-5. SPI0_FLG Bit Mapping to PFx Pins

Bit	Name	Function	PFx Pin
0		Reserved	
1	FLS1	SPI0SEL1 Enable	PF2
2	FLS2	SPI0SEL2 Enable	PF4
3	FLS3	SPI0SEL3 Enable	PF6
4	FLS4	SPI0SEL4 Enable	PF8
5	FLS5	SPI0SEL5 Enable	PF10
6	FLS6	SPI0SEL6 Enable	PF12
7	FLS7	SPI0SEL7 Enable	PF14
8		Reserved	
9	FLG1	SPI0SEL1 Value	PF2
10	FLG2	SPI0SEL2 Value	PF4
11	FLG3	SPI0SEL3 Value	PF6
12	FLG4	SPI0SEL4 Value	PF8
13	FLG5	SPI0SEL5 Value	PF10
14	FLG6	SPI0SEL6 Value	PF12
15	FLG7	SPI0SEL7 Value	PF14

Table 10-6. SPI1_FLG Bit Mapping to PFx Pins

Bit	Name	Function	PFx Pin
0		Reserved	
1	FLS1	SPI1SEL1 Enable	PF3
2	FLS2	SPI1SEL2 Enable	PF5
3	FLS3	SPI1SEL3 Enable	PF7
4	FLS4	SPI1SEL4 Enable	PF9
5	FLS5	SPI1SEL5 Enable	PF11
6	FLS6	SPI1SEL6 Enable	PF13
7	FLS7	SPI1SEL7 Enable	PF15
8		Reserved	
9	FLG1	SPI1SEL1 Value	PF3
10	FLG2	SPI1SEL2 Value	PF5
11	FLG3	SPI1SEL3 Value	PF7
12	FLG4	SPI1SEL4 Value	PF9
13	FLG5	SPI1SEL5 Value	PF11
14	FLG6	SPI1SEL6 Value	PF13
15	FLG7	SPI1SEL7 Value	PF15

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 SPIx_CTL enables this feature. When PSSE = 1, the $\overline{\texttt{SPISS}}$ input is the master-mode error input. Otherwise, $\overline{\texttt{SPISS}}$ is ignored. The state of these input pins can be observed in the Flag Clear register (FIO_FLAG_C) or the Flag Set register (FIO_FLAG_S).

Multiple Slave SPI Systems

The FLSx bits in SPIx_FLG are used in a multiple slave SPI environment. For example, if there are eight SPI devices in the system including an ADSP-BF535 processor master, the master ADSP-BF535 processor can support the SPI mode transactions across the other seven devices. This configuration requires only one master ADSP-BF535 processor in this multislave environment. For example, assume that SPI0 is the master. The seven flag pins (PF2, PF4, PF6, PF8, PF10, PF12, and PF14) on the ADSP-BF535 processor master can be connected to each of the slave SPI device's $\overline{\text{SPISS}}$ pins. In this configuration, the FLSx bits in SPIx_FLG can be used in three cases.

In cases 1 and 2, the ADSP-BF535 processor is the master and the seven microcontrollers/peripherals with SPI interfaces are slaves. The ADSP-BF535 processor can:

- Transmit to all seven SPI devices at the same time in a broadcast mode. Here, all FLSx bits are set.
- 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 ADSP-BF535 processors.

- If all the slaves are also ADSP-BF535 processors, then the requestor can receive data from only one ADSP-BF535 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 ADSP-BF535 processor as a master with three ADSP-BF535 processors (or other SPI compatible devices) as slaves.

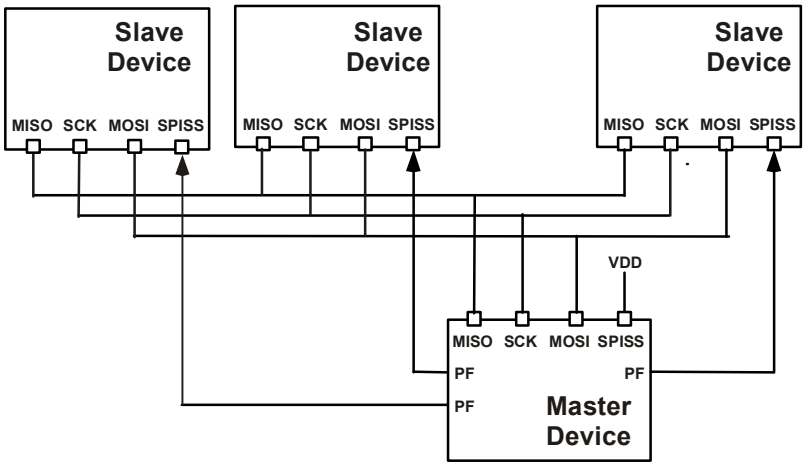


Figure 10-6. Single Master, Multiple Slave Configuration

SPIx Status Register (SPIx_ST)

The SPI Status register (SPIx_ST) is used to detect when an SPI transfer is complete or if transmission/reception errors occur. The SPIx_ST register can be read at any time.

Some of the bits in SPIx_ST are read-only and other bits are sticky. Bits that just provide information 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 SPIx_ST. For example, if the TXE bit is set, the user must write a 1 to bit 2 of SPIx_ST to clear the TXE error condition. This allows the user to read SPIx_ST without changing its value.

Sticky bits can only be cleared by writing a 1 to them. Writing 0 does not have any effect on a sticky bit. This is known as a write-1-to-clear (W1C) bit.

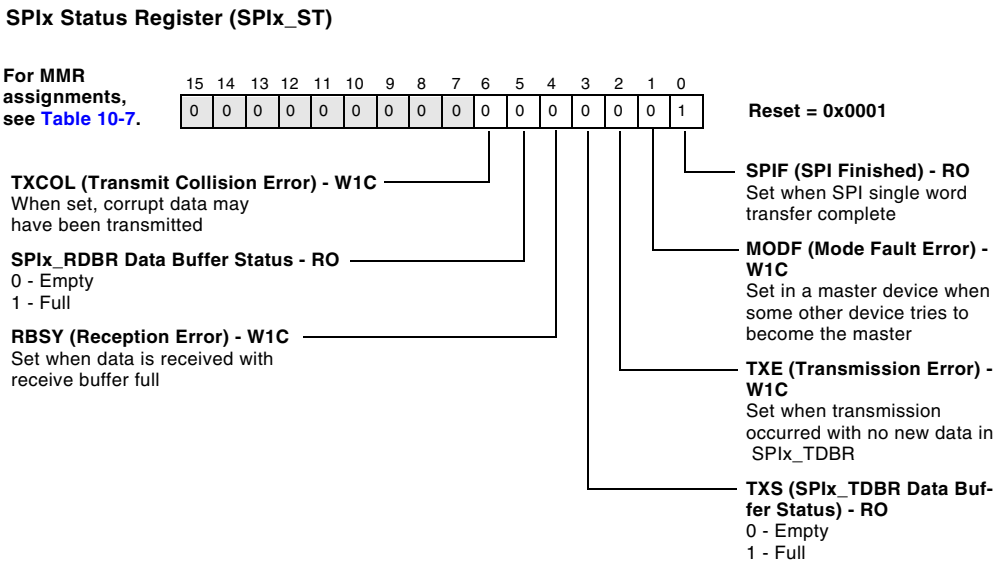


Figure 10-7. SPIx Status Register

Table 10-7. SPIx Status Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_ST	0xFFC0 3004
SPI1_ST	0xFFC0 3404

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.

SPIx Transmit Data Buffer Register (SPIx_TDBR)

The Transmit Data Buffer register (SPIx_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 SPIx_TDBR is loaded into the Shift Data register (SFDR). A read of SPIx_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 SPIx_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 SPIx_TDBR are repeatedly transmitted. A write to SPIx_TDBR is permitted in this mode, and this data is transmitted.

If multiple writes to SPIx_TDBR occur while a transfer is already in progress, only the last data written is transmitted. None of the intermediate values written to SPIx_TDBR are transmitted. Multiple writes to SPIx_TDBR are possible, but not recommended.

SPIx Transmit Data Buffer Register (SPIx_TDBR)

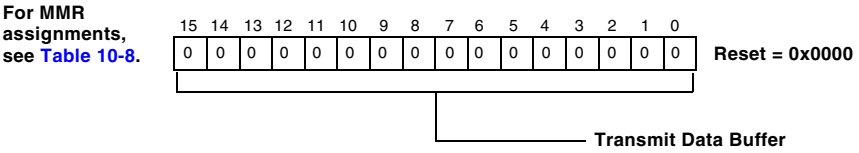


Figure 10-8. SPIx Transmit Data Buffer Register

Table 10-8. SPIx Transmit Data Buffer Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_ST	0xFFC0 3004
SPI1_ST	0xFFC0 3404

SPIx Receive Data Buffer Register (SPIx_RDBR)

The Receive Data Buffer register (SPIx_RDBR) is a 16-bit read-only register. At the end of a data transfer, the data in the shift register is loaded into SPIx_RDBR. During a DMA receive operation, the data in SPIx_RDBR is automatically read by the DMA.

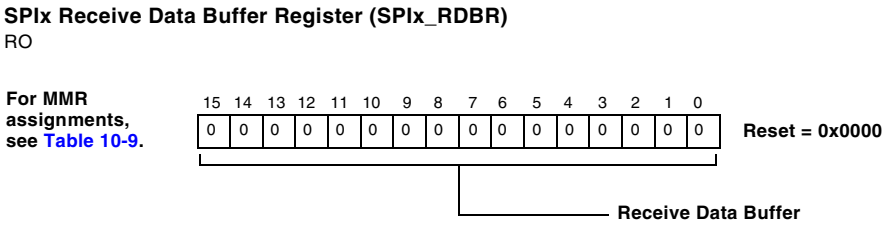


Figure 10-9. SPIx Receive Data Buffer Register

Table 10-9. SPIx Receive Data Buffer Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_RDBR	0xFFC0 3008
SPI1_RDBR	0xFFC0 3408

SPIx RDBR Shadow Register (SPIx_SHADOW)

A shadow register for the receive data buffer, SPIx_RDBR, has been provided for use in debugging software. This register is at a different address than SPIx_RDBR, but its contents are identical to that of SPIx_RDBR. When a software read of SPIx_RDBR occurs, the RXS bit in SPIx_ST is cleared and

an SPI transfer may be initiated (if TIMOD = 00 in SPIx_CTL). No such hardware action occurs when the shadow register is read. SPIx_SHADOW is a read-only register.

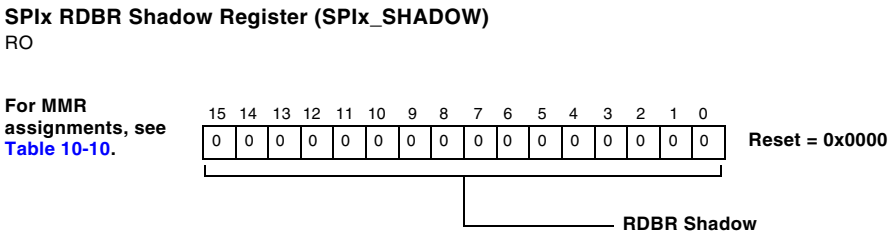


Figure 10-10. SPIx RDBR Shadow Register

Table 10-10. SPIx RDBR Shadow Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_SHADOW	0xFFC0 300C
SPI1_SHADOW	0xFFC0 340C

DMA Registers

The SPI DMA has two basic modes of operation—autobuffer and descriptor. In autobuffer mode, all registers are read/write except the Current Descriptor Pointer register, which is read-only. In this mode, the DMA operates as a simple circular buffer. Autobuffer mode is enabled by writing to bit 4 of the DMA configuration register. In descriptor mode, descriptor work blocks are read from memory. In this mode, many registers and configuration bits are no longer writable, since their values are loaded via descriptors. These include the DMA Start Address and Count registers and DMA Configuration register bits such as TRAN, DATA SIZE BIT 0, DATA SIZE BIT 1, and INTERRUPT ENABLE. Descriptor mode is initiated by writing to the Next Descriptor Pointer register, followed by a write to the DMA Configuration register. For more information about DMA, see “Direct Memory Access” on page 9-1.

SPIx DMA Current Descriptor Pointer Register (SPIx_CURR_PTR)

This 16-bit read-only register holds the lower 16 bits of the pointer to the current descriptor block for the SPI DMA receive operation. The full 32-bit address to the current pointer is formed by concatenating the Next Descriptor Base Pointer register (DB_NDBP) with the DMA Current Descriptor Pointer register.

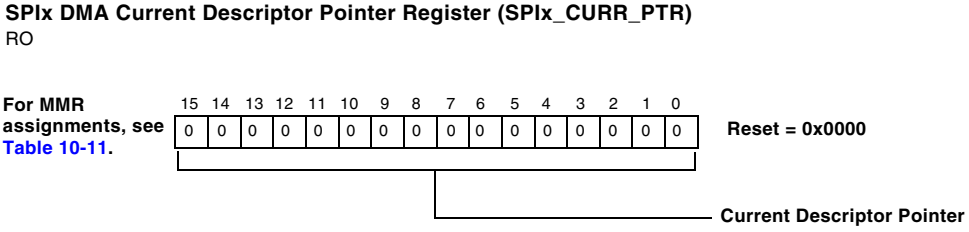


Figure 10-11. SPIx DMA Current Descriptor Pointer Register

Table 10-11. SPIx Current Descriptor Pointer Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_CURR_PTR	0xFFC0 3200
SPI1_CURR_PTR	0xFFC0 3600

SPIx DMA Configuration Register (SPIx_CONFIG)

The SPIx_CONFIG register, shown in Figure 10-12, is one of five registers that make up the descriptor block for a DMA transfer. These registers are accessible through the DMA bus.

SPIx DMA Configuration Register (SPIx_CONFIG)

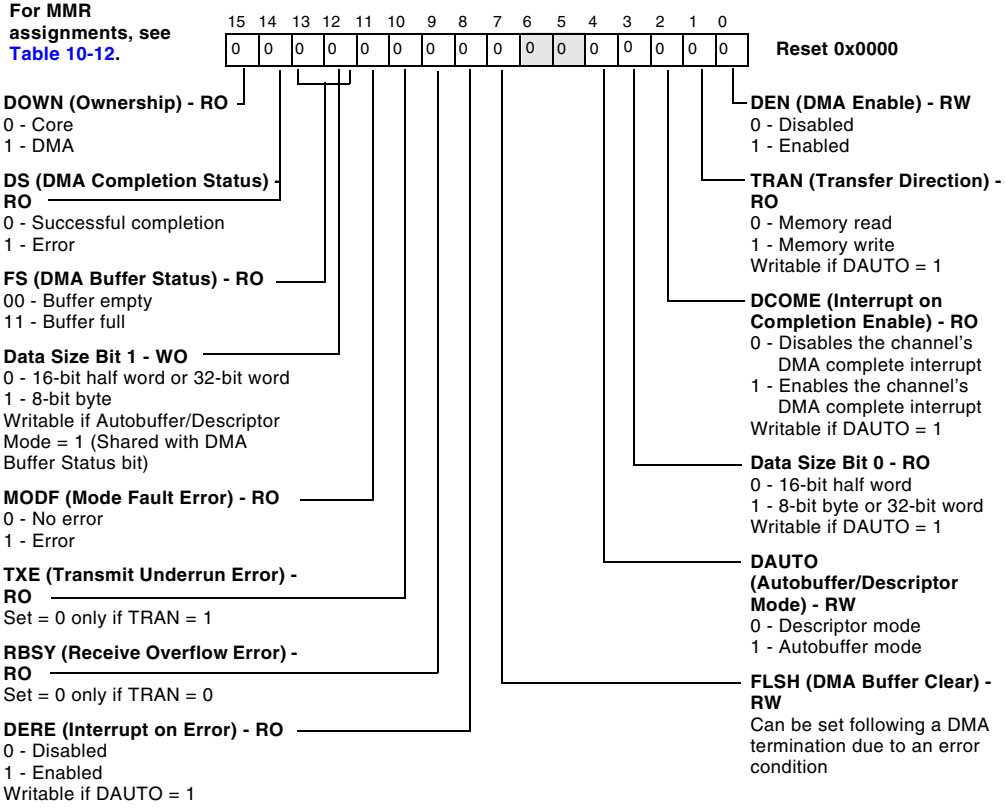


Figure 10-12. SPIx DMA Configuration Register

Table 10-12. SPIx DMA Configuration Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_CONFIG	0xFFC0 3202
SPI1_CONFIG	0xFFC0 3602

SPIx DMA Start Address High Register (SPIx_START_ADDR_HI) and SPIx DMA Start Address Low Register (SPIx_START_ADDR_LO)

The SPIx_START_ADDR_HI and SPIx_START_ADDR_LO registers, shown in Figure 10-13, hold a pointer to the DMA address that is being accessed.

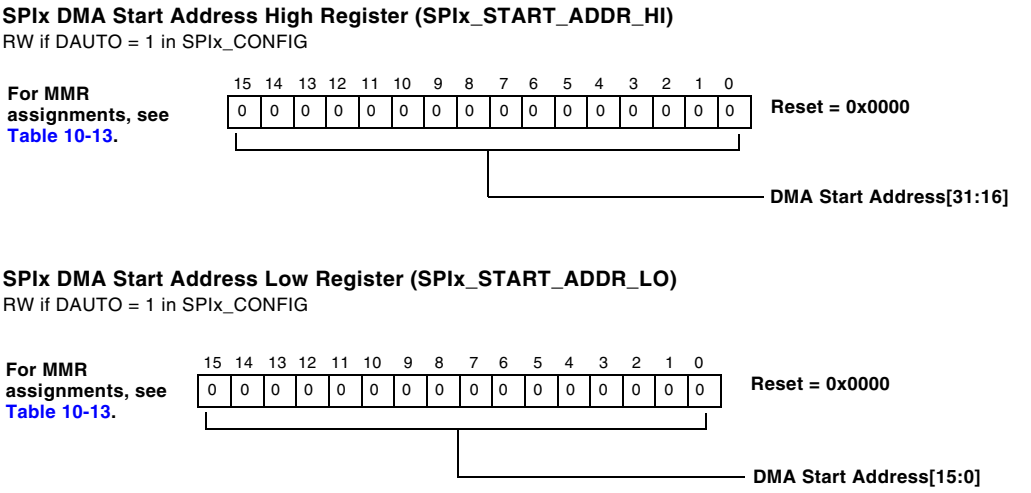


Figure 10-13. SPIx DMA Start Address High and Low Registers

Table 10-13. SPIx Start Address High and SPIx Start Address Low Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_START_ADDR_HI	0xFFC0 3204
SPI1_START_ADDR_HI	0xFFC0 3604
SPI0_START_ADDR_LO	0xFFC0 3206
SPI1_START_ADDR_LO	0xFFC0 3606

SPIx DMA Count Register (SPIx_COUNT)

The `SPIx_COUNT` register, shown in Figure 10-14, holds the block word count (number of remaining words in the transfer).

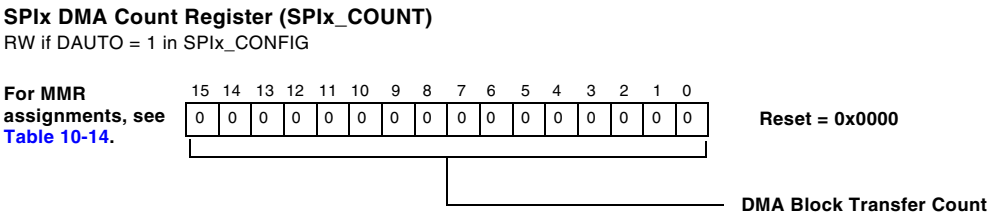


Figure 10-14. SPIx DMA Count Register

Table 10-14. SPIx DMA Count Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_COUNT	0xFFC0 3208
SPI1_COUNT	0xFFC0 3608

SPIx DMA Next Descriptor Pointer Register (SPIx_NEXT_DESCR)

The SPIx_NEXT_DESCR register, shown in Figure 10-15, is used to write to the head of a descriptor list. It holds the address for the next transfer control block in a DMA operation. The Next Descriptor address [31:0], which provides a pointer to the address of the next descriptor block, is the concatenation of two registers:

Next Descriptor Address [31:0] = Next Descriptor Base Pointer [15:0] || Next Descriptor Pointer [15:0]

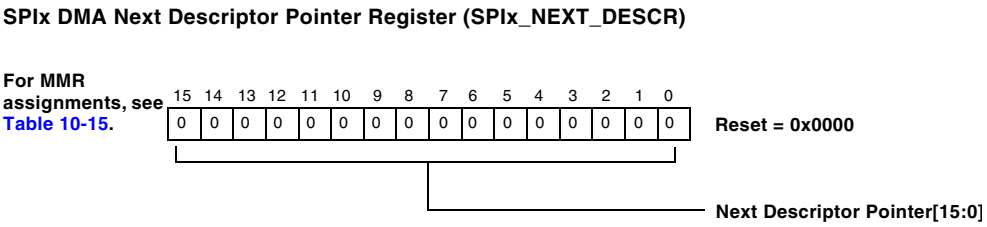


Figure 10-15. DMA Next Descriptor Pointer Register

Table 10-15. SPIx Next Descriptor Pointer Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_NEXT_DESCR	0xFFC0 320A
SPI1_NEXT_DESCR	0xFFC0 360A

SPIx DMA Descriptor Ready Register (SPIx_DESCR_RDY)

The SPIx_DESCR_RDY register, shown in Figure 10-16, is used to reactivate a descriptor fetch in the event of a stall (DOWN = 0). Write a 1 to bit 0 to reactivate a descriptor fetch.

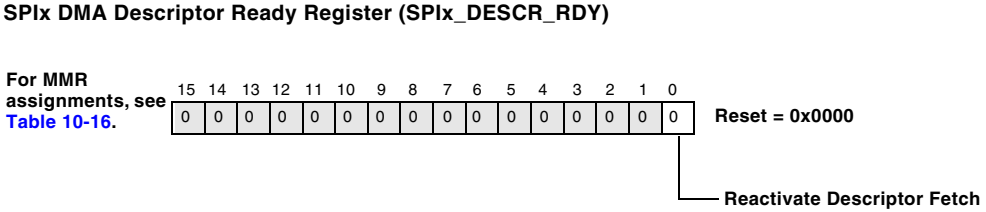


Figure 10-16. SPIx DMA Descriptor Ready Register

Table 10-16. SPIx DMA Descriptor Ready Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_DESCR_RDY	0xFFC0 320C
SPI1_DESCR_RDY	0xFFC0 360C

SPIx DMA Interrupt Register (SPIx_DMA_INT)

The SPIx_DMA_INT register, shown in Figure 10-17, indicates the SPI DMA interrupt status. Write a 1 to the corresponding bit to clear.

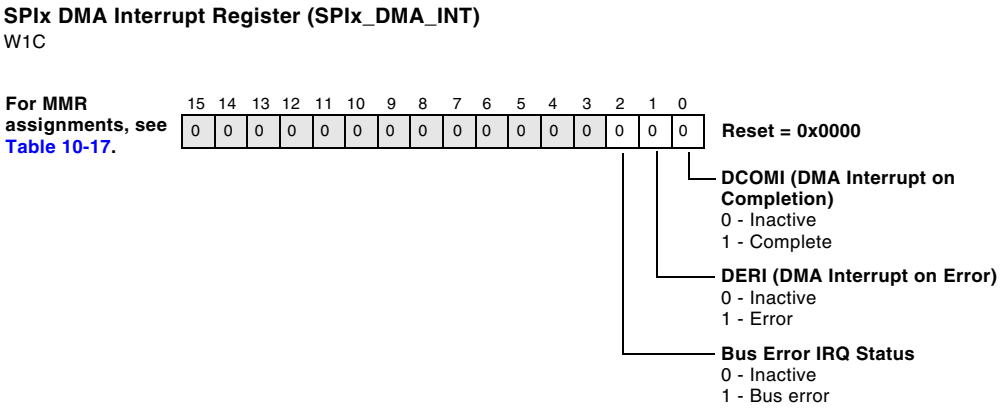


Figure 10-17. SPIx DMA Interrupt Register

Table 10-17. SPIx DMA Interrupt Register MMR Assignments

Register Name	Memory-Mapped Address
SPI0_DMA_INT	0xFFC0 320E
SPI1_DMA_INT	0xFFC0 360E

Register Functions

Table 10-18 shows the functions of the SPI registers.

Table 10-18. SPI Register Mapping

Register Name	Function	Notes
SPIx_CTL	SPI port control	SPE and MSTR bits can also be modified by hardware (when MODF is set)
SPIx_FLG	SPI port flag	
SPIx_ST	SPI port status	SPIF bit can be set by clearing SPE in SPIx_CTL
SPIx_TDBR	SPI port transmit data buffer	Register contents can also be modified by hardware (by DMA and/or when SZ = 1 in SPIx_CTL)
SPIx_RDBR	SPI port receive data buffer	When register is read, hardware events are triggered
SPIx_BAUD	SPI port baud control	
SPIx_SHADOW	SPI port data	Register has the same contents as SPIx_RDBR, but no action is taken when it is read
SPIx_CURR_PTR	SPI port DMA current pointer	Register is always read-only
SPIx_CONFIG	SPI port DMA configuration	Five of the control bits (TRAN, DCOME, DERE, DATA SIZE BITS 0 and 1) can only be written to via software when the DAUTO bit is set
SPIx_START_ADDR_HI	SPI port DMA start address (upper 16 bits)	Register can only be written to via software when the DAUTO bit in SPIx_CONFIG is set
SPIx_START_ADDR_LO	SPI port DMA start address (lower 16 bits)	Register can only be written to via software when the DAUTO bit in SPIx_CONFIG is set
SPIx_COUNT	SPI port DMA count	Register can only be written to via software when the DAUTO bit in SPIx_CONFIG is set
SPIx_NEXT_DESCR	SPI port DMA next descriptor pointer	Register is concatenated with Next Descriptor Base Pointer register to form head address

Table 10-18. SPI Register Mapping (Cont'd)

Register Name	Function	Notes
SPIx_DESCR_RDY	SPI port DMA descriptor ready	Write a 1 to bit 0 to reactivate a descriptor fetch
SPIx_DMA_INT	SPI port interrupt status	All three Interrupt Status bits are sticky; write a 1 to the corresponding bit to clear

SPI Transfer Formats

The ADSP-BF535 SPI supports four different combinations of serial clock phase and polarity, selectable using the CPOL and CPHA bits in SPIx_CTL.

The SPI transfer protocols shown in Figure 10-18 and Figure 10-19 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 MSB first (LSBF = 0). Any combination of the SIZE and LSBF bits of SPIx_CTL is allowed. For example, a 16-bit transfer with 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, SPISS, must be inactive (high) between each serial transfer. This is controlled automatically by the SPI hardware logic. When CPHA = 1, SPISS may either remain active (low) between successive transfers or be inactive (high). This must be controlled by the software by manipulating SPIx_FLG.

Figure 10-18 shows the SPI transfer protocol for CPHA = 0. Note that SCK starts toggling in the middle of the data transfer, SIZE = 0, and LSBF = 0.

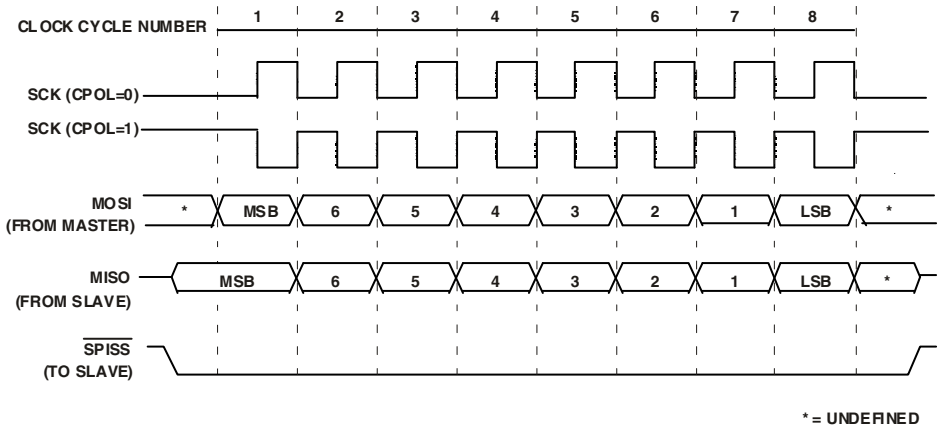


Figure 10-18. SPI Transfer Protocol for CPHA = 0

Figure 10-19 shows the SPI transfer protocol for CPHA = 1. Note that SCK starts toggling at the beginning of the data transfer, SIZE = 0, and LSBF = 0.

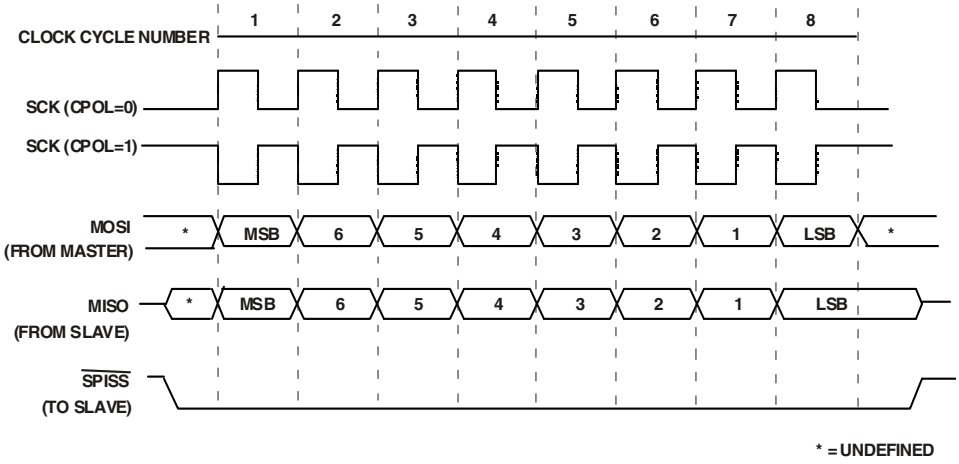


Figure 10-19. SPI Transfer Protocol for CPHA = 1

SPI General Operation

The SPI in ADSP-BF535 processors 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 the 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 (except 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 feature. When WOM is set and the ADSP-BF535 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 ADSP-BF535 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 SPIx_BAUD. For slave devices, the value in SPIx_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 (the *active edge*) and sampled on the opposite edge of the clock (the *sampling edge*). Clock polarity and clock phase relative to data are programmable into SPIx_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.

- The core writes to SPIx_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.
- The core writes to the SPIx_CTL and SPIx_BAUD 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.
- If CPHA = 1, the core activates the desired slaves by clearing one or more of the SPI flag bits (FLGx) of SPIx_FLG.
- The TIMOD bits in SPIx_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 (SPIx_TDBR) or a data read of the receive data buffer (SPIx_RDBR).
- 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 SPIx_TDBR register. At the end of the transfer, the contents of the shift register are loaded into SPIx_RDBR.
- 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 SPIx_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 SPIx_RDBR buffer. If GM = 0 and the receive buffer is full, the incoming data is discarded and SPIx_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 SPIx_CTL. Based on those two bits and the status of the interface, a new transfer is started upon either a read of SPIx_RDBR or a write to SPIx_TDBR. Transfer initiation is summarized in Table 10-19.

Table 10-19. Transfer Initiation

TIMOD	Function	Transfer Initiated Upon	Action, Interrupt
00	Transmit and Receive	Initiate new single-word transfer upon read of SPIx_RDBR and previous transfer completed	Interrupt active when receive buffer is full Read of SPIx_RDBR clears interrupt
01	Transmit and Receive	Initiate new single-word transfer upon write to SPIx_TDBR and previous transfer completed	Interrupt active when transmit buffer is empty Writing to SPIx_TDBR clears interrupt

Table 10-19. Transfer Initiation (Cont'd)

TIMOD	Function	Transfer Initiated Upon	Action, Interrupt
10	Transmit or Receive with DMA	Initiate new multiword transfer upon write to DMA enable bit. Individual word transfers begin with either a DMA write to SPIx_TDBR or a DMA read of SPIx_RDBR (depending on TRAN bit), and last transfer complete	Interrupt active upon DMA error or multiword transfer complete Write 1 to SPIx_DMA_INT register clears interrupt
11	Reserved	N/A	N/A

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 $\overline{\texttt{SPISS}}$ 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:

- The core writes to SPIx_CTL to define the mode of the serial link to be the same as the mode setup in the SPI master.
- To prepare for the data transfer, the core writes data to be transmitted into SPIx_TDBR.
- Once the $\overline{\texttt{SPISS}}$ falling edge is detected, the slave starts sending and receiving data on active SCK edges.
- Reception/transmission continues until $\overline{\texttt{SPISS}}$ is released or until the slave has received the proper number of clock cycles.
- The slave device continues to receive/transmit with each new falling edge transition on $\overline{\texttt{SPISS}}$ and/or active 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 SPIx_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 SPIx_RDBR. If GM = 0 and the receive buffer is full, the incoming data is discarded, and SPIx_RDBR is not updated.

Slave Ready for a Transfer

Table 10-20 shows the actions necessary to prepare the device for a new transfer, when a device is enabled as a slave.

Table 10-20. Transfer Preparation

TIMOD	Function	Action, Interrupt
00	Transmit and Receive	Interrupt active when receive buffer is full Read of SPIx_RDBR clears interrupt
01	Transmit and Receive	Interrupt active when transmit buffer is empty Writing to SPIx_TDBR clears interrupt
10	Transmit or Receive with DMA	Interrupt configured in SPIx_CONFIG Interrupt active upon DMA error or multiword transfer complete Writing 1 to SPI_DMA_INT clears interrupt
11	Reserved	N/A

Error Signals and Flags

The status of a device is indicated by the SPIx_ST register. See "SPIx Status Register (SPIx_ST)" on page 10-15 for more information.

Mode Fault Error (MODF)

The MODF bit is set in SPIx_ST 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 SPIx_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 SPIx_CTL is cleared, configuring the SPI interface as a slave
- The SPE control bit in SPIx_CTL is cleared, disabling the SPI system
- The MODF status bit in SPIx_ST is set
- An SPI 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. The state of the input pin is observable in the Flag Clear register (FIO_FLAG_C) or the Flag Set register (FIO_FLAG_S).

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 ADSP-BF535 processor. To ensure that the slave select output drivers are disabled once a 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 writing to the Flag Direction register (FIO_DIR) 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 SPIx_ST when all the conditions of transmission are met, but there is no new data in SPIx_TDBR (SPIx_TDBR is empty). In this case, the contents of the transmission depend on the state of the SZ bit in SPIx_CTL. The TXE bit is sticky (W1C).

Reception Error (RBSY)

The RBSY flag is set in SPIx_ST when a new transfer is completed before the previous data can be read from SPIx_RDBR. The state of the GM bit in SPIx_CTL determines whether SPIx_RDBR is updated with the newly received data. The RBSY bit is sticky (W1C).

Transmit Collision Error (TXCOL)

The TXCOL flag is set in SPIx_ST when a write to SPIx_TDBR coincides with the load of the shift register. The write to SPIx_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 SPIx_TDBR may not match what was transmitted. This error can easily be avoided by proper software control. The TXCOL bit is sticky (W1C).

The TXCOL bit is never set when the SPI is configured as a slave with CPHA = 0. A collision may occur, but it cannot be detected.

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, the CPHA mode selected, and the transfer initiation mode (TIMOD) selected. For a master SPI with CPHA = 0, a transfer starts when either SPIx_TDBR is written to or SPIx_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 in SPIx_STATUS 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, SPIx_RDBR. RXS is set shortly after the last sampling edge of SCK. The latency is typically a few SCLK cycles and is independent of CPHA, TIMOD, and the baud rate. If configured to generate an interrupt when SPIx_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 set at the same time as RXS. For a master device, SPIF is set one-half SCK period after the last SCK edge, regardless of CPHA or CPOL.

Thus, 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 (SPIx_BAUD < 4). SPIF is set before RXS is set, and consequently before new data is latched

into SPIx_RDBR, because of the latency. Therefore, for SPIx_BAUD = 2 or SPIx_BAUD = 3, RXS must be set before SPIF to read SPIx_RDBR. For larger SPIx_BAUD settings, RXS is guaranteed to be set before SPIF is set.

11 SERIAL PORT CONTROLLERS

The ADSP-BF535 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 multi-processor system.

In this text, the naming conventions for registers and pins use a lowercase `x` to represent a digit. In this chapter, for example, the name `DTx` pins indicates `DT0` and `DT1` (corresponding to SPORT0 and SPORT1, respectively).

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 ADSP-BF535 processor provides asynchronous RS-232 data transfer via the UARTs. Each SPORT is a full duplex device, capable of simultaneous data transfer in both directions. Each SPORT has one group of pins (data, clock, and frame sync) for transmit and a second set of pins for receive. The receive and transmit functions are programmed separately. The SPORTs can be programmed for bit rate, frame sync, and bits per word by writing to memory-mapped registers.

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 1/2 the full clock rate of SCLK (where SCLK is the peripheral clock). Independent transmit and receive functions provide greater flexibility for serial communications. SPORT data can be automatically transferred to and from on-chip memory using DMA block transfers. Additionally, each of the SPORTs offers a TDM (time division multiplexed) multichannel mode.

SPORT clocks and frame syncs can be internally generated by the core or received from an external source. The SPORTs can operate with little endian or big endian transmission formats, with word lengths selectable from 3 to 16 bits. They offer selectable transmit modes and optional µ-law or A-law companding in hardware.

Each of the SPORTs offers these features and capabilities:

- Provides independent transmit and receive functions.
- Transfers serial data words from three to sixteen bits in length, either most significant bit first (MSB) or least significant bit first (LSB).
- Double buffers data (both receive and transmit functions have a data buffer register and a shift register), providing additional time to service the SPORT.
- Performs A-law and µ-law hardware companding on transmitted and received words. (See "Companding" on page 11-52 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.

- Performs interrupt-driven, single word transfers to and from on-chip memory under core 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 channels of a time-division-multiplexed serial bitstream multiplexed into up to 128 channels. This mode can be useful as a network communication scheme for multiple processors.
- 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.

Table 11-1 shows the pins for each SPORT.

Table 11-1. Serial Port (SPORT) Pins

Pin	Description
DTx	Transmit Data
DRx	Receive Data
TCLKx	Transmit Clock
RCLKx	Receive Clock
TFSx	Transmit Frame Sync
RFSx	Receive Frame Sync

A SPORT receives serial data on its DR input and transmits serial data on its DT output. It can receive and transmit simultaneously for full duplex operation. For both transmit and receive data, the data bits (DR or DT) are synchronous to the serial clocks (RCLK or TCLK); this is an output if the processor generates this clock or an input if the clock is externally generated. Frame synchronization signals RFS and TFS are used to indicate the start of a serial data word or stream of serial words.

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.

Figure 11-1 shows a simplified block diagram of a single SPORT. Data to be transmitted is written from an internal processor register to the SPORT's memory-mapped SPORTx_TX register. This data is optionally compressed by the hardware then automatically transferred to the transmit shift register. The bits in the shift register are shifted out on the SPORT's DT pin, MSB first or LSB first, synchronous to the serial clock on the TCLK pin. The receive portion of the SPORT accepts data from the DR pin synchronous to the serial clock. When an entire word is received, the data is optionally expanded, then automatically transferred to the SPORT's memory-mapped SPORTx_RX register, where it is available to the processor.

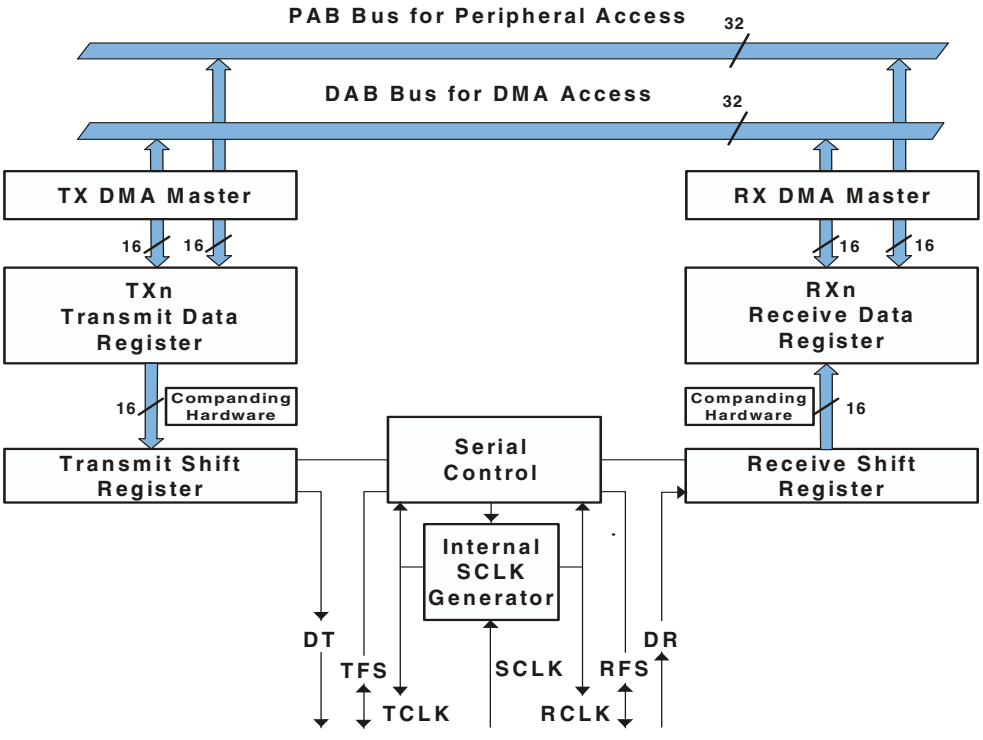


Figure 11-1. SPORT Block Diagram

Figure 11-2 shows the port connections for the ADSP-BF535 processor SPORTs.

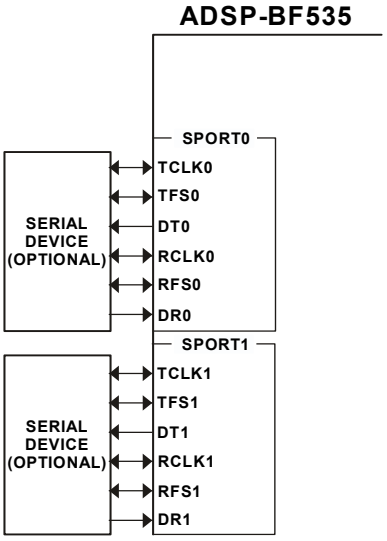


Figure 11-2. SPORT Connections

SPORT Operation

This section provides an example of SPORT operation, illustrating the most common use of a SPORT. Since the SPORT functionality is configurable, this example 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 to the internal transmit shift register. The bits are then sent, beginning with either the MSB or the LSB as specified. Each bit is shifted out on the rising edge of SCK. After the first bit of a word has been trans-

ferred, the SPORT generates the transmit interrupt. The SPORTx_TX register is then available for the next data word, even though the transmission of the first word continues.

As a SPORT receives bits, they accumulate in an internal receive register. When a complete word has been received, it is written to the SPORTx_RX register and the receive interrupt for that SPORT is generated. Interrupts are generated differently if DMA block transfers are performed. For information about DMA, see "Direct Memory Access" on page 9-1.

SPORT Disable

The SPORTs are automatically disabled by a 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_TX_CONFIG control register and RSPEN in the SPORTx_RX_CONFIG control register, respectively). Each method has a different effect on the SPORT.

A reset disables the SPORTs by clearing the SPORTx_TX_CONFIG and SPORTx_RX_CONFIG control registers (including the TSPEN and RSPEN enable bits) and the TSCLKDIVx, RSCLKDIVx, SPORTx_TFSDIVx, and SPORTx_RFSDIVx clock and frame sync divisor registers. Any ongoing operations are aborted.

Disabling 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 after the SPORT has been enabled, set the SPORT to receive an external clock.

The SPORTs are ready to start transmitting or receiving data three serial clock cycles after they are enabled in the SPORTx_TX_CONFIG or SPORTx_RX_CONFIG control register. No serial clock cycles are lost from this point on.

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 the SPORTx_TSCLKDIV and SPORTx_RSCLKDIV registers, which can be modified while the SPORT is enabled). To change values in all other SPORT configuration registers, disable the SPORT by clearing TSPEN in SPORTx_TX_CONFIG and/or RSPEN in SPORTx_RX_CONFIG.

Each SPORT has its own set of control registers and data buffers. These registers are described in detail in the following sections.

The SPORT control registers are programmed by writing to the appropriate address in memory. All control and status bits in the SPORT registers are active high unless otherwise noted.

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 TSCLKDIV/RSCLKDIV registers and multichannel configuration registers.

SPORT Registers

The following sections describe the SPORT registers.

Transmit and Receive Configuration Registers (SPORTx_TX_CONFIG, SPORTx_RX_CONFIG)

The main control registers for each SPORT are the transmit configuration register, SPORTx_TX_CONFIG, shown in Figure 11-3 on page 11-10, and the receive configuration register, SPORTx_RX_CONFIG, shown in Figure 11-4 on page 11-14.

A SPORT is enabled for transmit if Bit 0 (TSPEN) of the Transmit Configuration register is set to 1; it is enabled to receive if Bit 0 (RSPEN) of the Receive Configuration register is set to 1. Both of these bits are cleared during either a hard reset or a soft reset, disabling all SPORT channels.

When the SPORT is enabled to transmit (TSPEN set) or receive (RSPEN set), corresponding SPORT configuration register writes are disabled except for SPORTx_RSCLKDIV, SPORTx_TSCLKDIV, and multichannel mode channel enable registers. Writes are always enabled to the SPORTx_TX buffer. SPORTx_RX is a read-only register.

After a write to a SPORT register, any changes to the control and mode bits generally take effect when the SPORT is re-enabled.

When changing operating modes, a SPORT control register should be cleared before the new mode is written to the register.

The TXS status bit in the SPORT Status register indicates whether the SPORTx_TX buffer is full (1) or empty (0).

The Transmit Underflow Status bit (TUVF) in the SPORT Status register is set whenever the TFS signal occurs from either an external or an internal source while the SPORTx_TX buffer 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.

When DITFS=0 (the default), the internal transmit frame sync signal (TFS) is dependent upon new data being present in the SPORTx_TX buffer; the TFS signal is only generated for new data. Setting DITFS to 1 selects data

independent frame syncs. This causes the TFS signal to be generated whether or not new data is present, transmitting the contents of the SPORTx_TX buffer regardless. SPORT DMA typically keeps the SPORTx_TX buffer full, and when the DMA operation is complete, the last word in SPORTx_TX is continuously transmitted. For information about DMA, see "Direct Memory Access" on page 9-1.

The SPORTx_TX_CONFIG and SPORTx_RX_CONFIG registers control the SPORTs' operating modes for the I/O processor.

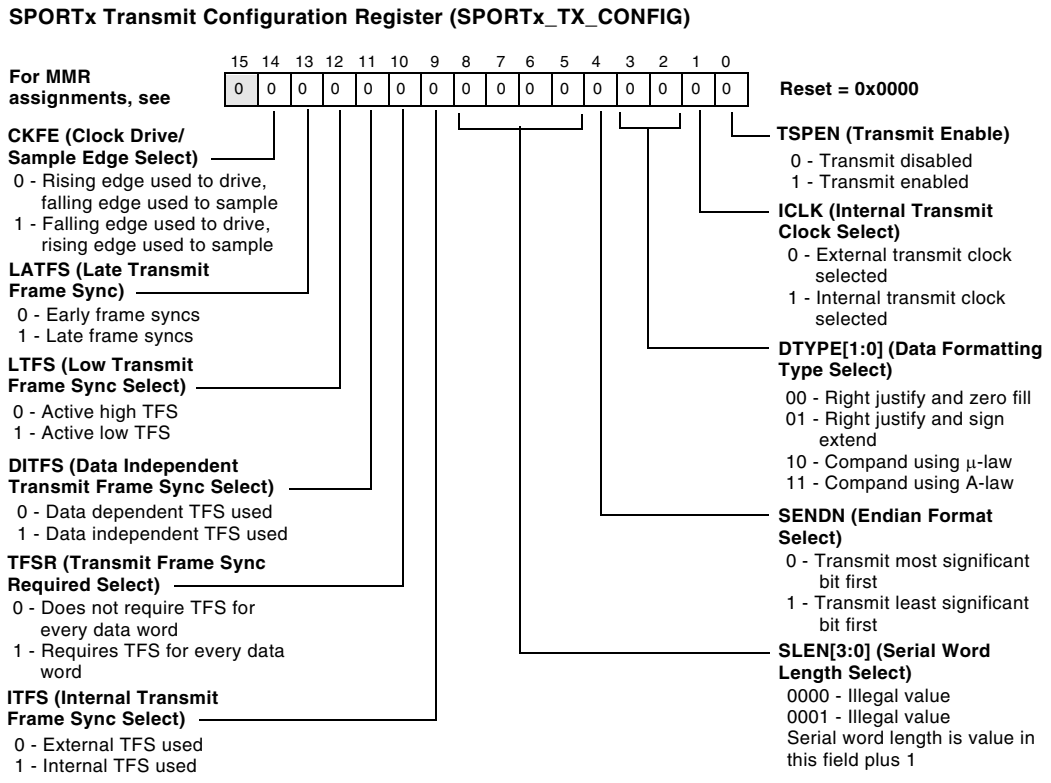


Figure 11-3. SPORTx Transmit Configuration Register

Table 11-2. SPORTx Transmit Configuration Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_TX_CONFIG	0xFFC0 2800
SPORT1_TX_CONFIG	0xFFC0 2C00

Additional information for the SPORTx_TX_CONFIG transmit configuration register bits:

- Transmit Enable. SPORTx_TX_CONFIG[0] (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 TD write code in the TX interrupt service routine. For this reason, the code should initialize the interrupt service routine and be ready to service TX interrupts before setting TSPEN.

  Clearing TSPEN causes the SPORT to stop driving data 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.

- Data Formatting Type Select. SPORTx_TX_CONFIG[3:2] (DTYPE). The DTYPE, SENDN, and SLEN bits configure the format of the data words transmitted over the SPORTs. The two DTYPE bits specify one of four data formats used for single and multichannel operation (00=right justify and zero fill unused most significant bits, 01=right justify and sign extend into unused most significant bits, 10=compand using μ-law, 11=compand using A-law).

- Endian Format Select. SPORTx_TX_CONFIG[4] (SENDN). The DTYPE, SENDN, and SLEN bits configure the format of the data words transmitted over the SPORTs. The SENDN bit selects the endian format (0=serial words are transmitted MSB first, 1=serial words are transmitted LSB first).

- Serial Word Length Select. SPORTx_TX_CONFIG[8:5] (SLEN). The DTYPE, SENDN, and SLEN bits configure the format of the data words transmitted over the SPORTs. 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 = SLEN + 1;
  ```

  The SLEN field can be set to a value of 2 to 15; 0 and 1 are illegal values for this field. Two common settings for the SLEN field are 15, to transmit a full 16-bit word, and 7, to transmit an 8-bit byte. The ADSP-BF535 processor is a 16-bit processor, so program instruction or DMA engine loads of the TX data register always move 16 bits into the register; the SLEN field tells the SPORT how many of those 16 bits to shift out of the register over the serial link.

The frame sync signal is controlled by the Frame Sync Divider register, 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. SPORTx_TX_CONFIG[9] (ITFS). This bit selects whether the SPORT uses an internal TFS (if set) or an external TFS (if cleared).

- Transmit Frame Sync Required Select. SPORTx_TX_CONFIG[10] (TFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a transfer frame sync for every data word.

The TFSR bit is normally set. A frame sync pulse is used to mark the beginning of each word or data packet, and most systems need frame sync to function properly.

- Data Independent Transmit Frame Sync Select. SPORTx_TX_CONFIG[11] (DITFS). This bit selects whether the SPORT uses a data independent TFS (sync at selected interval, if set) or uses a data dependent TFS (sync when data in SPORTx_TX, if cleared).

  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 TX register has been loaded or not; if DITFS is cleared, the frame sync pulse is only generated if the TX data register has been loaded. If the receiver demands regular frame sync pulses, DITFS should be set, and the core 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 core is late in loading the TX register.

- Low Transmit Frame Sync Select. SPORTx_TX_CONFIG[12] (LTFS). This bit selects an active low TFS (if set) or active high TFS (if cleared).

- Late Transmit Frame Sync. SPORTx_TX_CONFIG[13] (LATFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).

- Clock Drive/Sample Edge Select. SPORTx_TX_CONFIG[14] (CKFE). 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.

SPORTx Receive Configuration Register (SPORTx_RX_CONFIG)

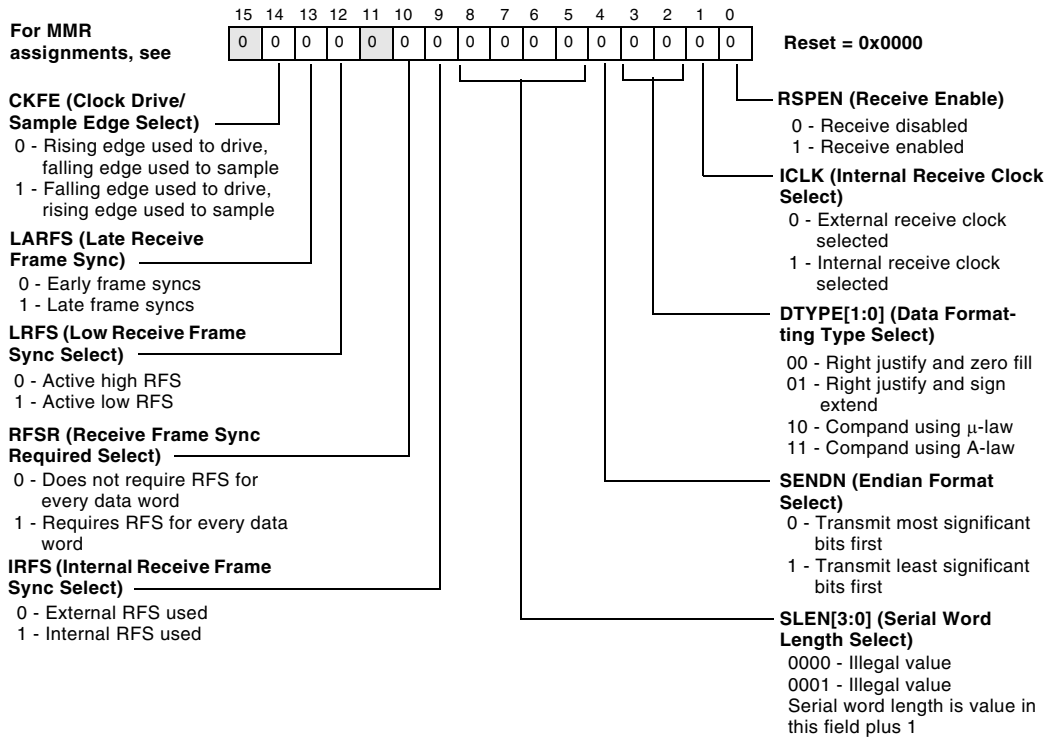


Figure 11-4. SPORTx Receive Configuration Register

Table 11-3. SPORTx Receive Configuration Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_RX_CONFIG	0xFFC0 2802
SPORT1_RX_CONFIG	0xFFC0 2C02

Additional information for the SPORTx_RX_CONFIG receive configuration register bits:

- Receive Enable. SPORTx_RX_CONFIG[0] (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 DRx pin as well as the RX bit clock and receive frame sync pins if so programmed.

All SPORT control registers should be programmed before RSPEN is set. Typical SPORT initialization code first writes SPORTx_RX_CONFIG with everything except RSPEN, then the last step in the code is to rewrite SPORTx_RX_CONFIG with all of the necessary bits including RSPEN.

Setting RSPEN enables the SPORT RX interrupt. For this reason, the code should initialize the interrupt service routine and be ready to service RX interrupts before setting RSPEN.

Clearing RSPEN causes the SPORT to stop receiving data; it also shuts down the internal SPORT circuitry. In low power applications, battery life can be extended by clearing RSPEN whenever the SPORT is not in use.

- Internal Receive Clock Select. SPORTx_RX_CONFIG[1] (ICLK). This bit selects the internal receive clock (if set) or external receive clock (if cleared).

- Data Formatting Type Select. SPORTx_RX_CONFIG[3:2] (DTYPE). The DTYPE, SENDN, and SLEN bits configure the format of the data words received over the SPORTs. The two DTYPE bits specify one of four data formats used for single and multichannel operation

(00=right justify and zero fill unused most significant bits, 01=right justify and sign extend into unused MSBs, 10=compand using μ-law, 11=compand using A-law).

- Endian Format Select. SPORTx_RX_CONFIG[4] (SENDN). The DTYPE, SENDN, and SLEN bits configure the format of the data words received over the SPORTs. The SENDN bit selects the endian format (0=serial words are received MSB first, 1=serial words are received LSB first).
- Serial Word Length Select. SPORTx_RX_CONFIG[8:5] (SLEN). The DTYPE, SENDN, and SLEN bits configure the format of the data words received over the SPORTs. 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 15; 0 and 1 are illegal values for this field.
- Internal Receive Frame Sync Select. SPORTx_RX_CONFIG[9] (IRFS). This bit selects whether the SPORT uses an internal RFS (if set) or an external RFS (if cleared).
- Receive Frame Sync Required Select. SPORTx_RX_CONFIG[10] (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. SPORTx_RX_CONFIG[12] (LRFS). This bit selects an active low RFS (if set) or active high RFS (if cleared).
- Late Receive Frame Sync. SPORTx_RX_CONFIG[13] (LARFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
- Clock Drive/Sample Edge Select. SPORTx_RX_CONFIG[14] (CKFE). This bit selects which edge of the RCLK 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.

SPORTx Transmit (SPORTx_TX) Registers

The SPORTx_TX register, shown in Figure 11-5, acts as the transmit data buffer for the SPORT. It is a 16-bit register which must be loaded with the data to be transmitted; the data is loaded either by the DMA controller or by the program running on the core. Word lengths of less than 16 bits are right justified.

The SPORT transmit registers act like a two-location FIFO because there is a data register (SPORTx_TX) with an accompanying output shift register as shown in Figure 11-1 on page 11-5. Two 16-bit words may be stored in these registers at any one time. When the SPORTx_TX register is loaded and any previous word has been transmitted, the contents are automatically loaded into the output shifter. An interrupt is generated when the output shifter has been loaded, signifying that the SPORTx_TX register is ready to accept the next word (the SPORTx_TX buffer is “not full”). This interrupt does not occur if SPORT DMA is enabled.

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, TUVF is set whenever the serial shift register is not loaded, when that transmission should begin on an enabled channel. The TUVF status bit is cleared on a hard reset. From that moment on, if it is set, it stays set until the SPORT is re-enabled (disabled and then enabled again). Once the SPORT is enabled, it takes at least 4 transmit clock cycles to clear the TUVF bit.

Once the TUVF bit is cleared, the DMA's Peripheral-Dependent Error IRQ Status bit in the DMA IRQ Status register can be cleared by writing 1 to it. See "Peripheral DMA IRQ Status Register" on page 9-28. Follow these steps:

1. Disable the SPORT.
2. Enable the SPORT.
3. Poll the TUVF bit of the SPORTx_STAT register, waiting for a 0.
4. Write 1 to the Peripheral-Dependent Error IRQ Status bit to clear it.

If the program causes the core processor to attempt a write to a full SPORTx_TX register, the new data overwrites the SPORTx_TX register. If it is not known whether the core processor can access the SPORTx_TX register without causing such an error, the register's full or empty status should be read first (in the SPORT Status register) to determine if the access can be made. See "SPORTx Status (SPORTx_STAT) Registers" on page 11-23.

The SPORTx_TX register can be read whether or not the SPORT is enabled.

SPORTx Transmit Registers (SPORTx_TX)

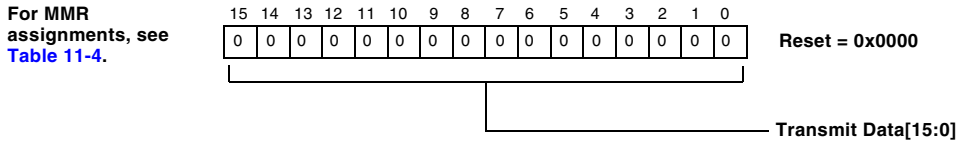


Figure 11-5. SPORTx Transmit Registers

Table 11-4. SPORTx Transmit Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_TX	0xFFC0 2804
SPORT1_TX	0xFFC0 2C04

SPORTx Receive (SPORTx_RX) Registers

The SPORTx_RX register, shown in Figure 11-6, acts as the receive data buffer for the SPORT. It is a 16-bit register which is automatically loaded from the receive shifter when a complete word has been received. Word lengths of less than 16 bits are right justified.

The SPORT receive registers act like a two-location FIFO buffer because there is a data register (SPORTx_RX) with an accompanying input shift register as shown in Figure 11-1 on page 11-5. Two 16-bit words may be stored in these registers at any one time. SPORTx_RX is read-only and the reset values are undefined.

Two 16-bit words can be stored in the SPORT receive registers at any one time. The third word overwrites the second if the first word has not been read out by the Master core or the DMA controller. When this happens, the receive overflow status bit (ROVF) is set in the SPORT Status register. The overflow status is generated on the last bit of the second word. The ROVF status bit is sticky and is only cleared by disabling the serial port.

An interrupt is generated when the SPORTx_RX register has been loaded with a received word (that is, the SPORTx_RX register is not empty). This interrupt is masked out if serial port DMA is enabled.

If the program causes the core processor to attempt a read from an empty SPORTx_RX register, any old data is read. If it is not known whether the core processor can access the SPORTx_RX register without causing such an error, the register's full or empty status should be read first (in the SPORT Status register) to determine if the access can be made.

The RXS and TXS status bits in the SPORT Status register are updated upon reads and writes from the core processor, even when the SPORT is disabled.

SPORTx Receive Registers (SPORTx_RX)
RO

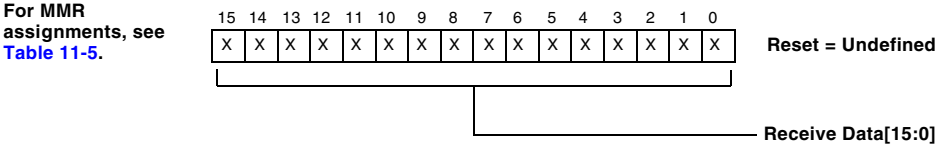


Figure 11-6. SPORTx Receive Registers

Table 11-5. SPORTx Receive Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_RX	0xFFC0 2806
SPORT1_RX	0xFFC0 2C06

SPORTx Transmit (SPORTx_TSCLKDIV) and Receive (SPORTx_RSCLKDIV) Serial Clock Divider Registers

The frequency of an internally generated clock is a function of the system clock frequency (as determined by SSEL, MSEL, and DF) and the value of the 16-bit serial clock divide modulus registers: SPORTx_TSCLKDIV and SPORTx_RSCLKDIV. These registers are shown in Figure 11-7 and Figure 11-8, respectively.

SPORTx Transmit Serial Clock Divider Register (SPORTx_TSCLKDIV)

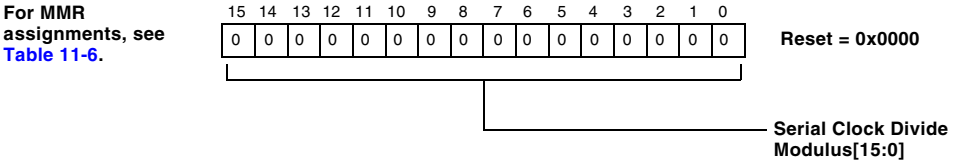


Figure 11-7. SPORTx Transmit Serial Clock Divider Register

Table 11-6. SPORTx Transmit Serial Clock Divider Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_TSCLKDIV	0xFFC0 2808
SPORT1_TSCLKDIV	0xFFC0 2C08

SPORTx Receive Serial Clock Divider Register (SPORTx_RSCLKDIV)

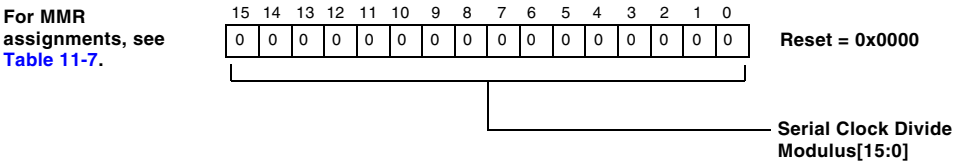


Figure 11-8. SPORTx Receive Serial Clock Divider Register

Table 11-7. SPORTx Receive Serial Clock Divider Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_RSCLKDIV	0xFFC0 280A
SPORT1_RSCLKDIV	0xFFC0 2C0A

SPORTx Transmit (SPORTx_TFSDIV) and Receive (SPORTx_RFSDIV) Frame Sync Divider Registers

These 16-bit registers 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. The SPORTx_TFDIV register is shown in Figure 11-9; the SPORTx_RFSDIV register is shown in Figure 11-10.

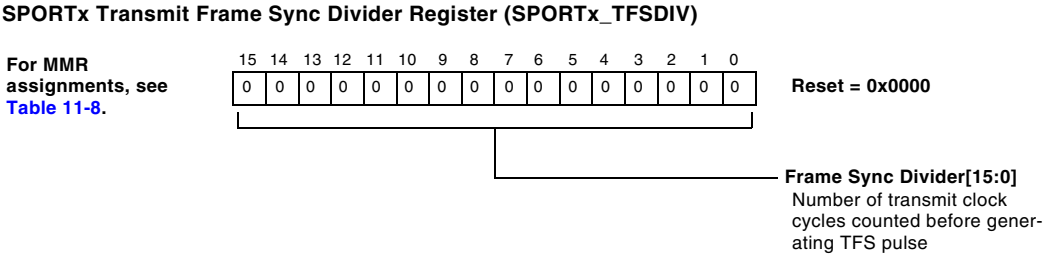


Figure 11-9. SPORTx Transmit Frame Sync Divider Register

Table 11-8. SPORTx Transmit Frame Sync Divider Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_TFSDIV	0xFFC0 280C
SPORT1_TFSDIV	0xFFC0 2C0C

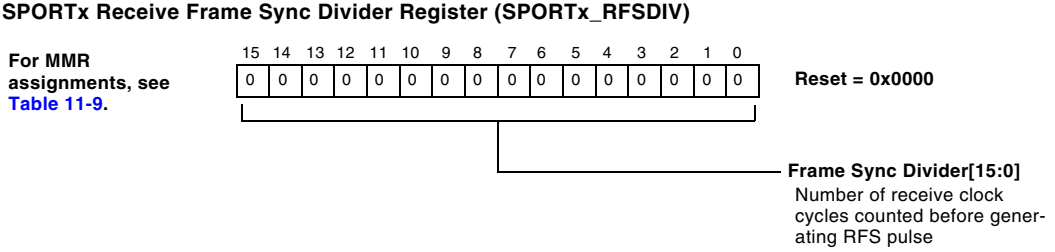


Figure 11-10. SPORTx Receive Frame Sync Divider Register

Table 11-9. SPORTx Receive Frame Sync Divider Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_RFSDIV	0xFFC0 280E
SPORT1_RFSDIV	0xFFC0 2C0E

SPORTx Status (SPORTx_STAT) Registers

The RXS and TXS status bits in the SPORT Status register are updated upon reads and writes from the core processor even when the serial port is disabled. The SPORT Status register is used to determine if the access to a SPORT RX or TX buffer can be made by determining their full or empty status. It is a read-only register. The reset value is undefined. The SPORTx Status register is shown in Figure 11-11.

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 SPORTx_TX register. The TUVF status bit is sticky and is only cleared by disabling the serial port.

When the SPORT RX buffer is full, the receive overflow status bit (ROVF) is set in the SPORT Status register. The overflow status is generated on the last bit of the second word. The ROVF status bit is sticky and is only cleared by disabling the serial port.

The 7-bit CHNL field is the read-only status indicator that shows which channel is currently selected during multichannel operation. CHNL[6:0] increments by one as each channel is serviced. Note that in Channel Select Offset mode, the CHNL value is reset to 0 after the offset has been completed. For example, with offset equal to 21 and a window of 8, in the regular mode the counter displays a value between 0 and 28, while in Channel Select Offset mode, the counter resets to 0 after counting up to 21, and then the frame completes when the CHNL bit reaches the 8th channel (value of 7).

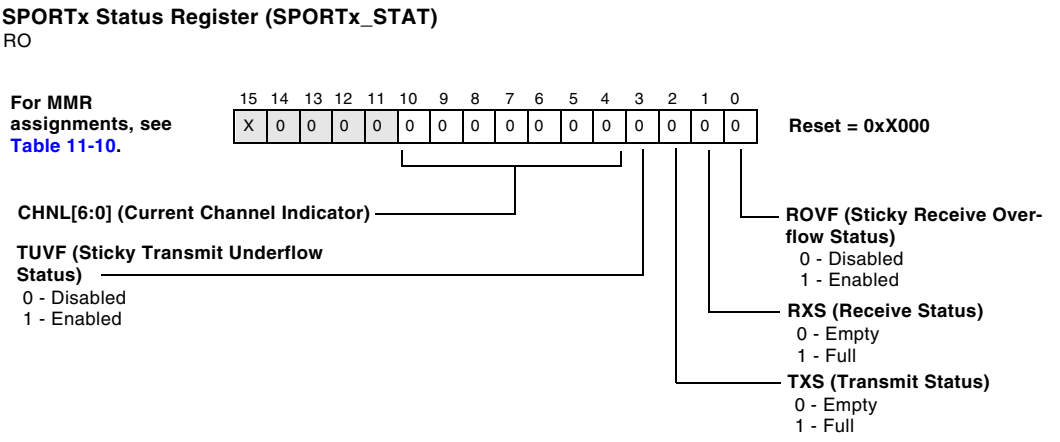


Figure 11-11. SPORTx Status Register

Table 11-10 lists the MMR assignments for the SPORTx Status registers.

Table 11-10. SPORTx Status Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_STAT	0xFFC0 2810
SPORT1_STAT	0xFFC0 2C10

SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers

The multichannel selection registers are used to enable and disable individual channels. The `SPORTx_MTCSx` register, shown in Figure 11-12, specifies the active transmit channels. Each register has 16 bits, corresponding to the 16 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_MTCSx register causes the serial port to transmit the word in that channel's position of the data stream. Clearing the bit in the SPORTx_MTCSx register causes the serial port's DT (data transmit) pin to three-state during the time slot of that channel.

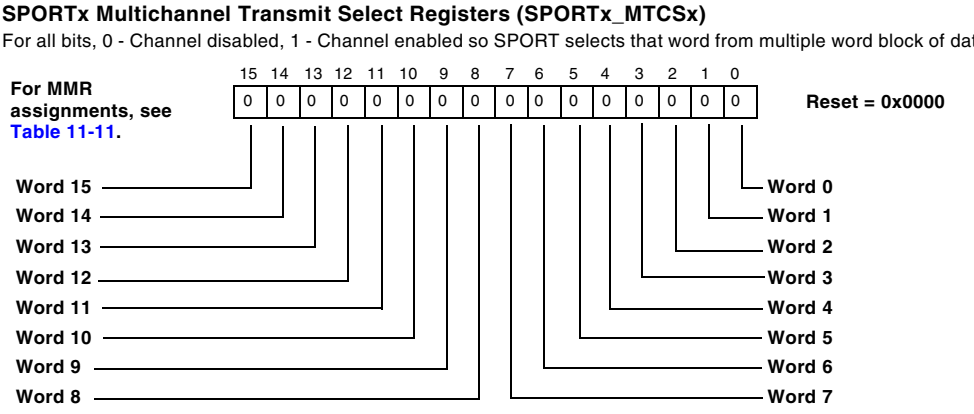


Figure 11-12. SPORTx Multichannel Transmit Select Registers

Table 11-11. SPORTx Multichannel Transmit Select Register MMR Assignments

SPORT0 Register Name	SPORT0 Memory-Mapped Address	SPORT1 Register Name	SPORT1 Memory-Mapped Address
SPORT0_MTCS0	0xFFC0 2812	SPORT1_MTCS0	0xFFC0 2C12
SPORT0_MTCS1	0xFFC0 2814	SPORT1_MTCS1	0xFFC0 2C14
SPORT0_MTCS2	0xFFC0 2816	SPORT1_MTCS2	0xFFC0 2C16
SPORT0_MTCS3	0xFFC0 2818	SPORT1_MTCS3	0xFFC0 2C18
SPORT0_MTCS4	0xFFC0 281A	SPORT1_MTCS4	0xFFC0 2C1A
SPORT0_MTCS5	0xFFC0 281C	SPORT1_MTCS5	0xFFC0 2C1C
SPORT0_MTCS6	0xFFC0 281E	SPORT1_MTCS6	0xFFC0 2C1E
SPORT0_MTCS7	0xFFC0 2820	SPORT1_MTCS7	0xFFC0 2C20

SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers

The multichannel selection registers are used to enable and disable individual channels. The SPORTx_MRCSx register, shown in Figure 11-13, specifies the active receive channels. Each register has 16 bits, corresponding to the 16 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_MRCSx register causes the serial port to receive the word in that channel's position of the data stream; the received word is loaded into the RX buffer. Clearing the bit in the SPORTx_MRCSx register causes the serial port to ignore the data.

SPORTx Multichannel Receive Select Registers (SPORTx_MRCSx)

For all bits, 0 - Channel disabled, 1 - Channel enabled so SPORT selects that word from multiple word block of data.

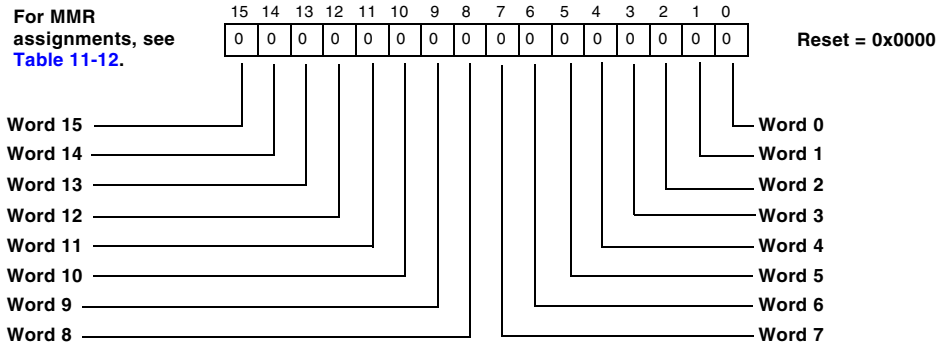


Figure 11-13. SPORTx Multichannel Receive Select Registers

Table 11-12. SPORTx Multichannel Receive Select Register MMR Assignments

SPORT0 Register Name	SPORT0 Memory-Mapped Address	SPORT1 Register Name	SPORT1 Memory-Mapped Address
SPORT0_MRCS0	0xFFC0 2822	SPORT1_MRCS0	0xFFC0 2C22
SPORT0_MRCS1	0xFFC0 2824	SPORT1_MRCS1	0xFFC0 2C24
SPORT0_MRCS2	0xFFC0 2826	SPORT1_MRCS2	0xFFC0 2C26
SPORT0_MRCS3	0xFFC0 2828	SPORT1_MRCS3	0xFFC0 2C28
SPORT0_MRCS4	0xFFC0 282A	SPORT1_MRCS4	0xFFC0 2C2A
SPORT0_MRCS5	0xFFC0 282C	SPORT1_MRCS5	0xFFC0 2C2C
SPORT0_MRCS6	0xFFC0 282E	SPORT1_MRCS6	0xFFC0 2C2E
SPORT0_MRCS7	0xFFC0 2830	SPORT1_MRCS7	0xFFC0 2C30

SPORTx Multichannel Configuration (SPORTx_MCMCx) Registers

There are two SPORTx_MCMCx registers for each SPORT. The SPORTx_MCMCx registers, represented in Figure 11-14 and Figure 11-15, are used to enable multichannel mode. Setting the MCM bit enables multichannel operation for both receive and transmit sides of the SPORT. A transmitting SPORT must therefore be in multichannel mode if the receiving SPORT is 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 T1 interface devices.

A value of zero 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.

SPORTx Multichannel Configuration 1 Registers (SPORTx_MCMC1)

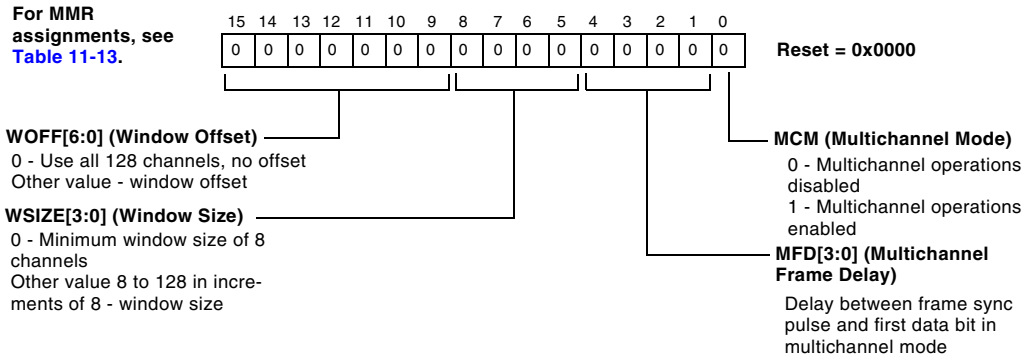


Figure 11-14. SPORTx Multichannel Configuration 1 Registers

Table 11-13. SPORTx Multichannel Configuration 1 Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_MCMC1	0xFFC0 2832
SPORT1_MCMC1	0xFFC0 2C32

SPORTx Multichannel Configuration 2 Registers (SPORTx_MCMC2)

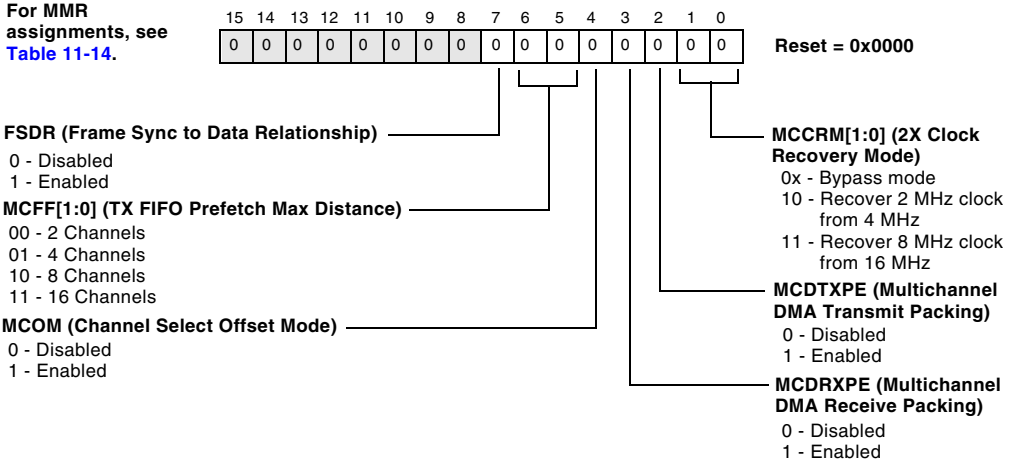


Figure 11-15. SPORTx Multichannel Configuration 2 Registers

Table 11-14. SPORTx Multichannel Configuration 2 Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_MCMC2	0xFFC0 2834
SPORT1_MCMC2	0xFFC0 2C34

SPORTx Receive DMA Current Descriptor Pointer (SPORTx_CURR_PTR_RX) Registers

This 16-bit read-only register holds the lower 16 bits of the pointer to the current descriptor block for the SPORT DMA receive operation. The full 32-bit address to the current pointer is formed by concatenating the Next Descriptor Base Pointer register and the SPORT Receive DMA Current Descriptor Pointer register. Figure 11-16 shows the SPORTx_CURR_PTR_ RX registers.

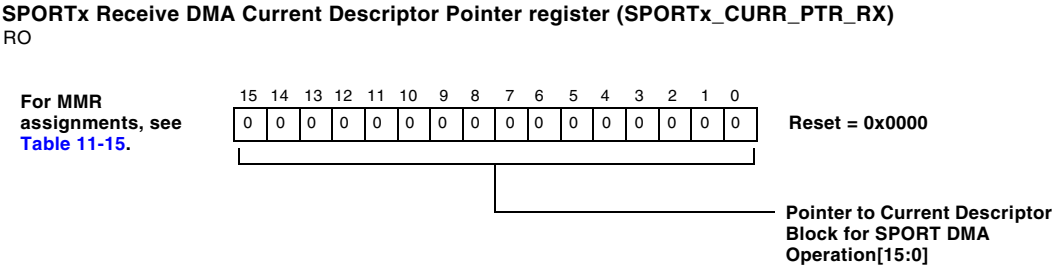


Figure 11-16. SPORTx Receive DMA Current Descriptor Pointer Register

Table 11-15. SPORTx Receive DMA Current Descriptor Pointer Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_CURR_PTR_RX	0xFFC0 2A00
SPORT1_CURR_PTR_RX	0xFFC0 2E00

SPORTx Receive DMA Configuration (SPORTx_CONFIG_DMA_RX) Registers

During SPORT initialization, the program can write the head address of the first DMA descriptor block to the Receive DMA Next Descriptor Pointer register and then set the DMA Enable bit in the Receive DMA

Configuration register. The DMA Configuration register maintains real-time DMA buffer status. Figure 11-17 represents the SPORTx_CONFIG_DMA_RX registers.

Each SPORT DMA channel has a DMA Enable bit (DEN) for each of the serial ports. When DMA is not enabled for a particular channel, the SPORT generates an interrupt every time it receives a data word.

When the Interrupt on Completion bit of the SPORTx_CONFIG_DMA_RX register is set, bit 0 of the SPORTx_IRQSTAT_RX register is set and a DMA interrupt is generated after the final transfer of data. Final transfer of data occurs when DMA count = 0, as specified by the descriptor work block.

DMA transfer size can be set to 8, 16, or 32 bits. This provides flexibility in how data is packed in memory. For DMA writes to memory in excess of the 16-bit FIFO size, the upper bits are padded with 0s. For DMA writes to memory smaller than the 16-bit FIFO size, only the LSBs of the FIFO are written. A DMA read of memory greater than the 16-bit FIFO size reads only the LSBs of memory into the FIFO. For DMA reads of memory smaller than the FIFO size, the least significant bits of the FIFO are loaded and the remaining FIFO bits have unknown values. DMA data transfer size is determined by setting the Data Size Bit 0 and Data Size Bit 1 bits in the SPORTx_CONFIG_DMA_RX register.

When the Interrupt on Error bit is set, a receive overflow error results in bit 1 of the SPORTx_IRQSTAT_RX register being set and a DMA interrupt being generated.

The Receive Overflow Error bit is set if an overflow condition occurs. This bit is sticky only during the current work block. It is cleared on the next descriptor fetch.

The DMA Completion Status bit reflects the current error condition. It is set if a receive overflow error occurs. When the current descriptor work block completes (DMA count = 0), the value of this bit is written to memory as a record of the completion result, and the bit is cleared for the next data transfer.

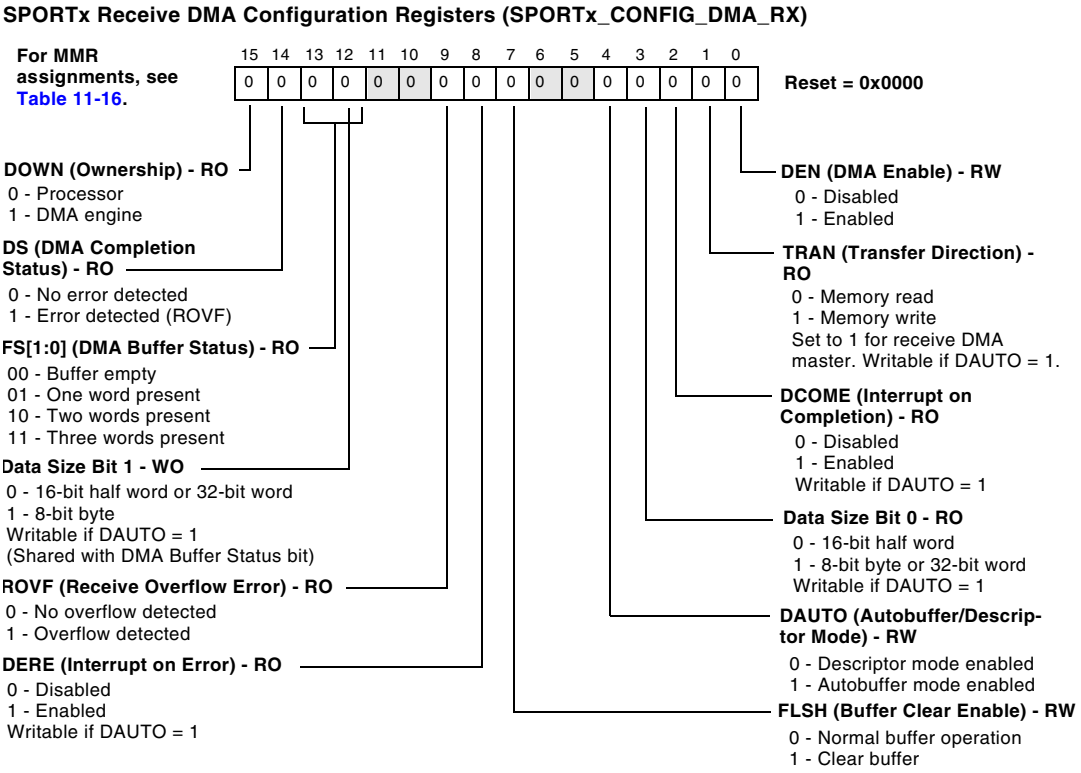


Figure 11-17. SPORTx Receive DMA Configuration Registers

Table 11-16. SPORTx Receive DMA Configuration Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_CONFIG_DMA_RX	0xFFC0 2A02
SPORT1_CONFIG_DMA_RX	0xFFC0 2E02

SPORTx Receive DMA Start Address High (SPORTx_START_ADDR_HI_RX) Registers

This register, shown in Figure 11-18, holds a running pointer to the DMA address that is being accessed. This register is read-only. It can be written in autobuffer mode. Together, SPORTx_START_ADDR_HI_RX and SPORTx_START_ADDR_LO_RX form the 32-bit address for data access.

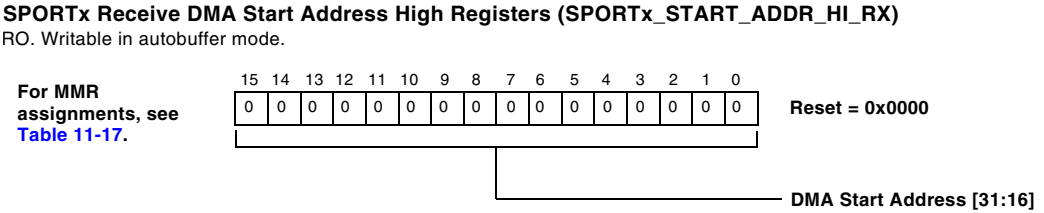


Figure 11-18. SPORTx Receive DMA Start Address High Registers

Table 11-17. SPORTx Receive DMA Start Address High Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_START_ADDR_HI_RX	0xFFC0 2A04
SPORT1_START_ADDR_HI_RX	0xFFC0 2E04

SPORTx Receive DMA Start Address Low (SPORTx_START_ADDR_LO_RX) Registers

The DMA Start Address Low register, shown in Figure 11-19, maintains a running pointer to the DMA address that is being accessed.

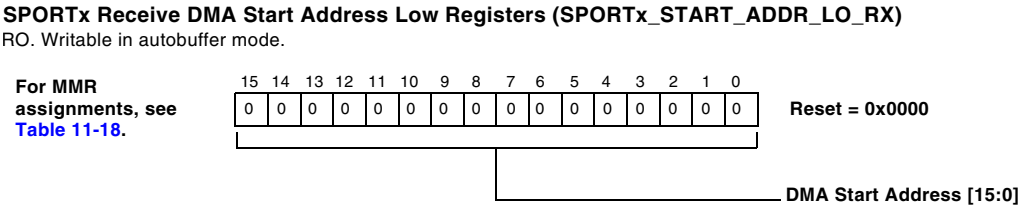


Figure 11-19. SPORTx Receive DMA Start Address Low Registers

It is a read-only register. It can be written in autobuffer mode. Together, SPORTx_START_ADDR_HI_RX and SPORTx_START_ADDR_LO_RX form the 32-bit address for data access.

Table 11-18. SPORTx Receive DMA Start Address Low Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_START_ADDR_LO_RX	0xFFC0 2A06
SPORT1_START_ADDR_LO_RX	0xFFC0 2E06

SPORTx Receive DMA Count (SPORTx_COUNT_RX) Registers

The SPORT Receive DMA Count register, shown in Figure 11-20, holds the number of remaining words in the transfer. This is a read-only register. It can be written in autobuffer mode.

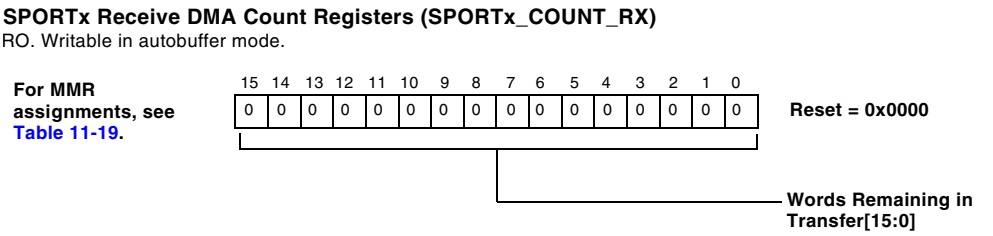


Figure 11-20. SPORTx Receive DMA Count Registers

Table 11-19. SPORTx Receive DMA Count Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_COUNT_RX	0xFFC0 2A08
SPORT1_COUNT_RX	0xFFC0 2E08

SPORTx Receive DMA Next Descriptor Pointer (SPORTx_NEXT_DESCR_RX) Registers

The SPORT Receive DMA Next Descriptor Pointer register, shown in Figure 11-21, maintains the 16 LSBs of the head address of the next DMA descriptor block. Together, the SPORTx_NEXT_DESCR_RX and the DB_NDBP registers form the 32-bit head address of the next descriptor block. During SPORT initialization, the programmer writes the head address of the first DMA descriptor block to the Receive DMA Next Descriptor Pointer register and then sets the DEN bit in the Transmit or Receive DMA Configuration Registers.

Once a DMA process has started, no further control of the SPORT controller or the DMA process should be performed by write accesses to the SPORT DMA control registers.

To initialize SPORTx_NEXT_DESCR_RX before starting a descriptor based DMA operation, first set the DAUTO bit in either the SPORTx_CONFIG_DMA_TX or the SPORTx_CONFIG_DMA_RX register. If the DAUTO bit is not set in at least one of these registers, writes to SPORTx_NEXT_DESCR_RX are ignored.

SPORTx Receive DMA Next Descriptor Pointer Registers (SPORTx_NEXT_DESCR_RX)
RO

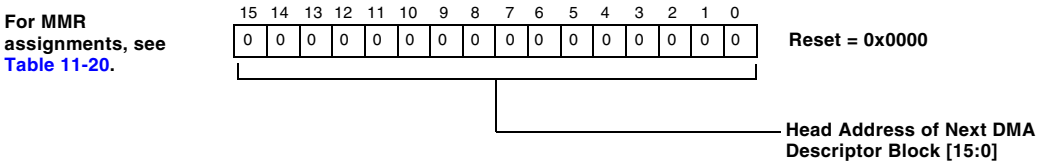


Figure 11-21. SPORTx Receive DMA Next Descriptor Pointer Registers

Table 11-20. SPORTx Receive DMA Next Descriptor Pointer Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_NEXT_DESCR_RX	0xFFC0 2A0A
SPORT1_NEXT_DESCR_RX	0xFFC0 2E0A

SPORTx Receive DMA Descriptor Ready (SPORTx_DESCR_RDY_RX) Registers

The DMA engine stalls if a low level ownership bit is detected during a descriptor block access. Writing a 1 to bit 0 of the SPORT Receive DMA Descriptor Ready register reactivates the processing of the descriptor

block. If the ownership bit is set, processing of the descriptor block resumes and bit 0 of the SPORT Receive DMA Descriptor Ready register is cleared. Figure 11-22 represents the SPORTx_DESCR_RDY_RX registers.

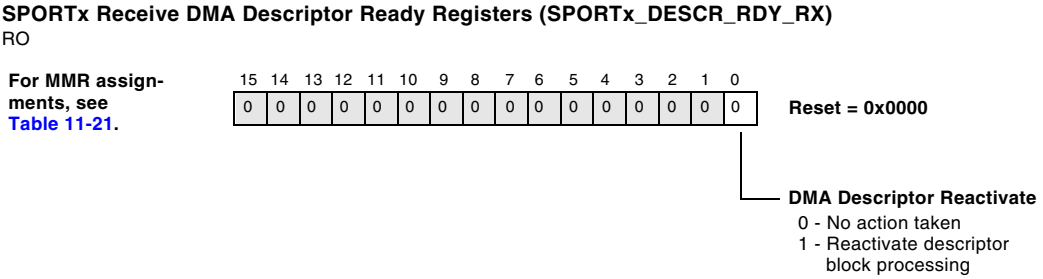


Figure 11-22. SPORTx Receive DMA Descriptor Ready Registers

Table 11-21. SPORTx Receive DMA Descriptor Ready Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_DESCR_RDY_RX	0xFFC0 2A0C
SPORT1_DESCR_RDY_RX	0xFFC0 2E0C

SPORTx Receive DMA IRQ Status (SPORTx_IRQSTAT_RX) Registers

Each SPORT DMA unit has three interrupt sources. Two of these sources are enabled by the Interrupt On Error and Interrupt On Completion bits within the SPORTx_CONFIG_DMA_RX configuration register. The third source, bus error, is not maskable at the DMA level. All three interrupt status bits contained within the SPORTx_IRQSTAT_RX register are sticky.

Bits are cleared by writing a 1 to them. All SPORTx_IRQSTAT_RX register bits are ORed to form a single DMA interrupt. Figure 11-23 represents the SPORTx_IRQSTAT_RX registers.

SPORTx Receive DMA IRQ Status Registers (SPORTx_IRQSTAT_RX)

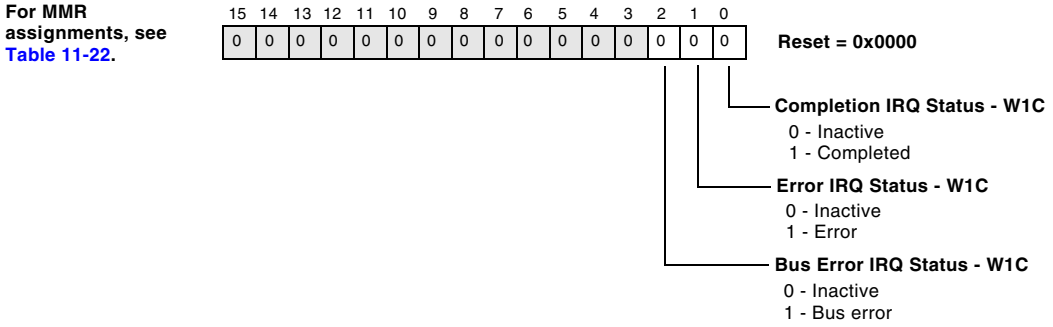


Figure 11-23. SPORTx Receive DMA IRQ Status Registers

Table 11-22. SPORTx Receive DMA IRQ Status Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_IRQSTAT_RX	0xFFC0 2A0E
SPORT1_IRQSTAT_RX	0xFFC0 2E0E

SPORTx Transmit DMA Current Descriptor Pointer (SPORTx_CURR_PTR_TX) Registers

This 16-bit read-only register holds the lower 16 bits of the pointer to the current descriptor block for the SPORT DMA transmit operation. The full 32-bit address to the current pointer is formed by concatenating the Next Descriptor Base Pointer register and the SPORT Transmit DMA Current Descriptor Pointer register, shown in Figure 11-24.

SPORTx Transmit DMA Current Descriptor Pointer Registers (SPORTx_CURR_PTR_TX)

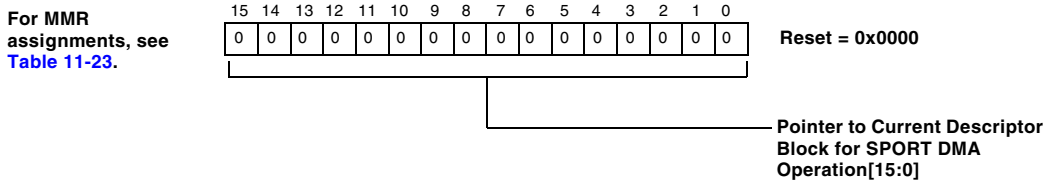


Figure 11-24. SPORTx Transmit DMA Current Descriptor Pointer Registers

Table 11-23. SPORTx Transmit DMA Current Descriptor Pointer Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_CURR_PTR_TX	0xFFC0 2B00
SPORT1_CURR_PTR_TX	0xFFC0 2F00

SPORTx Transmit DMA Configuration (SPORTx_CONFIG_DMA_TX) Registers

During SPORT initialization, the program can write the head address of the first DMA descriptor block to the Transmit DMA Next Descriptor Pointer register, shown in Figure 11-29 on page 11-46, and then set the

DMA Enable bit in the Transmit DMA Configuration registers, shown in Figure 11-25. The DMA Configuration register maintains real time DMA buffer status.

Each SPORT DMA channel has an enable bit (DEN) in these registers for each of the serial ports. When DMA is not enabled for a particular channel, the SPORT generates an interrupt every time it starts to transmit a data word.

When the Interrupt on Completion bit of the SPORTx_CONFIG_DMA_TX register is set, bit 0 of the SPORTx_IRQSTAT_TX register is set and a DMA interrupt is generated after the final transfer of data. Final transfer of data occurs when DMA count = 0, as specified by the descriptor work block.

DMA transfer size can be set to 8, 16, or 32 bits. This provides flexibility in how data is packed in memory. For DMA writes to memory in excess of the 16-bit FIFO size, the upper bits are padded with zeros. For DMA writes to memory smaller than the 16-bit FIFO size, only the LSBs of the FIFO are written. A DMA read of memory greater than the 16-bit FIFO size reads only the LSBs of memory into the FIFO. For DMA reads of memory smaller than the FIFO size, the least significant bits of the FIFO are loaded and the remaining FIFO bits have unknown values. DMA data transfer size is determined by setting the Data Size Bit 0 and Data Size Bit 1 bits in the SPORTx_CONFIG_DMA_TX register.

When the Interrupt on Error bit is set, a transmit underflow error results in bit 1 of the SPORTx_IRQSTAT_TX register being set and a DMA interrupt being generated.

The Transmit Underflow Error bit is set if an underflow condition occurs. This bit is sticky only during the current work block. It is cleared on the next descriptor fetch.

The DMA Completion Status bit reflects the current error condition. It is set if a transmit underflow error occurs. When the current descriptor work block completes (DMA count = 0), the value of this bit is written to memory as a record of the completion result, and the bit is cleared for the next data transfer.

SPORTx Transmit DMA Configuration Registers (SPORTx_CONFIG_DMA_TX)

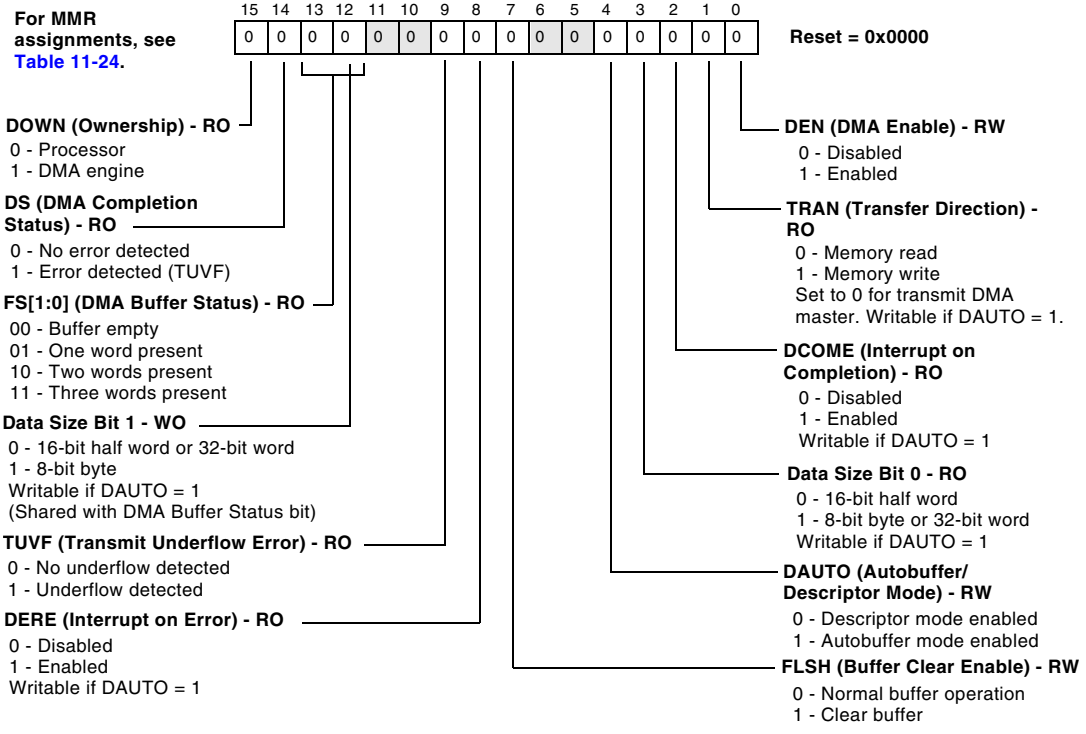


Figure 11-25. SPORTx Transmit DMA Configuration Registers

Table 11-24. SPORTx Transmit DMA Configuration Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_CONFIG_DMA_TX	0xFFC0 2B02
SPORT1_CONFIG_DMA_TX	0xFFC0 2F02

SPORTx Transmit DMA Start Address High (SPORTx_START_ADDR_HI_TX) Registers

This register, shown in Figure 11-26 holds a running pointer to the DMA address that is being accessed. This register is read-only. It can be written in autobuffer mode.

Together, SPORTx_START_ADDR_HI_TX and SPORTx_START_ADDR_LO_TX form the 32-bit address for data access.

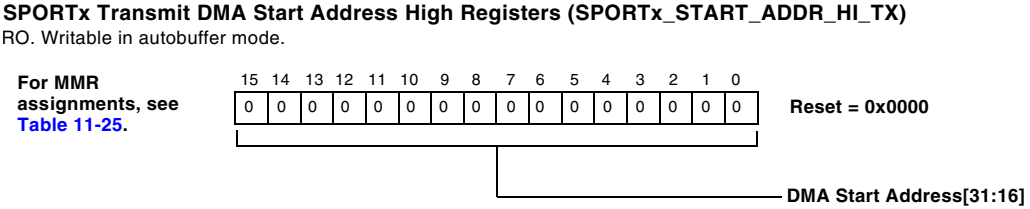


Figure 11-26. SPORTx Transmit DMA Start Address High Registers

Table 11-25. SPORTx Transmit DMA Start Address High Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_START_ADDR_HI_TX	0xFFC0 2B04
SPORT1_START_ADDR_HI_TX	0xFFC0 2F04

SPORTx Transmit DMA Start Address Low (SPORTx_START_ADDR_LO_TX) Registers

The DMA Start Address Low register, shown in Figure 11-27, maintains a running pointer to the DMA address that is being accessed.

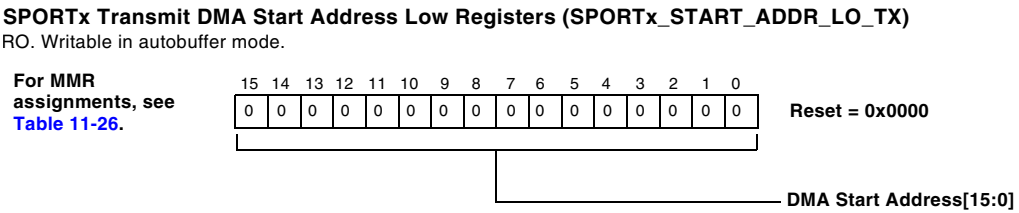


Figure 11-27. SPORTx Transmit DMA Start Address Low Registers

It is a read-only register. It can be written in autobuffer mode. Together, SPORTx_START_ADDR_HI_TX and SPORTx_START_ADDR_LO_TX form the 32-bit address for data access.

Table 11-26. SPORTx Transmit DMA Start Address Low Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_START_ADDR_LO_TX	0xFFC0 2B06
SPORT1_START_ADDR_LO_TX	0xFFC0 2F06

SPORTx Transmit DMA Count (SPORTx_COUNT_TX) Registers

The SPORT Transmit DMA Count register, shown in Figure 11-28, holds the number of remaining words in the transfer. This is a read-only register. It can be written in autobuffer mode.

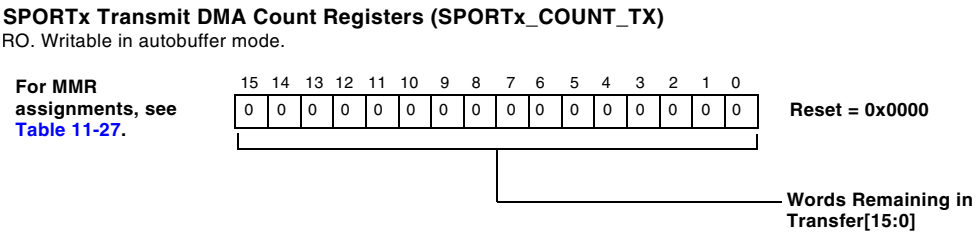


Figure 11-28. SPORTx Transmit DMA Count Registers

Table 11-27. SPORTx Transmit DMA Count Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_COUNT_TX	0xFFC0 2B08
SPORT1_COUNT_TX	0xFFC0 2F08

SPORTx Transmit DMA Next Descriptor Pointer (SPORTx_NEXT_DESCR_TX) Registers

The SPORT Transmit DMA Next Descriptor Pointer register, shown in Figure 11-29, holds the 16 LSBs of the head address of the next DMA descriptor block. Together, the SPORTx_NEXT_DESCR_TX and the DB_NDBP registers form the 32-bit head address of the next descriptor block. During SPORT initialization, the programmer writes the head address of the first DMA descriptor block to the Transmit DMA Next Descriptor Pointer register and then sets the DEN bit in the Transmit or Receive DMA Configuration Registers.

Once a DMA process has started, no further control of the SPORT controller or the DMA process should be performed by write accesses to the SPORT DMA control registers.

To initialize SPORTx_NEXT_DESCR_TX before starting a descriptor based DMA operation, first set the DAUTO bit in either the SPORTx_CONFIG_DMA_TX or the SPORTx_CONFIG_DMA_RX register. If the DAUTO bit is not set in at least one of these registers, writes to SPORTx_NEXT_DESCR_TX are ignored.

SPORTx Transmit DMA Next Descriptor Pointer Registers (SPORTx_NEXT_DESCR_TX)
RO

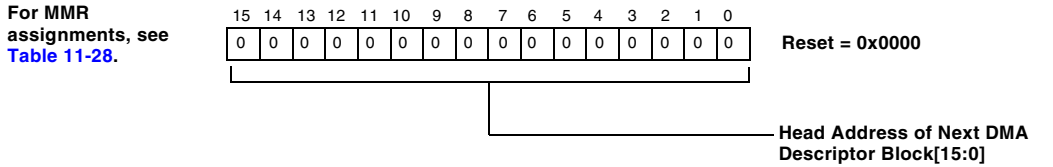


Figure 11-29. SPORTx Transmit DMA Next Descriptor Pointer Registers

Table 11-28. SPORTx Transmit DMA Next Descriptor Pointer Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_NEXT_DESCR_TX	0xFFC0 2B0A
SPORT1_NEXT_DESCR_TX	0xFFC0 2F0A

SPORTx Transmit DMA Descriptor Ready (SPORTx_DESCR_RDY_TX) Registers

The DMA engine stalls if a low level ownership bit is detected during a descriptor block access. Writing a 1 to bit 0 of the SPORT Transmit DMA Descriptor Ready register reactivates the processing of the descrip-

tor block. If the ownership bit is set, processing of the descriptor block resumes and bit 0 of the SPORT Transmit DMA Descriptor Ready register is cleared (see Figure 11-30).

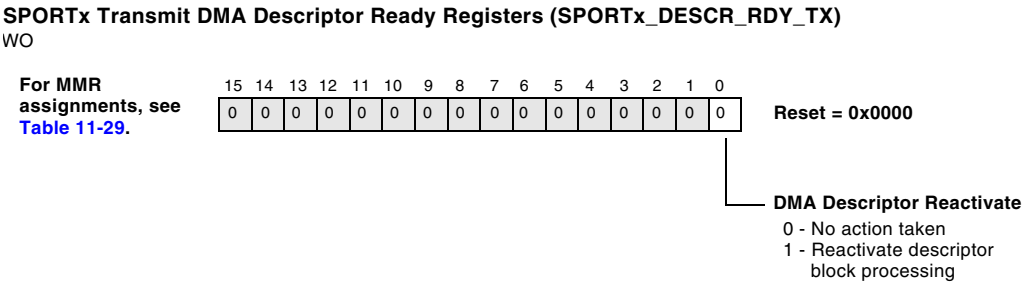


Figure 11-30. SPORTx Transmit DMA Descriptor Ready Registers

Table 11-29. SPORTx Transmit DMA Descriptor Ready Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_DESCR_RDY_TX	0xFFC0 2B0C
SPORT1_DESCR_RDY_TX	0xFFC0 2F0C

SPORTx Transmit DMA IRQ Status (SPORTx_IRQSTAT_TX) Registers

Each SPORT DMA unit has three interrupt sources. Two of these sources are enabled by the Interrupt On Error and Interrupt On Completion bits within the `SPORTx_CONFIG_DMA_TX` configuration register. The third source, bus error, is not maskable at the DMA level. All three interrupt status bits contained within the `SPORTx_IRQSTAT_TX` register are sticky.

Bits are cleared by writing a 1 to them. All SPORTx_IRQSTAT_TX register bits are ORed to form a single DMA interrupt. Figure 11-31 shows the SPORTx_IRQSTAT_TX register.

SPORTx Transmit DMA IRQ Status Registers (SPORTx_IRQSTAT_TX)

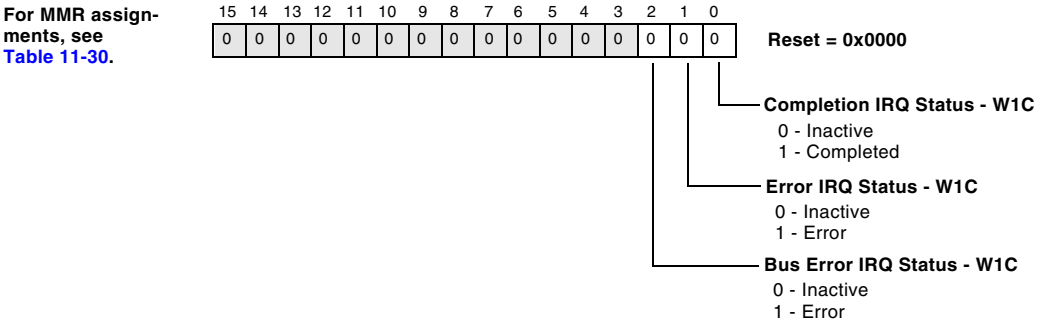


Figure 11-31. SPORTx Transmit DMA IRQ Status Registers

Table 11-30. SPORTx Transmit DMA IRQ Status Register MMR Assignments

Register Name	Memory-Mapped Address
SPORT0_IRQSTAT_TX	0xFFC0 2B0E
SPORT1_IRQSTAT_RX	0xFFC0 2E0E

Register Writes and Effect Latency

When the SPORT is disabled (TSPEN and RSPEN cleared), SPORT register writes are internally completed at the end of the next SCLK cycle after which they occurred and the register reads back the newly written value on the next cycle after that.

When the SPORT is enabled to transmit (TSPEN set) or receive (RSPEN set), corresponding SPORT configuration register writes are disabled (except for SPORTx_RSCLKDIV, SPORTx_TSCLKDIV, and multichannel mode channel registers). SPORTx_TX register writes are always enabled; SPORTx_RX is a read-only register.

After a write to a SPORT register, any changes to the control and mode bits generally take effect when the SPORT is re-enabled.

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 (as seen at the SCLK0 or CLKOUT_SCLK1 pin) and the value of the 16-bit serial clock divide modulus registers, SPORTx_TSCLKDIV and SPORTx_RSCLKDIV.

```
SPxTCLK frequency = (SCLK frequency) / (2 × (SPORTx_TSCLDIV + 1))

SPxRCLK frequency = (SCLK frequency) / (2 × (SPORTx_RSCLDIV + 1))
```

If the value of SPORTx_TSCLKDIV or SPORTx_RSCLKDIV is changed while the internal serial clock is enabled, the change in TCLK or RCLK frequency takes effect at the start of the rising edge of TCLK or RCLK that follows the next leading edge of TFS or RFS.

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 serial clocks between frame sync assertions = xFSDIV + 1
```

Use the following equations to determine the correct value of xFSDIV, given the serial clock frequency and desired frame sync frequency:

```
SPxTFSCLK frequency = (CLKOUT frequency) / (SPORTx_TFSDIV + 1)

SPxRFSCLK frequency = (CLKOUT frequency) / (SPORTx_RFSDIV + 1)
```

The frame sync would thus be continuously active if xFSDIV=0. However, the value of xFSDIV should not be less than the serial word length minus one (the value of the SLEN field in the transmit or receive control register); a smaller value of xFSDIV could cause an external device to abort the current operation or have other unpredictable results. If the SPORT is not being used, the xFSDIV 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 *ADSP-BF535 Blackfin Embedded Processor Data Sheet* for exact timing specifications.

Be careful when operating with externally generated clocks near the frequency of SCLK/2. There is a delay between the clock signal's arrival at the TCLK pin and the output of the data, and this delay may limit the receiver's speed of operation. See *ADSP-BF535 Blackfin Embedded Processor Data Sheet* for exact timing specifications. At full speed serial clock rate, the safest practice is to use an externally generated clock and externally generated frame sync (ICLK=0 and IRFS=0).

Data Word Formats

The format of the data words transferred over the SPORTs is configured by the DTYPE, SENDN, and SLEN bits of the SPORTx_TX_CONFIG and SPORTx_RX_CONFIG registers.

Word Length

Each SPORT channel (transmit and receive) independently handles word lengths of 3 to 16 bits. The data is right-justified in the SPORT data registers if it is fewer than 16 bits long, residing in the LSB positions. The value of the serial word length (SLEN) field in the SPORTx_TX_CONFIG and SPORTx_RX_CONFIG 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 zero or one; values from 2 to 15 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 of longer (so SLEN ≥ 3).

Endian Format

Endian format determines whether the serial word is transmitted most significant bit (MSB) first or least significant bit (LSB) first. Endian format is selected by the SENDN bit in the SPORTx_TX_CONFIG and SPORTx_RX_CONFIG registers. When SENDN=0, serial words are transmitted (or received) MSB first. When SENDN=1, serial words are transmitted (or received) LSB first.

Data Type

The DTYPE field of the SPORTx_TX_CONFIG and SPORTx_RX_CONFIG registers specifies one of four data formats for both single and multichannel operation (See Table 11-31).

Table 11-31. DTYPE and Data Formatting

DTYPE	Data Formatting
00	Right justify, zero fill unused MSBs
01	Right justify, sign extend into unused MSBs
10	Compand using μ-law
11	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 ADSP-BF535 processor's 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. 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. If the magnitude of the 16-bit value is greater than the 13-bit A-law or 14-bit μ-law maximum, the value is automatically compressed to the maximum positive or negative value.

Clock Signal Options

Each SPORT has a transmit clock signal (TCLK) and a receive clock signal (RCLK). The clock signals are configured by the ICLK and CKFE bits of the SPORTx_TX_CONFIG and SPORTx_RX_CONFIG registers. Serial clock frequency is configured in the SPORTx_TSCLKDIV and SPORTx_RSCLKDIV registers.

The receive clock pin may be tied to the transmit clock if a single clock is desired for both input and output.

Both transmit and receive clocks can be independently generated internally or input from an external source. The ICLK bit of the SPORTx_TX_CONFIG and SPORTx_RX_CONFIG registers determines the clock source.

When ICLK=1, the clock signal is generated internally by the core and the TCLK or RCLK pin is an output. The clock frequency is determined by the value of the serial clock divisor in the SPORTx_TSCLKDIV or SPORTx_RSCLKDIV registers.

When ICLK=0, the clock signal is accepted as an input on the TCLK or RCLK pins, and the serial clock divisors in the SPORTx_TSCLKDIV/SPORTx_RSCLKDIV registers are ignored. The externally generated serial clock need not be synchronous with the core system clock.

Frame Sync Options

Framing signals indicate the beginning of each serial word transfer. The framing signals for each SPORT are TFS (transmit frame synchronization) and RFS (receive frame synchronization). A variety of framing options are available; these options are configured in the SPORT control registers. 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) and RFSR (receive frame sync required) control bits determine whether frame sync signals are required. These bits are located in the SPORTx_TX_CONFIG and SPORTx_RX_CONFIG 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 buffer before the previous word is shifted out and transmitted. "Data Independent Transmit Frame Sync" on page 11-59.

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.

When DMA is enabled in this mode, with frame syncs not required, DMA requests may be held off by chaining or may not be serviced frequently enough to guarantee continuous unframed data flow.

Figure 11-32 illustrates framed serial transfers, which have these characteristics:

- TFSR and RFSR bits in the SPORTx_TX_CONFIG and SPORTx_RX_CONFIG 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_TX_CONFIG and SPORTx_RX_CONFIG registers.

See "Timing Examples" on page 11-69 for more timing examples.

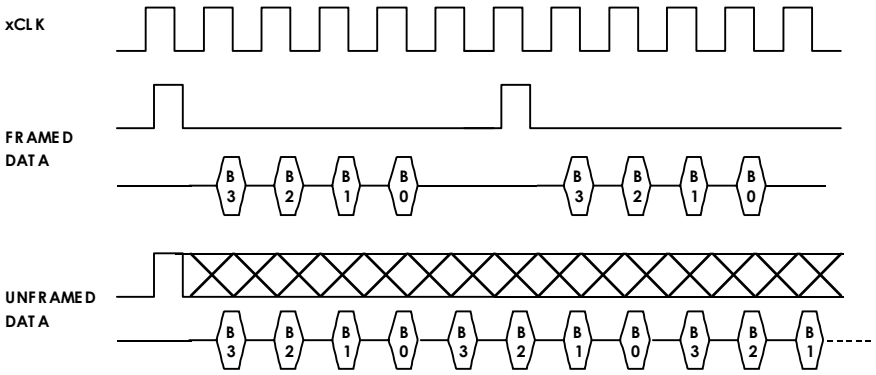


Figure 11-32. Framed Versus Unframed Data

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_TX_CONFIG and SPORTx_RX_CONFIG 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 registers.

When ITFS=0 or IRFS=0, the corresponding frame sync signal is accepted as an input on the TFS pin or RFS pins, 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_TX_CONFIG and SPORTx_RX_CONFIG registers determine the frame syncs' logic level:

- 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 CKFE bit of the SPORTx_TX_CONFIG and SPORTx_RX_CONFIG registers selects the driving and sampling edges for the serial data and frame syncs.

For the SPORT transmitter, setting CKFE=1 in the SPORTx_TX_CONFIG register selects the falling edge of TCLKx to be used to drive data and internally generated frame syncs and selects the rising edge of TCLKx to be used to sample externally generated frame syncs. Setting CKFE=0 selects the rising edge of TCLKx to be used to drive data and internally generated frame syncs and selects the falling edge of TCLKx to be used to sample externally generated frame syncs.

For the SPORT receiver, setting CKFE=1 in the SPORTx_RX_CONFIG register selects the falling edge of RCLKx to be used to drive internally generated frame syncs and selects the rising edge of RCLKx to be used to sample data and externally generated frame syncs. Setting CKFE=0 selects the rising edge of RCLKx to be used to drive internally generated frame syncs and selects the falling edge of RCLKx to be used 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 clock (CKFE=1 in the SPORTx_TX_CONFIG register), the frame sync must be transmitted on the falling edge of clock.

The transmit and receive functions of two SPORTs connected together should always select the same value for CKFE in the transmitter and receiver, so that the transmitter drives the data on one edge and the receiver samples the data on the opposite edge.

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_TX_CONFIG and SPORTx_RX_CONFIG 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 is the case when 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 (so 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 11-33 illustrates the two modes of frame signal timing. In frame signal timing:

- The LATFS or LARFS bits of the SPORTx_TX_CONFIG and SPORTx_RX_CONFIG registers configure early or late frame syncs. A value of LATFS=0 or LARFS=0 is used for early frame syncs. A value of LATFS=1 or LARFS=1 is used for late frame syncs.
- With early framing, the frame sync precedes data by one cycle. With late framing, the frame sync is checked on the first bit only.

- Data is transmitted MSB first (if SENDN=0) or LSB first (if SENDN=1).
- The frame sync and clock can be generated internally or externally.

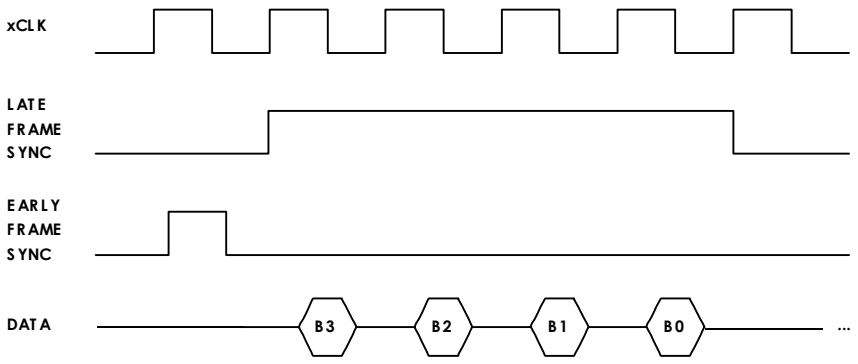


Figure 11-33. Normal Versus Alternate Framing

See "Timing Examples" on page 11-69 for more timing examples.

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 mode bit (DITFS) allows the continuous generation of the TFS signal, with or without new data. The DITFS bit of the SPORTx_TX_CONFIG 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 retransmitted 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.

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 bit stream occupies a separate channel. Each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of 24 channels.

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. In other words, the SPORT can do any of these 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 DT pin is always driven (not three-stated) if the SPORT is enabled (TSPEN=1 in the SPORTx_TX_CONFIG register), unless it 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/SPORTx_RX operation. This is especially important in DMA data unpacked mode, since SPORT FIFO operation begins immediately after SPORTx_TX/SPORTx_RX is enabled and depends on the values of these registers. Enable MCM_EN before enabling SPORTx_TX or SPORTx_RX operation.

Figure 11-34 shows example timing for a multichannel transfer (for example, receive on channels 0 and 2, and transmit on channels 1 and 2). Multichannel transfer has these characteristics:

- Uses 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.

See “Timing Examples” on page 11-69 for more timing examples.

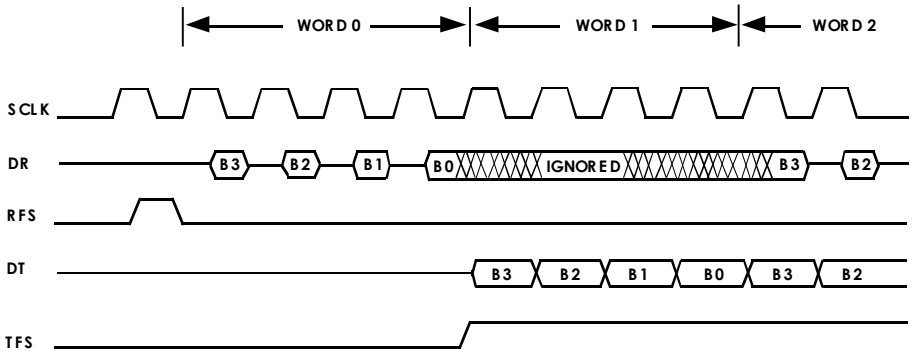


Figure 11-34. Multichannel Operation

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 occurs at 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 MCM configuration, both SPORTx_RX configuration registers should always be programmed the same way as the SPORTx_TX configuration register, even if SPORTx_RX operation is not enabled.

In multichannel mode, 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 DT pin is three-stated when the time slot is not active, and the TFS signal serves as an output enabled signal for the DT pin. The SPORT drives TFS in multichannel mode whether or not ITFS is cleared.

Once the initial FS is received, and a frame transfer has started, all other FS signals are ignored by the SPORT until the complete frame has been transferred.

In multichannel mode, the RFS signal is used for the block or frame start reference, after which the transfers are performed continuously with no FS required. Therefore, internally generated frame syncs are always data independent.

Multichannel Frame Delay

The 4-bit MFD field in the multichannel configuration control register 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 defines the range of the channels that can be enabled/disabled in the current configuration. It can be any value in the range of 8 to 128, in increments of 8; the default value of 0 corresponds to a minimum window size of 8 channels. Since the DMA buffer size is always fixed, it is possible to define a smaller window size (for example, 32 bits), resulting in a smaller DMA buffer size (in this example, 32 bits instead of 128 bits) to save DMA bandwidth. The window size cannot be changed while the SPORT is enabled.

Window Offset

The window offset specifies where in the 127-channel range to place the start of the window. A value of 0 specifies no offset and permits using all 128 channels. As an example, a program could define a window with a window size of 5 and an offset of 93. This 5-channel window would reside in the range from 93 to 97. The window offset cannot be changed while the SPORT is enabled.

If the combination of the window size and the window offset would place the window outside of the range of the channel enable registers, none of the channels in the frame are enabled, since this combination is invalid.

Other Multichannel Fields in SPORTx_TX_CONFIG, SPORTx_RX_CONFIG

A multichannel frame contains more than one channel, as specified by the window size and window offset; the multichannel frame is a combined sequence of the window offset and the channels contained in the window. The total number of channels in the frame is calculated by adding the window size to the window offset.

The channel select offset mode is bit 4 in the SPORTx_MCMC2 register. When this mode is selected, the first bit of the SPORTx_MTCSx or SPORTx_MRCSx register is linked to the first bit directly following the offset of the window. If the channel select offset mode is not enabled, the first bit of the SPORTx_MTCSx or SPORTx_MRCSx register is placed at offset 0.

The 7-bit CHNL field in the SPORTx_STAT register indicates which channel is currently selected during multichannel operation. This field is a read-only status indicator. CHNL[6:0] increments by one as each channel is serviced, and in channel select offset mode the value of CHNL is reset to 0 after the offset has been completed. So, as an example, for a window of 8 and an offset of 21, the counter displays a value between 0 and 28 in the regular mode, but in channel select offset mode the counter resets to 0 after counting up to 21 and the frame completes when the CHNL reaches a value of 7, indicating the eighth channel.

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 (FSDR=0), the data is transmitted on the same edge that the TFS is generated. For example, a positive edge TFS causes data to be transmitted on the positive edge of the SCK, 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 Registers

A channel is a multibit word from 3 to 16 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 channels are available. The SPORTx_MRCSx and SPORTx_MTCSx registers are used to enable and disable individual channels; the SPORTx_MRCSx registers specify the active receive channels, and the SPORTx_MTCSx registers specify the active transmit channels.

Each register has 16 bits, corresponding to the 16 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). For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in the SPORTx_MTCSx register causes the SPORT to transmit the word in that channel's position of the data stream. Clearing the bit in the SPORTx_MTCSx register causes the SPORT's DT (data transmit) pin to three-state during the time slot of that channel.

Setting a particular bit in the SPORTx_MRCSx register causes the SPORT to receive the word in that channel's position of the data stream; the received word is loaded into the SPORTx_RX buffer. Clearing the bit in the SPORTx_MRCSx register causes the SPORT to ignore the data.

Companding may be selected on an all-or-none channel basis. A-law or μ-law companding is selected with the DTYPE bit 1 in the SPORTx_TX_CONFIG and SPORTx_RX_CONFIG registers, and applies to all active channels. (See "Companding" on page 11-52 for more information about companding.)

Multichannel Enable

Setting the MCM bit in the multichannel mode configuration control register 1 enables multichannel mode. When MCM=1, multichannel operation is enabled; when MCM=0, all multichannel operations are disabled.

Setting the MCM 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.

Multichannel DMA Data Packing

Multichannel DMA data packing and unpacking are specified with the MCDTXPE and MCDRXPE bits in the SPORTx_MCMC2 multichannel configuration registers.

If the bits are set, indicating that data is packed, the SPORT expects that 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 of the frames. 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 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 has no restrictions on changing the number of enabled channels while the SPORT is enabled.

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. Both methods are interrupt-driven, using the same internally generated interrupts.

When SPORT DMA is not enabled in the SPORTx_TX_CONFIG or SPORTx_RX_CONFIG registers, the SPORT generates an interrupt every time it has received a data word or has started to transmit a data word. 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. Service routines can then operate on the block of data rather than on single words, significantly reducing overhead.

For information about DMA, see "Direct Memory Access" on page 9-1.

Support for Standard Protocols

The ADSP-BF535 processor supports the H.100 standard protocol. These SPORT parameters must be set to support this standard.

- SPORTx_TFSDIVx = SPORTx_RFSDIVx = 0x03FF (1024 clock cycles per frame, 122 ns wide, 125 µs period frame sync)
- TFSR/RFSR set (FS required)
- LTFS/LRFS set (active low FS)
- TSCLKDIV = RSCLKDIV = 8 (for 8.192 MHz (+/- 2%) bit clock)
- MCM 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 (SCK) 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 the 2 MHz or 8 MHz clock from the incoming 4 MHz or 16 MHz clock with the proper phase relationship. A 2-bit mode signal chooses the applicable clock mode, which includes a non-divide/bypass mode for normal operation.

SPORT Pin/Line Terminations

The ADSP-BF535 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 11-54, "Early Versus Late Frame Syncs (Normal Versus Alternate Timing)" on page 11-57, and "Frame Syncs In Multichannel Mode" on page 11-62). This section contains additional examples to illustrate more 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-BF535 Blackfin 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 11-35 through Figure 11-40 show framing for receiving data.

In Figure 11-35 and Figure 11-36, the normal framing mode is shown for non-continuous data (any number of SCK cycles between words) and continuous data (no SCK cycles between words).

Figure 11-37 and Figure 11-38 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 11-39 and Figure 11-40 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 SCK 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 11-41 through Figure 11-46 show framing for transmitting data and are very similar to Figure 11-35 through Figure 11-40.

In Figure 11-41 and Figure 11-42, the normal framing mode is shown for non-continuous data (any number of SCK cycles between words) and continuous data (no SCK cycles between words).

Figure 11-43 and Figure 11-44 show non-continuous and continuous transmission in the alternate framing mode. As noted previously for the receive timing diagrams, the TFS output meets the TFS input timing requirement.

Figure 11-45 and Figure 11-46 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 SCK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode).

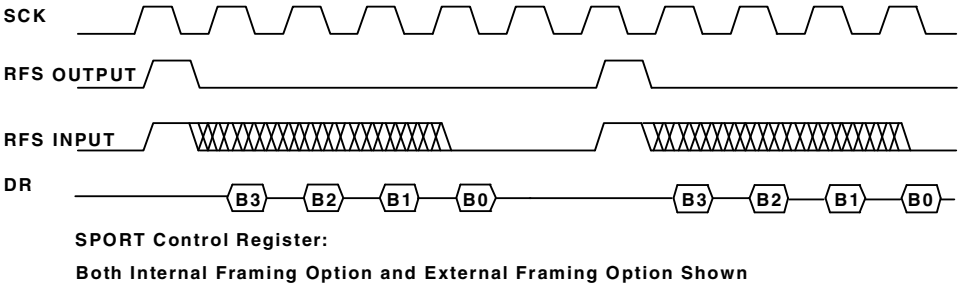


Figure 11-35. SPORT Receive, Normal Framing

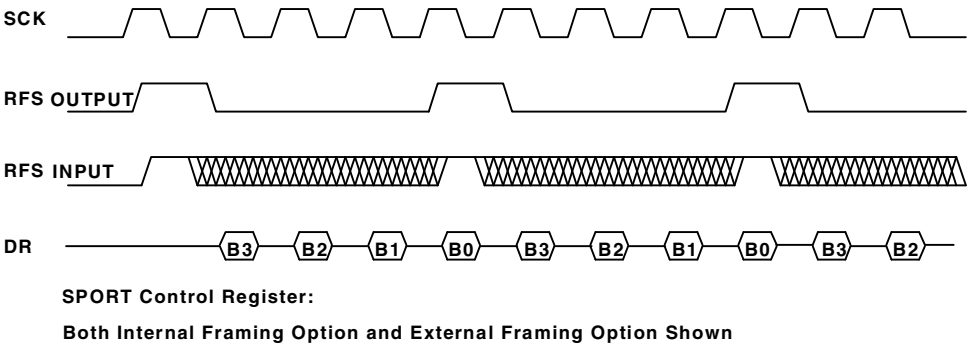
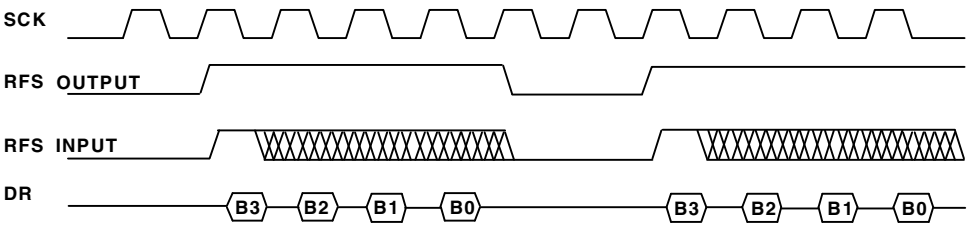
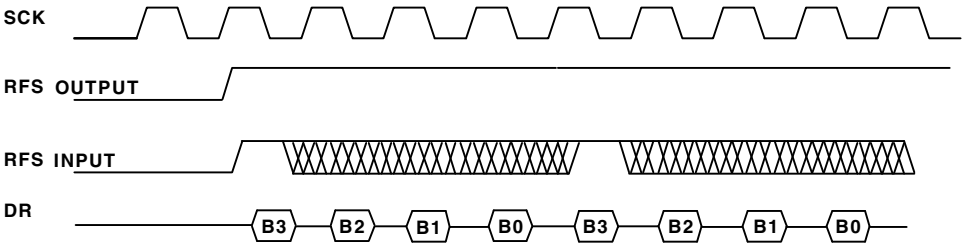


Figure 11-36. SPORT Continuous Receive, Normal Framing



SPORT Control Register:
Both Internal Framing Option and External Framing Option Shown

Figure 11-37. SPORT Receive, Alternate Framing



SPORT Control Register:
Both Internal Framing Option and External Framing Option Shown

Figure 11-38. SPORT Continuous Receive, Alternate Framing

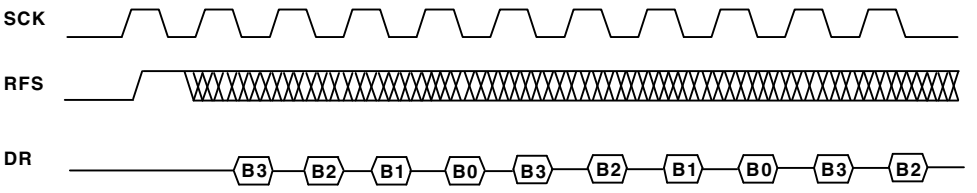


Figure 11-39. SPORT Receive, Unframed Mode, Normal Framing

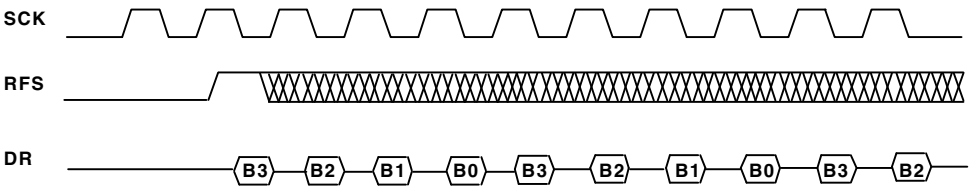


Figure 11-40. SPORT Receive, Unframed Mode, Alternate Framing

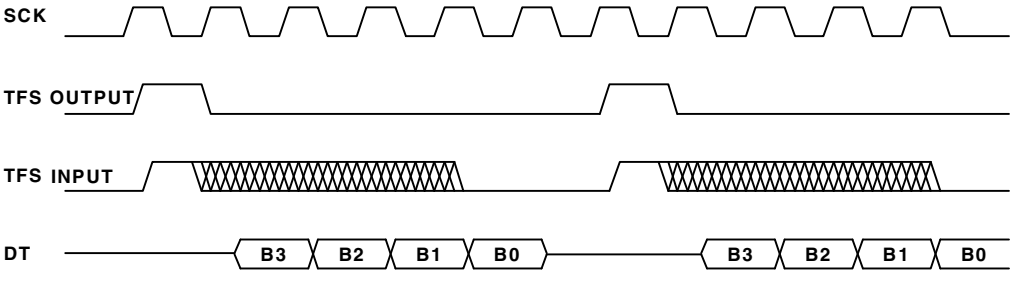


Figure 11-41. SPORT Transmit, Normal Framing

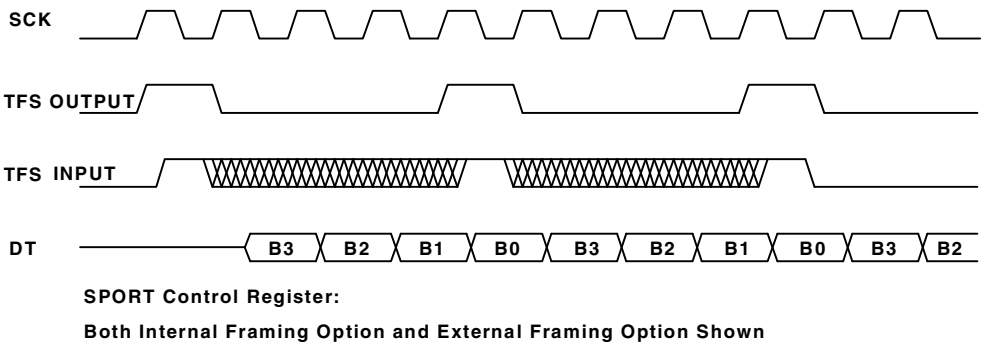
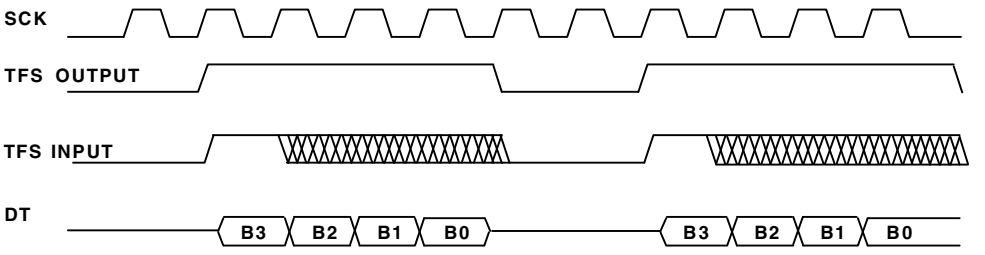


Figure 11-42. SPORT Continuous Transmit, Normal Framing

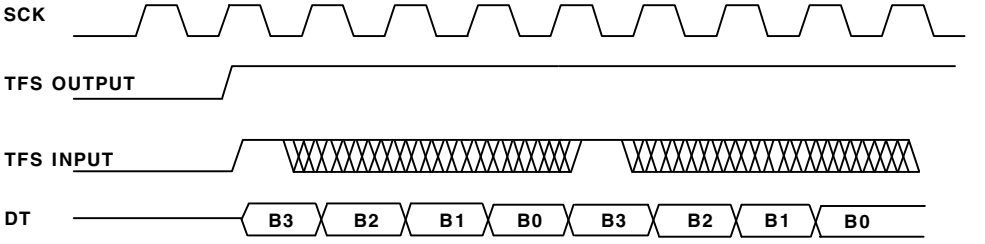


SPORT Control Register:

Both Internal Framing Option and External Framing Option Shown

Note: There is an asynchronous delay between TFS input and DT. See the appropriate datasheet for specifications.

Figure 11-43. SPORT Transmit, Alternate Framing



SPORT Control Register:

Both Internal Framing Option and External Framing Option Shown

Note: There is an asynchronous delay between TFS input and DT. See the appropriate datasheet for specifications.

Figure 11-44. SPORT Continuous Transmit, Alternate Framing

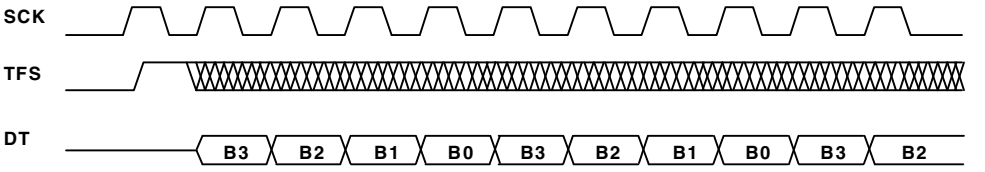
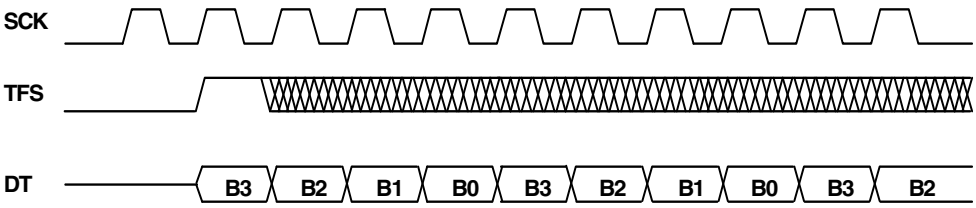


Figure 11-45. SPORT Transmit, Unframed Mode, Normal Framing



Note: There is an asynchronous delay between TFS input and DT. See the appropriate datasheet for specifications.

Figure 11-46. SPORT Transmit, Unframed Mode, Alternate Framing

12 UART PORT CONTROLLER

The ADSP-BF535 processor's 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 also includes modem control and interrupt handling hardware. Interrupts can be generated from 12 different events.

The ADSP-BF535 processor features two independent UART devices: UART0 and UART1. UART0 enhances the standard functionality and supports the half duplex IrDA® SIR (9.6/115.2 Kbps rate) protocol.

Modem status and control registers are available, but only the transmit (TX) and receive (RX) data signals are routed to external pins.

The UARTs are DMA capable peripherals with support for separate TX and RX DMA master channels. They 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.

Timers 0, 1, and 2 can be used to provide a hardware assisted autobaud detection mechanism for use with the UART. See "Timers" on page 16-1 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 x 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. Data is always transmitted and received least significant bit (LSB) first.

Figure 12-1 shows a typical physical bit stream measured on the TX pin.

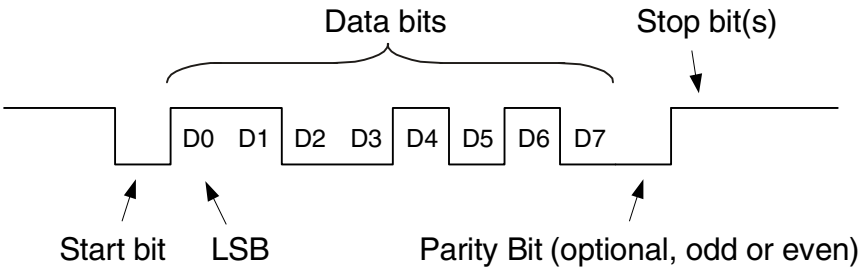


Figure 12-1. Bit Stream on the TX Pin

UARTx Control and Status Registers

The ADSP-BF535 processor provides a set of PC-style industry standard control and status registers for each UART. These MMRs are byte-wide registers that are accessed as 16-bit 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 (UARTx_DLH and UARTx_DLL) share their addresses with the Transmit Holding registers (UARTx_THR), the Receive Buffer registers (UARTx_RBR), and the Interrupt Enable registers (UARTx_IER). The Divisor Latch Access bit (DLAB) in the Line Control Register (UARTx_LCR) controls which sets of registers are accessible at a given time.

UARTx Line Control Registers (UARTx_LCR)

The UARTx Line Control register (UARTx_LCR) controls the format of received and transmitted character frames. Figure 12-2 shows the bits in this register.

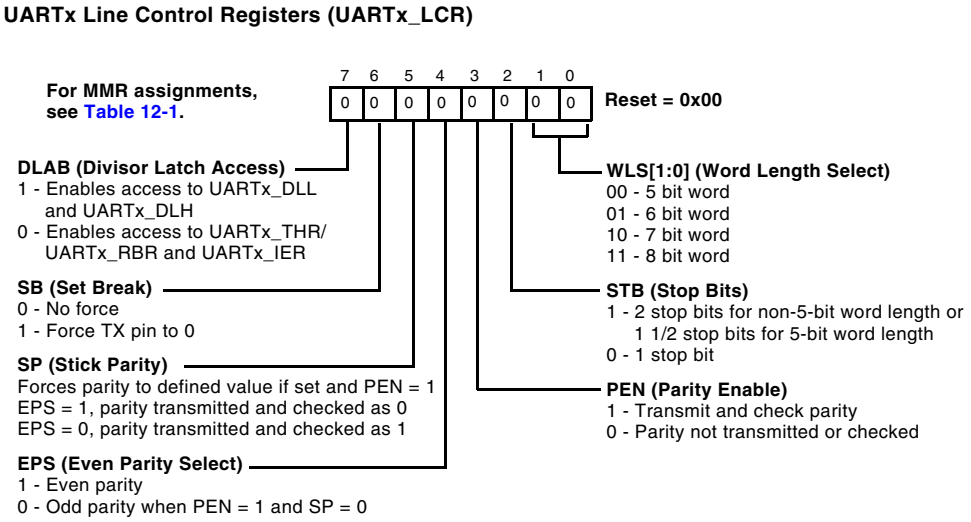


Figure 12-2. UARTx Line Control Registers

Table 12-1. UARTx Line Control Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_LCR	0xFFC0 1806
UART1_LCR	0xFFC0 1C06

UARTx Line Status Registers (UARTx_LSR)

The UARTx Line Status register contains UART status information, as shown in Figure 12-3.

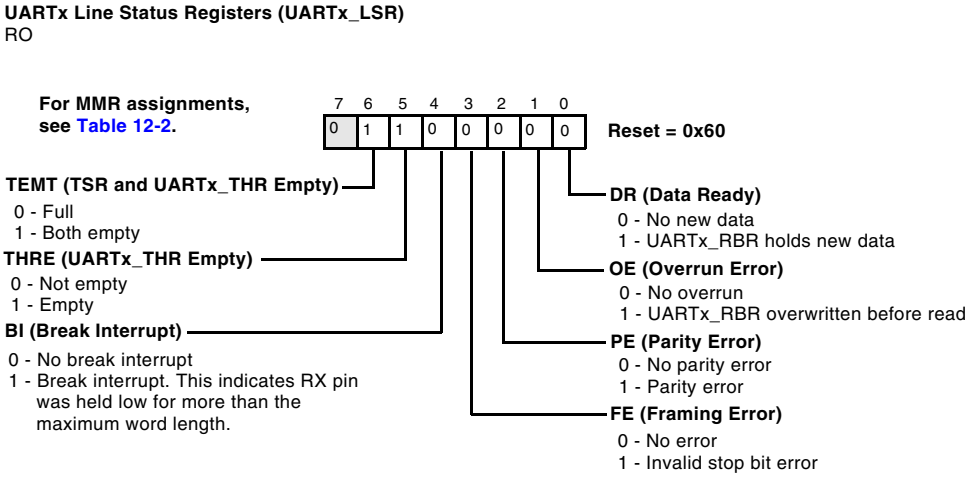


Figure 12-3. UARTx Line Status Registers

Table 12-2. UARTx Line Status Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_LSR	0xFFC0 180A
UART1_LSR	0xFFC0 1C0A

The Break Interrupt (BI), Overrun Error (OE), Parity Error (PE) and Framing Error (FE) bits are cleared when the UART Line Status register (UARTx_LSR) is read. The Data Ready (DR) bit is cleared when the UART Receive Buffer register (UARTx_RBR) is read. Because of the destructive nature of these read operations, special care should be taken. See "Speculative Load Execution" on page 6-81 and "Conditional Load Behavior" on page 6-82 for more information.

UARTx Transmit Holding Registers (UARTx_THR)

A write to the UARTx Transmit Holding register (UARTx_THR), shown in Figure 12-4, 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 UARTx_THR to the Transmit Shift register sets the Transmit Holding Register Empty (THRE) status flag in the UARTx Line Status register (UARTx_LSR).

The UARTx_THR register is mapped to the same address as UARTx_RBR and UARTx_DLL. To access UARTx_THR, the DLAB bit in UARTx_LCR must be cleared. When the DLAB bit is cleared, writes to this address target the UARTx_THR register, and reads from this address return the UARTx_RBR register.

Note that data is transmitted and received least significant bit first (bit 0) followed by the most significant bits.

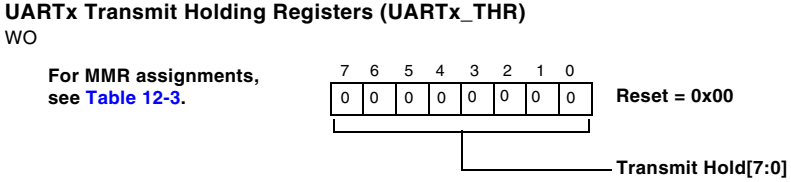


Figure 12-4. UARTx Transmit Holding Registers

Table 12-3. UARTx Transmit Holding Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_THR	0xFFC0 1800
UART1_THR	0xFFC0 1C00

UARTx Receive Buffer Registers (UARTx_RBR)

The receive operation uses the same data format as the transmit configuration, except that the number of stop bits is always 1. After detection of the start bit, the received word is shifted into the Receive Shift register (RSR) at a bit rate of SCLK/(16 × Divisor). After the appropriate number of bits (including stop bits) is received, the data and any status are updated and the Receive Shift register is transferred to the UARTx Receive Buffer register (UARTx_RBR), shown in Figure 12-5. After the transfer of the received word to the UARTx_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 resynchronized 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 UARTx_RBR register is mapped to the same address as UARTx_THR and UARTx_DLL. To access UARTx_RBR, the DLAB bit in UARTx_LCR must be cleared. When the DLAB bit is cleared, writes to this address target the UARTx_THR register, while reads from this address return the UARTx_RBR register.

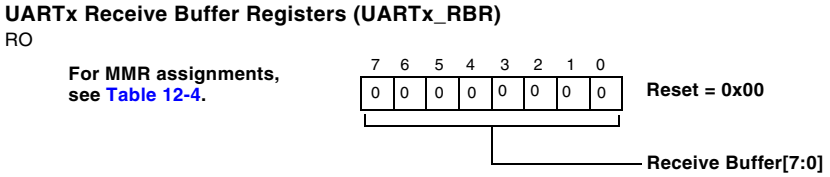


Figure 12-5. UARTx Receive Buffer Registers

Table 12-4. UARTx Receive Buffer Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_RBR	0xFFC0 1800
UART1_RBR	0xFFC0 1C00

UARTx Interrupt Enable Registers (UARTx_IER)

The UARTx_IER register, shown in Figure 12-6, is used only in non-DMA mode. Four different interrupt sources are ORed to share a single IRQ channel, the UART RX interrupt. While the Receive Interrupt itself must be unmasked within the System Interrupt Controller (SIC), the four sources can be enabled individually by the control bits in this register. An Interrupt Service Routine (ISR) evaluates the UARTx_IIR register to determine the signaling interrupt source.

The UARTx_IER register is mapped to the same address as UARTx_DLH. To access UARTx_IER, the DLAB bit in UARTx_LCR must be cleared.

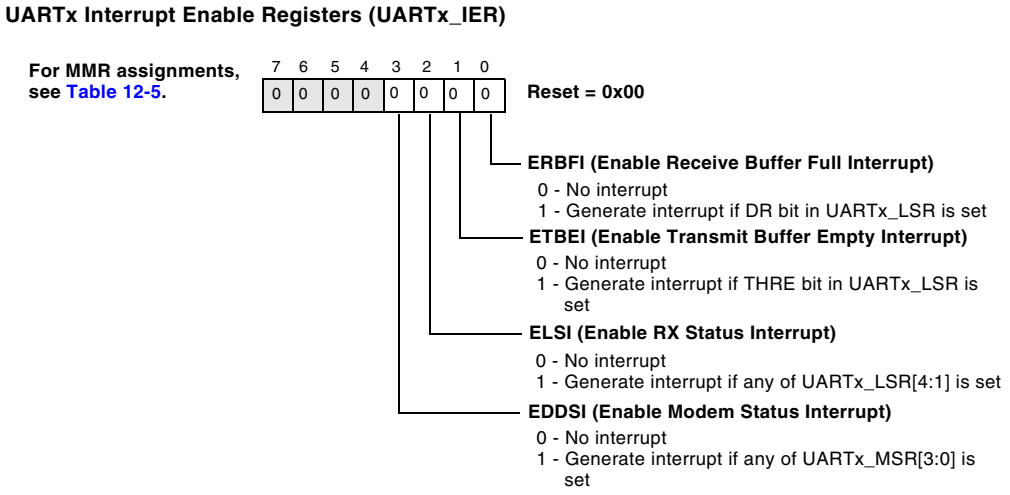


Figure 12-6. UARTx Interrupt Enable Registers

Table 12-5. UARTx Interrupt Enable Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_IER	0xFFC0 1802
UART1_IER	0xFFC0 1C02

The ELSI bit enables interrupt generation when any of the following conditions are raised by the respective bit in the UARTx Line Status register (UARTx_LSR):

- Receive Overrun Error (OE)
- Receive Parity Error (PE)
- Receive Framing Error (FE)
- Break Interrupt (BI)

The EDDSI bit enables interrupt generation when any of the following conditions are raised by the respective bit in the UARTx Modem Status register (UARTx_MSR):

- Delta CTS (DCTS)
- Delta DSR (DDSR)
- Trailing Edge RI (TERI)
- Delta DCD (DDCD)

On the ADSP-BF535 processor, the Modem Status interrupt is not generated if the LOOP bit in the UARTx Modem Control register (UARTx_MCR) is not set.

UARTx Interrupt Identification Registers (UARTx_IIR)

In non-DMA mode, a UART interrupt service routine (ISR) reads UARTx_IIR to determine the exact interrupt source, whenever more than one bit in the UARTx_IER register is set.

When cleared, the Pending Interrupt bit (NINT) signals that an interrupt is pending. The STATUS field indicates the highest priority pending interrupt (see Figure 12-7).

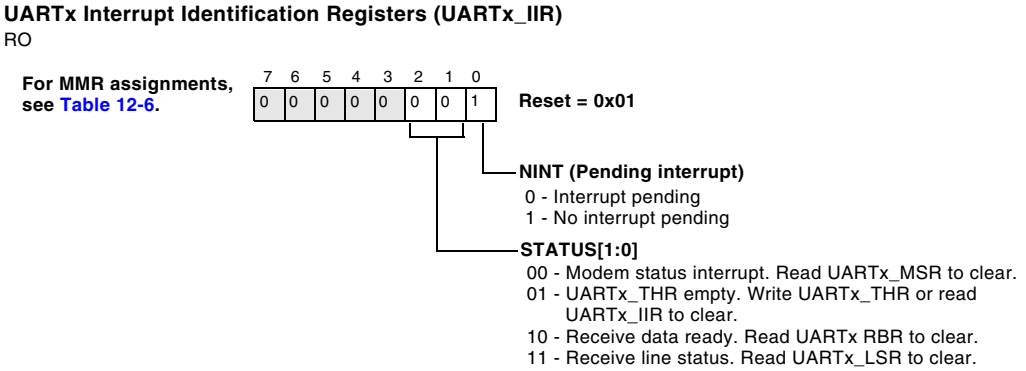


Figure 12-7. UARTx Interrupt Identification Registers

Table 12-6. UARTx Interrupt Identification Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_IIR	0xFFC0 1804
UART1_IIR	0xFFC0 1C04

Because of the destructive nature of these read operations, special care should be taken. See "Speculative Load Execution" on page 6-81 and "Conditional Load Behavior" on page 6-82 for more information.

UARTx Divisor Latch Registers (UARTx_DLL, UARTx_DLH)

The bit rate is characterized by the system clock (SCLK) and the 16-bit Divisor. The Divisor is split into the Divisor Latch Low Byte register (UARTx_DLL) and the Divisor Latch High Byte register (UARTx_DLH), as shown in Figure 12-8. These registers form a 16-bit divisor. The baud clock is divided by 16 so that:

```
BAUD RATE = SCLK/(16 × Divisor)
```

Divisor = 65,536 when UARTx_DLL = UARTx_DLH = 0

The UARTx_DLL register is mapped to the same address as the UARTx_THR and UARTx_RBR registers. The UARTx_DLH register is mapped to the same address as the Interrupt Enable register (UARTx_IER). The DLAB bit in UARTx_LCR must be set before the UARTx Divisor Latch registers can be accessed.

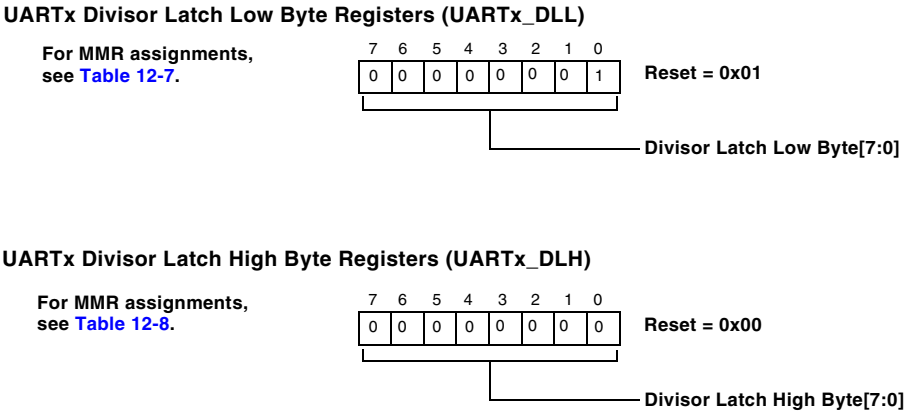


Figure 12-8. UARTx Divisor Latch Low Byte Registers and UARTx Divisor Latch High Byte Registers

Table 12-7. UARTx Divisor Latch Low Byte Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_DLL	0xFFC0 1800
UART1_DLL	0xFFC0 1C00

Table 12-8. UARTx Divisor Latch High Byte Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_DLH	0xFFC0 1802
UART1_DLH	0xFFC0 1C02

Note the 16-bit divisor resets to 0x0001 resulting in the highest possible clock frequency by default. If the UART is not used, changing the value of this register may help save power.

Table 12-9 provides example divide factors required to support most standard baud rates.

Table 12-9. 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.

UARTx Modem Control Registers (UARTx_MCR)

The UARTx_MCR register contains modem control and test signals, as shown in Figure 12-9.

Loopback mode forces the TX pin to high and disconnects the RX pin from the Receive Shift register (RSR), so the RSR input is directly connected to the Transmit Shift register (TSR) output. When Loopback mode is enabled, modem control signals in the UARTx Modem Control Register

(UARTx_MCR) are directly connected to the fields in the UARTx Modem Status Register (UARTx_MSR): RTS to CTS, DTR to DSR, OUT1 to RI, OUT2 to DCD.

The four modem control bits do not have any effect on the processor when the LOOP bit is not set. This is a read/write register that enables compatibility with PC standard UART devices.

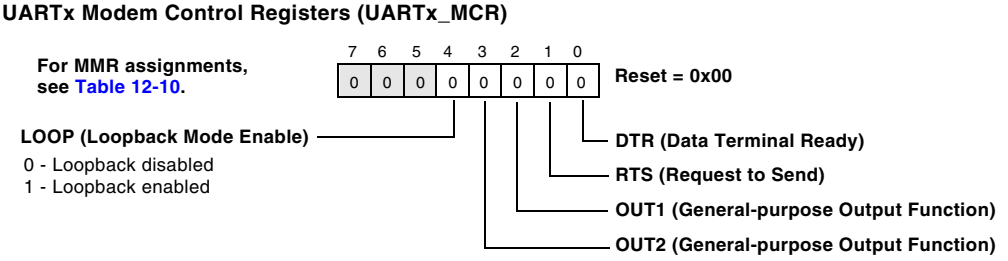


Figure 12-9. UARTx Modem Control Registers

Table 12-10. UARTx Modem Control Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_MCR	0xFFC0 1808
UART1_MCR	0xFFC0 1C08

UARTx Modem Status Registers (UARTx_MSR)

The UARTx_MSR register contains modem control status bits, as shown in Figure 12-10.

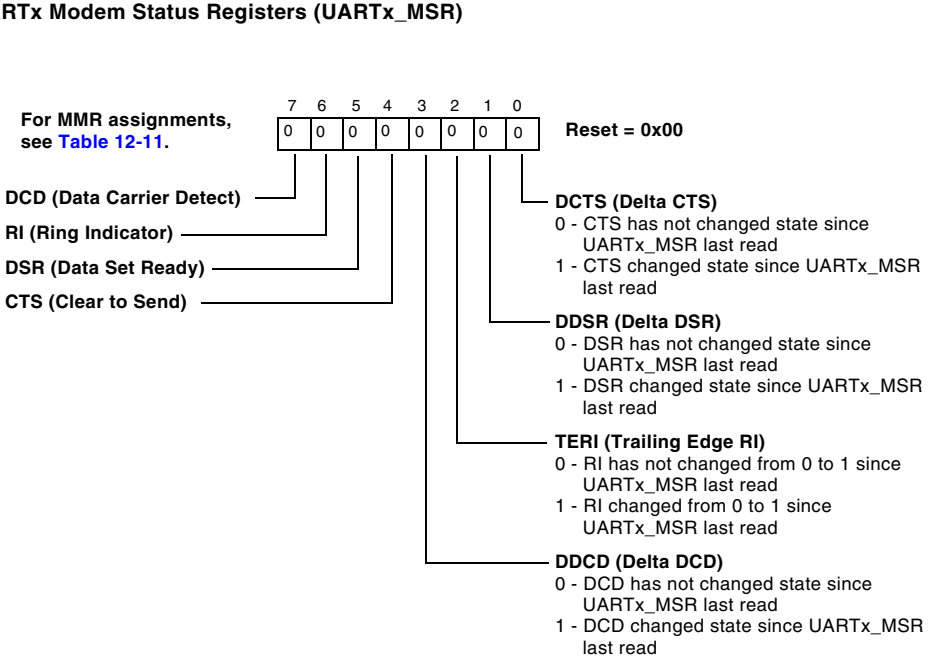


Figure 12-10. UARTx Modem Status Registers

Table 12-11. UARTx Modem Status Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_MSR	0xFFC0 180C
UART1_MSR	0xFFC0 1C0C

The Delta CTS (DCTS), Delta DSR (DDSR), Trailing Edge RI (TERI) and Delta DCD (DDCD) fields are cleared when the UARTx Modem Status register (UARTx_MSR) is read.

UARTx Scratch Registers (UARTx_SCR)

The contents of the 8-bit UARTx Scratch register (Figure 12-11) are not altered on reset. It is used for general-purpose data storage and does not control the UART hardware in any way.

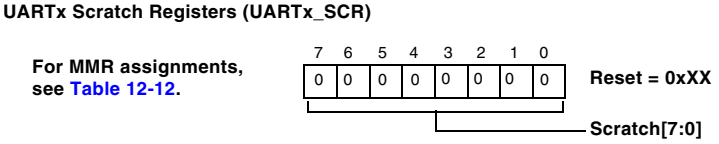


Figure 12-11. UARTx Scratch Registers

Table 12-12. UARTx Scratch Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_SCR	0xFFC0 180E
UART1_SCR	0xFFC0 1C0E

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 UARTx_THR. Received Data can be read from UARTx_RBR.

The ADSP-BF535 processor must write and read one character at time.

To prevent any loss of data and misalignments of the serial data stream, the UARTx Line Status Register (UARTx_LSR) provides two status flags for handshaking: THRE and DR.

The THRE flag is set when UARTx_THR is ready for new data and cleared when the processor loads new data into UARTx_THR. Writing UARTx_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 UARTx_RBR. This flag is cleared automatically when the processor reads from UARTx_RBR. Reading UARTx_RBR when it is not full returns the previously received value.

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.

Alternatively, UART writes and reads can be accomplished by ISRs. Both UART devices feature independent interrupt channels. In non-DMA mode, the UART TX interrupt channel is not used. All interrupt sources within one UART device share the same interrupt request, the UART RX interrupt channel. As controlled by UARTx_IER, this interrupt can be raised by any of the following:

- THR Empty (THRE) event
- RBR Full (DR) event
- Modem Status event
- Receive Error condition

The ISR can evaluate the Status bit field within the UARTx Interrupt Identification register (UARTx_IIR) to determine the exact interrupt source. Interrupts also must be assigned and unmasked by the ADSP-BF535 processor's interrupt controller.

DMA Mode

In this mode, separate receive (RX) and transmit (TX) DMA channels move data between the UART and memory. The processor 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 channels, 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 "Direct Memory Access" on page 9-1.

The DMA interrupt mechanism works independently from the UARTx_IER and the UARTx_IIR registers. The TX DMA has its own dedicated interrupt channel. The receive interrupt channel handles RX DMAs as well as receive error conditions when enabled in the UARTx Receive DMA Configuration register (UARTx_CONFIG_RX). DMA interrupt routines must explicitly write ones to the corresponding DMA IRQ status registers explicitly to clear the latched request of the pending interrupt.

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 that both UARTx_THR and the internal Transmit Shift register (TSR) are empty by testing the THRE and the TEMT status bits in UARTx_LSR. Otherwise, the processor must wait until the 2-bit DMA Buffer Status field within the appropriate UARTx Transmit DMA Configuration register (UARTx_CONFIG_TX) is clear.

UART DMA Receive Registers

Each of the two UART modules provides two complete independent DMA channels. The transmit channel reads from memory and the receive channel writes to memory. Every DMA channel features its own set of control registers. For more information on DMA registers, see "Direct Memory Access" on page 9-1.

UARTx Receive DMA Current Descriptor Pointer Registers (UARTx_CURR_PTR_RX)

Figure 12-12 shows the UARTx Receive DMA Current Descriptor Point registers.

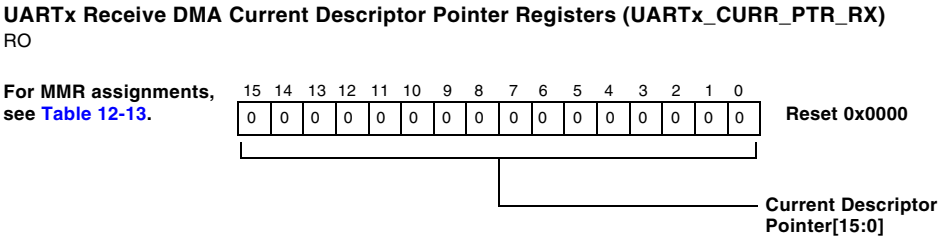


Figure 12-12. UARTx Receive DMA Current Descriptor Pointer Registers

Table 12-13. UARTx Receive DMA Current Descriptor Pointer Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_CURR_PTR_RX	0xFFC0 1A00
UART1_CURR_PTR_RX	0xFFC0 1E00

This register defines the lower 16 bits of the Current Descriptor Pointer. The complete 32-bit address is created by combining this value with the upper 16 bits from the DMA Descriptor Base Pointer Register (DMA_DBP). For more information on the DMA Descriptor Base Pointer Register, see “Direct Memory Access” on page 9-1.

UARTx Receive DMA Configuration Registers (UARTx_CONFIG_RX)

Figure 12-13 shows the UARTx Receive DMA Configuration registers.

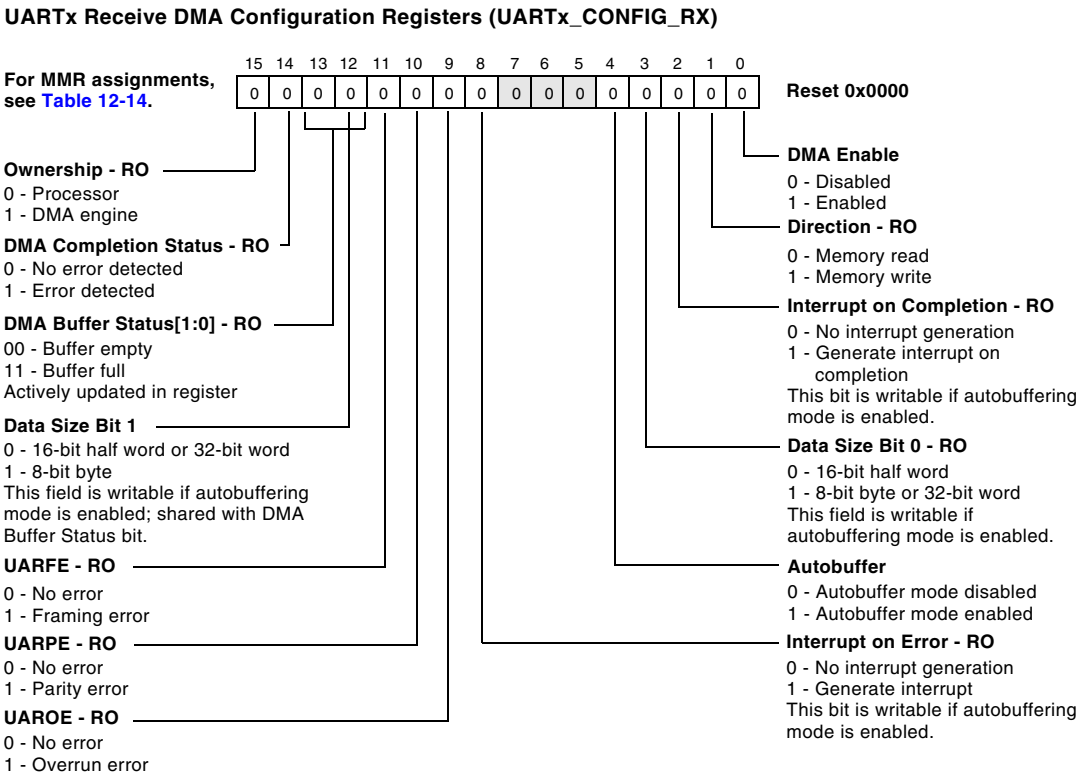


Figure 12-13. UARTx Receive DMA Configuration Registers

Table 12-14. UARTx Receive DMA Configuration Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_CONFIG_RX	0xFFC0 1A02
UART1_CONFIG_RX	0xFFC0 1E02

When autobuffering is enabled, Data Size Bit 0 and Data Size Bit 1 are used to determine the size of a DMA transfer. Table 12-15 defines the allowed DMA transfer sizes.

Table 12-15. DMA Transfer Sizes

Data Size Bit 1	Data Size Bit 0	Transfer Size
0	0	16-bit half word
0	1	32-bit word
1	0	Reserved
1	1	8-bit byte

Note the receive DMA always writes to memory, so the Direction bit should always be set.

If the Interrupt on Error bit is set, a UARTx RX interrupt is issued when either an overflow error, a parity error, or a framing error has been detected. The RX interrupt service routine should evaluate bit 1 of the `UARTx_IRQSTAT_RX` register in order to determine whether it was requested by a line error condition rather than by the DMA completion. If this bit is set, the service routine should read the `UARTx_LSR` register to determine the error condition.

Reading `UARTx_LSR` also clears the UART's interrupt request. Since the error request has also been latched in the DMA engine, the error handler must also write a 1 to bit 1 of the `UARTx_IRQSTAT_RX` afterward.

In descriptor DMA mode, individual descriptors may be configured to disable interrupt generation. The erroneous DMA sequence can still be located. In this case, the UART error flags are mirrored in the `UARTx_CONFIG_RX` register and this one is copied to DMA descriptor in memory afterward.

UARTx Receive DMA Start Address High Registers (UARTx_START_ADDR_HI_RX)

Figure 12-14 shows the UARTx Receive DMA Start Address High registers.

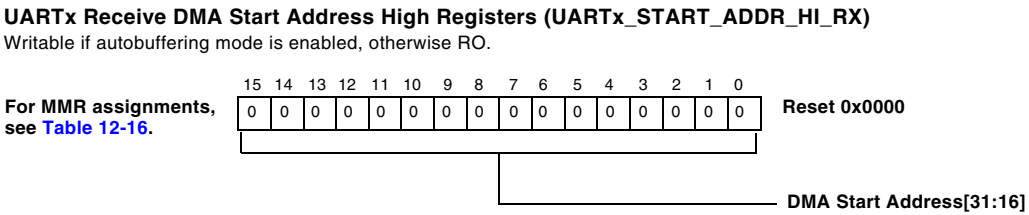


Figure 12-14. UARTx Receive DMA Start Address High Registers

Table 12-16. UARTx Receive DMA Start Address High Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_START_ADDR_HI_RX	0xFFC0 1A04
UART1_START_ADDR_HI_RX	0xFFC0 1E04

UARTx Receive DMA Start Address Low Registers (UARTx_START_ADDR_LO_RX)

Figure 12-15 shows the UARTx Receive DMA Start Address Low registers.

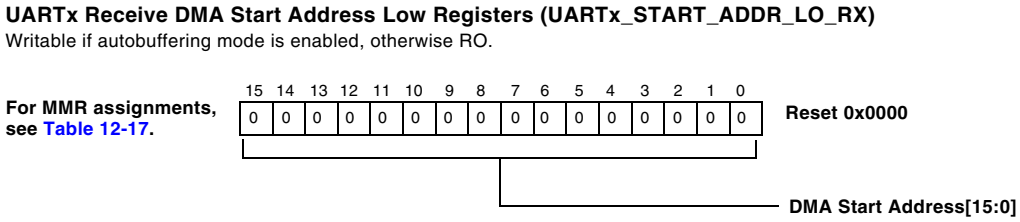


Figure 12-15. UARTx Receive DMA Start Address Low Registers

Table 12-17. UARTx Receive DMA Start Address Low Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_START_ADDR_LO_RX	0xFFC0 1A06
UART1_START_ADDR_LO_RX	0xFFC0 1E06

UARTx Receive DMA Count Registers (UARTx_COUNT_RX)

Figure 12-16 shows the UARTx Receive DMA Count registers.

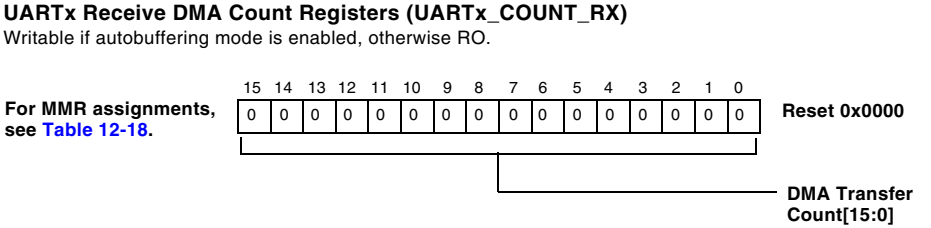


Figure 12-16. UARTx Receive DMA Count Registers

Table 12-18. UARTx Receive DMA Count Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_COUNT_RX	0xFFC0 1A08
UART1_COUNT_RX	0xFFC0 1E08

UARTx Receive DMA Next Descriptor Pointer Registers (UARTx_NEXT_DESCR_RX)

Figure 12-17 shows the UARTx Receive DMA Next Descriptor Pointer registers.

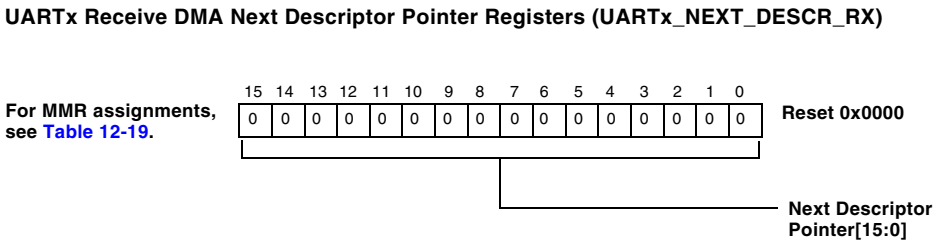


Figure 12-17. UARTx Receive DMA Next Descriptor Pointer Registers

Table 12-19. UARTx Receive DMA Next Descriptor Pointer Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_NEXT_DESCR_RX	0xFFC0 1A0A
UART1_NEXT_DESCR_RX	0xFFC0 1E0A

This register defines the lower 16 bits of the Next Descriptor Pointer. The complete 32-bit address is created by combining this value with the upper 16 bits from the DMA Descriptor Base Pointer Register (`DMA_DBP`). For more information on the DMA Descriptor Base Pointer Register, see "Direct Memory Access" on page 9-1.

UARTx Receive DMA Descriptor Ready Registers (UARTx_DESCR_RDY_RX)

Figure 12-18 shows the UARTx Receive DMA Descriptor Ready registers.

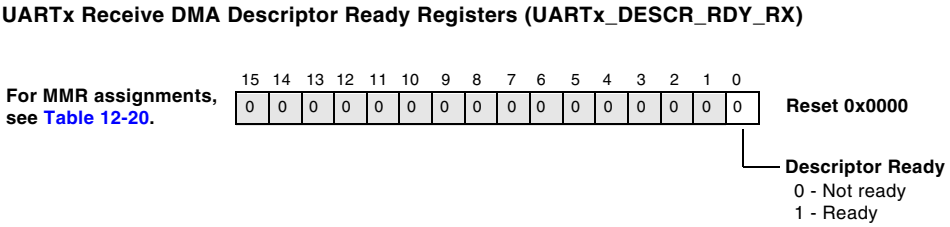


Figure 12-18. UARTx Receive DMA Descriptor Ready Registers

Table 12-20. UARTx Receive DMA Descriptor Ready Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_DESCR_RDY_RX	0xFFC0 1A0C
UART1_DESCR_RDY_RX	0xFFC0 1E0C

UARTx Receive DMA IRQ Status Registers (UARTx_IRQSTAT_RX)

Figure 12-19 shows the UARTx Receive DMA IRQ Status registers.

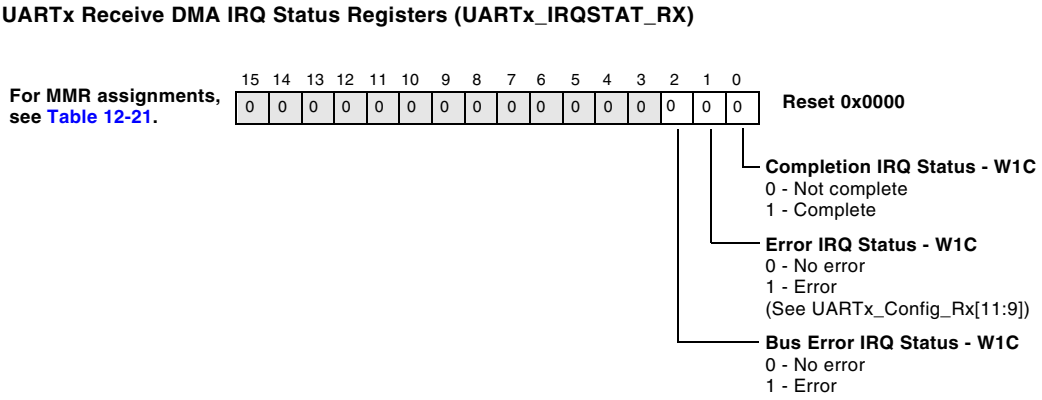


Figure 12-19. UARTx Receive DMA IRQ Status Registers

Table 12-21. UARTx Receive DMA IRQ Status Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_IRQSTAT_RX	0xFFC0 1A0E
UART1_IRQSTAT_RX	0xFFC0 1E0E

UART DMA Transmit Registers

For more information on DMA registers, see “Direct Memory Access” on page 9-1.

UARTx Transmit DMA Current Descriptor Pointer Registers (UARTx_CURR_PTR_TX)

Figure 12-20 shows the UARTx Transmit DMA Current Descriptor Pointer registers.

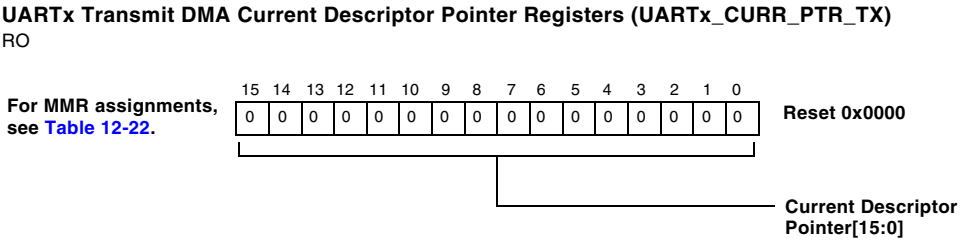


Figure 12-20. UARTx Transmit DMA Current Descriptor Pointer Registers

Table 12-22. UARTx Transmit DMA Current Descriptor Pointer Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_CURR_PTR_TX	0xFFC0 1B00
UART1_CURR_PTR_TX	0xFFC0 1F00

This register defines the lower 16 bits of the Current Descriptor Pointer. The complete 32-bit address is created by combining this value with the upper 16 bits from the DMA Descriptor Base Pointer Register (DMA_DBP). For more information on the DMA Descriptor Base Pointer Register, see "Direct Memory Access" on page 9-1.

UARTx Transmit DMA Configuration Registers (UARTx_CONFIG_TX)

Figure 12-21 shows the UARTx Transmit DMA Configuration registers.

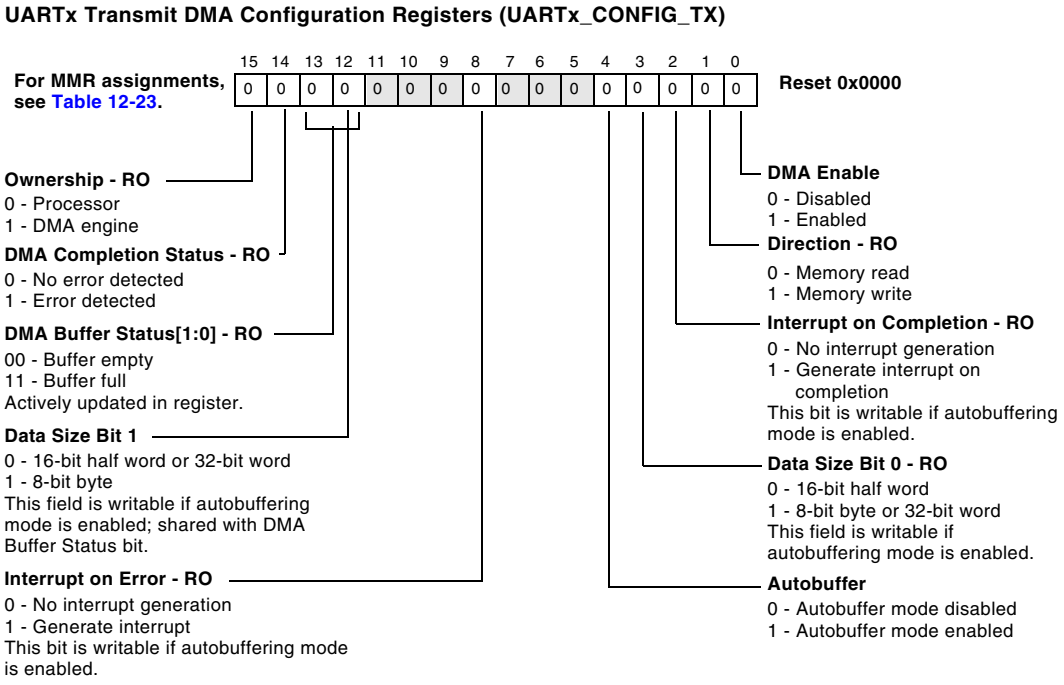


Figure 12-21. UARTx Transmit DMA Configuration Registers

Table 12-23. UARTx Transmit DMA Configuration Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_CONFIG_TX	0xFFC0 1B02
UART1_CONFIG_TX	0xFFC0 1F02

When autobuffering is enabled, Data Size Bit 0 and Data Size Bit 1 are used to determine the size of a DMA transfer. Table 12-24 defines the allowed DMA transfer sizes.

Table 12-24. DMA Transfer Sizes

Data Size Bit 1	Data Size Bit 0	Transfer Size
0	0	16-bit half word
0	1	32-bit word
1	0	Reserved
1	1	8-bit byte

UARTx Transmit DMA Start Address High Registers (UARTx_START_ADDR_HI_TX)

Figure 12-22 shows the UARTx Transmit DMA Start Address High registers.

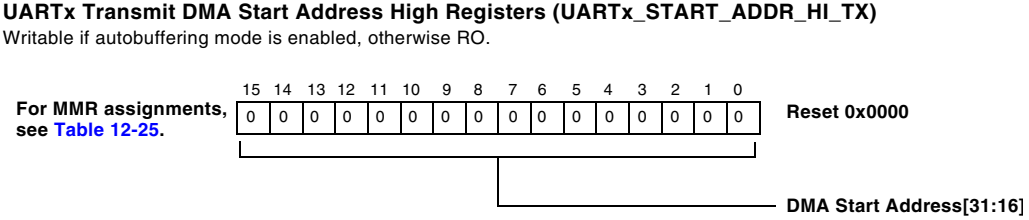


Figure 12-22. UARTx Transmit DMA Start Address High Registers

Table 12-25. UARTx Transmit DMA Start Address High Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_START_ADDR_HI_TX	0xFFC0 1B04
UART1_START_ADDR_HI_TX	0xFFC0 1F04

UARTx Transmit DMA Start Address Low Registers (UARTx_START_ADDR_LO_TX)

Figure 12-23 shows the UARTx Transmit DMA Start Address Low registers.

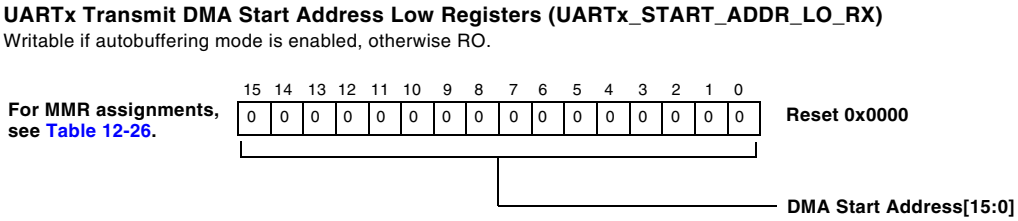


Figure 12-23. UARTx Transmit DMA Start Address Low Registers

Table 12-26. UARTx Transmit DMA Start Address Low Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_START_ADDR_LO_TX	0xFFC0 1B06
UART1_START_ADDR_LO_TX	0xFFC0 1F06

UARTx Transmit DMA Count Registers (UARTx_COUNT_TX)

Figure 12-24 shows the UARTx Transmit DMA Count registers.

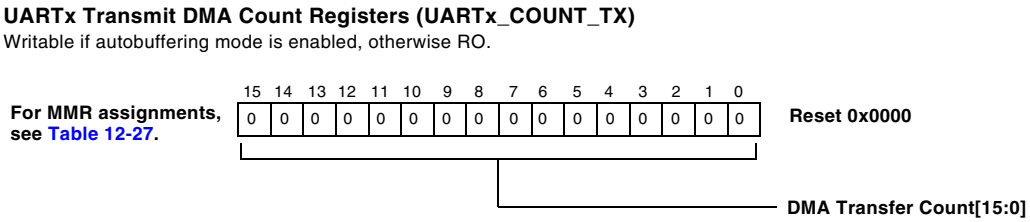


Figure 12-24. UARTx Transmit DMA Count Registers

Table 12-27. UARTx Transmit DMA Count Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_COUNT_TX	0xFFC0 1B08
UART1_COUNT_TX	0xFFC0 1F08

UARTx Transmit DMA Next Descriptor Pointer Registers (UARTx_NEXT_DESCR_TX)

Figure 12-25 shows the UARTx Transmit DMA Next Descriptor Pointer registers.

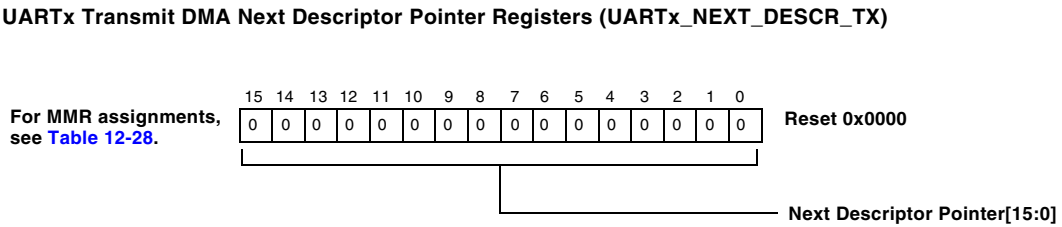


Figure 12-25. UARTx Transmit DMA Next Descriptor Pointer Registers

Table 12-28. UARTx Transmit DMA Next Descriptor Pointer Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_NEXT_DESCR_TX	0xFFC0 1B0A
UART1_NEXT_DESCR_TX	0xFFC0 1F0A

This register defines the lower 16 bits of the Next Descriptor Pointer. The complete 32-bit address is created by combining this value with the upper 16 bits from the DMA Descriptor Base Pointer Register (DMA_DBP). For more information on the DMA Descriptor Base Pointer Register, see "Direct Memory Access" on page 9-1.

UARTx Transmit DMA Descriptor Ready Registers (UARTx_DESCR_RDY_TX)

Figure 12-26 shows the UARTx Transmit DMA Descriptor Ready registers.

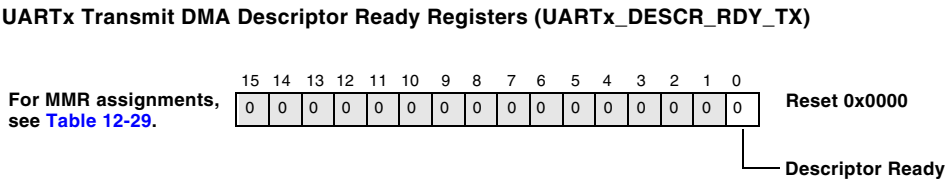


Figure 12-26. UARTx Transmit DMA Descriptor Ready Registers

Table 12-29. UARTx Transmit DMA Descriptor Ready Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_DESCR_RDY_TX	0xFFC0 1B0C
UART1_DESCR_RDY_TX	0xFFC0 1F0C

UARTx Transmit DMA IRQ Status Registers (UARTx_IRQSTAT_TX)

Figure 12-27 shows the UARTx Transmit DMA IRQ Status registers.

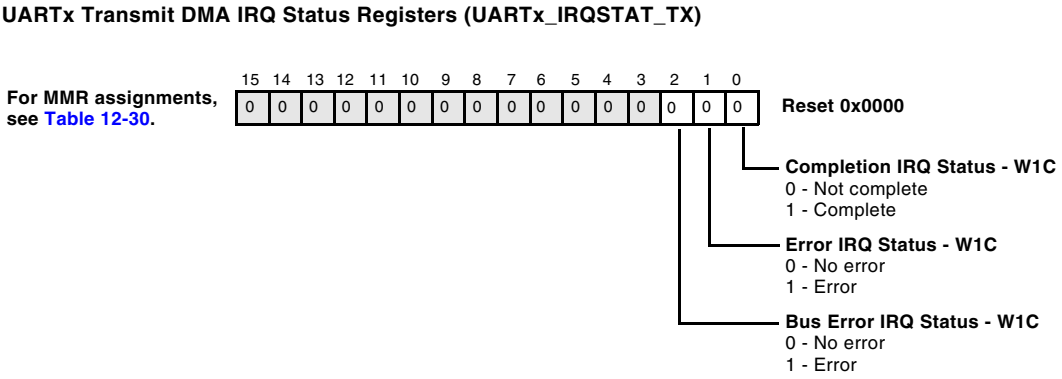


Figure 12-27. UARTx Transmit DMA IRQ Status Registers

Table 12-30. UARTx Transmit DMA IRQ Status Register MMR Assignments

Register Name	Memory-Mapped Address
UART0_IRQSTAT_TX	0xFFC0 1B0E
UART1_IRQSTAT_TX	0xFFC0 1F0E

IrDA Support

Aside from the standard UART functionality, UART0 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 date 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. The minimum pulse duration, that is, the time required for valid 0 receive data to be detected, is programmable.

IrDA support is enabled by setting the `IREN` bit in the UART0 Infrared Control register. The IrDA application requires external transceivers.

UART0 Infrared Control Register (UART0_IRCR)

The `UART0_IRCR` register, shown in Figure 12-28, contains bits that control all IrDA related features. These include the enable bit for IrDA mode of operation, bits to specify the minimum receive pulse width, and a bit to specify the polarity of the data received.

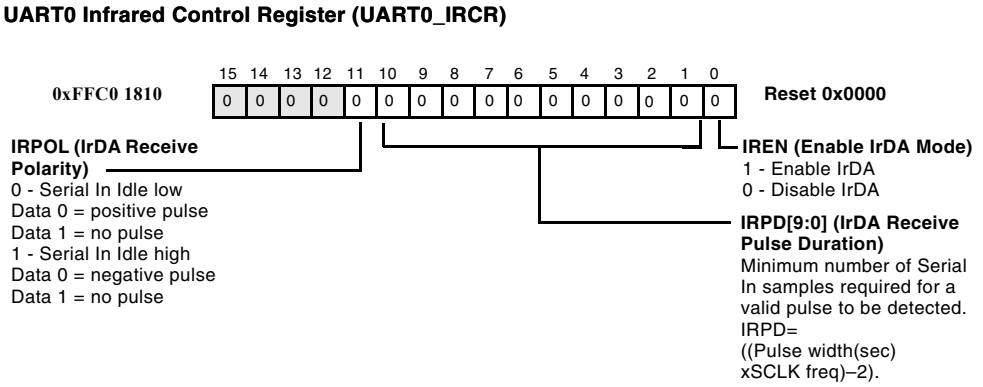


Figure 12-28. UART0 Infrared Control Register

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 12-29. The final IrDA pulse is fed to the off-chip infrared driver.

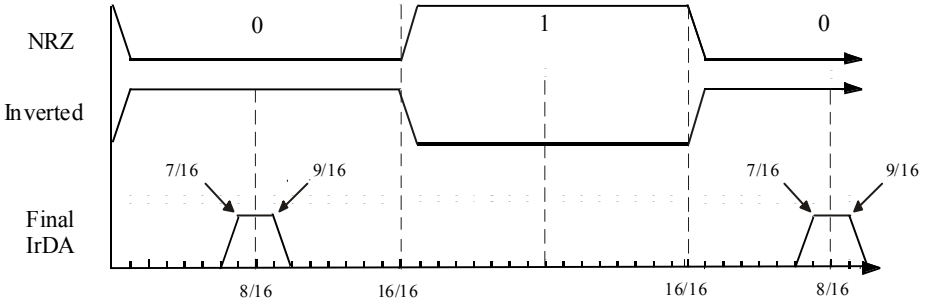


Figure 12-29. IrDA Transmit Pulse

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 12-9 on page 12-12, 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. In addition, pulse widths of less than 1 UART clock period must be detected per existing standards. Consequently, both the position of valid data relative to the UART clock and the pulse duration must be considered.

The filter function enables IrDA sampling from the sixth through the eleventh 16x clock periods. The window is wider than the transmitted IrDA pulse to allow for clock frequency discrepancies between the transmitter and receiver. Within this window, the receiver incorporates minimum pulse duration logic to detect valid data.

The pulse duration is specified by the IRPD bits using this formula:

IRPD = (Desired Pulse Width(sec) * SLCK frequency)–2

For example, the Infrared Data Association specifies a receive pulse duration minimum of 1.41 µS at 9.6 Kbits/second. For a SCLK frequency of 100 MHz, IRPD should contain a value of 0x8b. In this example, receive data during the sampling window, whose duration is greater than 141 SCLK cycles, is treated as valid data. Figure 12-30 shows the receiver functionality.

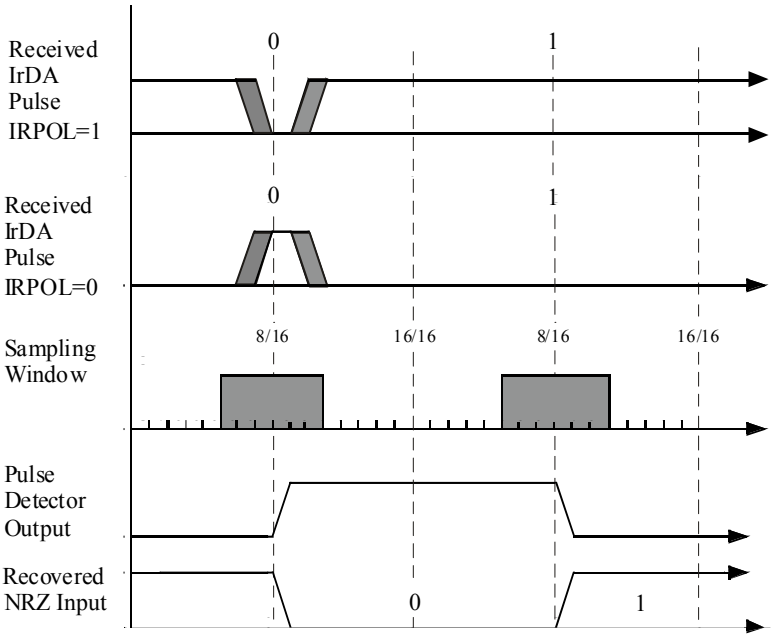


Figure 12-30. IrDA Receiver Pulse Detection

The receive sampling window is determined by a counter, clocked at the 16x bit-time sample clock. The sampling window is resynchronized 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 12-30 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.

13 PCI BUS INTERFACE

The ADSP-BF535 processor includes a Revision 2.2-compliant, 33 MHz, 32-bit Peripheral Component Interconnect (PCI) bus interface. The PCI interface provides a bus bridge function between an external PCI bus and the memories, the peripherals, and the processor core which are internal to the ADSP-BF535 processor. The PCI bus interface supports these functions:

- PCI device, in which an ADSP-BF535-processor-based system can be easily interfaced to a 3.3V Revision 2.2-compliant PCI bus.
- Host-to-PCI bridge, in which ADSP-BF535 processor resources (the core, internal and external memory, and the memory DMA controller) provide the necessary hardware components to act as the system controller of a PCI bus. This provides peripheral expansion for the ADSP-BF535 processor, from the perspective of one or more PCI devices.

In either configuration, the PCI can function as either the PCI initiator (master) or the PCI target (slave) for a PCI bus transaction.

The PCI bus interface consists of a PCI controller, interfaces to internal buses, and drivers for the external PCI bus (see the block diagram in Figure 13-1).

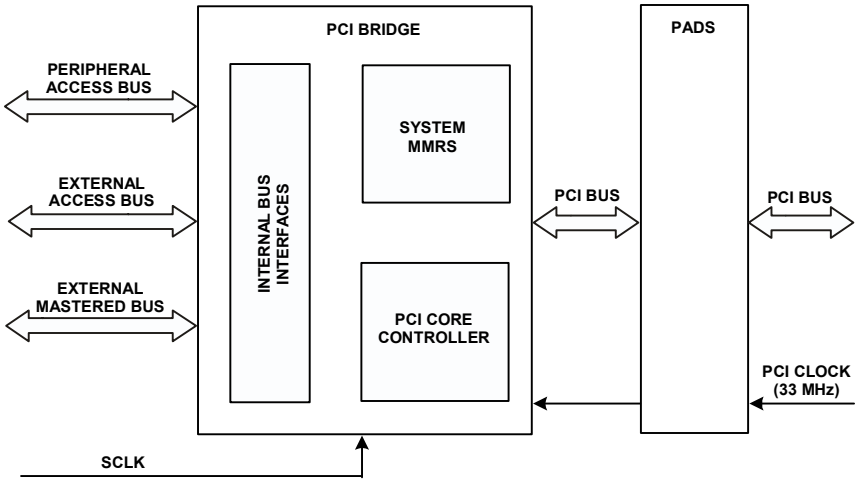


Figure 13-1. PCI Bridge Block Diagram

The PCI Bridge includes the PCI core controller, control and status memory-mapped registers, and the logic required to interface to the three internal buses.

The PCI external interface (PADs in the block diagram) includes special I/O drivers and dedicated power supplies that support the PCI electrical characteristics. The PCI operates at frequencies up to 33 MHz at 3.3 V, and is asynchronous with respect to the ADSP-BF535 processor system clock. The PCI bus multiplexes data transfers on the same bus used for address transfers and supports a maximum transfer rate of 132 MB/second.

PCI Specification

The PCI bus interface conforms to PCI Local Bus Specification Rev. 2.2. The specification is maintained by the PCI Special Interest Group and can be obtained from http://www.pcisig.com.

The PCI specification refers to a 32-bit wide word as a double-word, and to 16 bits as a word. This document defines 32 bits as a word, 16 bits as a half word, and 8 bits as a byte.

PCI Device Function

The PCI device function supports a glueless logical and electrical interface to a 3.3V PCI Revision 2.2-compliant bus platform (such as a PC) as a single function device. This enables the development of intelligent PCI peripheral systems based on the ADSP-BF535 processor. The ADSP-BF535 processor PCI bus interface does not support multifunction device applications; it supports one configuration space. The ADSP-BF535 processor can also support a PCI/USB bridge function.

As a PCI device, the PCI controller can use the Memory DMA controller to perform DMA transfers as required by the PCI host.

PCI Host Function

The ADSP-BF535 processor provides the PCI host (platform) functions required to support and control numerous off-the-shelf PCI device functions (ethernet controllers, bus bridges, and so on) in a system in which the core processor is the host.

Processor Core Access to PCI Space

The Blackfin architecture defines only memory space; it does not define I/O or configuration address spaces. The three memory spaces of PCI space (memory, I/O, and configuration space) are mapped into the flat 32-bit core memory space. Since the PCI memory space is as large as the full memory address space of the ADSP-BF535 processor, a segmented, or windowed, approach must be employed, so that only portions of the PCI

address spaces are visible to the processor core at any one time. The diagram in Figure 13-2 shows the ADSP-BF535 processor external memory map, including the PCI-space components, and their address ranges.

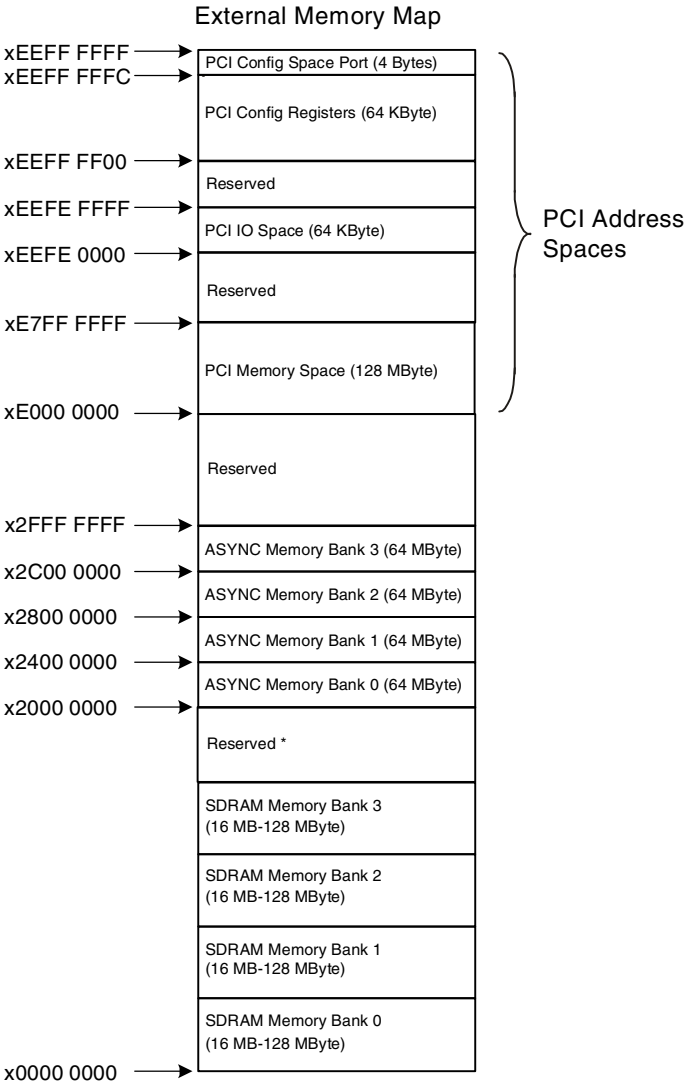
External PCI Requirements

Although most functions are integrated into the ADSP-BF535 processor PCI device, several are not, including:

- PCI bus arbitration. An external device, such as a programmable array logic (PAL) device, is required to implement this function.
- Pull-up resistors on signals that may not always be driven to a valid state.
- Non-volatile memory. The intent is that EE/flash PROM requirements for a PCI initiator can be supported through programmable ROM attached to the EBIU.
- Direct method for selecting devices when configured in host mode. The contents of the `PCI_CBAP` register are placed directly on the bus when performing a configuration read or write. External hardware is needed to generate device selection IDs (`IDSEL`s). ID selecting device 0 (the ADSP-BF535 processor device) results in a master abort.

Device Mode Operation

The ADSP-BF535 processor is in device mode when the Host/device bit in the PCI Bridge Control register is cleared (its default setting).



Note: Reserved off-chip memory areas are labeled in the diagram above. All other off-chip system resources are addressable by both the core and the system.
* This reserved block does not exist if all four blocks of SDRAM are configured to their full 128 MByte size.

Figure 13-2. PCI Memory Map

Outbound Transactions (ADSP-BF535 Processor as PCI Initiator)

This section describes outbound transactions with the ADSP-BF535 processor as the PCI initiator.

General Outbound Operation

Read and write transactions to PCI are achieved by first writing to the appropriate Base Address register to position the mapping windows in PCI address space. Write to the PCI_MBAP register for memory accesses, to the PCI_IBAP register for I/O accesses, or to the PCI_CBAP register for configuration accesses. After writing to the appropriate register, a read or write to the appropriate data window initiates a PCI transaction.

For memory and I/O accesses, the address put onto the PCI Address/Data bus (PCI AD) is a concatenation of the upper bits of the Base Address pointer (MBAP or IBAP) and the lower bits of the load or store target address. These lower bits are actually the offset into the static-sized PCI Memory or I/O windows. Table 13-1 shows the PCI data windows for outbound reads and writes.

Table 13-1. Data Windows in EAB for Outbound Transactions

Name	Lower Address	Upper Address	Size
Memory Data Window	0xE000 0000	0xE7FF FFFF	128 MB
I/O Data Window	0xEEFE 0000	0xEEFE FFFF	64 KB
Config Data Port	0xEEFF FFFC	0xEEFF FFFF	1 Word (32 bits)

For all outbound read and write transactions, the control signals of the transactions are placed into a FIFO that is written in the SCLK domain and read by the PCI core in the PCI clock domain. For writes, the data written

to the EAB bus is put into the master TX FIFO to be put onto the PCI AD bus during the data phase. For reads, the PCI AD bus data is placed into the master RX FIFO during the data phase.

Each of the data FIFOs is 8 × 32-bit words deep. The transaction FIFO is 4 transactions deep. Writes are posted as long as there is room in both the transaction and the TX data FIFOs. Reads are delayed (slave inserts wait states) until there is room in the transaction FIFO, then until the read transaction makes it through to the PCI side of the FIFO, and then until the data is returned through the RX FIFO. Thus, reads from PCI space clear all three of the FIFOs.

If the transaction FIFO is full, all transactions are delayed until the next spot in the FIFO is opened. Even if the transaction FIFO only has one element in it, the TX FIFO could be either full or not have enough space for the size burst being attempted, so the write is again delayed. Three status bits have been provided in the PCI_STAT register that let you interrogate whether the TX FIFO is full or empty and whether the transaction FIFO is full. The PCI Master TX FIFO Empty status bit is sticky and can be masked by the PCI_ICTL register to generate an interrupt to improve efficiency when doing burst writes to PCI.

Outbound Error Detection and Reporting

All addresses between 0xE000 0000 and 0xEEFF FFFF are decoded to select the PCI interface, but only part of this space is valid. There are four windows in this space that have been marked for specific use by the PCI interface. Three of these are defined in Table 13-1. The fourth consists of the PCI configuration registers, from 0xEEFF FF00 through 0xEEFF FFFB. Any accesses made between these four valid areas (that is, above 0xE000 0000, below 0xEEFF FFFF, and not within one of the windows) results in an error. The logic in the PCI module generates a bus error and completes the transaction without actually reading or writing any data. The Unsupported EAB Access sticky bit in the PCI_STAT register is set, and causes an interrupt if masked to do so in the PCI_ICTL register.

Any accesses that are valid on the EAB bus but cause fatal errors on PCI do not result in a bus error, but the transaction completes without doing any actual data reads or writes. A fatal error causes the PCI Fatal Error sticky bit in the PCI_STAT register to be set, and an interrupt occurs to the core if masked to do so in the PCI_ICTL register. PCI fatal errors occur in these cases:

- The requested PCI target does not respond. This usually indicates an invalid address. The core executes a master abort cycle.
- The requested PCI target terminates the cycle with a target abort.
- The requested PCI target responds with a PCI_DEVSEL#, but does not follow with PCI_TRDY or PCI_STOP to allow the cycle to complete. The master abandons the cycle after 128 cycles.
- The PCI target device retried beyond the retry count. The master abandons the cycle after 128 retries.

If a parity error occurs for an access to the PCI bus during the actual transaction or at the end of the transaction, the PCI Parity Error sticky bit in the PCI_STAT register is set. An interrupt to the core occurs if masked to do so in the PCI_ICTL register.

Supported Transactions to PCI

The PCI interface is capable of handling these types of transactions:

- Single 8-bit, 16-bit, and 32-bit
- 4-beat incrementing 8-bit, 16-bit, and 32-bit bursts
- 8-beat incrementing 8-bit, 16-bit, and 32-bit bursts

It is important to note that the PCI interface does not support wrapping bursts. Any access identified as a wrap signals a bus error and sets the Unsupported EAB Access bit in the PCI_STAT register. The interface treats wrapping bursts as incrementing bursts, but this behavior should not be relied upon for normal operation.

Since core cache burst transactions are wrapping bursts, all core access to the PCI module must be cache inhibited. This is accomplished by correct configuration of the core Cacheability Protection Lookaside Buffers (CPLBs), which describe the access characteristics of the core memory map. The data CPLB entry or entries used to describe the PCI memory and I/O spaces must have the CPLB_L1_CHBL bit cleared. Note that core instruction fetch from PCI space is not supported.

The PCI protocol is incapable of doing a 4-sequential byte burst, or a 2-sequential 16-bit burst. For bursts smaller than 32 bits, the data is packed into words and bursted to PCI if necessary. Table 13-2 shows the PCI transaction produced for each supported burst size.

Table 13-2. Burst Transaction Conversion to PCI

Supported EAB Burst Transaction	Number and Type of PCI Transaction Produced
4-beat incrementing 8-bit	Single 32-bit word transaction
8-beat incrementing 8-bit	Two 32-bit word bursts
4-beat incrementing 16-bit	Two 32-bit word bursts
8-beat incrementing 16-bit	Four 32-bit word bursts
4-beat incrementing 32-bit	Four 32-bit word bursts
8-beat incrementing 32-bit	Eight 32-bit word bursts

Inbound Transactions (ADSP-BF535 Processor as PCI Target)

This section describes inbound transactions with the ADSP-BF535 processor as the PCI target.

General Inbound Operation

To participate on a PCI bus as a target, the ADSP-BF535 processor must first be configured. First, the processor core must write to the Memory and I/O BAR Mask registers. Every bit set by the core in each of these masks allows the corresponding bit in the memory base address register (BAR) or I/O BAR, respectively, to be written by the PCI host during configuration. The number of upper bits writable by the PCI determines the size in bits of the base address register that configuration software on the PCI side programs into the BAR. This determines the size of the window for which the PCI interface can claim the transaction as the intended target.

Each bit of the mask registers must be set to 1 by the processor core, starting with the uppermost bit (bit 31). To prevent the core from claiming either memory or I/O transactions, the corresponding mask should be programmed with all 0s (the default state). The pattern of all 1s is assumed by the PCI interface to be contiguous, stray 0s make the bit non-writable, and cause unpredictable results.

The processor core must also program the `PCI_TMBAP` and `PCI_TIBAP` registers. These registers are the base address pointers (BAPs) for incoming memory and I/O transactions, respectively. The number of bits programmed into the upper portion of each of these registers must be the same as the number of bits set in the corresponding mask. Any bits written in the space where 0s exist in the mask are ignored, but the write to the BAP completes successfully. If memory or I/O transactions are not needed, then the corresponding BAP register is ignored and does not need to be programmed. When the processor core has completed writing to

both masks and both BAPs, it should allow PCI configuration software running on the host processor to configure the PCI interface from the PCI bus. This is done by setting the PCI Enable bit in the PCI_CTL register. Any configuration accesses to the PCI module before this bit is set are retried by the PCI module. It is a stipulation of the PCI protocol that this bit must be set. That is, the device has up to 2^{25} PCI cycles to initialize itself after PCI_RST# is asserted.

When PCI configuration software has configured the PCI interface, it is configured with a memory and/or I/O address range for which it claims the current transaction as the target.

There is no incoming transaction FIFO, so only one transaction can occur at a time. Each of the data FIFOs is 8 × 32-bit words deep. If a burst greater than 8 words comes in, then each time the FIFO fills up, the PCI bus is retried until space is available for the next word.

Two status bits in the PCI_STAT register are set whenever the Slave TX Data FIFO or Slave RX Data FIFO have data written into them. These bits are sticky and can be configured by the PCI_ICTL register to generate interrupts. They can be used to determine PCI target bandwidth efficiency or for general debugging of PCI target operation.

The PCI interface can also perform fast back-to-back accesses, which make multiple word accesses (not bursts) more efficient by removing the usually mandatory single idle cycle between transfers. The PCI interface can only perform fast back-to-back transfers when configuration software sets the Fast Back-to-Back Enable bit in the command register of the core PCI configuration space. This bit is set when all of the slaves on the PCI bus are fast back-to-back capable. The ADSP-BF535 processor must also enable its own ability to do fast back-to-back transfers by setting the Fast Back-to-Back Enable bit in the PCI_CTL register. This bit may be set and cleared at any time during operation to enable and disable fast back-to-back transfers whenever necessary.

Inbound Error Detection and Reporting

Inbound transactions to ADSP-BF535 processor resources that are not accessible to the PCI bus result in an error. Reads to non-accessible resources result in the Error on Inbound Read bit being set in the PCI_STAT register and an interrupt to the processor core, if masked to do so in the PCI_ICTL register. The waiting read on the PCI side is aborted. Writes to non-accessible resources result in the Error on Inbound Write bit being set in the PCI_STAT register and an interrupt to the core, if masked to do so in the PCI_ICTL register. Writes are posted so that the transaction may already be finished. However, if the transaction (for example, a burst) is long enough, it is aborted by the error.

Supported Transactions From PCI

The PCI interface supports single 32-bit word memory and I/O accesses. It also supports memory bursts of any number of 32-bit words. However, the data FIFOs for inbound accesses are only 8 words deep, and if the write data FIFO becomes full in the middle of a burst, the interface *retries* the PCI transaction. I/O bursts to the ADSP-BF535 processor are not supported. The PCI interface also supports inbound fast back-to-back transactions, a method of doing more than one single 32-bit word transfer quickly, without the usual idle cycle between transfers. Inbound fast back-to-back transfers are passed through as ordinary single word accesses.

Unsupported Transactions From PCI

PCI as a protocol is capable of addressing any combination of the 4 bytes in a single word using its byte enable bits. This means that the ADSP-BF535 processor could receive a transfer that it does not support, such as a read or a write to bytes 0 and 2 of a word. On reads, this is not a problem because the whole word can be read and the master of the transaction can take the data it wants.

A problem arises on writes because the internal bus protocols employed on the ADSP-BF535 processor support only byte, half word, and word transfers. In an effort to avoid the complexity of breaking writes into multiple transactions, all writes are assumed to have contiguous byte enables and the number of bits written to is rounded up to the nearest supported transfer size. This means that attempting to write from PCI to only the first and third bytes results in writing to the whole word, with possible corruption of data.

Host Mode Operation

When the ADSP-BF535 processor is in host mode, the core acts as the system processor with the PCI interface acting as its host-to-PCI bridge. The ADSP-BF535 processor is put into host mode by setting the Host/Device Switch bit in the `PCI_CTL` register. In host mode, the core must perform PCI configuration on any devices attached to the PCI bus. The core must also configure the ADSP-BF535 processor PCI configuration space, since it is not accessible from the PCI bus side in host mode. The programmer should carefully review the values put into the PCI configuration registers to be sure that all required functionality is enabled correctly.

Outbound Transactions (ADSP-BF535 Processor as PCI Initiator)

Outbound transactions in host mode are accomplished by the same method as in device mode. The only difference is that the ADSP-BF535 processor is responsible for configuring the devices on the PCI bus for certain memory ranges.

Inbound Transactions (ADSP-BF535 Processor as PCI Target)

Inbound transactions in host mode are accomplished by the same method as in device mode. However, only memory accesses are claimed by the PCI interface, and PCI memory space is mapped directly to ADSP-BF535 processor memory space. In other words, the ADSP-BF535 processor memory map becomes the system memory map, and all resources in the ADSP-BF535 processor memory map that are accessible from PCI lie at the same address in PCI memory space. In host mode, a register (`PCI_HMCTL`) is provided at the bottom of the PCI configuration registers to configure which ADSP-BF535 processor resources are accessible from PCI.

This register contains four enable bits, one for each region of ADSP-BF535 processor space that can be used for PCI accesses in host mode. The regions are system MMRs, L2 memory, asynchronous memory, and SDRAM. The register also contains two size fields, one for the asynchronous memory and one for SDRAM memory, that specify how much of each of these spaces is available to PCI. For the asynchronous memory region, the size field is 2 bits and specifies the number of contiguous banks that are accessible. For the SDRAM region, the size field is 5 bits and specifies the number of contiguous 32 MB blocks that are accessible. These size fields are 0 based. This means, for example, enabling asynchronous memory and leaving its size field all 0s results in a single bank enabled. The size field can be thought of as the “size minus one” field. You must write a 0 to the enable bit to completely disable asynchronous memory or SDRAM. This functionality allows the ADSP-BF535 processor to prevent the PCI from accessing space above the valid memory region when SDRAM or asynchronous memory is not fully populated. It also allows the ADSP-BF535 processor to protect upper parts of SDRAM or asynchronous memory from PCI accesses. Any space between these 4 regions is available for assignment to PCI devices during configuration. Any space made available by disabling one of these regions or by sizing down one of the RAM regions is also available for assignment to PCI

devices. Do not assign an address range so that one of the enabled regions in the ADSP-BF535 processor overlaps with any external PCI device. Doing so would cause contention on the PCI bus, since both the ADSP-BF535 processor and the external PCI device would assert DEVSEL# to claim the transaction.

Outbound Configuration Transactions

When in host mode, the ADSP-BF535 processor is responsible for the configuration of the devices on the PCI bus. The ADSP-BF535 processor is responsible for determining the memory or I/O window size for each device, and assigning it a base address where there is sufficient room for the device window. Because the ADSP-BF535 processor only claims transactions in memory space when in host mode, *all* of the I/O space is available for the ADSP-BF535 processor to assign to different devices on the bus. Memory, however, must be allocated with care. The default configuration of the PCI_HMCTL register disables all ADSP-BF535 processor resources: system MMRs, asynchronous memory, and SDRAM. Change this configuration if the PCI system needs these resources. External devices' need for memory space must be balanced against the ADSP-BF535 processor's need for memory space.

To configure devices on the PCI bus, the ADSP-BF535 processor must perform Type 0 and Type 1 transactions, depending on whether the targeted device is on the bus directly connected to the PCI core or is on a bus on the other side of a PCI-to-PCI bridge. See the PCI specification for more information. The PCI core does not behave as a normal bridge—it does not convert a Type 1 transaction into a Type 0 transaction if the bus number connected to the PCI core matches the bus number in the configuration address. The PCI core behavior is consistent with that of the host-to-PCI-bridge behavior.

The PCI core does not generate IDSEL lines for the bus connected to it. For accomplishing configuration accesses, the PCI_CBAP address register and the accompanying Configuration Data port are used. The PCI_CBAP register contents are placed directly on the address bus during the address phase, to allow for maximum flexibility. The system designer must decide how to implement the IDSEL lines. This can be done with the upper address bits, with external logic, or by other means, and the addresses in the PCI_CBAP must be formatted according to the system design requirements.

Reset Behavior and Control

The PCI interface is a bridge between the PCI and the ADSP-BF535 processor systems, so it must gracefully handle a reset from either system in both host and device modes. The PCI reset signal (PCI_RST) comes into the ADSP-BF535 processor and resets some registers in the PCI clock domain. PCI registers that are not affected by a PCI reset are PCI_CFG_DIC, PCI_CFG_VIC, PCI_CFG_STAT, PCI_CFG_CMD, PCI_CFG_MLT, PCI_CFG_CLS, PCI_CFG_MBAR, and PCI_CFG_IBAR. It signals all state machines interfacing the PCI with the internal buses to reset to IDLE. This enables the PCI module to completely recover from a PCI_RST, even if it was asserted during a transaction to or from the ADSP-BF535 processor. The assertion of PCI_RST sets the PCI Reset bit of the PCI_STAT register, and generates an interrupt to the core if masked to do so in the PCI_ICTL register. This enables the core to recognize what has happened, especially in the case of the reset being asserted during a transaction. A system reset running to the PCI module within the ADSP-BF535 processor resets all logic in the PCI module, including the registers and state machines in the PCI core. This method enables PCI core logic to be reset during the reset of either system. It also allows the surrounding logic to return to a known state but register the PCI_RST and only completely reset during an ADSP-BF535 processor reset.

The ADSP-BF535 processor can also cause a reset to the PCI using the two bits provided in the PCI_CTL register. The first bit is the RST to PCI Enable bit, which is the output enable on the PCI_RST pad. The second bit is the RST to PCI bit, which when set, causes the PCI_RST signal to be asserted. Resetting the PCI system in this manner has the same effect on the PCI core and surrounding logic as the PCI_RST input signal. The logic of the PCI module does not restrict the ADSP-BF535 processor from causing a PCI_RST in device mode, but doing this is recommended only when the ADSP-BF535 processor is in host mode.

Note all PCI signals are active low, but the bits in the PCI_CTL and the PCI_STAT registers are active high. This means that setting the RST to PCI bit in the PCI_CTL register asserts PCI_RST, and the PCI Reset bit in the PCI_STAT register is set when the PCI_RST line is asserted.

Interrupt Behavior and Control

All four interrupt lines from PCI (INTA-INTD) have associated sticky status bits in the PCI_STAT register. Whenever one of these interrupt lines is asserted, the appropriate bit is set and an interrupt is generated if masked to do so in the PCI_ICTL register. These bits are valid in host or device mode, but are most useful when the ADSP-BF535 processor is acting as the system host.

The INTA interrupt line is defined as an input/output because the ADSP-BF535 processor is capable of generating an interrupt to the system processor by setting the INTA to PCI bit in the PCI Bridge Control register. This bit asserts INTA in device mode or host mode, but it is most useful in device mode. In host mode it makes more sense to use a core interrupt, if necessary.

Note all PCI signals are active low, but the bits in the PCI_CTL and the PCI_STAT registers are active high. So setting the INTA to PCI bit asserts the INTA signal and the INTA-INTD bits are set when the corresponding line is asserted.

PCI Programming Model

The PCI interface programming model includes memory-mapped control and status registers, as well as memory-mapped PCI spaces (configuration, I/O, and memory). The memory-mapped PCI spaces are described in "Processor Core Access to PCI Space" on page 13-3. This section describes the memory-mapped control and status registers.

There are two sets of memory-mapped registers (MMRs) on the ADSP-BF535 processor. The control and status registers clocked in the internal ADSP-BF535 processor system clock domain are mapped into the system MMR space. The control and status registers clocked in the PCI clock domain are mapped into the PCI Configuration Registers memory range (see the PCI memory map in Figure 13-2 on page 13-5). Each group of registers is described in the sections that follow.

Bus Operation Ordering

Processor core accesses to the PCI interface always occur in the order that the processor core initiated the system access. This implies:

- Since the processor core allows reads to occur before writes, with respect to instruction order, the programmer must explicitly order the load and store instruction by inserting an `SSYNC` instruction between them in the cases where a PCI read must see the effect of a previous PCI write. This is not required if the load and store address and size are the same, for example, a write to a PCI Status register followed by a read from the same register.
- ADSP-BF535 processor hardware does not enforce ordering among the EAB, PAB, and EMB buses. Software must explicitly manage any coherency issues among these buses. Shared resources should be protected by software locks. With respect to program order, EAB reads and PAB accesses are always completed in order, since the core is stalled until they complete.

System MMR Control and Status Registers

This section describes each of the memory-mapped registers (MMRs) for PCI. All defined registers are writable and readable. Reserved bits within a register always read 0. Do not access reserved register space. The address range for these registers is 0xFFC0 4000 - 0xFFC0 43FF. Refer to "System MMR Assignments" on page B-1 for the address of a particular register.

The PCI peripheral also has several registers mapped into the memory space. This is because these registers reside in the PCI clock domain, therefore they have high access latency from the SCLK domain.

See "Configuration Space Control and Status Registers" on page 13-26 for a description of the configuration space MMRs. Also see "System MMR Assignments" on page B-1 for the PCI MMRs mapped into the PAB system MMR space.

PCI Bridge Control Register (PCI_CTL)

This register, shown in Figure 13-3, controls the operation of the application interface logic to the PCI core.

PCI Bridge Control Register (PCI_CTL)

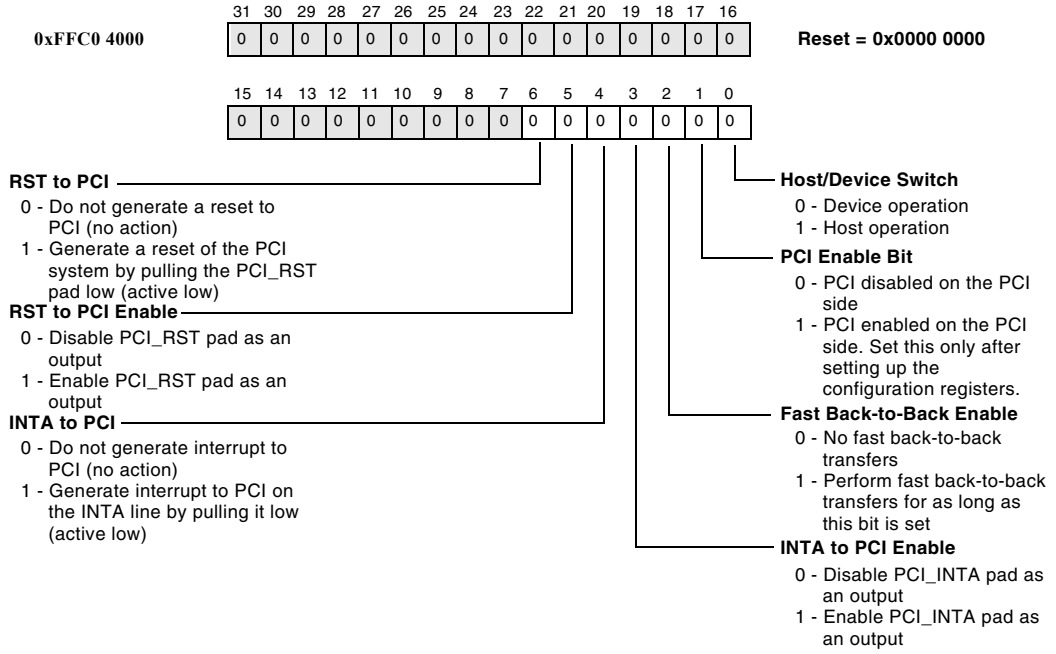


Figure 13-3. PCI Bridge Control Register

It is used to put the interface in host or device mode. This register includes the PCI Enable bit used to enable PCI access after the configuration registers have been set up, and it includes the enable bit for outbound fast back-to-back transactions.

PCI Status Register (PCI_STAT)

The PCI_STAT register includes all PCI status flags visible to the core (see Figure 13-4).

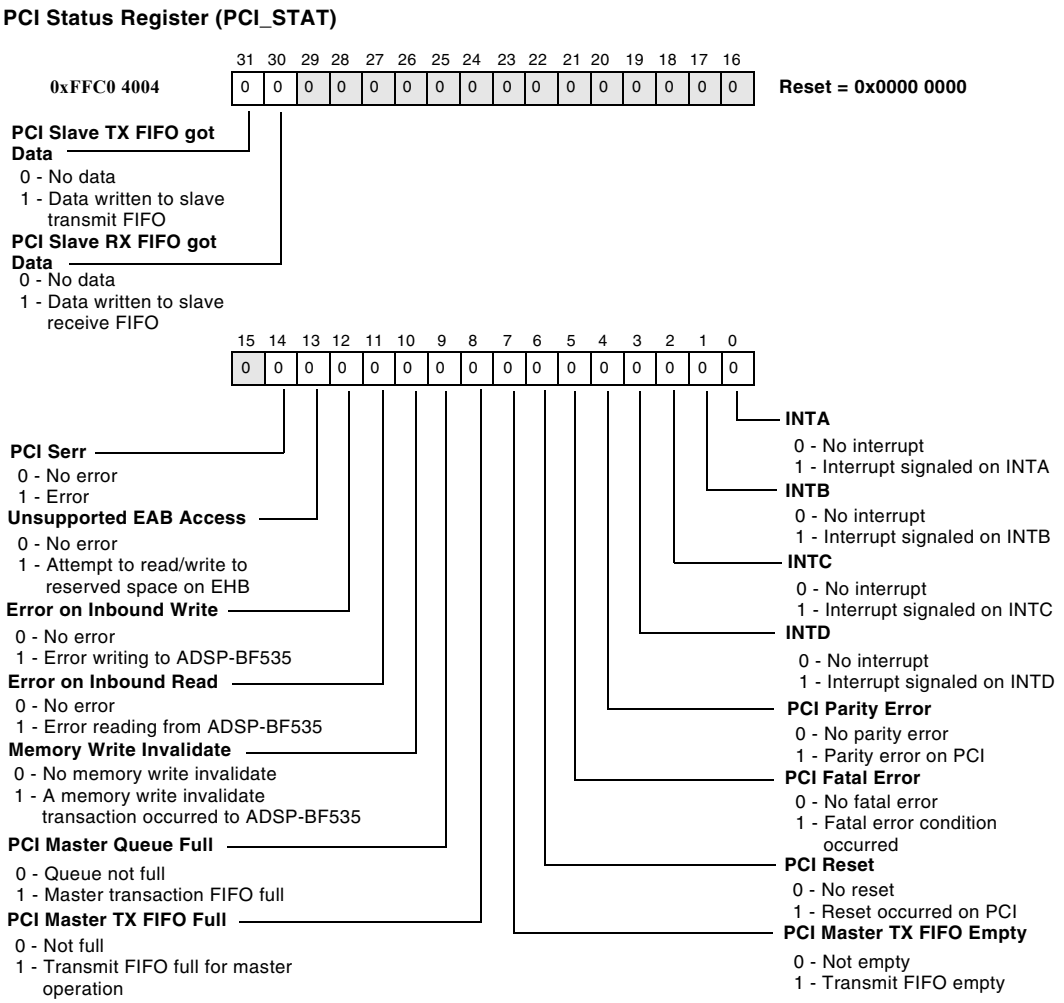


Figure 13-4. PCI Status Register

Each of these flags can also generate a PCI interrupt to the core by setting the corresponding bit in the PCI_ICTL register. The PCI Master Queue Full bit and the PCI Master TX FIFO Full bit are status bits only. All the other bits in this register can be configured to cause interrupts via the PCI_ICTL register. These bits are sticky and all are write-1-to-clear. If these interrupts are enabled in the PCI_ICTL register, they should be cleared in the interrupt service routine to prevent them from continuously generating interrupts, by writing a 1 to the appropriate bits.

PCI Interrupt Controller Register (PCI_ICTL)

The bits in this register, shown in Figure 13-5, enable the flags in PCI Status register, except the PCI Master Queue Full bit and the PCI Master TX FIFO Empty bit. Those two bits are status bits only.

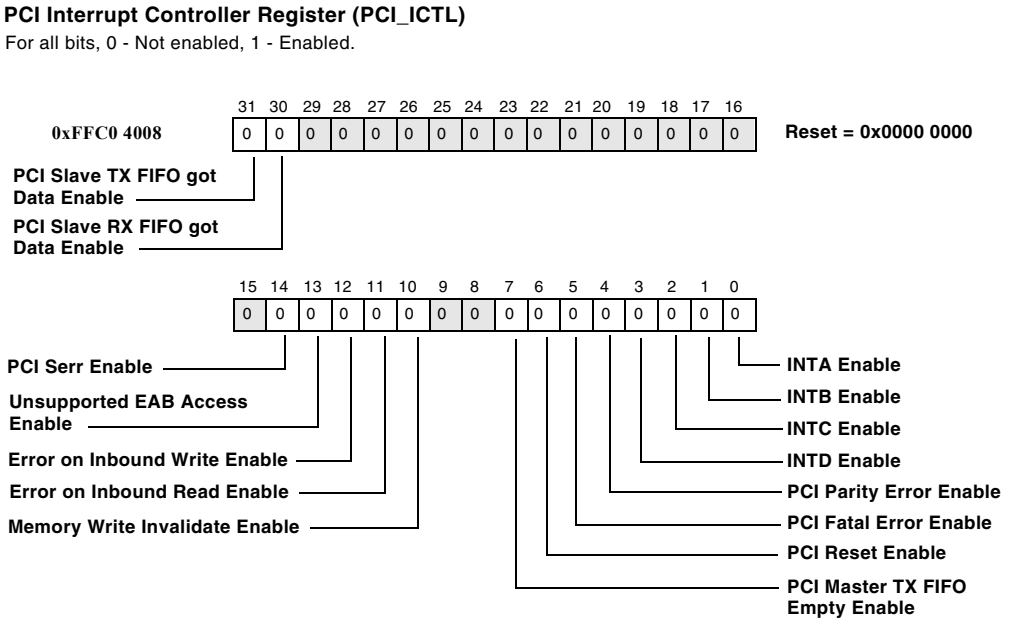


Figure 13-5. PCI Interrupt Controller Register

PCI Outbound Memory Base Address Register (PCI_MBAP)

This register, shown in Figure 13-6, is the PCI memory base address pointer that is prepended to the EAB address for outgoing PCI memory transfers. The 5-bit PCI Space Address is prepended to the 27-bit offset into the PCI memory space window in the ADSP-BF535 processor EAB space. Write to this register before attempting a read or write to or from PCI memory space.

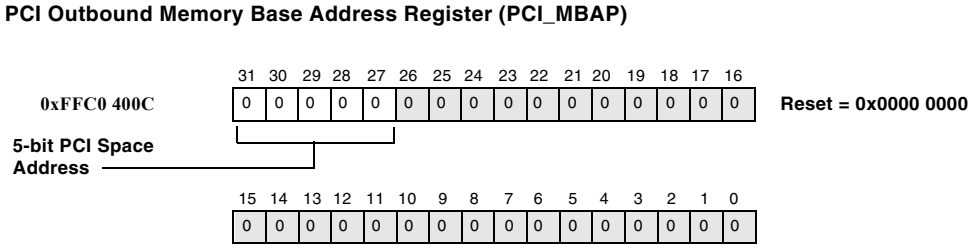


Figure 13-6. PCI Outbound Memory Base Address Register

PCI Outbound I/O Base Address Register (PCI_IBAP)

This register, shown in Figure 13-7, is the I/O base address pointer.

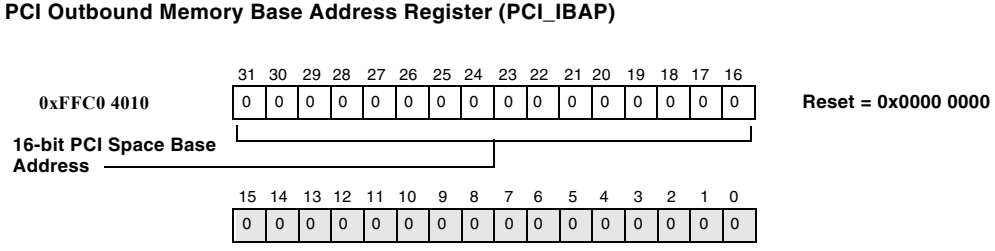


Figure 13-7. PCI Outbound I/O Base Address Register

This address is prepended to the EAB address for outgoing PCI I/O transfers. The 16-bit PCI Space Base Address is prepended to the 16-bit offset into the PCI I/O space window in ADSP-BF535 processor EAB space. Write to this register before attempting a read or write to or from PCI I/O space.

PCI Outbound I/O Configuration Address Register (PCI_CBAP)

This register, shown in Figure 13-8, is the PCI configuration space address port. The contents of this register are put directly on the PCI AD bus during the address phase for outgoing PCI configuration space transfers. The register can then be used for Type 1 or Type 0 addresses (with IDSELs on the upper bits) or a combination of the two address types if IDSEL generation is moved off chip. The software must generate Type 1 and Type 0 accesses. The core has no logic for conversion or IDSEL generation. External hardware is needed to generate IDSELs.

There is no direct method for selecting devices for doing configuration in host mode.

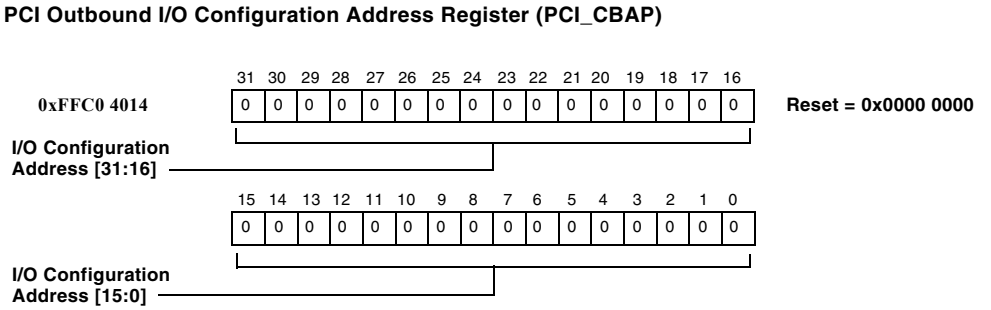


Figure 13-8. PCI Outbound I/O Configuration Address Register

PCI Inbound Memory Base Address Register (PCI_TMBAP)

This register specifies a base address in the Blackfin ADSP-BF535 processor's memory space. This is the base address of the window that a host processor may use to set for access to a buffer in PCI memory space. This register has meaning only when the PCI interface is configured as a device. The address written to this register by the Blackfin processor core is the start address of the section of memory that will be used by the host processor on the PCI bus for this purpose. The size of the window is specified by the mask loaded into the `PCI_DMBARM` register. The base address loaded into the `PCI_TMBAP` register must be aligned on a boundary that is an integer multiple of this size.

The MMR address for this register is 0xFFC0 4018. Its reset value is 0x0000 0000.

PCI Inbound I/O Base Address Register (PCI_TIBAP)

This register specifies a base address in the Blackfin ADSP-BF535 processor's memory space. This is the base address of the window that a host processor may use to set for access to a buffer in PCI I/O space. This register has meaning only when the PCI interface is configured as a device. The address written to this register by the Blackfin processor core is the start address of the section of memory that will be used by the host processor on the PCI bus for this purpose. The size of the window is specified by the mask loaded into the `PCI_DIBARM` register. The base address loaded into the `PCI_TIBAP` register must be aligned on a boundary that is an integer multiple of this size.

The MMR address for this register is 0xFFC0 401C. Its reset value is 0x0000 0000.

Configuration Space Control and Status Registers

This section describes each of the MMRs for PCI that are accessible in the PCI Configuration Registers address space. All defined registers are writable and readable. Reserved bits within a register read 0. Do not access reserved register name space. The address range for these registers is 0xEEFF FF00 - 0xEEFF FFFF. Refer to "System MMR Assignments" on page B-1 for the address of a particular register.

The configuration address space-based register space includes all PCI clock domain MMRs for the PCI peripheral. This includes the PCI device basic configuration registers consisting of sixty-four 32-bit words.

This register space also includes the memory space and I/O space BAR masks. These registers define the size of the respective memory areas made available to the PCI bus when the PCI peripheral functions as a PCI device. The corresponding BARs are part of the PCI configuration space and are programmed by the PCI bus host/bridge.

PCI Device Memory BAR Mask Register (PCI_DMBARM)

This register is a bit mask that specifies the size of the window available to a host processor on the PCI bus for access into the Blackfin processor's memory space using memory space PCI transactions. The `PCI_DMBARM` register is shown in Figure 13-9. It is filled by the Blackfin processor core with 1s starting from the most significant bit (bit 31) down to the bit position corresponding to the window size available to the host processor. For example, if a 16 MB window is available for the host to use as a frame buffer in the Blackfin processor's memory space, bits 31 through 24 would contain 1s and bits 0 through 23 would contain 0s. Bits 0 through 3 of this register are not writable and always contain 0s. As a result, the smallest non zero window that can be specified is 16 bytes long. Similarly, bit

31 must always be used, limiting the largest window to 2 GB. Writing all 0s to this register disables PCI memory space access by external bus masters.

During configuration, the PCI host processor loads the PCI_CFG_MBAR register using this mask, setting the base address in PCI memory space of the window into the Blackfin processor's memory system.

This register has meaning only when the PCI interface is configured as a device.

PCI Device Memory BAR Mask Register (PCI_DMBARM)

Bits [3:0] are reserved and always 0.

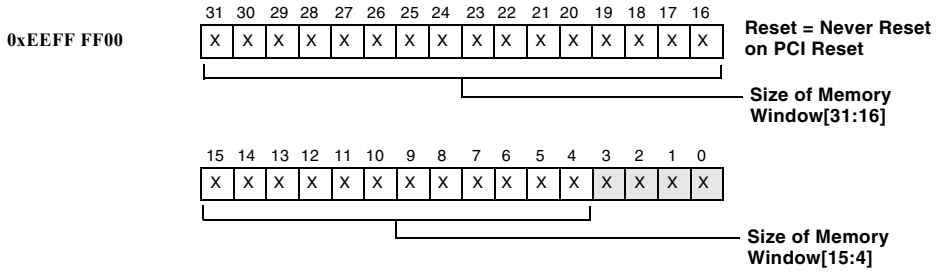


Figure 13-9. PCI Device Memory BAR Mask Register

PCI Device I/O BAR Mask Register (PCI_DIBARM)

This register is a bit mask that specifies the size of the window available to a host processor on the PCI bus for access into the Blackfin processor's memory space using I/O space PCI transactions. The PCI_DIBARM register is shown in Figure 13-10. It is filled by the Blackfin processor core with 1s starting from the most significant bit (bit 31) down to the bit position corresponding to the window size available to the host processor. For example, if a 64 byte window is available for the host to use as an I/O window in the Blackfin processor's memory space, bits 31 through 6 would

contain 1s and bits 0 through 5 would contain 0s. Bits 0 and 1 of this register are not writable and always contain 0s. As a result, the smallest non-zero window that can be specified is 4 bytes long. Similarly, bit 31 through 8 must always be used, limiting the largest window to 256 bytes. Writing all 0s to this register disables PCI I/O space access by external bus masters.

During configuration, the PCI host processor loads the `PCI_CFG_IBAR` register using this mask, setting the base address in PCI I/O space of the window into the Blackfin processor's memory system.

This register has meaning only when the PCI interface is configured as a device.

PCI Device I/O BAR Mask Register (PCI_DIBARM)

Bits [1:0] are reserved and always 0.

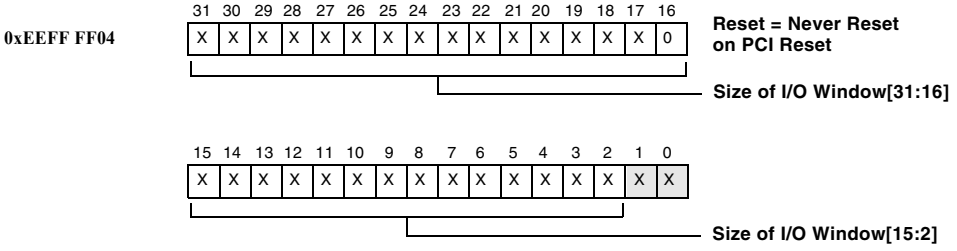


Figure 13-10. PCI Device I/O BAR Mask Register

PCI Configuration Device ID Register (PCI_CFG_DIC)

This register, shown in Figure 13-11, is the Device ID from the PCI configuration registers.

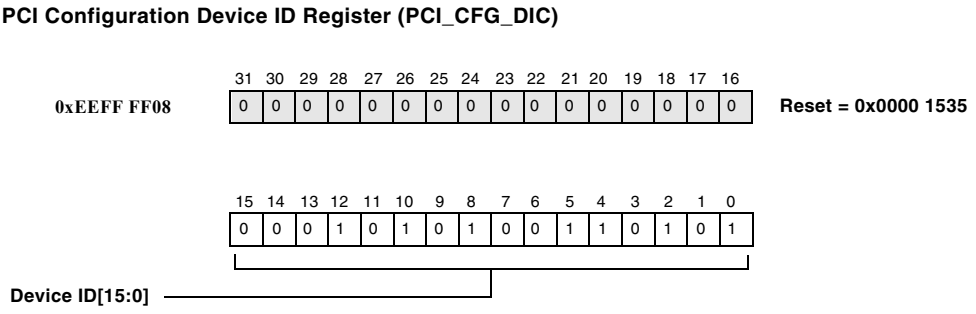


Figure 13-11. PCI Configuration Device ID Register

It is not used in host mode. Program this register before enabling the PCI in device mode. For more information, see the PCI Local Bus Specification Rev. 2.2.

PCI Configuration Vendor ID Register (PCI_CFG_VIC)

The PCI_CFG_VIC register, shown in Figure 13-12, contains the Vendor ID from the PCI configuration registers.

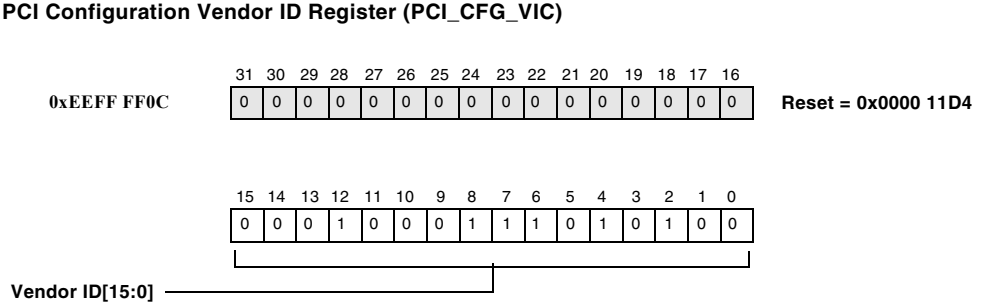


Figure 13-12. PCI Configuration Vendor ID Register

It is not used in host mode. Program this register before enabling the PCI in device mode. For more information, see the PCI Local Bus Specification Rev. 2.2.

PCI Configuration Status Register (PCI_CFG_STAT)

This register, shown in Figure 13-13, is the status register for PCI configuration registers.

PCI Configuration Status Register (PCI_CFG_STAT)

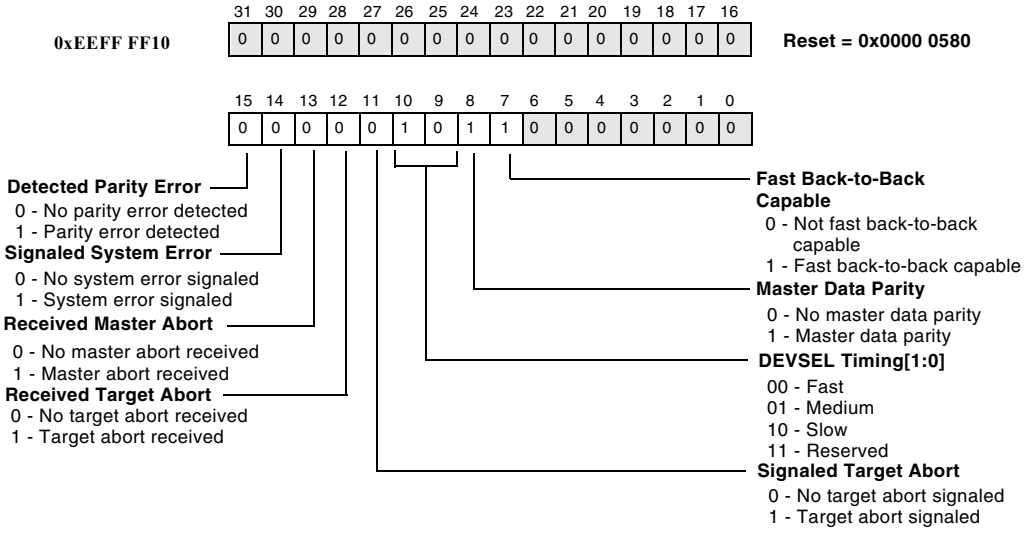


Figure 13-13. PCI Configuration Status Register

The information in this register applies to both host mode and device mode. For more information, see the PCI Local Bus Specification Rev. 2.2.

PCI Configuration Command Register (PCI_CFG_CMD)

This register, shown in Figure 13-14, is the command register from the PCI configuration registers.

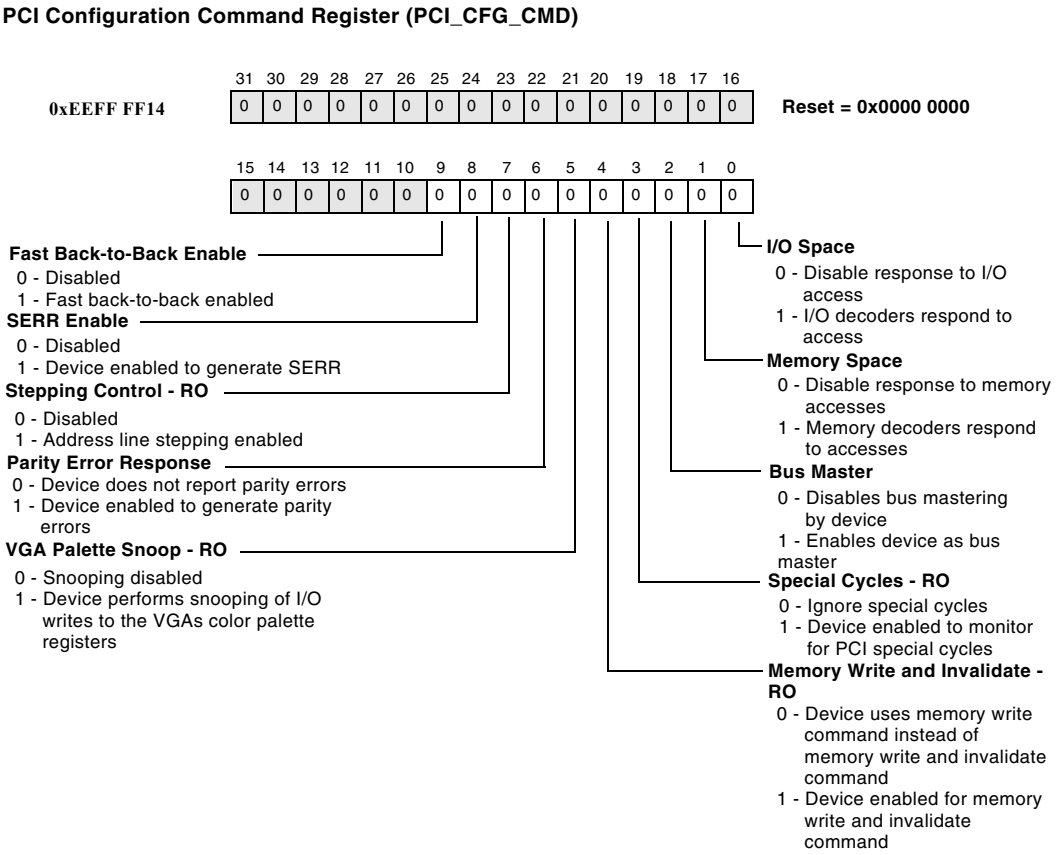


Figure 13-14. PCI Configuration Command Register

This register is always writable, except for the reserved bits. Write in host mode only during configuration of the PCI system. In device mode, the system processor writes to this register. For more information, see the PCI Local Bus Specification Rev. 2.2.

PCI Configuration Class Code Register (PCI_CFG_CC)

This register, shown in Figure 13-15, holds the class code from the PCI configuration registers. It is not used in host mode. Program this register before enabling the PCI in device mode. For more information, see the PCI Local Bus Specification Rev. 2.2.

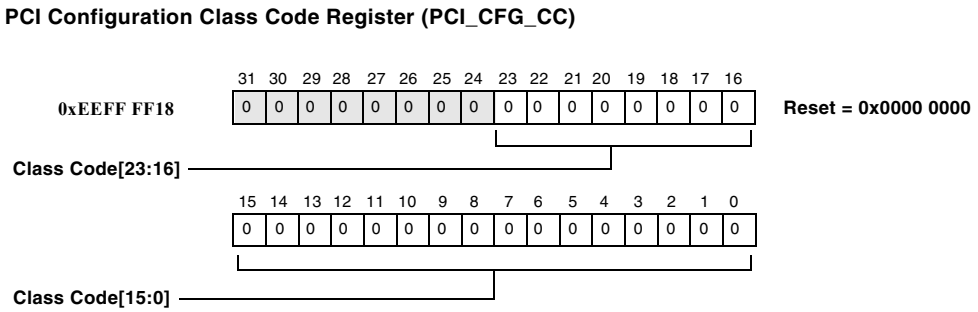


Figure 13-15. PCI Configuration Class Code Register

PCI Configuration Revision ID Register (PCI_CFG_RID)

This register, shown in Figure 13-16, holds the revision ID from the PCI configuration registers.

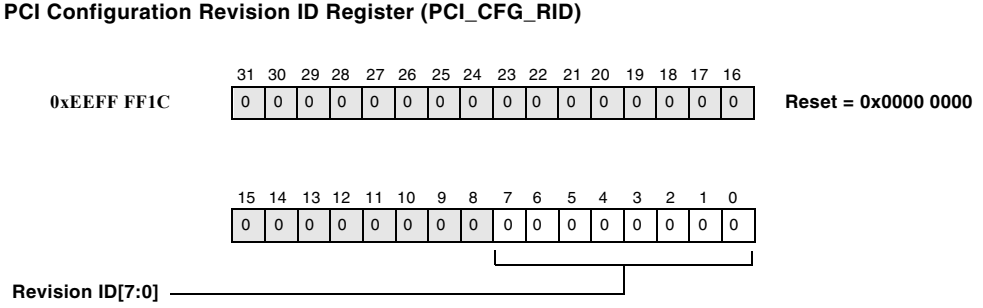


Figure 13-16. PCI Configuration Revision ID Register

It is not used in host mode. Program this register before enabling the PCI in device mode. For more information, see the PCI Local Bus Specification Rev. 2.2.

PCI Configuration BIST Register (PCI_CFG_BIST)

This register, shown in Figure 13-17, holds the value for the built-in self test (BIST) from the PCI configuration registers.

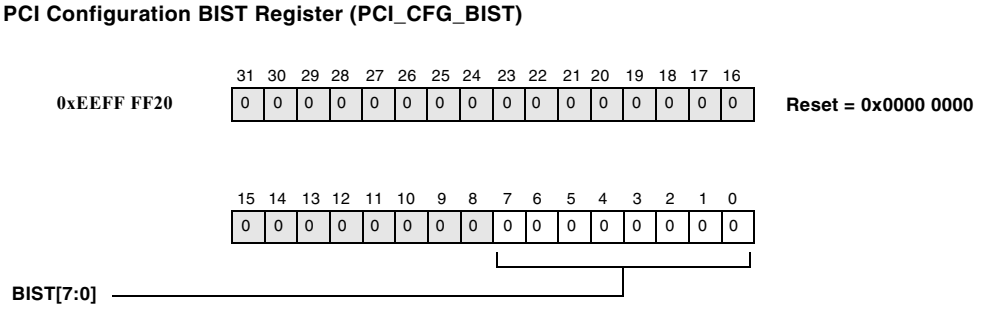


Figure 13-17. PCI Configuration BIST Register

The PCI core does not support BIST. For more information, see the PCI Local Bus Specification Rev. 2.2.

PCI Configuration Header Type Register (PCI_CFG_HT)

This register, shown in Figure 13-18, holds the header type from the PCI configuration registers.

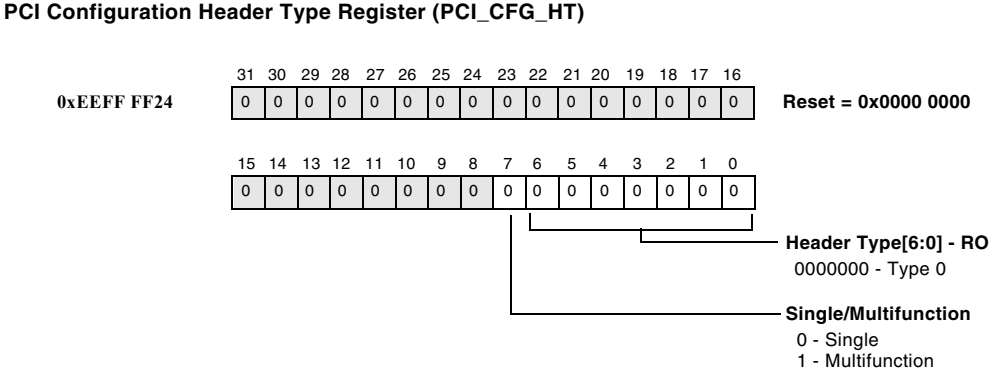


Figure 13-18. PCI Configuration Header Type Register

The PCI core can only be single function and the configuration registers conform to Type 0. For more information, see the PCI Local Bus Specification Rev. 2.2.

PCI Configuration Memory Latency Timer Register (PCI_CFG_MLT)

The `PCI_CFG_MLT` register, shown in Figure 13-19, holds the memory latency timer from the PCI configuration registers. It is always writable, except where reserved, and for two hardwired bits. Write to this register only during PCI system configuration in host mode. In device mode, the system processor writes to this register. The lowest two bits are read only and hardwired to 0 so that latency values are always in increments of four PCI clock cycles.

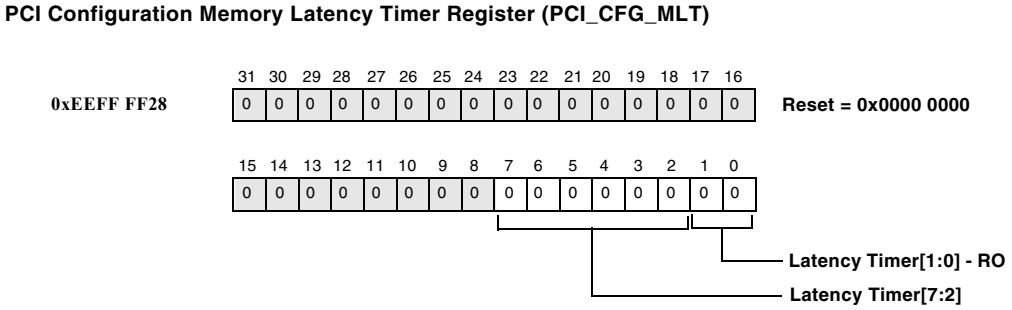


Figure 13-19. PCI Configuration Memory Latency Timer Register

PCI Configuration Cache Line Size Register (PCI_CFG_CLS)

The PCI_CFG_CLS register, shown in Figure 13-20, holds the cache line size from the PCI configuration registers. It is writable from the PCI and from the EAB side. It is unused because the memory write Invalidate (MWINV) commands are disabled in the control register and the Inbound Cacheline Wrap is not supported by the ADSP-BF535 processor. For more information, see the PCI Local Bus Specification Rev. 2.2.

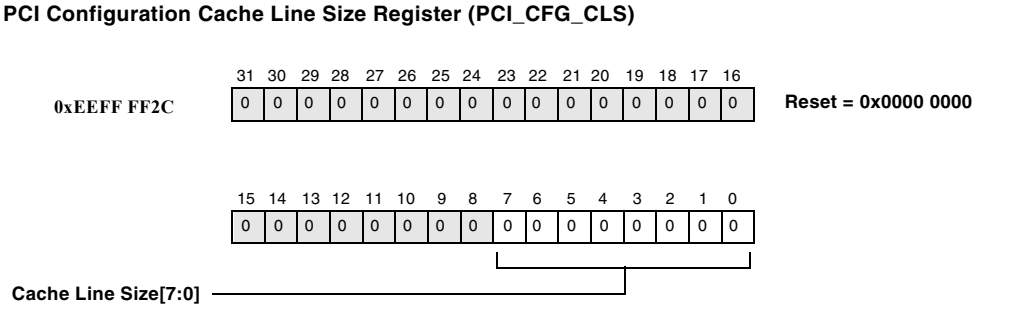


Figure 13-20. PCI Configuration Cache Line Size Register

PCI Configuration Memory Base Address Register (PCI_CFG_MBAR)

The PCI_CFG_MBAR register, shown in Figure 13-21, holds the memory base address from the PCI configuration registers. The lower 4 bits are hardcoded, and tell the configuration software to use this register for a prefetchable memory base address and to place it in the lower 4 Gbytes (32 bits) of address space. The PCI_DMBARM register configures the upper bits to be writable from the PCI side. Configuration software detects the size requested by looking at the number of writable upper bits and places a base address in this space. This register is unused in Host mode. For more information, see the PCI Local Bus Specification Rev. 2.2.

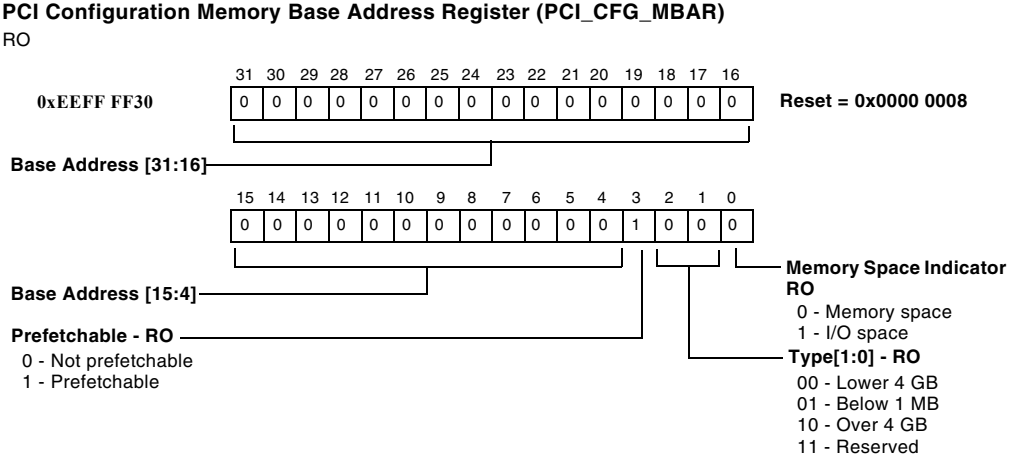


Figure 13-21. PCI Configuration Memory Base Address Register

PCI Configuration I/O Base Address Register (PCI_CFG_IBAR)

This register, shown in Figure 13-22, holds the I/O base address from the PCI configuration registers.

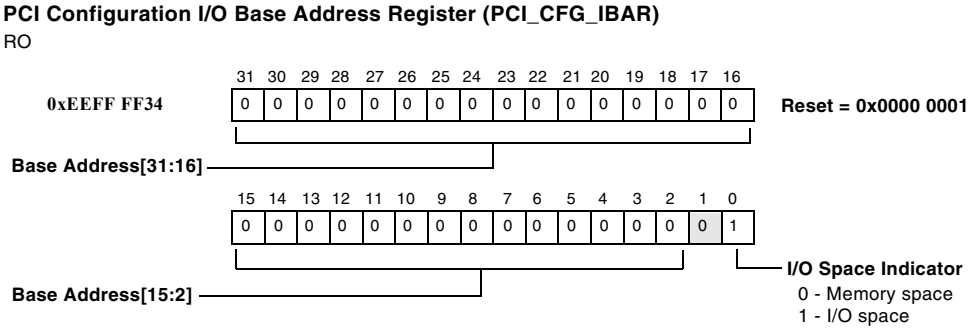


Figure 13-22. PCI Configuration I/O Base Address Register

The hardcoded lower bits tell the configuration software to use this register for an I/O base address. The PCI_DIBARM register configures the upper bits to be writable from the PCI side. The configuration software detects the size by looking at the number of writable upper bits and places a base address in this space. This register is unused in Host mode. For more information, see the PCI Local Bus Specification Rev. 2.2.

PCI Configuration Subsystem ID Register (PCI_CFG_SID)

The PCI_CFG_SID register, shown in Figure 13-23, holds the Subsystem ID from the PCI configuration registers.

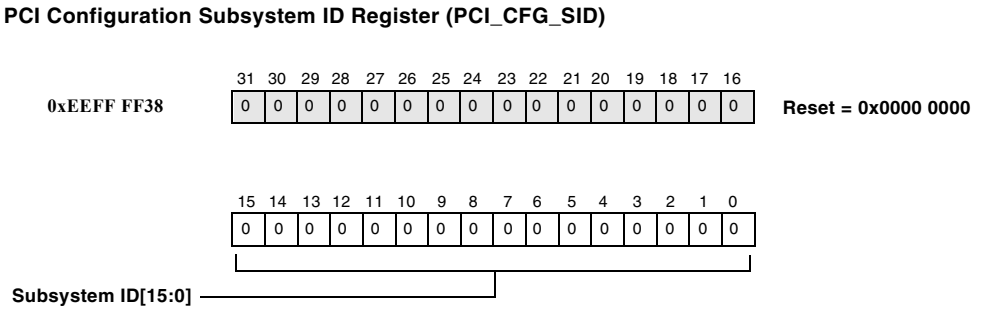


Figure 13-23. PCI Configuration Subsystem ID Register

Program this register before enabling the PCI in device mode. This register is not used in host mode. For more information, see the PCI Local Bus Specification Rev. 2.2.

PCI Configuration Subsystem Vendor ID Register (PCI_CFG_SVID)

The PCI_CFG_SVID register, shown in Figure 13-24, holds the subsystem vendor ID from the PCI configuration registers. Program this register before enabling the PCI in device mode. This register is not used in host mode. For more information, see the PCI Local Bus Specification Rev. 2.2.

PCI Configuration Subsystem Vendor ID Register (PCI_CFG_SVID)

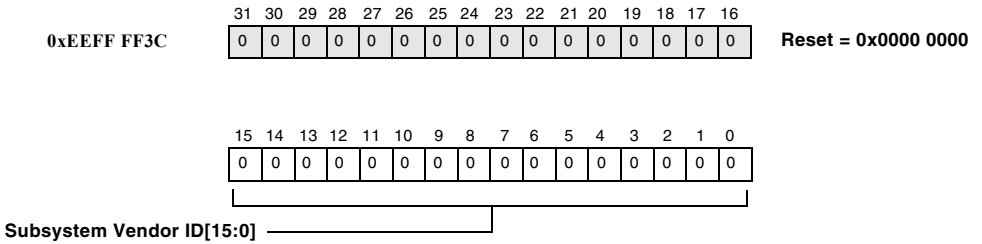


Figure 13-24. PCI Configuration Subsystem Vendor ID Register

PCI Configuration Maximum Latency Register (PCI_CFG_MAXL)

This register, shown in Figure 13-25, holds `Max_Lat` from the PCI configuration registers. Program this register before enabling the PCI in device mode. This register is optional for master operation (host or device). For more information, see the PCI Local Bus Specification Rev. 2.2.

PCI Configuration Maximum Latency Register (PCI_CFG_MAXL)

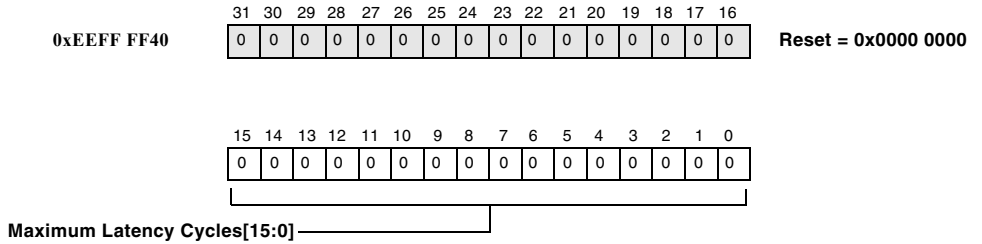


Figure 13-25. PCI Configuration Maximum Latency Register

PCI Configuration Minimum Grant Register (PCI_CFG_MING)

This register, shown in Figure 13-26, holds `Min_Gnt` from the PCI configuration registers. Program this register before enabling the PCI in device mode. This register is optional for master operation (host or device). For more information, see the PCI Local Bus Specification Rev. 2.2.

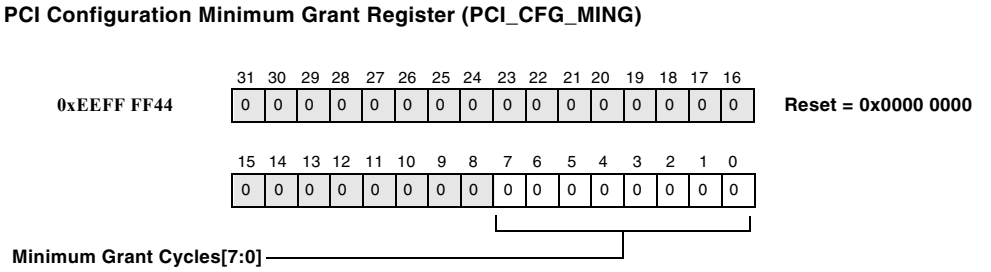


Figure 13-26. PCI Configuration Minimum Grant Register

PCI Configuration Interrupt Pin Register (PCI_CFG_IP)

This register, shown in Figure 13-27, holds the interrupt pin value from the PCI configuration registers. For more information, see the PCI Local Bus Specification Rev. 2.2.

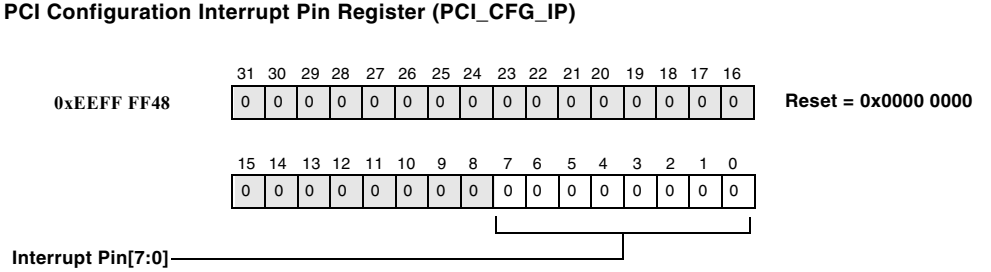


Figure 13-27. PCI Configuration Interrupt Pin Register

PCI Configuration Interrupt Line Register (PCI_CFG_IL)

This register, shown in Figure 13-28, holds the interrupt line from the PCI configuration registers. For more information, see the PCI Local Bus Specification Rev. 2.2.

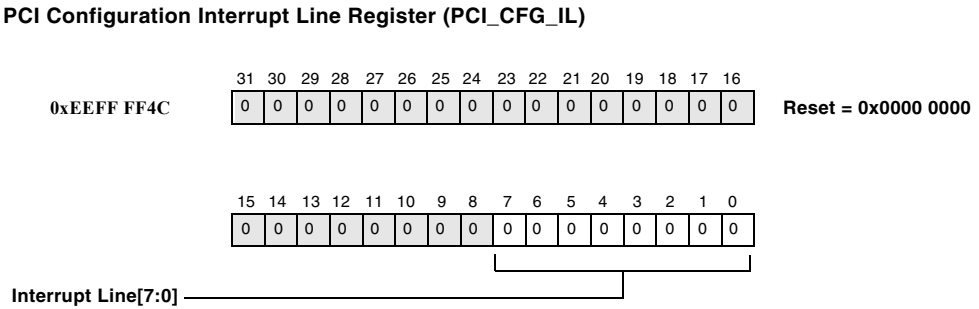


Figure 13-28. PCI Configuration Interrupt Line Register

PCI Host Memory Control Register (PCI_HMCTL)

This register, shown in Figure 13-29, configures which portions of ADSP-BF535 processor memory space are accessible from any initiators on the PCI bus. There is one-to-one mapping of addresses between the PCI and the ADSP-BF535 processor in host mode. The PCI core asserts DEVSEL and claims the transaction for any region indicated as enabled in this register. The asynchronous memory access is valid only if the Async Mem Access Enable bit is set and specifies the size of the accessible Async Mem window in 64-MB blocks. The SDRAM access size is valid only if the SDRAM Access Enable bit is set. It specifies the size of the accessible SDRAM window in 32-MB blocks. This register is valid only in host mode.

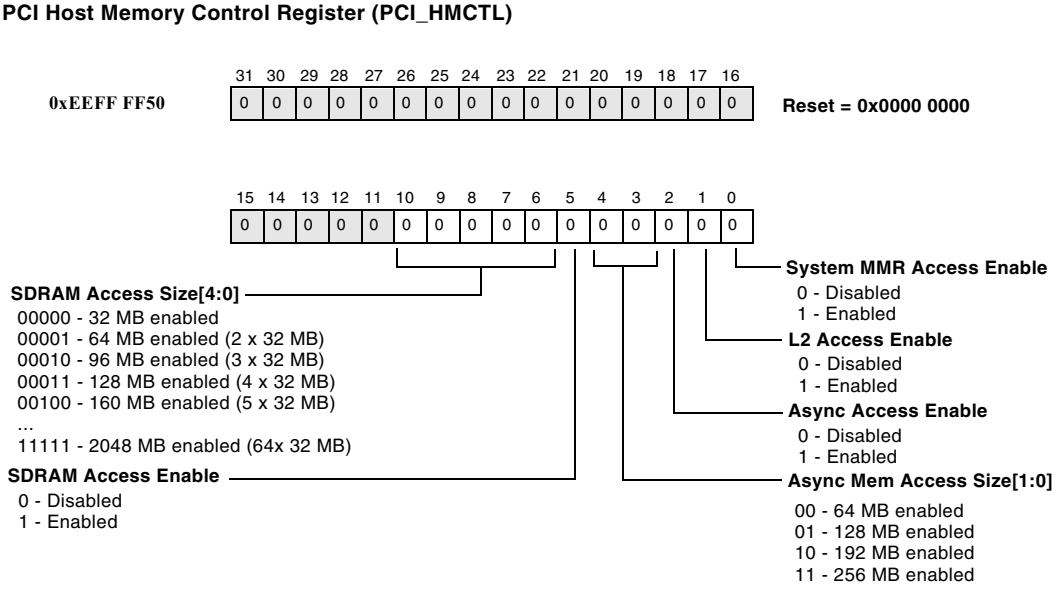


Figure 13-29. PCI Host Memory Control Register

PCI I/O Issues

To meet the requirements of the PCI 2.2 specification, the PCI interface employs special I/O receivers and drivers.

Reflected Wave Switching

PCI specifies reflected wave switching bus topology, such that the output driver only charges/discharges the signal line part way to the desired logic level. The reflected wave from the end of the unterminated signal trace then fully drives the voltage to the desired level, before the next rising edge of the PCI clock. This approach minimizes noise and termination issues at the board level, and lowers overall system power consumption.

Power Sequencing

The PCI interface operates on two power domains. The internal logic is on the same power domain as the rest of the internal logic of the ADSP-BF535 processor. The PCI I/O is powered through a dedicated set of power pins. These two domains can operate at different voltage levels. In general, it is recommended that power be applied to both domains at the same time, or to the I/O power domain first then to the internal logic power domain.

PCI Clock Requirements

The 33 MHz PCI clock requirements are >= 30 ns cycle time, with a maximum skew of <= 2 ns across all PCI component clock inputs. The clock may be stopped, but only in the low state. This clock can be driven completely asynchronously with respect to the internal ADSP-BF535 processor's clocks and `CLKIN`.

14 USB DEVICE

The Universal Serial Bus Device (USBD) module in the ADSP-BF535 processor contains a USB Device Controller core (UDC) and a front-end interface. The UDC handles all USB protocol requirements and provides a simple read/write interface to the front end. The front end adds hardware to the UDC to connect it to the ADSP-BF535 processor's peripheral buses and to support an operating system.

The USBD module also includes memory-mapped registers, an interrupt subsystem, a configuration and clock control for the UDC, and a DMA controller for the transfer of USB endpoint data between system memory and the USB.

On the system side, the USBD module interfaces with the Data Access Bus (DAB) and the Peripheral Access Bus (PAB). On the USB side, it interfaces with an off-chip USB transceiver.

This chapter includes a USBD block diagram and functional description. A list of references is provided at the end of the chapter.

Convention

All transfer directions use the USB host as the point of reference. An IN transfer indicates the host is receiving data, and an OUT transfer indicates the host is transmitting data.

Requirements

The ADSP-BF535 processor supports a wide range of applications, including, but not limited to, multifunction peripherals such as laser printer/scanner/copier systems, and personal digital assistants. The USB device can use up to eight endpoints.

The USB protocol imposes a number of requirements on the design of a USB device. This is to ensure that USB devices comply with USB specifications.

USB Functionality

The ADSP-BF535 processor's USB peripheral includes:

- Eight physical USB endpoints
- The ADSP-BF535 processor can support one configuration, with the configuration having two interfaces and each interface having two alternates. There are a total of 64 logical endpoints. *All the logical endpoints must be programmed for correct functioning, even if an application does not require all logical endpoints.*
- Support for all USB transfer types
- Low power capabilities, and clocks that can be stopped
- Connection to the ADSP-BF535 interrupt system with a single interrupt line
- Connection to the PAB as a slave only for system access to registers
- Connection to the DAB as a master only for access to L2 memory, through a DMA Master module

- Arbitration granted to the USB DMA channel within 3 to 5 USB cycles (with SCLK=100 MHz) after the assertion of its request from a host

If the arbitration requirement is not met, then USB Bulk, Interrupt, and Control In packet transfers may be terminated prematurely, resulting in degraded USB performance. Potentially, this can add complexity to the USB driver code. To minimize degraded USB performance, a system clock (SCLK) minimum of 100 MHz is recommended, but the *ADSP-BF535 Blackfin Embedded Processor Data Sheet* should be consulted.

Because SPORTs have a higher arbitration priority than the USB on the DAB bus, USB data transfers may be affected when there is excessive SPORT activity coupled with USB transactions on the DAB bus. This condition could also occur on an external memory access using MemDMA to a high latency external memory.

USB Requirements

The references listed at the end of this chapter describe the USB protocol in detail. This section gives a brief description of some features of the protocol used for this particular USB device implementation.

Master and Slave Buses

The USB is a master-slave bus, in which a single master generates data transfer requests to the attached slaves, and allocates bandwidth on the serial cable according to a specific algorithm.

The bus master is referred to as the USB host, and the bus slaves are referred to as USB devices. Each USB device implements one or more USB endpoints, which are similar to virtual data channels. Each endpoint on a USB device operates independently of all the others.

Data Flow and Traffic Scheduling

Data flows between the USB host and the attached devices in units of packets, which are normally 8, 16, 32 or 64 bytes. An exception is the isochronous data type, where packets can be as large as 1023 bytes. The packets are grouped into larger units called transfers.

To allocate the serial bus bandwidth fairly across all of the attached USB devices and endpoints, the USB host implements a traffic scheduling algorithm. From the point of view of a USB device, this algorithm is not deterministic. Although the specific scheduling algorithm is standardized, the bus dynamically reallocates bandwidth based on factors such as packet error conditions and flow control. So far as the device is concerned, it can receive transfer requests for any endpoint at any time.

Depending on the bus loading, the USB host may request packets back-to-back, or it may request packet transfers to each endpoint in a round-robin fashion. The USB protocol also allows for detection and retransmission of packets in cases of bit errors and flow control problems.

Each USB device implementation must maintain state information for each endpoint that allows large data transfers to occur a packet at a time. Each device must also maintain state information for each packet to be retransmitted.

USB Implementation

The ADSP-BF535 processor's USBD implementation includes:

- The peripheral has a single interrupt output (`USBD_INTR`).
- System software is responsible for managing interrupt priority and requeueing low priority events as software interrupts.
- Memory access for the USB endpoints is by means of the DAB as a bus master.

- The DMA Master Channel module defines a 2 KB memory region in the system's memory space; the region is partitioned into 16-byte blocks. The peripheral requests transfers to individual blocks within the 2 KB memory space. Allocation of the 16-byte blocks within the 2 KB memory space is at the discretion of the programmer.
- Registers within the USB device module maintain the address offsets and byte counts for each endpoint.
- The registers are structured to allow multiple buffering of each endpoint's data, resulting in streaming operation on the USB. The registers support the ability of the USB to perform packet retries, and the ability to break large data buffers into individual USB packets.
- The suspend capability of the USB device can be exploited by a clock and power management scheme implemented in software.
- The endpoints are configured as unidirectional. For example, once an endpoint is configured as a type IN endpoint, it should not be used as type OUT on any configuration.

The USBD has multiple clock domains. Although the `PLL_IOCK` register allows shutting down the system clock for the front-end interface in USBD, the suspend capabilities allow for a software management scheme of the UDC's clocks.

Block Diagram

A block diagram of the USBD module is shown in Figure 14-1.

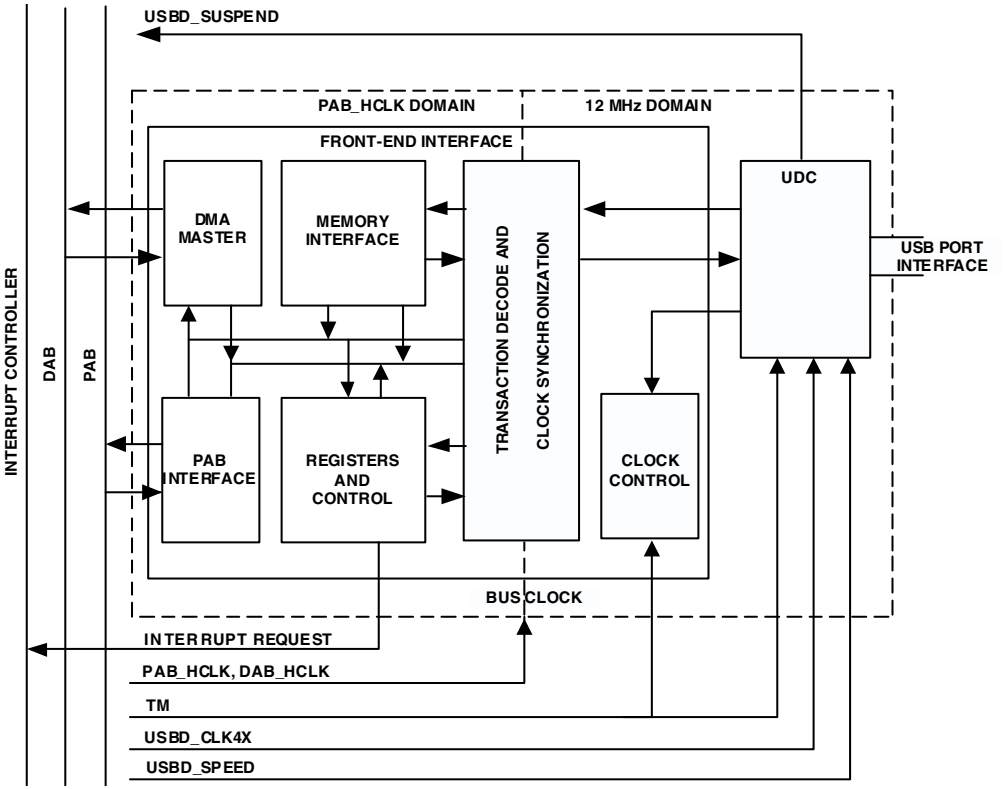


Figure 14-1. USBD Module Block Diagram

UDC Block

The UDC module implements the low-level USB protocol. It manages interaction with the USB host using the USB serial link, and presents data and command transactions to the application by means of a simple application bus.

Front-End Interface Block

The Front-End Interface consists of all blocks in the block diagram except the UDC. These blocks are:

- Clock Control
- Transaction Decode and Clock Synchronization
- Registers and Control
- Memory Interface
- DMA Master
- PAB Interface

Each is described in the following sections.

Clock Control Block

To reduce power consumption, the UDC module uses gated clocks. The gated-clock control circuit ensures error-free clocking to the UDC module in a DFT (design for testability) friendly way.

Transaction Decode and Clock Synchronization Block

The transaction decode module includes synchronizer elements to connect the system clock domain logic at the system clock rate to the USB logic at 12 MHz.

It also decodes transactions that occur on the UDC's application bus for presentation to the other submodules. This module routes endpoint transactions to and from the Memory Interface module, and control, configuration, and status functions to and from the Registers and Control module.

Registers and Control Block

The Registers and Control module implements the general-purpose registers for the USBD. These registers implement these types of functions:

- Module configuration
- Module status
- Interrupt status and masking
- Frame-related controls such as start of frame, current frame number, and frame number match

This module is connected to the PAB peripheral bus interface by means of a simple internal peripheral bus. A single interrupt request line connects the interrupt subsystem in USBD to the ADSP-BF535 processor's interrupt controller.

Memory Interface Block

The Memory Interface module connects the USB endpoints to specific memory resources in the system. It contains registers for software to assign each USB endpoint to specific regions in system memory and to control USB transactions. A small FIFO is included in this module to minimize the effects of DMA latency on USB packet throughput.

The Memory Interface module contains all the endpoint-specific registers of the USBD. Software uses the endpoint registers to route USB endpoint data transfers to specific regions within the module's memory space.

This module is connected to the PAB interface by a simple internal peripheral bus.

For each USB packet transfer, the Memory Interface reports this information to the DMA master:

- Current USB endpoint number
- Buffer offset
- Transfer direction
- FIFO status

DMA Master Block

The DMA Master module connects the Memory Interface module to the DAB bus. The USB DMA channel must win arbitration within 22 DAB cycles of the assertion of its request.

This module maintains registers to indicate:

- Base address of the USBD module's access space within system memory
- Burst count for the current transfer
- Direction for the current transfer
- Status of the DMA module

In normal operation, the device software programs the DMA master with the base address of a 2 KB block of memory that the USBD module can use to buffer its data transfers. Transfer offset addresses generated by the Memory Interface module are concatenated with the DMA base address to produce a physical address on the DAB. Device software is responsible for allocating the 2 KB buffer space among all active endpoints.

The DMA address, transfer count, direction, and interrupt status are available for monitoring.

The DMA Master module is a standard component used to implement DAB bus accesses.

PAB Interface Block

The peripheral bus interface connects the PAB bus to the USBD module's internal registers.

Features and Modes

This section describes the functions supported by the USBD module.

Endpoint Types

The USBD module supports four different transfer types using all their supported packet sizes:

- Bulk
- Interrupt
- Isochronous
- Control

Any endpoint (1 through 7) can be defined as Bulk, Interrupt, or Isochronous. In accordance with the USB specification, endpoint 0 is always defined as a Control endpoint. After each hard reset, specific hardware configuration information must be downloaded to the USBD module, as described in "USBD Device Initialization" on page 14-46.

From the point of view of the USB device, Bulk and Interrupt transfers function identically on the device. The differences between these transfer types are implemented in the USB host controller, the USB device's descriptor tables, and in the software.

Data Transfers

The four USB transfer types fall into three broad data transfer categories which must be implemented in hardware: Bulk, Isochronous, and Control.

Bulk Data Transfers

Bulk data transfers include Interrupt and Bulk endpoints. Bulk data is transferred using an error checking protocol. Packet sizes for bulk data are limited to 8, 16, 32, and 64 bytes. From the USB host's perspective, the scheduling rules differ for each endpoint type. However, the data transfer mechanism is identical.

Isochronous Data Transfers

Isochronous endpoints are generally used for raw audio or video data. During each 1.0 ms frame, isochronous data transfers are given a guaranteed percentage of the USB's bandwidth. Because of this guaranteed bandwidth, the isochronous endpoints have no error retry capability.

Depending on the application, isochronous packets can range from 0 to 1023 bytes. Each isochronous endpoint has a specific maximum packet size that is guaranteed to it in each USB frame. On a frame-by-frame basis, the application can choose to use all available bandwidth or only part of it.

Control Transfers

Control transfers implement a COMMAND/DATA/STATUS protocol so the host can send command sequences to the device. Each DATA transfer within this sequence consists of bulk data; that is, error checking/retry is included, and the range of packet sizes is limited.

The command sequences include:

- Standard requests from Chapter 9 of the USB specification
- Class requests
- Vendor commands

The UDC module handles almost all standard requests directly without system intervention. Class requests are standardized and defined in the USB class specifications (for example, Printer, Imaging, Mass Storage). Vendor commands are commands that are specific to a vendor or device and not defined in a class specification.

In the ADSP-BF535 processor's UDC configuration, device software handles these requests:

- `SYNC_FRAME`, `SET_DESCRIPTOR`, `GET_DESCRIPTOR` standard requests
- All class and vendor requests

Device software must monitor the interrupts to determine when a setup packet has been received on a control endpoint and respond accordingly. Interrupts are provided for determining when problems have occurred during a transfer, and for gracefully recovering from them.

UDC Configuration Control

The UDC module supports one USB configuration. The configuration can support two interfaces, and each interface can support two alternate interfaces. The ADSP-BF535 processor has 8 physical endpoints for USB. Each endpoint can have a different definition depending on the interface and alternate interface used. For communications to occur, the specific device configuration must be downloaded into the module after system startup. See "Configuration of the UDC Module" on page 14-44.

Suspend Operation

The USBD_SUSPEND output is provided on the module interface to put the USB transceivers into suspend mode. System hardware can also use this signal to stop and start the module clocks when the USB is suspended. When the USBD_SUSPEND signal asserts, the system can stop all clocks.

On the inactive edge of USBD_SUSPEND, the system must restart all clocks within 10 ms.

USB Suspend mode is selected by the USB host. When the USB host wants to put a device into the suspended mode, it stops sending traffic to the device. After 3 ms of inactivity on the bus, the device automatically drops into a suspended state. After the USB host generates resume signaling on the bus, the device reenters its active state and resumes communication on the bus.

Clocking

The 4X USB Clock (USBD_CLK4X) is nominally 48 MHz +/- 500 ppm for full-speed operation. The UDC uses this clock internally to drive a clock recovery circuit and to generate a 12 MHz clock (full-speed operation for the UDC module).

The rest of the USBD module operates at system clock frequencies.

USB Transceiver

A USB buffer chip, such as the Philips PDI USB P11A, must be added in order to use the USB device functionality of the ADSP-BF535 processor. The diagram in Figure 14-2 shows how to connect a Philips PDI USB P11A transceiver to the ADSP-BF535 processor.

If the external transceiver supports a low power mode, it can be controlled by using one of the programmable flag pins.

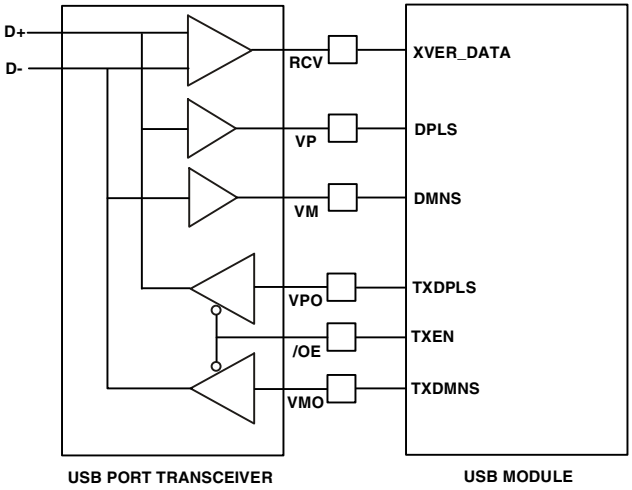


Figure 14-2. Connecting a USB Transceiver

Full Speed vs. Low Speed USB

The USB 1.1 specification supports two speed grades: full speed (12 Mbits/sec) and low speed (1.5 Mbits/sec). The ADSP-BF535 processor is configured to support full-speed operation at all times. The external transceiver must also be configured to support the fast slew rate of the full-speed USB mode.

Registers

This section describes the register set for the USBD module. All USBD memory-mapped registers are 16 bits wide, and 16-bit aligned.

The registers are grouped into three categories: general registers, DMA registers, and endpoint registers.

The general registers handle configuration, control, and status. These registers are described in later sections of this chapter. They are:

- USBD Device ID register (USBD_ID)
- Current USB Frame Number register (USBD_FRM)
- Match Value for USB Frame Number register (USBD_FRMAT)
- Enable Download of Configuration Into UDC Core register (USBD_EPBUF)
- USBD Module Status register (USBD_STAT)
- USBD Module Configuration and Control register (USBD_CTRL)
- Global Interrupt register (USBD_GINTR)
- Global Interrupt Mask register (USBD_GMASK)

The DMA registers handle configuration and control of the DMA master channel embedded within the module. These registers are described in later sections of this chapter. They are:

- DMA Master Channel Configuration register (USBD_DMACFG)
- DMA Master Channel Base Address Low register (USBD_DMABL)
- DMA Master Channel Base Address High register (USBD_DMABH)
- DMA Master Channel Count register (USBD_DMACT)
- DMA Master Channel DMA Count register (USBD_DMAIRQ)

The USB Endpoint registers handle endpoint specific interrupts and control of the datapath. The number of endpoint specific registers is dependent on the number of endpoints with which the design is compiled.

The ADSP-BF535 processor's design is configured for eight endpoints. The endpoint registers are described in later sections of this chapter. They are:

- USB Endpoint x Interrupt registers (USBD_INTRx)
- USB Endpoint x Mask registers (USBD_MASKx)
- USB Endpoint x Control registers (USBD_EPCFGx)
- USB Endpoint x Address Offset registers (USBD_EPADRx)
- USB Endpoint x Buffer Length registers (USBD_EPLENx)

USB Device ID Register (USBD_ID)

This register, shown in Figure 14-3, is used to identify the module to the system. Using binary-coded decimal, the USBD_SPEC field indicates the version of the USB specification that the module is compliant to. A value of 0x110 corresponds to the USB Specification, Revision 1.1. The USBD_PID field indicates the silicon revision of the module. Use this value to allow software drivers to adapt to different hardware revisions.

The specific vendor ID (for example, 0x0456 for Analog Devices, Inc.) and the product ID (for example, 0x2153 for the ADSP-BF535 processor) should not be programmed using this register. They should be programmed using the Device Descriptor set.

USB Device ID Register (USBD_ID)
RO

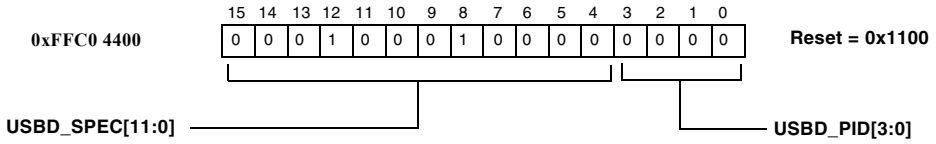


Figure 14-3. USB Device ID Register

Current USB Frame Number Register (USBD_FRM)

This register, shown in Figure 14-4, reports the current USB frame number. This number updates each time a start of frame (SOF) token is received on the USB.

Current USB Frame Number Register (USBD_FRM)
RO

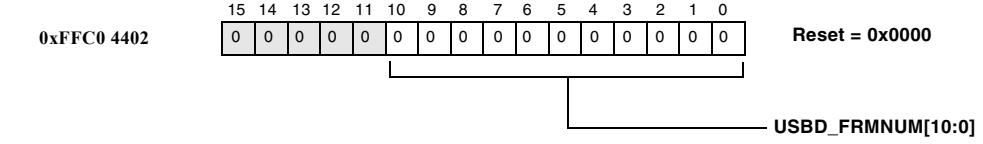


Figure 14-4. Current USB Frame Number Register

Match Value for USB Frame Number Register (USBD_FRMAT)

This register, shown in Figure 14-5, provides a mechanism for software to wait for the arrival of a specific USB frame number. This feature is typically used with isochronous endpoints for synchronization between the

data source and sink. When the value in this register matches the value received on the Current USB Frame Number register (USBD_FRM), an interrupt is generated in the Global Interrupt register (USBD_GINTR).

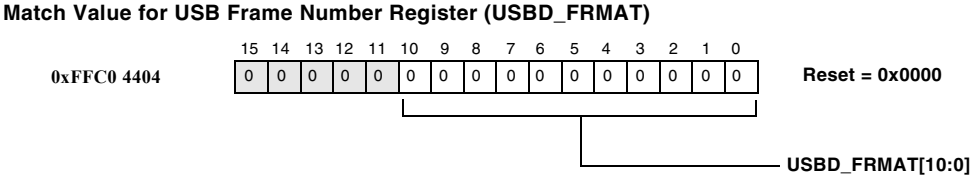


Figure 14-5. Match Value for USB Frame Number Register

Enable Download of Configuration Into UDC Core Register (USBD_EPBUF)

This register, shown in Figure 14-6, allows the system to download endpoint buffer data into the UDC module after hard reset events. The EpBuf structures hold configuration information for each possible endpoint on the device including the configuration number, packet size, direction, type, and the interface/alternates to which the endpoint is attached.

The total number of physical endpoints on the device is hardwired, but the specific configuration of each endpoint is downloaded into the device by the firmware.

The USBD_EPDOWN field allows software to write data into the UDC core's internal EpBuf structures. On reads, this field always returns 0s.

Enable Download of Configuration Into UDC Core Register (USBD_EPBUF)

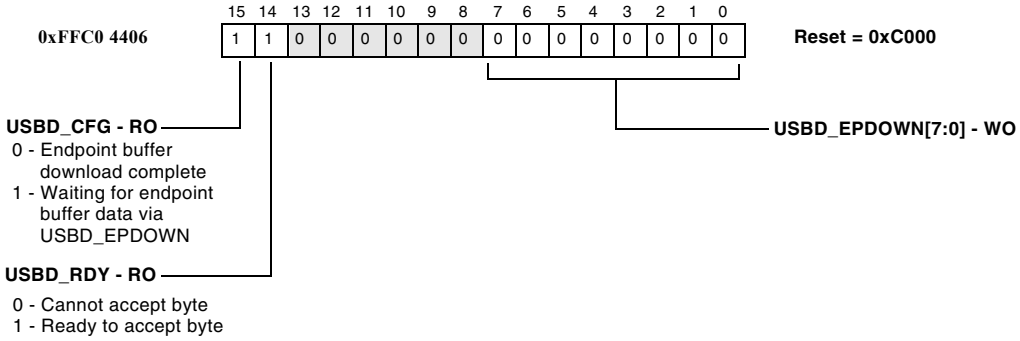


Figure 14-6. Enable Download of Configuration Into UDC Core Register

USBD Module Status Register (USBD_STAT)

This register reports various status conditions from the USB and from the module's internal logic (see Figure 14-7).

The USBD_CFG field contains the current USB configuration number. It is updated when the USB host sends a SET_CONFIGURATION request to the USB device.

The USBD_IF field contains the USB interface number. It is updated when the USB host sends a SET_INTERFACE request to the device. It changes along with the USBD_AIF field.

The USBD_AIF field contains the alternate interface number. This field is updated when the USB host sends a SET_INTERFACE request to the device. It changes along with the USBD_IF field.

The USBD_SIP bit is valid when the USBD_PIP bit is set. The USBD_PIP bit asserts at the start of the packet request from the UDC module to the front-end interface and deasserts at the end of a packet. It is used to indicate when the UDC module is actively transferring data on the USB.

The USBD_EP field holds the currently accessed endpoint, 0 through 7. This register bit field is valid only when the USBD_PIP bit is set. Note this is the endpoint number *within the* USBD *hardware*. It may not match the actual USB endpoint number due to mapping which occurs in the UDC hardware.

USBD Module Status Register (USBD_STAT)
RO

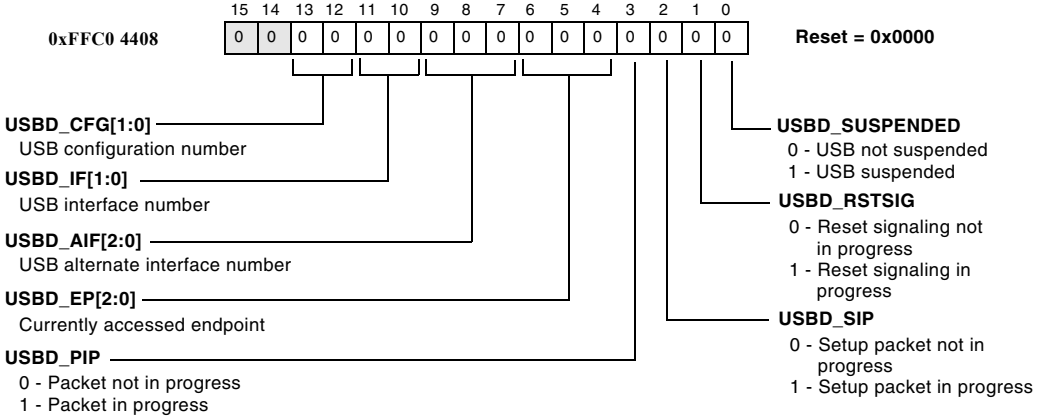


Figure 14-7. USBD Module Status Register

USBD Module Configuration and Control Register (USBD_CTRL)

This register, shown in Figure 14-8, provides software control of various features of the USBD module. Note when the USBD_ENA bit is programmed to disable the USBD module, the module does not accept command and data packets from the USB, but suspend/resume can still occur, as well as the 1 ms start-of-frame interrupt.

The USBD_EPxSTALL bits assert a stall request on the corresponding endpoint at the next request from the USB host. The bit automatically clears after the stall takes effect.

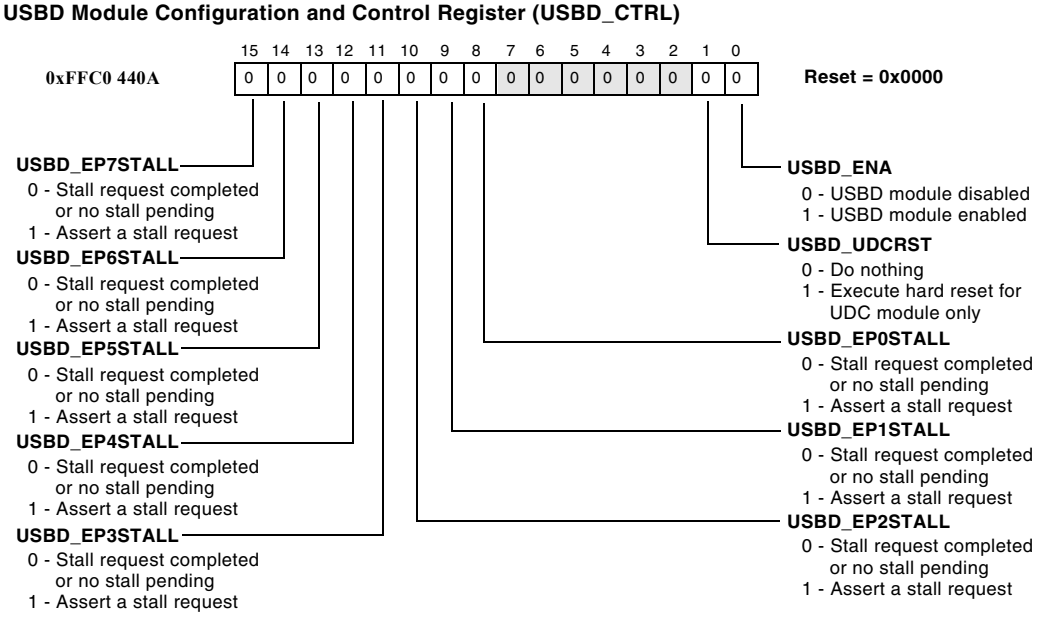


Figure 14-8. USBD Module Configuration and Control Register

Global Interrupt Register (USBD_GINTR)

The Global Interrupt register, shown in Figure 14-9, is the top-tier interrupt register for the module. This register tracks the status of all high-priority interrupts, and it includes one interrupt bit for each endpoint.

All bits are write-1-to-clear (W1C). Writing a 0 to any bit in this register has no effect.

The Global Interrupt Mask register (`USBD_GMASK`) can mask all bits. An interrupt may be masked or unmasked at any time. When masked, an interrupt does not affect the state of the module's `USBD_INTR` output. To determine the interrupt status, software must poll a masked interrupt.

The interrupt circuit includes a protection mechanism for simultaneous clear and set operations. If software tries to clear an interrupt at the same time as the hardware is trying to set the interrupt, the set operation takes priority over the clear.

All the interrupts are edge triggered. Modifying the mask bit in an endpoint interrupt register may trigger an interrupt from an earlier endpoint interrupt.

See "Interrupt Descriptions" on page 14-37 for detailed explanations of each interrupt.

Global Interrupt Register (USBD_GINTR)
All bits are write-1-to-clear.

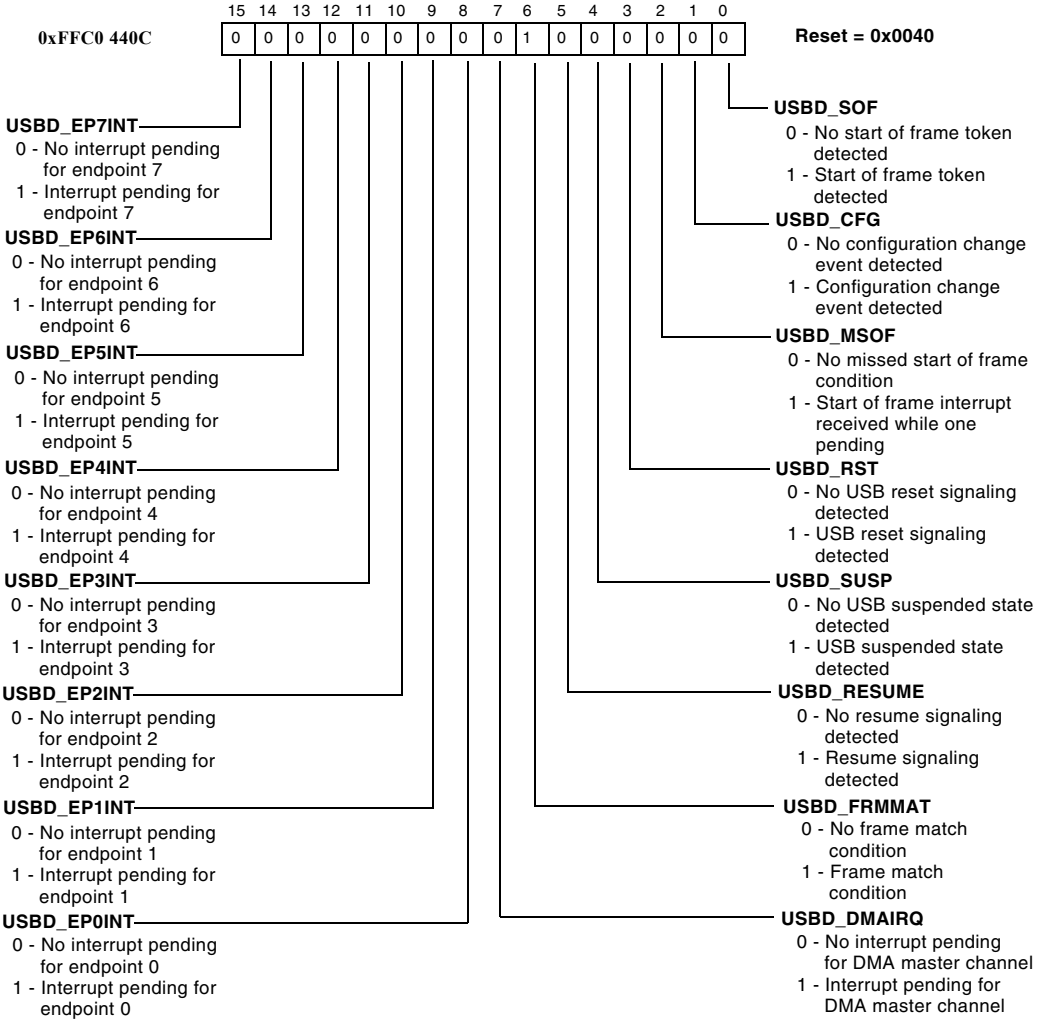


Figure 14-9. Global Interrupt Register

Global Interrupt Mask Register (USBD_GMASK)

The Global Interrupt Mask register, shown in Figure 14-10, allows masking of the interrupts in the Global Interrupt register. Setting a bit in the Global Interrupt Mask register masks the interrupt output from asserting for that interrupt. Clearing a bit in the Global Interrupt Mask register enables the interrupt output to assert for that interrupt.

Global Interrupt Mask Register (USBD_GMASK)
For all bits, 0 - Enable interrupt to assert, 1 - Mask the interrupt output.

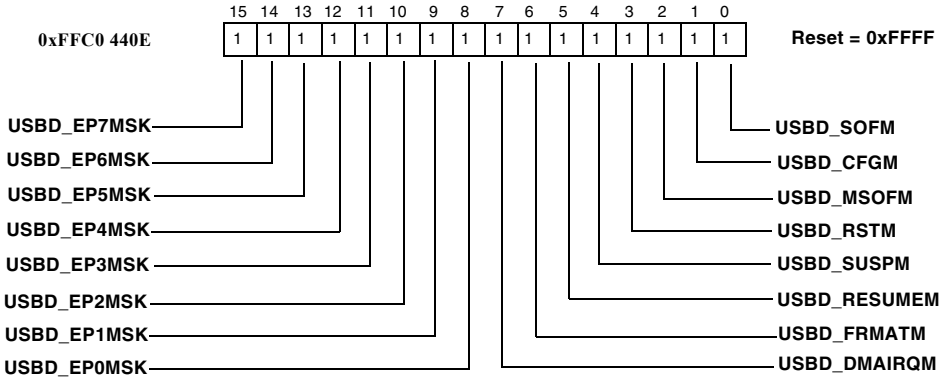


Figure 14-10. Global Interrupt Mask Register

DMA Master Channel Configuration Register (USBD_DMACFG)

This register, shown in Figure 14-11, is used to configure the DMA master channel.

The USBD_DMABC bit is the DMA buffer clear bit. It forces the DMA FIFO to be cleared. Writing a 1 initiates and holds the buffer in a clear state. A write of 0 must follow before normal operation can resume. It is recommended that the DMA be disabled and the USB be prevented from accessing the FIFO before USBD_DMABC is set.

The USBD_DMAEN bit indicates the DMA-enabled and active status.

DMA Master Channel Configuration Register (USBD_DMACFG)
DMA configuration register. USBD_IOE bit has no effect on DMA transactions.

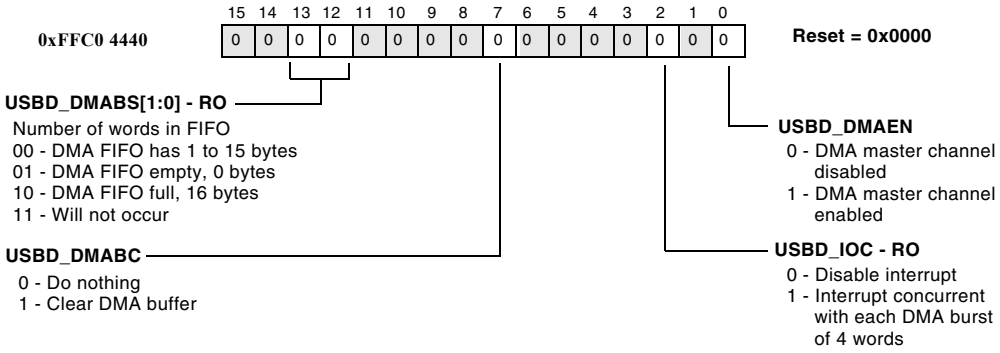


Figure 14-11. DMA Master Channel Configuration Register

DMA Master Channel Base Address Low Register (USBD_DMABL)

This register, Figure 14-12, contains the low order bits of the base address for the memory block to be accessed for USB buffers.

DMA Master Channel Base Address Low Register (USBD_DMABL)
Low order bits of base address.

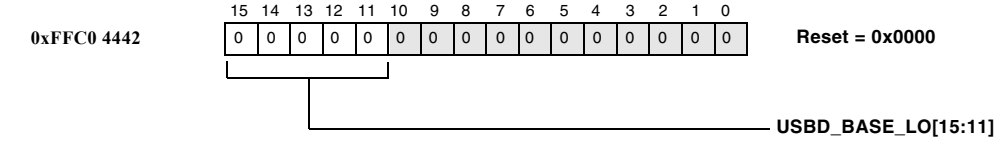


Figure 14-12. DMA Master Channel Base Address Low Register

To produce a physical address on a per-transfer basis, the DMA master channel replaces bits [10:0] with an endpoint specific offset value. Bits [10:0] always read back 0.

DMA Master Channel Base Address High Register (USBD_DMABH)

This register, shown in Figure 14-13, contains the high order bits of the base address for the memory block to be accessed for USB buffers.

DMA Master Channel Base Address High Register (USBD_DMABH)
High order bits of base address.

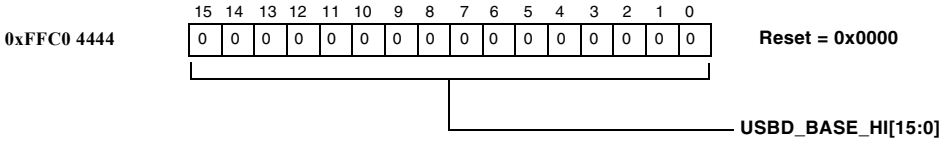


Figure 14-13. DMA Master Channel Base Address High Register

It is combined with the bits in the DMA Master Channel Base Address Low register and an endpoint specific offset value to produce a physical address on the DAB.

DMA Master Channel Count Register (USBD_DMACT)

This register, shown in Figure 14-14, holds the DMA count for the current transfer. This reads back the number of bytes pending for the DMA transfer on the currently selected endpoint, and its value ranges from 0 to 4 words (16 bytes).

DMA Master Channel Count Register (USBD_DMACT)

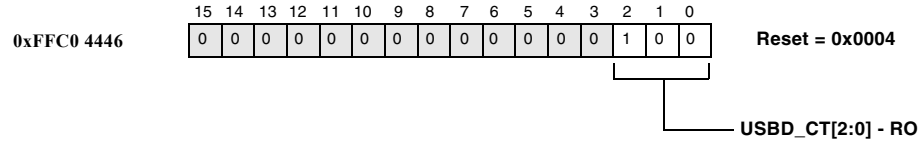


Figure 14-14. DMA Master Channel Count Register

DMA Master Channel DMA Interrupt Register (USBD_DMAIRQ)

The DMA Interrupt Request feeds into the Global Interrupt register (USBD_GINTR). To mask the DMA Complete (DMA_COMP) interrupt in the USBD_DMAIRQ register (Figure 14-15), program the corresponding enable bit in the DMA Master Channel Configuration register. DMA_ERROR is not maskable at the DMA level. All bits are write-1-to-clear. Writing 0 to any bit in this register has no effect.

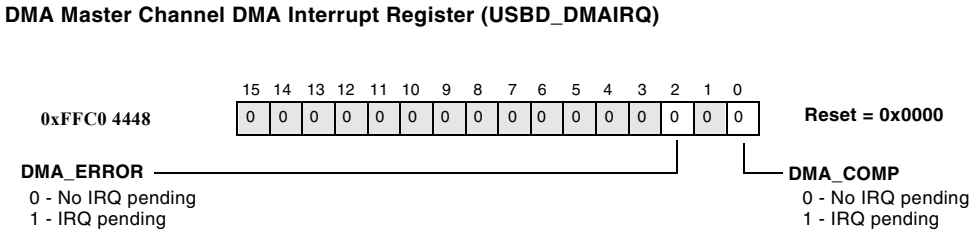


Figure 14-15. DMA Master Channel DMA Interrupt Register

USB Endpoint x Interrupt Registers (USBD_INTRx)

The USB Endpoint x Interrupt registers (Figure 14-16) report interrupt status for endpoint x. All bits are write-1-to-clear.

The USBD_MSETUP bit is set when a UDC_SETUP interrupt is pending, and another setup packet is received. This condition can occur if the USB device's ACK of a setup packet is corrupted, and the USB host retries the setup packet. If both USBD_MSETUP and USBD_SETUP are pending, and software tries to clear them both at the same time, a third setup packet is received, and USBD_MSETUP clears, while USBD_SETUP remains set.

The USBD_SETUP bit is set when a setup packet is received on the current endpoint.

The USBD_MERR bit is 1 if the USBD_BCSTAT interrupt is set, and another packet request has come in for the current endpoint. This allows software to determine whether the device is NAKing requests from the USB host.

The USBD_BCSTAT bit is 1 if the byte count for the endpoint has decremented to 0. This interrupt should be cleared before loading a new value into the USBD_EPLENx registers.

The USBD_PC bit is 1 if a complete USB packet has just transferred across the USB on the current endpoint. This interrupt normally asserts at the end of every USB packet transfer. However, in the case of isochronous transfers, this interrupt does not assert if the packet transferred with errors. Software can use the USBD_PC and USBD_TC interrupts, along with the endpoint configuration registers, to determine the fate of isochronous packets.

The USBD_TC bit is 1 if a USB transfer has completed on the current endpoint. This interrupt asserts for Bulk and Interrupt endpoints when a short packet is transferred. For Control endpoints, this interrupt asserts at the end of the control transfer's data phase—either from a short packet transfer or from the expiration of the wLength counter from the setup packet. For Isochronous endpoints, this interrupt asserts at the end of every packet. The error detection mechanism used for isochronous transfers requires the software to examine the endpoint configuration registers and determine how much data was actually transferred.

USB Endpoint x Interrupt Registers (USBD_INTRx)

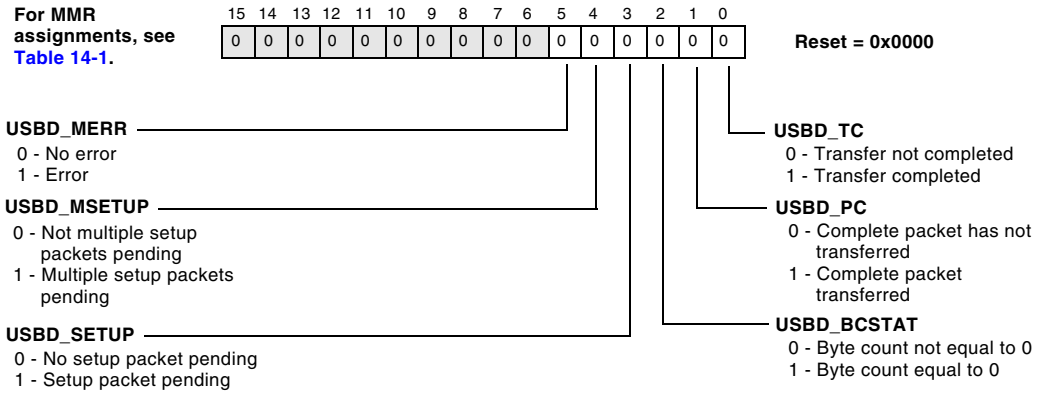


Figure 14-16. USB Endpoint x Interrupt Registers

Table 14-1. USB Endpoint x Interrupt Register MMR Assignments

Register Name	Memory-Mapped Address
USBD_INTR0	0xFFC0 4480
USBD_INTR1	0xFFC0 448A
USBD_INTR2	0xFFC0 4494
USBD_INTR3	0xFFC0 449E
USBD_INTR4	0xFFC0 44A8
USBD_INTR5	0xFFC0 44B2
USBD_INTR6	0xFFC0 44BC
USBD_INTR7	0xFFC0 44C6

USB Endpoint x Mask Registers (USBD_MASKx)

The USB Endpoint x Mask registers, shown in Figure 14-17, can be used to mask the interrupts in the USB Endpoint x Interrupt registers (`USBD_INTRx`). Setting a bit in the USB Endpoint x Mask registers masks

Global Interrupt register (USBD_GINTR) bits from asserting for that interrupt. Clearing a bit in the USB Endpoint x Mask register enables the interrupt bit to assert for that interrupt.

USB Endpoint x Mask Registers (USBD_MASKx)

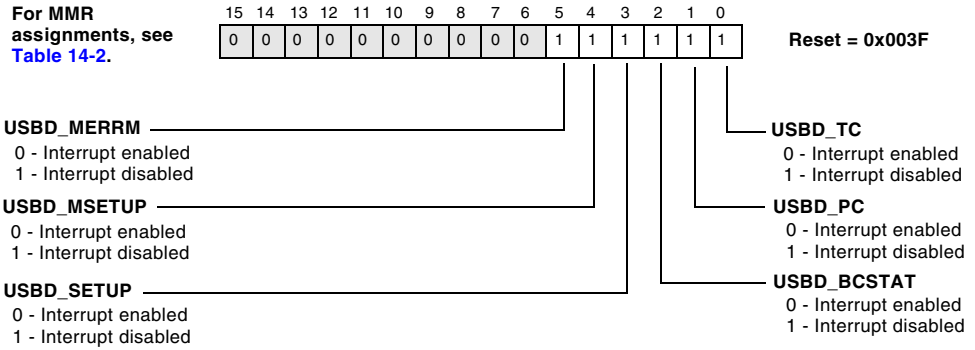


Figure 14-17. USB Endpoint x Mask Registers

Table 14-2. USB Endpoint x Mask Register MMR Assignments

Register Name	Memory-Mapped Address
USBD_MASK0	0xFFC0 4482
USBD_MASK1	0xFFC0 448C
USBD_MASK2	0xFFC0 4496
USBD_MASK3	0xFFC0 44A0
USBD_MASK4	0xFFC0 44AA
USBD_MASK5	0xFFC0 44B4
USBD_MASK6	0xFFC0 44BE
USBD_MASK7	0xFFC0 44C8

USB Endpoint x Control Registers (USBD_EPCFGx)

The USB Endpoint x Control registers, shown in Figure 14-18, allow programming the characteristics of individual endpoints.

The USBD_ARM bit is set to 1 to arm an endpoint. When the endpoint is armed, the USBD module can respond to transfer requests on the endpoint. If USBD_ARM = 1 and USBD_BC = 0, then the module generates a 0-length data packet. This bit clears automatically when USBD_BC decrements to 0 or when an end-of-transfer condition is detected.

The USBD_ARM bit is set to 0 to disarm an endpoint. If a packet transfer is in progress, clearing this bit does not stop the current packet transfer, but it does prevent succeeding packets in a multipacket transfer from occurring. For example, consider a 64-byte packet, 128-byte count. If the system clears USBD_ARM during the transfer of the first packet, then the first packet completes normally, but the second one is unable to transfer until the endpoint is armed again.

The USBD_DIR bit indicates the transfer direction between the USB device and the USB host. If the USB host requests a packet transfer and the direction of the request does not agree with the endpoint direction, then the module NAKs the current packet. This can occur on USB control transfers when the endpoint is set to OUT for the setup packet and the USB host requests an IN data-phase packet before the device has recognized the presence of the setup packet.

The USBD_TYP field sets the USB endpoint type. For Endpoint 0, the USBD_TYP field is hardwired to 00. For all other endpoints, only Bulk, Interrupt, and Isochronous type codes are valid, with 01 for Bulk, 10 for Interrupt, and 11 for Isochronous.

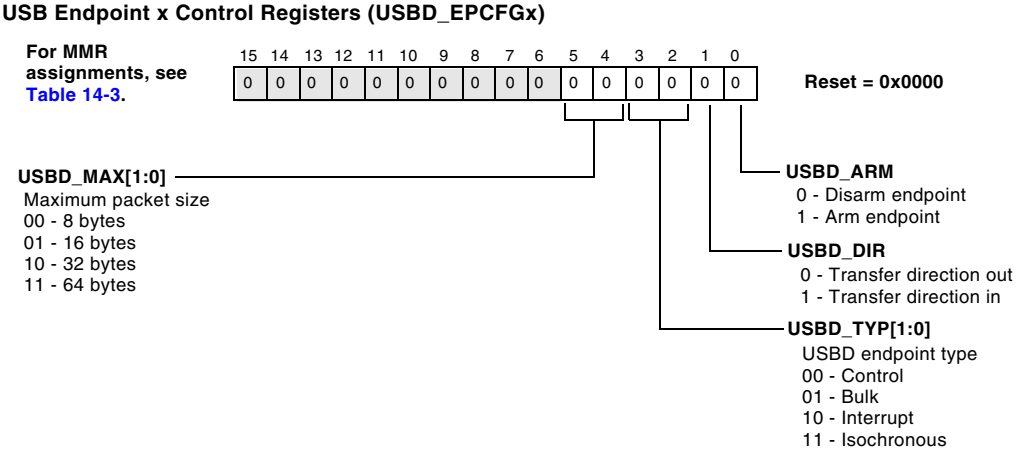


Figure 14-18. USB Endpoint x Control Registers

Table 14-3. USB Endpoint x Mask Register MMR Assignments

Register Name	Memory-Mapped Address
USBD_EPCFG0	0xFFC0 4484
USBD_EPCFG1	0xFFC0 448E
USBD_EPCFG2	0xFFC0 4498
USBD_EPCFG3	0xFFC0 44A2
USBD_EPCFG4	0xFFC0 44AC
USBD_EPCFG5	0xFFC0 44B6
USBD_EPCFG6	0xFFC0 44C0
USBD_EPCFG7	0xFFC0 44CA

USB Endpoint x Address Offset Registers (USBD_EPADRx)

These registers, represented in Figure 14-19, are used to program the memory buffer offset for data transfers.

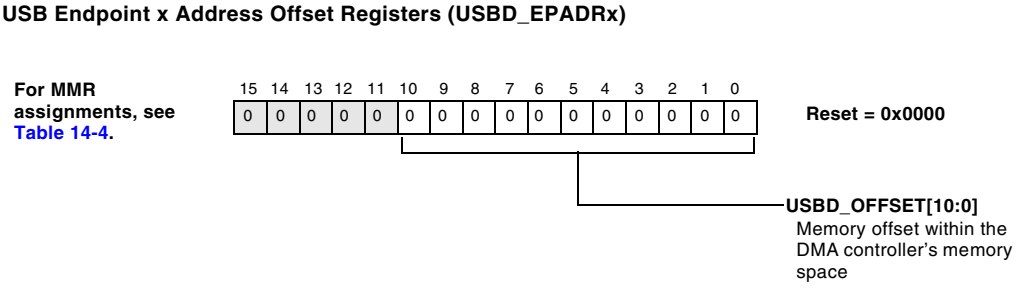


Figure 14-19. USB Endpoint x Address Offset Registers

They hold the memory offset within the DMA controller's memory space for the transfer associated with this endpoint.

DMA transfers occur on 16-byte aligned offsets within the controller's memory space (2 KB). As each packet is transferred on the USB, the `USBD_OFFSET` increments by the packet size.

For the USB Endpoint x Address Offset registers and the USB Endpoint x Buffer Length registers (`USBD_EPLENx`), it is assumed that software is keeping track of the original buffer addresses and can calculate the bytes transferred and the end of valid data based on reading back these registers at the end of a transfer.

Table 14-4. USB Endpoint x Address Offset Register MMR Assignments

Register Name	Memory-Mapped Address
USBD_EPADR0	0xFFC0 4486
USBD_EPADR1	0xFFC0 4490
USBD_EPADR2	0xFFC0 449A
USBD_EPADR3	0xFFC0 44A4
USBD_EPADR4	0xFFC0 44AE
USBD_EPADR5	0xFFC0 44B8
USBD_EPADR6	0xFFC0 44C2
USBD_EPADR7	0xFFC0 44CC

USB Endpoint x Buffer Length Registers (USBD_EPLENx)

These registers, shown in Figure 14-20, are used to program the memory buffer byte counts for data transfers.

Packet transfers can be up to 1023 bytes (the maximum size isochronous packet allowed by USB specification). If the byte count for an endpoint is larger than the endpoint's packet size, the transfer is partitioned into a number of individual packet transfers at the endpoint's maximum packet size.

For non-isochronous endpoints, this field decrements by the packet size as each packet is transferred on the USB. For isochronous endpoints, this field decrements by the DMA burst size (16 bytes) at the end of each DMA burst.

USB Endpoint x Buffer Length Registers (USBD_EPLENx)

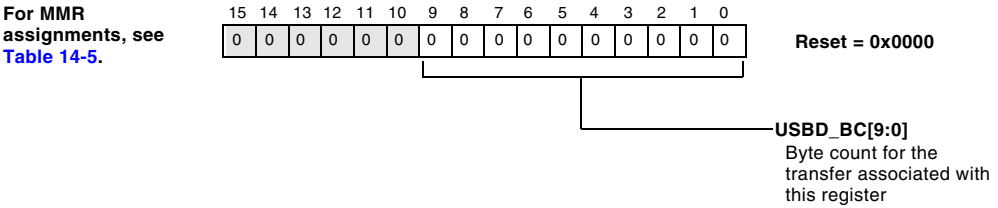


Figure 14-20. USB Endpoint x Buffer Length Registers

Table 14-5. USB Endpoint x Buffer Length Register MMR Assignments

Register Name	Memory-Mapped Address
USBD_EPLEN0	0xFFC0 4488
USBD_EPLEN1	0xFFC0 4492
USBD_EPLEN2	0xFFC0 449C
USBD_EPLEN3	0xFFC0 44A6
USBD_EPLEN4	0xFFC0 44B0
USBD_EPLEN5	0xFFC0 44BA
USBD_EPLEN6	0xFFC0 44C4
USBD_EPLEN7	0xFFC0 44CE

UDC Endpoint Buffer Register

The UDC Endpoint Buffer register, shown in Figure 14-21, holds the configuration of each logical endpoint. The register is loaded using the endpoint buffer information in `USBD_EPBUF`, see "Enable Download of Configuration Into UDC Core Register (USBD_EPBUF)" on page 14-18. This register cannot be accessed through the peripheral bus. This register is not a memory-mapped register.

Each logical endpoint is broken into 5 bytes with each byte loaded with an access using the USBD_EPBUF register.

The MSB of the UDC Endpoint Buffer register should be loaded first.

All endpoint buffer registers must be programmed for correct function of the USB device. Program all unused endpoint buffers to zeros.

The ADSP-BF535 processor supports 8 *physical* endpoints, one configuration, two interfaces, and two alternate interfaces. Given this configuration, the EPNUM field (physical endpoints) can hold values from 0 to 7; the EP_CONFIG field can be 0 or 1, the EP_INTERFACE and the EP_ALTSETTING fields can be 0 or 1.

The EP_MAXPKTSIZE field holds the maximum packet size. It can take values from 0 to 1023 with 8, 16, 32, or 64 as the only possible values for physical endpoints defined as Control, Bulk, and Interrupt.

The EP_BUFADRPTR field reflects the endpoint being used for the transaction. For the ADSP-BF535 processor, the EP_BUFADRPTR should reflect the endpoint on bits [0:2].

UDC Endpoint Buffer Register

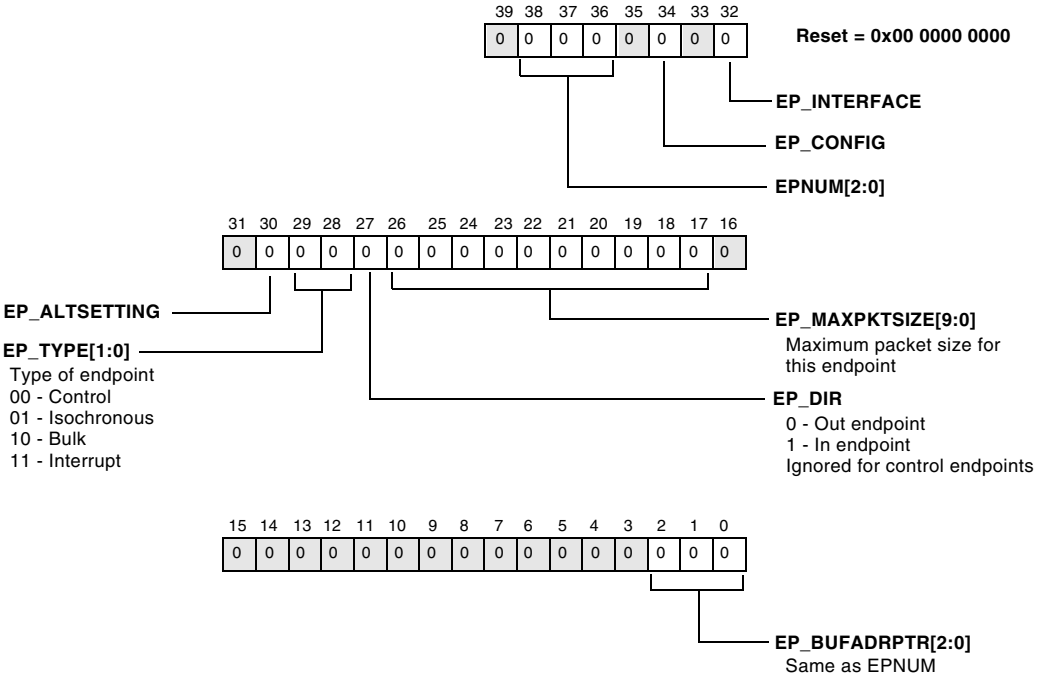


Figure 14-21. UDC Endpoint Buffer Register

Interrupt Descriptions

This section provides detailed descriptions of each interrupt bit and the operation of the interrupt subsystem.

The interrupt subsystem is based on a two-tiered approach. The USBD module presents a single interrupt output to the system, but internally it manages separate interrupt registers for critical module events, the DMA master channel, and each endpoint.

Software can mask each interrupt individually. Under this scheme, it is possible to enable software polling of the interrupts or to enable the hardware interrupt.

USB General Interrupts

The general interrupts are the top tier of the interrupt subsystem. This tier has the general interrupts and an interrupt for each endpoint.

USBD_SOF – Start of Frame

The `USBD_SOF` interrupt indicates that the USB device has received a Start of Frame (SOF) token from the USB host. The USB host broadcasts the SOF token at 1.0 ms intervals as a timebase for isochronous transfers. Isochronous endpoints can use this interrupt as part of their synchronization mechanism.

USBD_CFG – USB Configuration Change

The `USBD_CFG` interrupt indicates to the device that a configuration change event occurred on the USB. The host has requested the device to change to a different configuration, or it has switched to a different alternate interface for one of the interfaces in the current configuration.

Timely service of this interrupt is critical to ensure proper device operation. Once this interrupt is received, the software must immediately reconfigure itself to match the new configuration settings, then clear the interrupt.

To prevent the device's configuration from becoming out of sync with the intended configuration (from the USB host's viewpoint), the device refuses all traffic from the USB while the interrupt is pending. Masking the interrupt does not disable the USB activity lockout feature.

USBD_MSOF – Missed Start of Frame

This interrupt, or missed SOF indicator, asserts when a USBD_SOF interrupt is already pending in the device, and another SOF is received. Software can use this interrupt, in combination with the USBD_SOF interrupt, to help maintain synchronization with application software running on the USB host system.

USBD_RST – Reset Signaling Detected

This interrupt asserts when reset signaling is present on the USB. Software can use this interrupt as an indicator that something significant is about to happen or has happened (and perhaps the software wants to disable in-progress DMA transfers). Software can monitor the USBD_RSTSIG bit in the USBD_STAT register to monitor the state of reset signaling.

USBD_SUSP – Device Suspended

This interrupt asserts when the USB device enters suspend mode after 3.0 ms of inactivity on the USB. This interrupt complements the USBD_SUSPEND signal on the USB Port Interface which can be used by the system to start and stop the clocks for low power operation.

USBD_RESUME – Resume Signaling

This interrupt asserts when the USB device leaves suspend mode. This interrupt complements the USBD_SUSPEND signal on the USB Port Interface, which can be used by the system to start and stop the clocks for low power operation.

Because internal logic is synchronous, this interrupt may not assert when the module leaves the suspended state, depending on how and when the system restarts the clocks.

USBD_FRMAT – Frame Match

The USBD_FRMAT interrupt indicates a match between the current USB frame number (stored in USBD_FRM) and the match value stored in the USB Frame Match (USBD_FRMAT) register. This feature is of interest for isochronous applications that need to synchronize operations between the data source and data sink.

The USBD_FRMMAT bit in the USBD_GINTR register is 1 when there is a frame match condition between the Current USB Frame Number register (USBD_FRM) and the Match Value for USB Frame Number register (USBD_FRMAT). After hard reset, this interrupt is set because both the USB frame number and the match value are 0.

USBD_EPxINT – Endpoint(x) Interrupt

This interrupt indicates that an unmasked interrupt is present in the USBD_INTRx register.

DMA Master Interrupts

The DMA Master channel reports two interrupts:

- DMA_COMP
- DMA_ERROR

Each interrupt source is sticky. Thus an enabled interrupt source which asserts an interrupt request by setting the appropriate bit remains set until cleared. Clearing an interrupt source is accomplished by writing a 1 to the corresponding bit.

DMA_COMP

The DMA_COMP interrupt is generated during a DMA burst transfer of data. Data is transferred over the DAB bus as a burst of four 32-bit words (16 bytes total). The interrupt is generated after the first two words have been transferred. This interrupt is not endpoint specific and is an indicator that the USB device is in the process of a data transfer.

DMA_ERROR

Two general conditions cause a DMA bus error:

- Data misalignment
- Illegal memory access

Data Misalignment

Misalignment of address to data size occurs when the USBD_OFFSET (USBD_EPADRx) is not word aligned. If the address is misaligned to the data:

- The DMA engine does not access the bus.
- The DMA enable bit (bit 0 of DMA Configuration register) is cleared.
- Bit 2 of the DMA Master Channel DMA Count register is set.

Illegal Memory Access

If an illegal memory access occurs, such as an access to unimplemented memory, the DMA engine responds with an error and:

- Completes the current burst transfer.
- Clears the DMA Enable bit (Bit 0 of the Configuration Word).
- Sets bit 2 of the DMA Master Channel DMA Count register.

USB Endpoint Interrupts

Each endpoint interrupt in the USB General Interrupt register is based on the interrupt status of all interrupt sources for the endpoint. If any of the endpoint interrupts is active and unmasked, an interrupt is generated and sets the corresponding bit in the Global Interrupt register (USBD_GINTR).

The individual interrupt behaviors are described in the following sections. Note that it is possible to cause the top-tier interrupt to retrigger by toggling the mask bit for the endpoint interrupts.

USBD_TC – Transfer Complete

The USBD_TC interrupt asserts when a transfer completes on the current endpoint. Because the USB protocol does not maintain a transfer count, the end of a transfer is assumed whenever a short packet is transferred on an endpoint. The USB protocol requires all bulk interrupt and control endpoints to transmit the maximum sized packet for the endpoint (that is, 8, 16, 32, 64) until the end of a data transfer.

At the end of the transfer, the remaining bytes are transmitted in a short packet. If the data transfer ends on a packet boundary, the short packet can contain 0 bytes. The 0 byte packet is not required for isochronous transfers.

This interrupt does not assert when a setup packet is received on a control endpoint.

USBD_PC – Packet Complete

This interrupt asserts when a packet transfer completes on the current endpoint. System software can use this interrupt to monitor the progress of a data transfer.

USBD_BCSTAT – Buffer Complete

This interrupt asserts when the byte count has counted down to 0 for an endpoint. Software can examine the USBD_EPADRx and USBD_EPLENx registers to determine the actual progress of the transfer.

USBD_SETUP – Setup Packet Received

This interrupt asserts when a setup packet is received on the current control endpoint. See USBD_MSETUP below for the device behavior if a USBD_SETUP interrupt is already pending and a new one arrives.

USBD_MSETUP – Multiple Setup Packets Received

This interrupt asserts when a USBD_SETUP interrupt is already pending and another setup packet is received, as could happen when the USB device's ACK of the first setup packet is not received by the USB host, and the USB host retransmits the command.

USBD_MERR – Memory Controller Error

This interrupt covers the case of a buffer overrun or underrun. This interrupt sets when the USBD_BC interrupt is pending for an endpoint and another packet request has been received from the USB host for the same endpoint. When this situation occurs, the USB device is NAKing requests from the USB host and must correct the situation to ensure maximum bandwidth usage.

The specific problem here is with OUT packets. Each time the USB host tries to send a 64-byte packet, for example, it sends the whole packet "in the blind" assuming that the USB device will accept it. If the USB device is not ready, then this is wasted bus bandwidth that could be given to another device which is ready to complete a packet transfer.

USB Programming Model

This section discusses the USBD module from a programming viewpoint, including programming issues related to device initialization and data transfers.

Configuration of the UDC Module

The UDC module maintains a set of internal buffers that describe the endpoints and configurations available on the device. These buffers must be reloaded whenever a hard reset event occurs. These buffers are referred to as the logical endpoints or EpBufs.

For each unique physical endpoint on the device, there can be more than one EpBuf. For example, it is possible to have endpoint 1 in USB configuration 1, as well as endpoint 1 in USB configuration 2. While the two endpoints have the same physical number as far as the USB is concerned, they are mapped to two separate EpBufs within the module. Because the endpoints are mapped to two different buffers, they can have completely different configurations.

The ADSP-BF535 processor is hardwired to support one configuration, two interfaces, two alternate interfaces per interface, and it has a pool of 64 logical endpoint buffers to program.

Note: Even if all 64 endpoint buffers are not used in a specific product, the core expects all 64 logical endpoint buffers to be allocated data after each hard reset.

The UDC module gives software access to the endpoint buffers via the USBD_EPBUF register. The following steps must be followed to ensure proper download.

1. Verify that the UDC is ready to accept configuration data by reading the USBD_RDY bit from the USB_EPBUF register.
2. Write the first/next byte of the endpoint buffer to the USBD_EPDOWN field of the USBD_EPBUF register.

   This action causes the USBD_RDY bit to change to 0. The USD_CFG bit remains at 1 until the download process is complete.
3. Wait for the USBD_RDY bit to return to 1.

   Depending on the clock frequency of the PAB, the wait can be 30 or more clock periods.
4. Go back to step 2.

   After all bytes have been written, read back the USBD_CFG bit from the USBD_EPBUF register and verify that it has switched from 1 to 0, indicating that the UDC has received all the bytes expected.

For each logical endpoint, you must download 5 bytes of data starting with the most significant byte. See the UDC Endpoint Buffer register in Figure 14-21 on page 14-37 for more information about the UDC register format.

For endpoint 0, program the configuration, interface, and alternate values to 0 in the UDC Endpoint Buffer register.

USBD Device Initialization

Device initialization consists of several steps that prepare the USBD module to transfer data. On a hard reset event, software should perform the following steps.

1. Program the UDC module's endpoint buffers. For more information, see "Configuration of the UDC Module" on page 14-44.
2. Program interrupt masks in USBD_GINTR, USBD_INTRx and the DMA interrupt enables in USBD_DMACFG.
3. Program the DMA master interface with the USB module's base address (USBD_DMABLO, USBD_DMABHI) in system memory and enable the DMA.
4. Program the endpoint registers for endpoint 0 in order to receive commands from the USB host:
    a. USBD_EPCFG0 – USBD_MAX = Endpoint's maximum packet size, USBD_DIR = OUT, USBD_ARM = 1.
    b. USBD_EPADR0 = As desired.
    c. USBD_EPLEN0 = As desired. Control endpoints typically specify an 8 byte packet size.
5. Enable the module for USB activity by setting the USBD_ENA bit in the USBD_CTRL register.
6. Wait for a USBD_CFG interrupt, or for a SETUP packet on endpoint 0.

    The most likely sequence of events from the USB is that the device comes out of reset and receives a GET_DESCRIPTOR request from the USB host for the first 8 bytes of the device descriptor, potentially followed by additional GET_DESCRIPTOR requests. At that point, the

USB host assigns an address to the device and then selects a configuration. Optionally, if the USBD_CFG interrupt is masked, software can poll the USBD_GINTR register.

7. Read a new USB configuration number from the USBD_STAT register, or decode and respond to the setup packet (For more information, see "Control Transfers" on page 14-11.).
8. Repeat steps 6 and 7 until a device configuration has been specified.
9. Program endpoint configuration and interrupt registers according to the current device configuration.
10. Initialization is complete.

USB Data Transfers

This section documents the programming of the endpoint configuration registers to support USB data transfers. It includes a number of cases typically encountered, such as Bulk IN/OUT, Isochronous IN/OUT, and Control transfers. Exception cases, such as packet retries, bad device requests, corrupted handshakes, and short transfers, are also discussed.

From the device's viewpoint, USB Interrupt transfers usually behave like Bulk transfers. The information that describes Bulk transfers also applied to USB Interrupt transfers. USB Interrupt transfers are not discussed separately.

USB Transfer Concepts

In general, the USB does not maintain a transfer count for data transfers between the device and the host. For isochronous endpoints, the transfer size is irrelevant, because they get the same bandwidth every frame, whether they need it or not. Control transfers specify the length of their

data phase in the setup packet, but the data phase may terminate early. Bulk and interrupt transfers, however, appear only as a continuous stream of packets to the USB.

To delineate blocks of data, the USB requires that all packets be sent at the maximum size for the endpoint, except for the last packet in a group. For example, if the endpoint's maximum packet size is 8 bytes, and a 63-byte transfer is being performed, the data source would send 7 packets at 8 bytes, and one last packet at 7 bytes. The software on the USB host or the USB device recognizes the shortened data packet as the end of a group of packets. If the data transfer ends on a packet boundary (that is, a 64-byte transfer, 8-byte packet size), the data source sends a last 0-byte packet on the USB.

How to Transfer Data

Preparing a USB endpoint to transfer data consists of several steps:

1. For the first memory block, program the buffer offset and length into the USBD_EPADRx and USBD_EPLENx registers. The offset address is a word-aligned address in the ADSP-BF535 processor's L2 memory.
2. Unmask the USBD_BCSTAT, USBD_PC, and USBD_TC interrupts.
3. Program the USB_EPCFGx register with these fields:
    - Set the USBD_MAX field to the code for the endpoint's maximum packet size.
    - Set USBD_TYP to the code for the endpoint type.
    - Set USBD_DIR to IN or OUT.
    - Set USBD_ARM to 1.
4. Wait for interrupts.

The USBD_PC interrupt can be used by software to monitor the progress of a transfer on a packet-by-packet basis. The USBD_BCSTAT interrupt indicates when the byte count for the transfer has expired, and a new buffer must be set up. The USBD_TC interrupt indicates a completed transfer.

5. For multiple block transfers, repeat steps 1 through 4 until all blocks have been transferred.

   If a bulk IN data transfer ends on a packet boundary, an additional block with a size of 0 bytes must be created to terminate the transfer.

Due to packet scheduling on the USB, resetting the endpoint registers between transfers is a time-critical operation to avoid generating NAKs on the USB.

Bulk Transfers

Bulk data transfers (as well as interrupt and control transfers) are based around a limited set of packet sizes. The USB 1.1 specification defines valid sizes of 8, 16, 32, and 64 bytes. Each USB endpoint is assigned one of these possible basic packet sizes and cannot switch between them.

All large data transfers are broken down into units of the basic packet sizes. For example, a 1044-byte file transfer across the USB to a 64-byte bulk endpoint consists of 17 packet transfers. The first 16 packets are 64 bytes, and the remaining 20 bytes are carried in a short packet. The USB host automatically assumes that the short packet represents an end-of-transfer indicator.

Scheduling individual packet transfers on the USB is non-deterministic from the device's standpoint. It does not know in advance when the host is going to make a request. The best the device can do to ensure maximum bandwidth use of the USB is to be ready for a transfer at all times. This means setting up multiple buffering for the endpoints.

Note that the endpoint configuration registers allow software to create data transfers larger than a single USB packet. The DMA master channel can address a 2 KB buffer space in system memory. Software must manage the space as a pool of buffers. By using the endpoint configuration registers, it can create transfers from 1 to 1023 bytes at various locations in the 2 KB buffer space. This can be a single 1023-byte isochronous endpoint buffer or many 16-byte bulk endpoint buffers.

Bulk In

Assume 64-byte packet size, packet buffer located at address 0x80 in the L2 memory space, and buffer size 0x80 (two packets). Programming flow is:

1. Clear interrupt registers: USBD_EPINTRx and the USBD_EPxINTR bit of the USBD_GINTR register.
2. Unmask interrupts:
   - USBD_MASKx: USBD_BCSTAT = 0, USBD_TC = 0.
   - USBD_GMASK: USBD_EPxMSK = 0.
3. Configure the endpoint:
   - USBD_EPCFGx: USBD_TYP = 01, USBD_DIR = 1, USBD_MAX = 11, USBD_ARM = 1.
   - USBD_EPADRx: USBD_OFFSET = 0x0080.
   - USBD_EPLEN0: USBD_BC = 0x0080.

4. Wait for interrupts.
    - If USBD_PC interrupt:
    - Software can monitor transfer progress on a packet-by-packet basis. After each packet transfers, software must clear the USBD_PC and USBD_EPxINTR interrupts. Go back to step 4.
    - If USBD_BCSTAT interrupt and not USBD_TC interrupt:
    - If more data is to be sent, clear interrupts and go back to step 2. Update USBD_EPADRx and USBD_EPLENx to point to the next buffer of data, then set USBD_ARM = 1 to re-arm the endpoint.
    - If no more data is to be sent, clear interrupts, program USBD_BC = 0, and set UDBD_ARM = 1. Wait for interrupts.
    - If USBD_TC interrupt, the transfer is complete.

Bulk Out

Assume 16-byte packet size, packet buffer located at address 0x80 in the L2 memory space, and buffer size 0x80 (four packets). Programming flow is:

1. Clear interrupt registers: USBD_EPINTRx and the USBD_EPxINTR bit of the USBD_GINTR register.
2. Unmask interrupts:
    - USBD_MASKx: USBD_BCSTAT = 0, USBD_TC = 0, USBD_PC = 0.
    - USBD_GMASK: USBD_EPxMSK = 0.

3. Configure the endpoint:
    - USBD_EPCFGx:
    - USBD_TYP = 01, USBD_DIR = 0, USBD_MAX = 01, USBD_ARM = 1.
    - USBD_EPADRx:
    - USBD_OFFSET = 0x0080.
    - USBD_EPLEN0:
    - USBD_BC = 0x0080.
4. Wait for USBD_BCSTAT/USBD_PC/USBD_TC interrupt.
    - If USBD_PC:
    - Software can monitor transfer progress on a packet by packet basis. After each packet transfers, software must clear the USBD_PC and USBD_EPxINTR interrupts.
    - If USBD_BCSTAT and not USBD_TC:
    - Clear interrupts, then go back to step 2. Update USBD_EPADRx and USBD_EPLENx to point to a new data buffer and then set USBD_ARM = 1 to re-arm the endpoint.
    - If USBD_TC:
    - The transfer is complete. Total length of transfer is equal to the number of complete buffers transferred multiplied by the buffer size, plus the buffer size minus the value remaining in the USBD_EPLENx register at the end of the transfer.
5. Clear the endpoint interrupts, take the steps required to complete the transfer in software, then go back to step 2 and update the registers for a new data transfer.

Isochronous Transfers

Isochronous endpoints are guaranteed exactly one packet transfer per USB frame. The maximum size of the packet is specified in the endpoint's descriptors (software), and in the EpBufs (hardware). In any given frame, the USB may transfer from 0 to the maximum number of bytes allocated for the endpoint.

The specific timing of the packet within a frame is dynamic based on the USB traffic load. In the worst-case scenario, a device can get an isochronous packet request at the end of frame X, and then another immediately at the beginning of frame X+1. For this reason, isochronous applications should consider prebuffering an entire packet one frame in advance of when it is needed.

For USB IN endpoints, prebuffering an entire packet involves setting up the transfer for the next frame as soon as the USBD_TC interrupt is received for the current frame. For USB OUT endpoints, this translates to putting a buffer in place as soon as the USBD_TC interrupt is received for the current frame.

For USB OUT endpoints, the device software should program the byte count to the exact size of the maximum packet for the endpoint (1 to 1023 bytes). For USB IN transfers, the device software should program the byte count to match the amount of data to be transferred in a given frame (the maximum packet size or less).

If the USB device does not have any data to transfer on a particular frame, it should disarm the endpoint by clearing the USBD_ARM bit in the USBD_EPCFGx register.

To monitor the progress of transfers, the device software monitors the USBD_SOF interrupt, and it can look at the status fields of the USBD_STAT register to see exactly what is occurring on the bus at any given time.

Specific examples of ISO IN and ISO OUT register programming are shown in the following sections.

Iso In

Assume 128-byte packet size, packet buffer located at address 0x80 in the L2 memory space, and buffer size 0x80 (one packet). Programming flow is:

1. Clear all interrupts: Write USBD_EPINTRx = all ones. Write USBD_EPxINTR bit of USBD_GINTR register = 1.
2. Unmask interrupts:
   - USBD_MASKx: USBD_BCSTAT = 0, USBD_PC = 0, USBD_TC = 0.
   - USBD_GMASK: USBD_EPxMSK = 0.
3. Configure the endpoint:
   - USBD_EPCFGx:
   - USBD_TYP = 11, USBD_DIR = 1, USBD_MAX = XX, USBD_ARM = 1.
   - USBD_EPADRx:
   - USBD_OFFSET = 0x0080.
   - USBD_EPLEN0:
   - USBD_BC = 0x0080.

4. Wait for interrupts.
    - If USBD_TC interrupt and USBD_PC interrupt, then the packet transfer was error free.
    - If USBD_TC interrupt and not USBD_PC interrupt, then the packet transfer had a problem of some kind. The specific problem must be handled by the programmer. Software must determinate whether this is an error or merely a warning.
5. Go back to step 2 to prepare for the next frame's data transfer.

Iso Out

Assume 1023-byte packet size, packet buffer located at address 0x80 in the memory space, and buffer size 0x3FF (one packet). Programming flow is:

1. Unmask interrupts:
    - USBD_MASKx: USBD_BCSTAT = 0, USBD_PC = 0, USBD_TC = 0.
    - USBD_GMASK: USBD_EPxMSK = 0.
2. Configure the endpoint:
    - USBD_EPCFGx:
    - USBD_TYP = 11, USBD_DIR = 0, USBD_MAX = XX, USBD_ARM = 1.
    - USBD_EPADRx:
    - USBD_OFFSET = 0x0080.
    - USBD_EPLEN0:
    - USBD_BC = 0x03FF.

3. Wait for interrupts.
    - If USBD_TC interrupt and USBD_PC interrupt, then the packet transfer was error free.
    - If USBD_TC interrupt and not USBD_PC interrupt, then the packet transfer had a problem of some kind. The specific problem must be handled by the programmer. Software must determine whether this is an error or merely a warning.
4. Number of bytes received is equal to the buffer size minus the value remaining in the USBD_EPLENx register when the USBD_TC interrupt is received. Take the steps required to complete the transfer in software, then go back to step 2 and update the registers accordingly. Note that no error retries are performed.

Control Transfers

In general, Control endpoints (usually endpoint 0) operate like Bulk endpoints. However, Control endpoints are bidirectional and follow a special protocol that allows the USB host to send commands to the USB device. Control transfers consist of three phases: a setup phase, a data phase, and a status phase.

During the setup phase, the USB host sends an 8-byte setup packet to the USB device. The format and contents of the packet are defined in the USB specification.

The setup packet specifies information such as the type of command (standard request, vendor-specific, device-class-specific), the direction of the data phase (if any), the size of the data phase (if any), and the specific command code.

After receiving the command, the device can choose to reject the command by stalling the endpoint, or it can process the command and proceed through the data and status phases.

The following sections show example flows for control transfers without a data phase and with a data phase.

Control Transfer, No Data Phase

A control transfer with no data phase consists of these steps:

1. At initialization time, device software unmasks the USBD_SETUP interrupt on the control endpoint(s).
2. Device initializes the USBD_EPADRx and USBD_EPLENx registers to point to storage buffers.
3. Device initializes the USBD_EPCFGx registers for OUT direction, USBD_TYP = 00, USBD_DIR = 0 (OUT direction). The USBD_MAX field is programmed to match the endpoint's maximum packet size. The USBD_ARM bit is set to 1.
4. Monitor the USBD_SETUP interrupt.
5. On receipt of a USBD_SETUP interrupt, read the setup packet from the memory buffer and decode the command. If the command is valid, execute it.
6. To complete the command, a status packet must be returned to the USB host. For a transfer with no data phase, this is an IN transfer of 0 bytes, or a STALL handshake. If the command completes successfully, set the endpoint registers (USBD_EPCFGx, USBD_EPLENx) for a 0 byte IN transfer, and set the USBD_ARM bit to 1. If the command does not complete successfully, set the USBD_EPxSTALL bit in the USBD_CTRL register.

Control Transfer With Data Phase

A control transfer can include a data phase between the setup and status phases. The data phase can run from 0 to 65,535 bytes, in either IN or OUT direction. The flow is:

1. At initialization time, device software unmasks the USBD_SETUP interrupt on the control endpoint(s).
2. Device initializes the USBD_EPADRx and USBD_EPLENx registers to point to a storage buffer.
3. Device initializes the USBD_EPCFGx registers for OUT direction, USBD_TYP = 00, USBD_DIR = 0 (OUT direction). The USBD_MAX field is programmed to match the endpoint's maximum packet size. The USBD_ARM bit is set to 1.
4. Monitor the USBD_SETUP interrupt.
5. On receipt of a USBD_SETUP interrupt, read the setup packet from the memory buffer and decode the command. If the command is valid, execute it. If the wLength field of the setup packet is nonzero, then a data phase is associated with the transfer. The direction of the transfer is specified by the bmAttributes[7] bit of the setup packet.
6. Program the endpoint registers to execute a data transfer as in the BULK IN example (see "Bulk Data Transfers" on page 14-11).

Under no circumstances should software attempt to transfer more bytes than specified in the wLength field of the setup packet. It can transfer fewer bytes than specified in the wLength field.

7. To complete the command, a status packet must be returned to the USB host. For a transfer with a data phase, this is a packet with 0 bytes, or a STALL handshake. The direction of the status phase is *opposite* that of the data transfer. For example, for a control transfer with an OUT data phase, set the endpoint registers

(USBD_EPCFGx, USBD_EPLENx) for a zero-byte IN transfer, and set the USBD_ARM bit to 1. If the command does not complete successfully, set the USBD_EPxSTALL bit in the USBD_CTRL register.

Control Transfers Gone Bad

The USB protocol is usually very reliable about ensuring that packets reach their intended destination. However, in one case the packets can be confused. The particular corner case occurs when the USB host sends a command to a USB device.

The host sends a setup packet to the device, and the device acknowledges it with an ACK handshake. If the ACK handshake is corrupted, the USB host waits a while, then tries to resend the setup packet. In this case, the device can wind up with 2 (or more) setup packets in its internal buffer, and must figure out which is the real one before moving on and processing the command.

The USBD module includes several features to help deal with this situation. The features are:

- The UDC correctly handles the situation within its own logic and understands that the first setup packet is invalid.
- The USBD_SETUP and USBD_MSETUP interrupts can be used to determine whether two setup packets await service. An internal lockout mechanism guarantees that if software tries to clear USBD_SETUP and USBD_MSETUP at the same time that another setup packet is received, the software gets another USBD_SETUP interrupt indicating the condition.
- The USBD_SIP and USBD_PIP status bits can be used to determine whether the USB module is in the process of receiving a new setup packet. Software can check these bits before moving to the DATA or STATUS phase of the control transfer to determine whether the command request that it is processing is about to be invalidated.

Exception Handling

Software must handle several exception cases. These exceptions consist of catastrophic endpoint errors, isochronous packets with data errors, and failed memory access requests.

Small Packets (Less Than 16-Byte Transfers)

The DMA master channel is required to address data in minimum bursts of 16 bytes.For this reason, using the USBD module with packet sizes of less than 16 bytes presents a challenge to system software.

To use 16-byte or smaller packets, the USB module pads the data to maintain the 16-byte boundary. It is up to the system software to read the USBD_EPLENx register to determine the valid data.

Endpoint Errors

Under normal circumstances, the hardware handles transfer errors on individual data packets transparently without software intervention. The USB protocol supports packet retries for Bulk, Control, and Interrupt packets.

In cases where a catastrophic problem occurs on a specific endpoint, and the device needs support from the application software (such as a configuration change), the device has the option to stall individual endpoints. When a device stalls an endpoint, the USB host is required to clear the stall condition before further data transfers can occur on the endpoint. Presumably, while clearing the stall condition, the application also fixes the underlying condition which caused the stall in the first place.

The USBD module includes a stall bit for each endpoint in the USBD_CTRL register. When software needs to indicate a stall condition, it sets the USBD_EPxSTALL bit. After the USBD_EPxSTALL bit has been set, application behavior is product specific. The application must clear the USBD_EPxSTALL bit in the USBD_CTRL register.

Isochronous Transfers Error Detection

While Control, Interrupt, and Bulk data transfers support a method for error correction, Isochronous transfers have only error detection capability. Occasionally, the device can expect that an entire packet is lost, but in most cases, the device has to deal with CRC errors or bitstuff violations. In these cases, the entire packet transfers, but the application is responsible for determining how to handle the error.

Under normal circumstances, the USBD module generates both end-of-packet and end-of-transfer interrupts (`USBD_PC`, `USBD_TC`) for isochronous packets.

If an error occurs (for example, CRC error, or bitstuff error), the module still asserts the `USBD_TC`, but it does not assert the `USBD_PC`. In this way, the application can detect and compensate for errors.

Reset Signaling Detected on USB

USB reset signaling is generated by the USB host. The Host uses reset signaling when it needs to re-enumerate devices on the bus. This can occur just after power-up, when devices are initially connected to a hub port, and any time the USB host runs into a catastrophic failure situation.

The USBD module reports the presence of USB reset signaling to the system through the `USBD_RSTSIG` bit of the `USBD_STAT` register and by means of the `USBD_RST` interrupt of the `USBD_GINTR` register.

When the system detects one of these cases, it should terminate any activities currently in progress and prepare for the device to be re-enumerated by the USB host. The system does not need to re-download the UDC endpoint buffer data, nor does it need to reinitialize the device.

Suspend/Resume Considerations

During the time that the USB is suspended (USBD_SUSPEND output active), the system may stop all clocks to the USBD module. If the USB device is powered by the bus, it may be required to stop all clocks to meet the current requirements during the suspended state.

On the inactive edge of USBD_SUSPEND, the module must immediately restart its clocks and guarantee stability within 10 ms (USB specification, section 9.2.6.2).

Note: The 10 ms window is cited in the USB specification and is supposedly guaranteed by the system software on the USB host. Because no USB compliance tests exist for USB hosts, the 10 ms window cannot be guaranteed across all operating system/BIOS combinations. The system designer is urged to restart the clocks as quickly as possible.

More information about USB power distribution is given in Chapter 7 of the USB specification.

References

These reference materials may be of interest to the system integrator:

- *Universal Serial Bus System Architecture*, Don Anderson, Mindshare Inc.
- *Universal Serial Bus Specification*, Rev 1.1, USB Implementers Forum, www.usb.org. Available online.
- *Open Host Controller Interface Specification*, Rev 1.0, USB Implementers Forum, www.usb.org. Available online.

This additional resources is readily available for USB product developers:

- *USB Device Class Specifications*, www.usb.org. The common class specifications provide a framework in which device developers can make use of standardized device drivers on the USB host. Examples are printers, disk drives, and broadband modems.

15 PROGRAMMABLE FLAGS

The ADSP-BF535 processor supports sixteen 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 an output, the state written to the Flag Set (FIO_FLAG_S) and Flag Clear (FIO_FLAG_C) registers determines the state driven by the output PFx pin. Reading either the Flag Set or Flag Clear register returns the state of each pin, regardless of whether each individual pin is configured as an input or an output.

Each PFx pin can be further 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). When 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 ADSP-BF535 processor provides two independent interrupt channels for the PFx pins. Identical in functionality, these are called Interrupt A and Interrupt B. Each interrupt has two mask registers associated with it, a Flag Interrupt Mask Set register (FIO_MASKx_S) and a Flag Interrupt Mask Clear register (FIO_MASKx_C). This flexible mechanism allows each bit to generate Flag Interrupt A, Flag Interrupt B, both Flag Interrupts A and B, or neither.

Each PFx pin is represented by a bit in each of the four 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 PFx pins are multiplexed for use by the Serial Port Interface (SPI) and Phase Locked Loop (PLL) circuitry. Refer to the "SPI" and the "Dynamic Power Management" chapters for more information.

Programmable Flag Memory-Mapped Registers (MMRs)

The programmable flag (PFx) 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.

Flag Direction Register (FIO_DIR)

The Flag Direction register, shown in Figure 15-1, is a read-write register. Each bit position corresponds to a PFx pin. A logic 1 configures a PFx pin as an output, and a logic 0 configures a PFx pin as an input. The reset value of this register is 0x0000, making all PFx pins inputs upon reset.

Flag Set (FIO_FLAG_S) and Flag Clear (FIO_FLAG_C) Registers

The Flag Set and Flag Clear registers, shown in Figure 15-2 and Figure 15-3, are used to sense the value of the PFx pins defined as inputs, to set the state of PFx pins defined as outputs, and to clear interrupts generated by the PFx pins. Each PFx pin is represented by a bit in each of these registers.

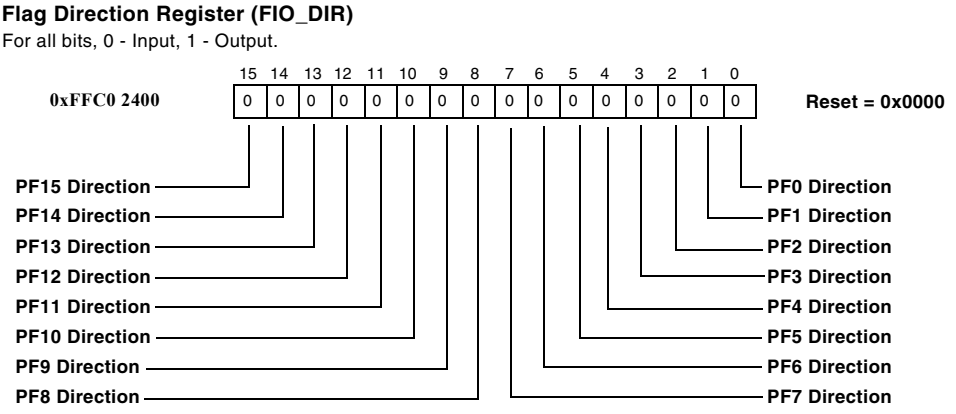


Figure 15-1. Flag Direction Register

Reading either the Flag Set or Flag Clear 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.

The Flag Set register is a write-1-to-set register, while the Flag Clear register is a write-1-to-clear register. These registers are used to set or clear the output state associated with each output PFx pin, and to set or clear the latched interrupt state captured from each input PFx pin. This mechanism allows for more straightforward coding and is less prone to bit manipulation errors than traditional read-modify-write mechanisms.

As an example, assume 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. Reading either register shows 0s for those PFx pins defined as outputs and driven low, 1s for those pins (including PF[0]) defined as outputs and driven high, and the present sense of those 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.

Flag Set Register (FIO_FLAG_S)

Write-1-to-set.

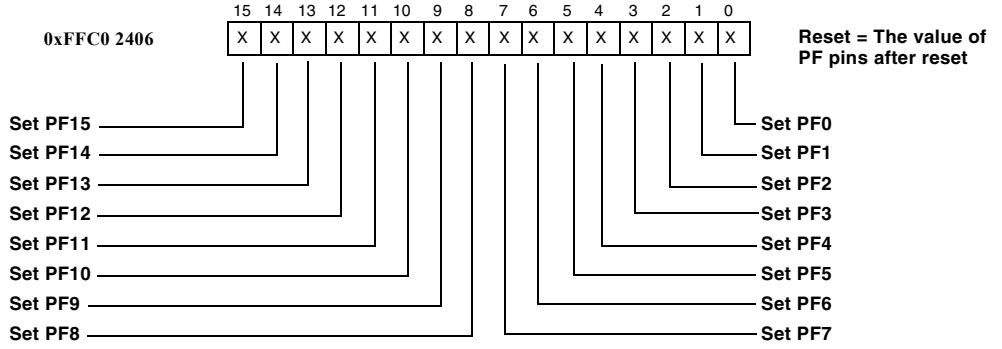


Figure 15-2. Flag Set Register

Flag Clear Register (FIO_FLAG_C)

Write-1-to-clear.

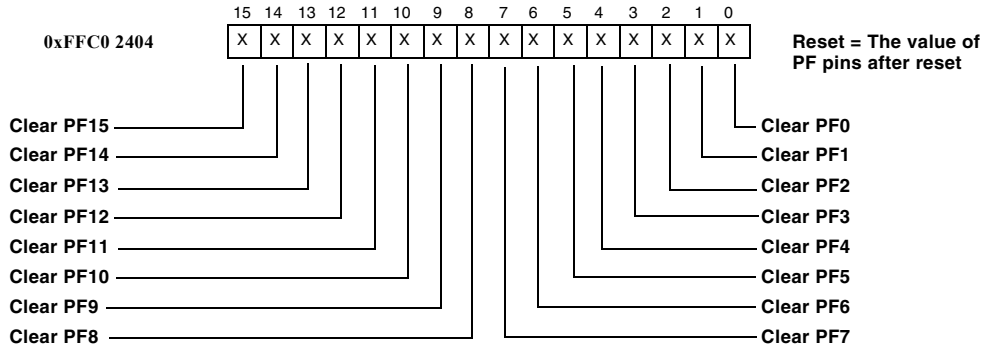


Figure 15-3. Flag Clear Register

Flag Interrupt Mask Registers (FIO_MASKA_C, FIO_MASKA_S, FIO_MASKB_C, FIO_MASKB_S)

Like the Flag Set and Flag Clear registers, the Flag Interrupt Mask registers are implemented as complementary pairs of write-1-to-set and write-1-to-clear 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.

Flag Interrupt A and Flag Interrupt B are each supported by a dedicated Flag Interrupt Mask Set register and a Flag Interrupt Mask Clear register (see Figure 15-4, Figure 15-5, Figure 15-6, and Figure 15-7). Each PFx pin is represented by a bit in each of the four 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.

Interrupt A and Interrupt B operate independently. For example, writing a 1 to a bit in the Flag Interrupt A Mask 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.

When using either rising or falling edge-triggered interrupts, the interrupt condition must be cleared each time a corresponding interrupt is serviced by writing a 1 to the appropriate FIO_FLAG_C bit.

Flag Interrupt A Mask Set Register (FIO_MASKA_S)

For all bits, 1 - Enable.

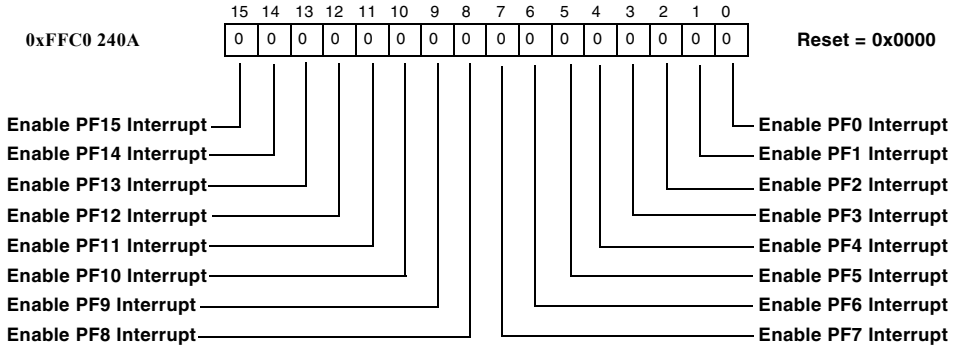


Figure 15-4. Flag Interrupt A Mask Set Register

Flag Interrupt A Mask Clear Register (FIO_MASKA_C)

For all bits, 1 - Disable.

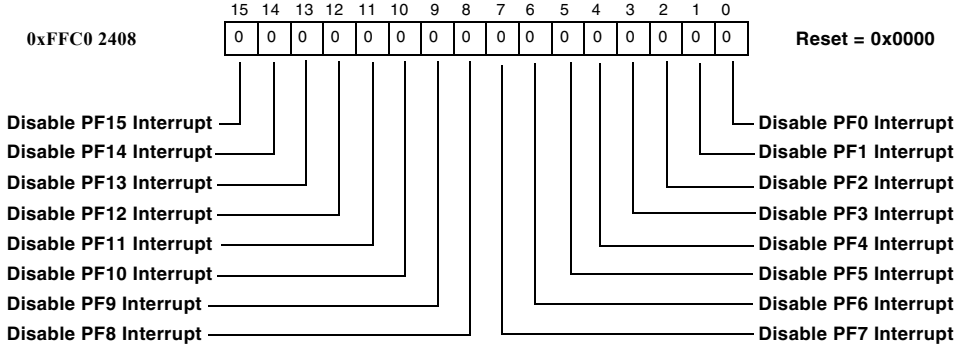


Figure 15-5. Flag Interrupt A Mask Clear Register

Flag Interrupt B Mask Set Register (FIO_MASKB_S)

For all bits, 1 - Enable.

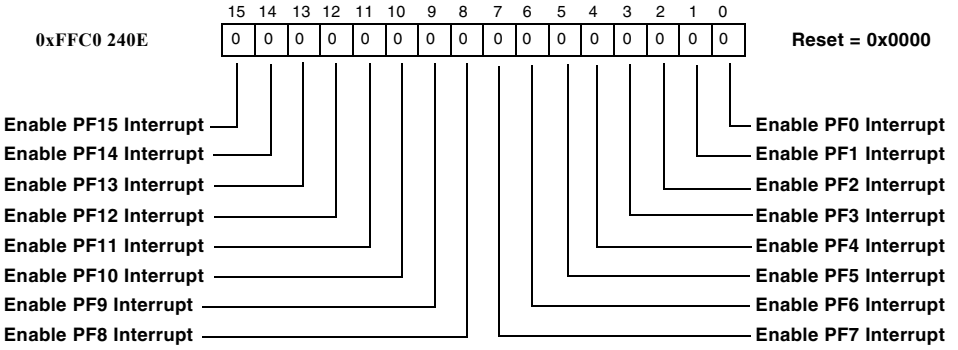


Figure 15-6. Flag Interrupt B Mask Set Register

Flag Interrupt B Mask Clear Register (FIO_MASKB_C)

For all bits, 1 - Disable.

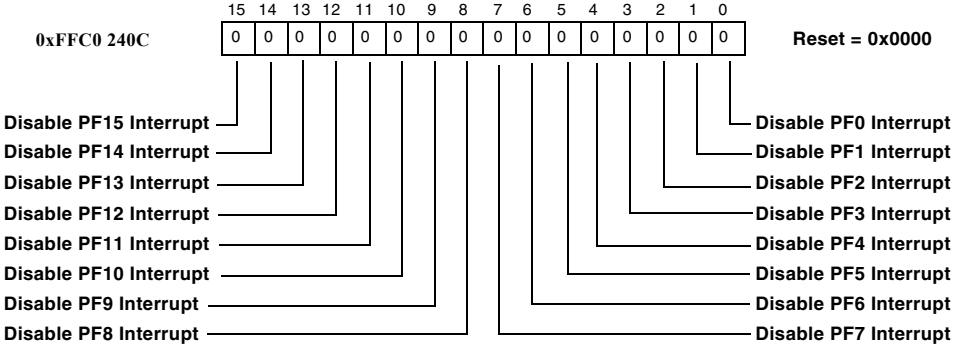


Figure 15-7. Flag Interrupt B Mask Clear Register

Figure 15-8 shows the process by which a Flag Interrupt A or a Flag Interrupt B event can be generated. Note that 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]. At reset, all interrupts are masked.

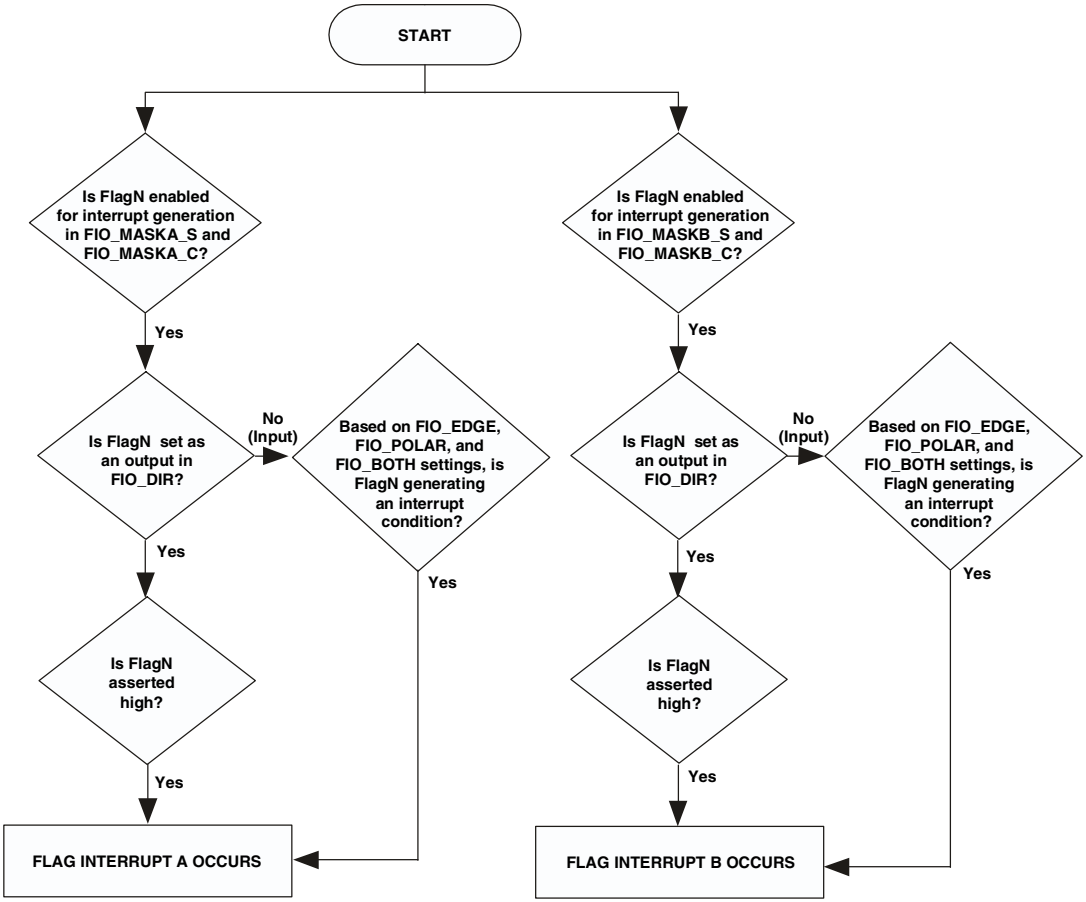


Figure 15-8. Flag Interrupt Generation Flow

Flag Polarity Register (FIO_POLAR)

The Flag Polarity register, shown in Figure 15-9, 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.

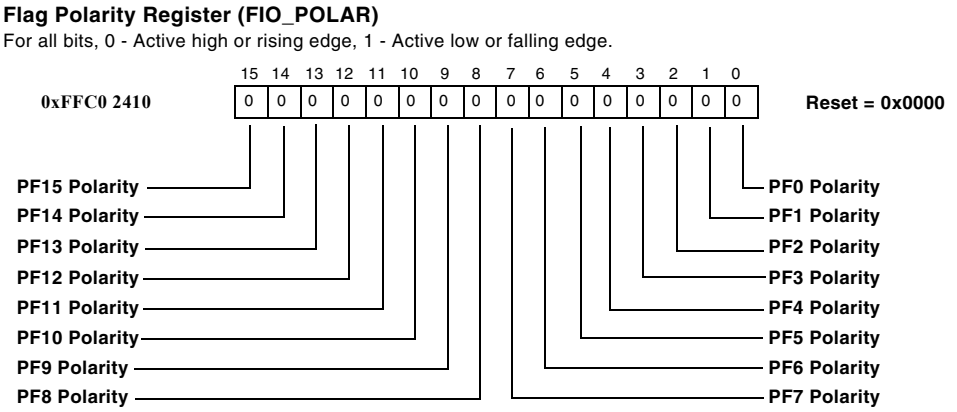


Figure 15-9. Flag Polarity Register

Flag Interrupt Sensitivity Register (FIO_EDGE)

The Flag Interrupt Sensitivity register, shown in Figure 15-10, is used to configure each of the flags as either a level sensitive or an edge sensitive source.

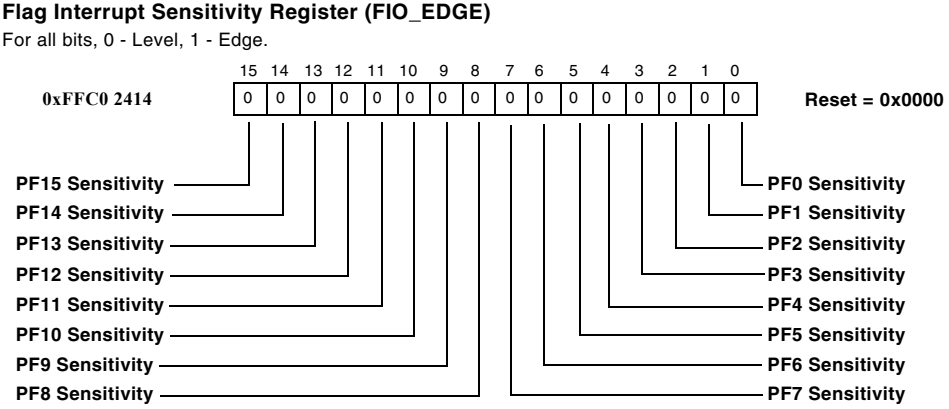


Figure 15-10. Flag Interrupt Sensitivity Register

When using an edge sensitive mode, an edge detection circuit is used to prevent a situation where a short event is missed due to the system clock rate.

The contents of this register are cleared at reset, defaulting to level sensitivity.

Flag Set on Both Edges Register (FIO_BOTH)

The Flag Set on Both Edges register, shown in Figure 15-11, is used to enable interrupt generation on both rising and falling edges.

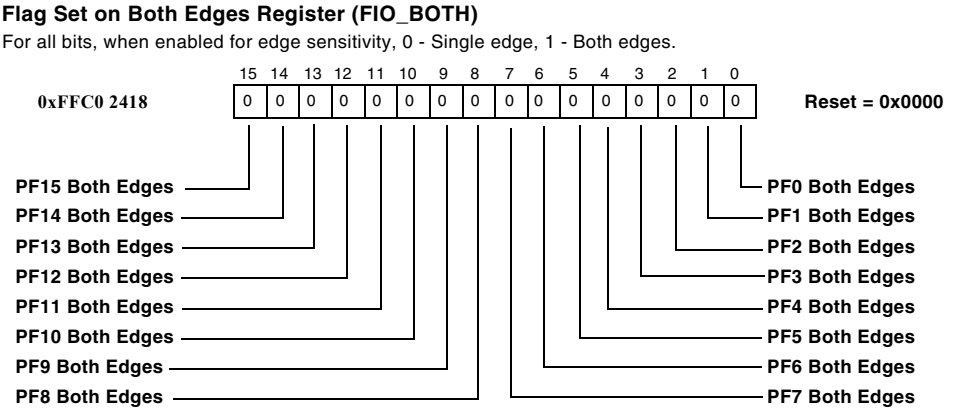


Figure 15-11. Flag Set on Both Edges Register

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.

Performance/Throughput

The PFx pins are synchronized to the system clock (SCLK). When configured as outputs, the programmable flags can transition once every 4 system clock cycles.

When configured as inputs, the overall system design should take into account the potential latency between the core and the system clock. 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.

16 TIMERS

The ADSP-BF535 processor features three general-purpose timers, a core timer, and a watchdog timer.

The general-purpose timers include five registers and can be configured in any of these three modes:

- Pulse Width Modulation mode (PWM_OUT)
- Pulse Width Count and Capture mode (WDTH_CAP)
- External Event Counter mode (EXT_CLK)

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 of the three general-purpose timers has one dedicated bidirectional pin, TMRx. This pin functions as an output pin in PWM_OUT mode and as an input pin in WDTH_CAP and EXT_CLK modes. When clocked internally, the clock source is the ADSP-BF535 processor's system clock (SCLK).

General-Purpose Timer Registers

Each general-purpose timer uses these 16-bit registers:

- Timer Status (TIMERx_STATUS)
- Timer Configuration (TIMERx_CONFIG)
- Timer Counter (TIMERx_COUNTER_LO/HI)
- Timer Period (TIMERx_PERIOD_LO/HI)
- Timer Width (TIMERx_WIDTH_LO/HI)

Timer registers are found in the memory range 0xFFC0 2000–0xFFC0 23FF and are shown in Table 16-1.

For the sake of readability, this chapter follows the notation specified in Table 16-2, which removes the LO/HI identifier from the TIMERx_COUNTER, TIMERx_PERIOD, and TIMERx_WIDTH registers, referring to each of these MMRs as a 32-bit entity. Table 16-1 shows the true register mappings that should be used in user code, as all timer registers are 16 bits wide.

Always access 16-bit timer registers using 16-bit word read or write commands.

Table 16-1. Timer Registers

Register Name	Memory-Mapped Address	Description
TIMER0_STATUS	0xFFC0 2000	Timer0 Status register
TIMER0_CONFIG	0xFFC0 2002	Timer0 Configuration register
TIMER0_COUNTER_LO	0xFFC0 2004	Timer0 Counter register, Low Word
TIMER0_COUNTER_HI	0xFFC0 2006	Timer0 Counter register, High Word
TIMER0_PERIOD_LO	0xFFC0 2008	Timer0 Period register, Low Word
TIMER0_PERIOD_HI	0xFFC0 200A	Timer0 Period register, High Word

Table 16-1. Timer Registers (Cont'd)

Register Name	Memory-Mapped Address	Description
TIMER0_WIDTH_LO	0xFFC0 200C	Timer0 Width register, Low Word
TIMER0_WIDTH_HI	0xFFC0 200E	Timer0 Width register, High Word
TIMER1_STATUS	0xFFC0 2010	Timer1 Status register
TIMER1_CONFIG	0xFFC0 2012	Timer1 Configuration register
TIMER1_COUNTER_LO	0xFFC0 2014	Timer1 Counter register, Low Word
TIMER1_COUNTER_HI	0xFFC0 2016	Timer1 Counter register, High Word
TIMER1_PERIOD_LO	0xFFC0 2018	Timer1 Period register, Low Word
TIMER1_PERIOD_HI	0xFFC0 201A	Timer1 Period register, High Word
TIMER1_WIDTH_LO	0xFFC0 201C	Timer1 Width register, Low Word
TIMER1_WIDTH_HI	0xFFC0 201E	Timer1 Width register, High Word
TIMER2_STATUS	0xFFC0 2020	Timer2 Status register
TIMER2_CONFIG	0xFFC0 2022	Timer2 Configuration register
TIMER2_COUNTER_LO	0xFFC0 2024	Timer2 Counter register, Low Word
TIMER2_COUNTER_HI	0xFFC0 2026	Timer2 Counter register, High Word
TIMER2_PERIOD_LO	0xFFC0 2028	Timer2 Period register, Low Word
TIMER2_PERIOD_HI	0xFFC0 202A	Timer2 Period register, High Word
TIMER2_WIDTH_LO	0xFFC0 202C	Timer2 Width register, Low Word
TIMER2_WIDTH_HI	0xFFC0 202E	Timer2 Width register, High Word

Table 16-2. Abbreviated MMR References from Table 16-1

Register Name	Memory-Mapped Address	Description
TIMER0_COUNTER	0xFFC0 2004	Timer0 Counter register
TIMER0_PERIOD	0xFFC0 2008	Timer0 Period register
TIMER0_WIDTH	0xFFC0 200C	Timer0 Width register
TIMER1_COUNTER	0xFFC0 2014	Timer1 Counter register

Table 16-2. Abbreviated MMR References from Table 16-1 (Cont'd)

Register Name	Memory-Mapped Address	Description
TIMER1_PERIOD	0xFFC0 2018	Timer1 Period register
TIMER1_WIDTH	0xFFC0 201C	Timer1 Width register
TIMER2_COUNTER	0xFFC0 2024	Timer2 Counter register
TIMER2_PERIOD	0xFFC0 2028	Timer2 Period register
TIMER2_WIDTH	0xFFC0 202C	Timer2 Width register

Timer Status Registers (TIMERx_STATUS)

Each Timer Status register (TIMERx_STATUS) indicates the status of all three timers and can be used to check the status of all three timers with a single read. Each TIMERx_STATUS register contains timer-enable bits that enable the corresponding timer (for example, TIMER0_STATUS enables TIMER0). Within a timer's TIMERx_STATUS register, that timer has a pair of sticky status bits that require a write-1-to-set (TIMENx) or write-1-to-clear (TIMDISx) to either enable or disable the timer.

While the timer is enabled, both its TIMENx and TIMDISx bits read 1. The timer starts counting three peripheral clock cycles after the TIMENx bit is set. Setting the timer TIMDISx bit stops the timer without waiting for any additional event.

Each TIMERx_STATUS register also contains a Timer Interrupt bit (IRQx) and a Timer Overflow bit (OVF_ERRx) for each timer. These sticky bits are set by the timer hardware and may be monitored by software. They need to be cleared in TIMERx_STATUS for each timer explicitly. To clear, write a 1 to the bit.

Interrupt and overflow bits may be cleared at the same time as timer disable bits.

To enable an individual timer, set the timer TIMENx bit in the TIMERx_STATUS register for that timer. To disable an individual timer, set the timer TIMDISx bit in the TIMERx_STATUS register for that timer.

For bits descriptions, see Figure 16-1, Figure 16-2, and Figure 16-3.

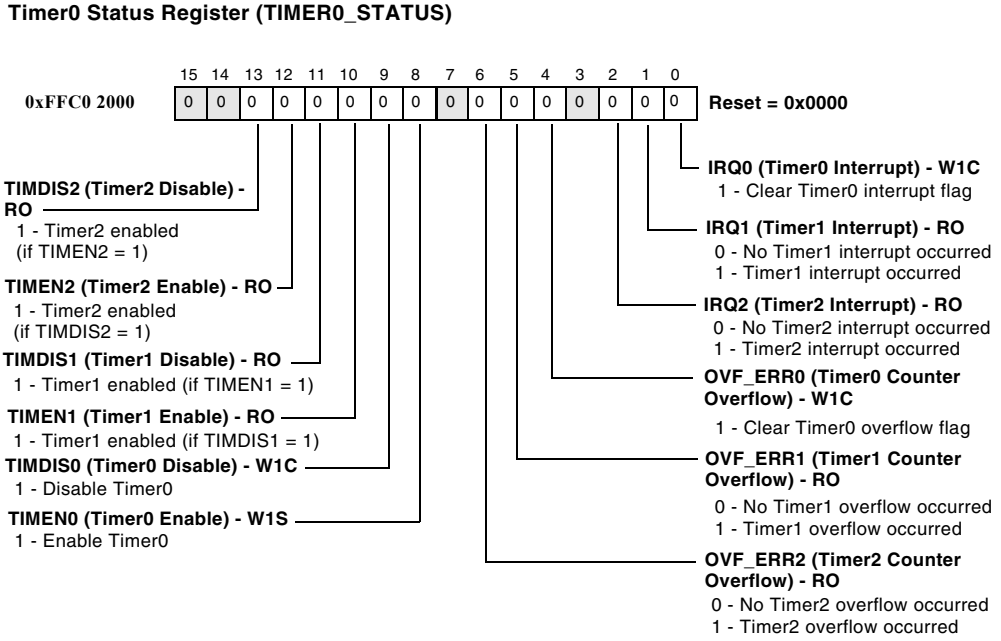


Figure 16-1. Timer0 Status Register

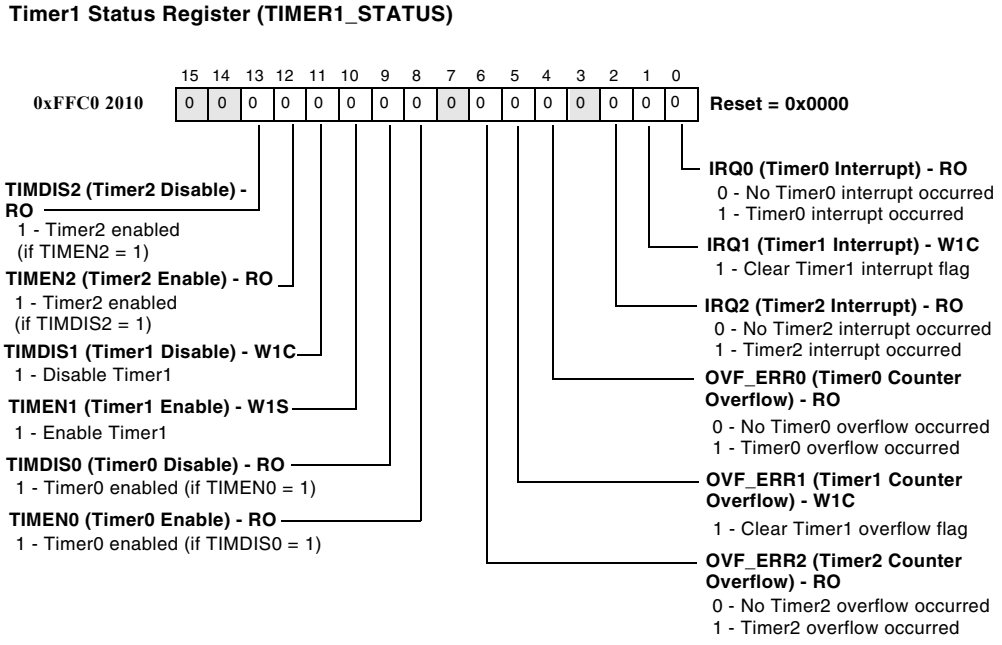


Figure 16-2. Timer1 Status Register

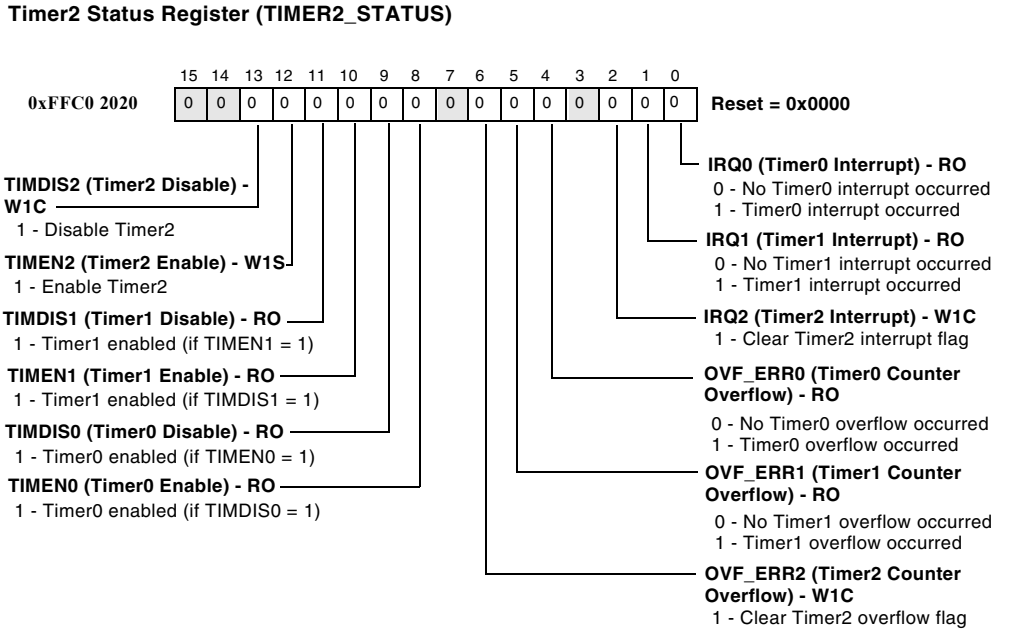


Figure 16-3. Timer2 Status Register

Timer Configuration Registers (TIMERx_CONFIG)

To enable timer interrupts, set the IRQ_ENA bit in the Timer Configuration register (TIMERx_CONFIG) and unmask the timer interrupt by setting the corresponding bit of the System Interrupt Mask register (SIC_IMASK). If the IRQ_ENA bit is cleared, a timer does not set its IRQx bits. To poll the IRQx bits without permitting a timer interrupt, set the IRQ_ENA bit while leaving the SIC_IMASK timer interrupt masked off.

If interrupts are enabled, make sure that the interrupt service routine (ISR) clears the IRQx 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 IRQx clear command from the RTI instruction, an

extra SSYNC instruction may be inserted. In External Event Counter mode (EXT_CLK), reset the IRQx bit in the TIMERx_STATUS register at the very beginning of the ISR to avoid missing any timer events.

Before enabling a timer, always program the corresponding Timer Configuration register (TIMERx_CONFIG). 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.

Figure 16-4 describes the Timer Configuration registers.

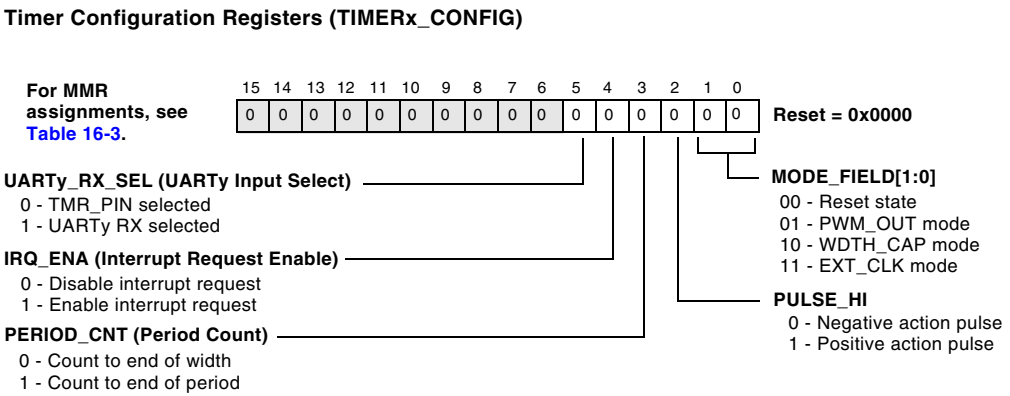


Figure 16-4. Timer Configuration Registers

The UARTy_RX_SEL bit applies to WDTH_CAP mode only. It is intended for UART autobaud detection. If this bit is set, the UARTy RX pin is captured instead of the TMRx pin, Timer 0 alternatively captures UART0, Timer 1 alternatively captures UART1, and Timer 2 alternatively captures UART1.

Table 16-3. Timer Configuration Register MMR Assignments

Register Name	Memory-Mapped Address
TIMER0_CONFIG	0xFFC0 2002
TIMER1_CONFIG	0xFFC0 2012
TIMER2_CONFIG	0xFFC0 2022

Figure 16-5 shows the timing for enabling and disabling the timers.

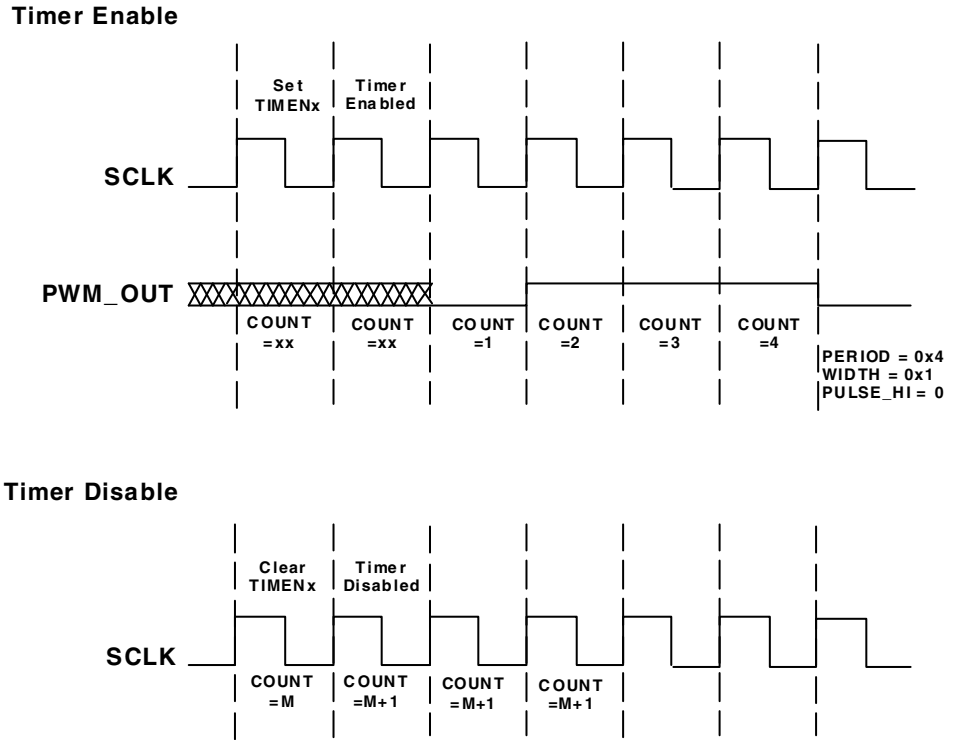


Figure 16-5. Timer Enable and Disable Timing

Timer Period Registers (TIMERx_PERIOD)

The Period registers, shown in Figure 16-6, perform these functions in the three timer modes:

- PWM_OUT: In conjunction with the Timer Width registers, these registers define output waveform characteristics.
- WDTH_CAP: These registers are read-only registers and contain the captured period value.
- EXT_CLK: These registers are write-only registers and contain the maximum timer external count.

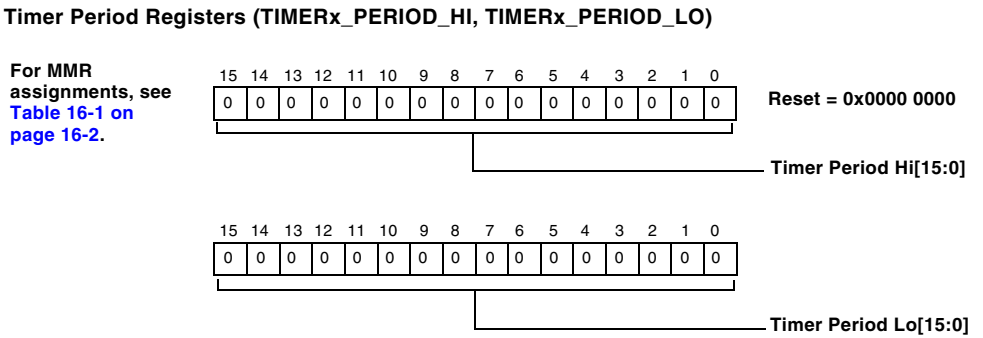


Figure 16-6. Timer Period Registers

Timer Width Registers (TIMERx_WIDTH)

The Timer Width registers, shown in Figure 16-7, perform these functions in the three timer modes:

- PWM_OUT: In conjunction with the Timer Period registers, these registers define output waveform characteristics.
- WDTH_CAP: These registers are read-only registers and contain the captured pulse width value.
- EXT_CLK: Not used.

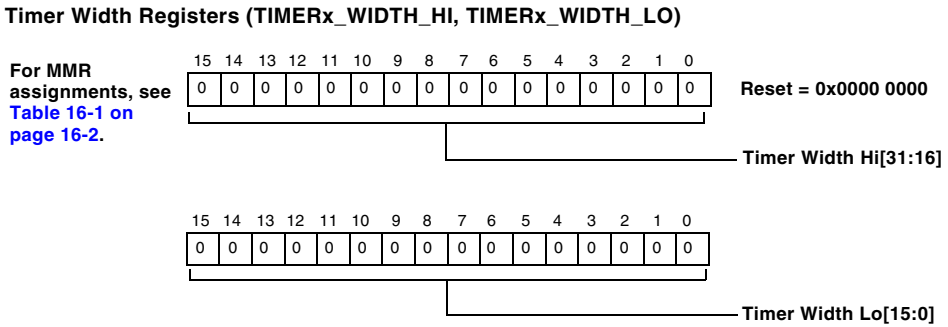


Figure 16-7. Timer Width Registers

Timer Counter Registers (TIMERx_COUNTER)

These read-only registers retain their state when disabled. When enabled, the counter is reinitialized from the Timer Period and Timer Width registers based on the mode selected in `TIMERx_CONFIG`.

The timer counter value cannot be set directly by the software. The counter should be read only when the respective timer is disabled. This prevents erroneous data from being returned.

If it is necessary to read the TIMERx_COUNTER registers while that timer is running, this algorithm must be followed to guarantee valid results:

1. Read TIMERx_COUNTER_HI
2. Read TIMERx_COUNTER_LO
3. Read TIMERx_COUNTER_HI again
4. Compare the two TIMERx_COUNTER_HI values
5. Repeat from step 1 if they are not equal

In External Event Capture mode (EXT_CLK), the TMRx pin (RX1 or RX0 for autobaud detection) is used to clock the timer counter. The counter is initialized with the period value and counts until the period expires. For more information about autobaud detection see "Autobaud Detection" on page 16-19.

Figure 16-8 describes the Timer Counter registers.

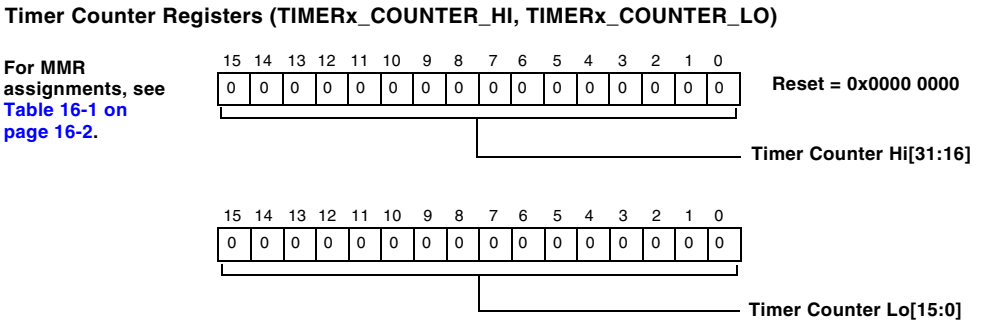


Figure 16-8. Timer Counter Registers

Timer Modes

The TMRx pins function as outputs in PWM_OUT mode and as inputs in WDTH_CAP and EXT_CLK modes. The timer mode is determined by the value of the MODE_FIELD bits in the TIMERx_CONFIG register.

Pulse Width Modulation Mode (PWM_OUT)

Setting the MODE_FIELD bits to 01 in TIMERx_CONFIG enables PWM Output mode (PWM_OUT). In PWM_OUT, the timer TMRx pin is an output.

In PWM_OUT mode, the TMRx pin is always driven low when the timer is disabled, regardless of the state of the PULSE_HI bit. When the timer is running, however, the TMRx pin polarity corresponds to the PULSE_HI bit setting.

The timer is clocked internally by SCLK. Depending on the state of the PERIOD_CNT bit in TIMERx_CONFIG, PWM_OUT either generates pulse width modulation waveforms or generates a single pulse on the TMRx pin.

When the timer is enabled, the timer checks the period and width values for validity (independent of PERIOD_CNT) and does *not* start to count when any of these invalid conditions is true:

- Width = 0
- Period value is lower than width value
- Width = period

The timer module tests these conditions on writes to the TIMERx_WIDTH_LO register. Before writing to TIMERx_WIDTH_LO, make sure that the TIMERx_WIDTH_HI and the TIMERx_PERIOD are set accordingly. On invalid conditions, the timer sets both the IRQx and the OVF_ERRx bits in TIMERx_STATUS after two SCLK cycles. The Timer Counter register is not altered. Note that after reset, the timer registers are all 0.

If period and width values are valid after the timer is enabled, the Timer Counter register is loaded with the value (0xFFFF FFFF – width). The timer counts upward to 0xFFFF FFFE. The timer then reloads the Timer Counter register with the value (0xFFFF FFFF – (period – width)) and repeats.

Pulse Width Modulation (PWM) Waveform Generation

If the PERIOD_CNT bit in the TIMERx_CONFIG register 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 core processing.

The Timer Period and Timer Width registers are programmed with the values of the timer count period and the 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 pulse width expires 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 TIMERx_CONFIG register is used. For a low assertion level, clear this bit. For a high assertion level, set this bit.

Writes to TIMERx_PERIOD_HI, TIMERx_PERIOD_LO, and TIMERx_WIDTH_HI do not become active until TIMERx_WIDTH_LO is written. Therefore, TIMERx_WIDTH_LO must always be written after changing TIMERx_PERIOD. When the TIMERx_WIDTH_LO value is not subject to change, the ISR may read back the current value of TIMERx_WIDTH_LO and write it again.

Listing 16-1. PWM Mode Initialization Sequence

```
/* A typical PWM mode initialization: */
#define lo(const32) (const32 & 0xFFFF)
#define hi(const32) ((const32 >> 16) & 0xFFFF)
P0.H = hi(TIMER0_CONFIG);
P0.L = lo(TIMER0_CONFIG);
R0.L = 0x0019;
W[P0] = R0.L;
/* */
P0.L = lo(TIMER0_PERIOD_HI);
W[P0] = R1.H;
P0.L = lo(TIMER0_PERIOD_LO);
W[P0] = R1.L;
/* */
P0.L = lo(TIMER0_WIDTH_HI);
W[P0] = R2.H;
P0.L = lo(TIMER0_WIDTH_LO);
W[P0] = R2.L;
/* */
P0.L = lo(TIMER0_STATUS);
R0.L = 0x0100;
W[P0] = R0.L;
SSYNC;
```

If enabled, a timer interrupt is generated at the end of each period. An interrupt service routine must clear the interrupt latch bit (IRQx) and might alter the period and/or width values. In PWM applications, the software may need to update period and width values while the timer is running. To guarantee coherency between Timer Period and Timer Width registers, a buffer mechanism is used.

In PWM_OUT, the counter is reloaded at the end of every period as well as at the end of every pulse. The generated waveform depends on whether TIMERx_WIDTH_LO is updated before or after the pulse width expires, due to the reload sequence described in "Pulse Width Modulation Mode (PWM_OUT)" on page 16-13.

Changing the pulse width while the timer is running is typically accomplished by having an interrupt service routine write new values to the width registers. As seen in Figure 16-9, this causes the generation of one incorrect period if the write to the TIMERx_WIDTH_LO register occurs before the ongoing pulse width expires. This is very likely because the interrupt is requested at the end of a period.

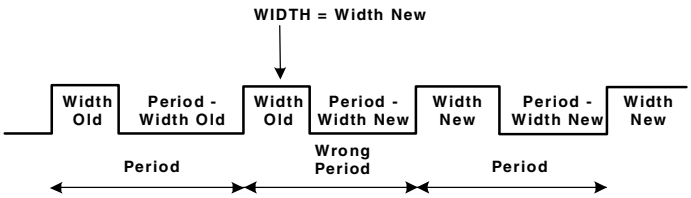


Figure 16-9. Possible Period Failure Due to Width Update with Timer Running

If an application forbids single misaligned PWM patterns, the procedure illustrated in Figure 16-10 can be used. It alters the period value temporarily and restores the original period value the very next PWM cycle in order to obtain constant PWM periods.

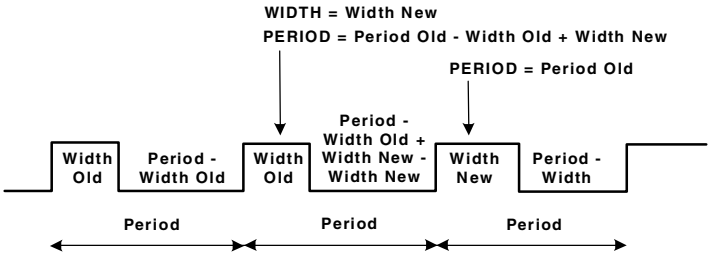


Figure 16-10. Recommended Width Update Procedure

Note the period settings can be altered without similar impacts.

When a timer is running in PWM_OUT mode with the PERIOD_CNT bit set, and it is subsequently stopped, it does not wait for the end of the current period before stopping. Once disabled, the TMRx pin goes low, and the last PWM cycle is not completed.

To generate the maximum frequency on the TMRx output pin, set the TIMERx_PERIOD value to 2 and the TIMERx_WIDTH value to 1. This makes TMRx toggle each SCLK cycle, producing a duty cycle of 50%.

Single Pulse Generation

If the PERIOD_CNT bit in TIMERx_CONFIG is cleared, PWM_OUT 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 Width register and the Timer Period register is not used.

At the end of the pulse the interrupt latch bit (IRQx) is 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.

The active low pulse configuration is not recommended, since the TMRx pin is always driven low when the timer is not running, regardless of the PULSE_HI bit.

Pulse Width Count and Capture Mode (WDTH_CAP)

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 MODE_FIELD bit in TIMERx_CONFIG to b#10 enables this mode. The period and width registers are read-only in WDTH_CAP mode.

When enabled in this mode, the timer resets the value in the TIMERx_COUNTER register to 0x0000 0001 and does not start counting until it detects the leading edge on the TMRx pin.

When the timer detects a first leading edge, it starts incrementing. When it detects the trailing edge of a waveform, the timer captures the current 32-bit value of TIMERx_COUNTER into TIMERx_WIDTH. At the next leading edge, the timer transfers the current 32-bit value of TIMERx_COUNTER into TIMERx_PERIOD. The TIMERx_COUNTER register is reset to 0x0000 0001, and the timer continues counting until it is either disabled or the count value reaches 0xFFFF FFFF.

In WDTH_CAP mode, software can simultaneously 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 TIMERx_CONFIG is set or cleared. If the PULSE_HI bit is cleared, the measurement is initiated by a falling edge, the value in the TIMERx_COUNTER register is captured to the TIMERx_WIDTH register on the rising edge, and the period is captured on the next falling edge.

With IRQ_ENA set, the width registers become sticky in WDTH_CAP mode. Once a pulse width event (trailing edge) has been detected and properly latched, the width registers do not update anymore unless the IRQx bit is cleared by software. The period registers still update every time a leading edge is detected.

The PERIOD_CNT bit in TIMERx_CONFIG controls whether an enabled interrupt is generated when the pulse width or pulse period is captured. If the PERIOD_CNT bit is set, the IRQx bit is set when the pulse period value is captured. If the PERIOD_CNT bit is cleared, the IRQx bit is set when the pulse width value is captured.

If the PERIOD_CNT bit is cleared, the first interrupt is generated before the first period value has been measured, so the period value is not valid. If the ISR reads the period value in this case, the timer returns a period value of 0.

If enabled, a timer interrupt is also generated if the TIMERx_COUNTER register reaches a value of 0xFFFF FFFF. At that point, the timer is disabled automatically, and the OVF_ERRx status bit is set, indicating a count overflow. IRQx and OVF_ERRx are sticky bits, and software must clear them explicitly.

Because of synchronizer latency, insert two NOP instructions between setting WDTH_CAP and setting the TIMENx bit. The timer can be subsequently disabled and re-enabled without additional NOP instructions.

Autobaud Detection

Timer0 provides autobaud detection for UART0, and Timer1 and Timer2 provide autobaud detection for UART1. When UARTx_RX_SEL is set in TIMERx_CONFIG, the timer samples the appropriate UART port receive data pin (RX0 or RX1) instead of the TMRx pin while enabled for WDTH_CAP mode. A software routine can measure the pulse widths of the bits received on the serial data line (RX). Because the timers operate synchronously with the UART operation (all are derived from the phase-locked loop (PLL) clock), the pulse widths can be used to calculate the baud-rate divider for the UART. To determine the UART baud rate, use this equation:

```
DIVISOR = TIMERx_WIDTH/(16 x (Number of captured UART bits))
```

Use the equation above if you are capturing either the pulse width of a single bit or the duration of a NULL-character frame, as shown in Figure 16-11.

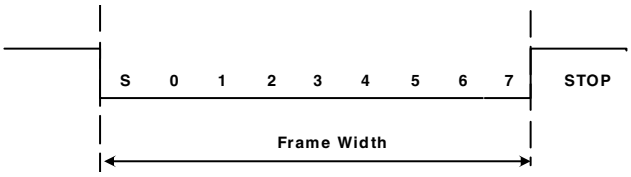


Figure 16-11. Autobaud Detection Character 0x00

Because the example frame in Figure 16-11 encloses 8 data bits and 1 start bit, apply the formula:

```
DIVISOR = TIMERx_WIDTH/(16 × 9)
```

At higher bit rates such pulse-width-based autobaud detection might not return sufficient results without additional analog signal conditioning. Therefore, it is strongly recommended to measure signal periods instead. For example, predefine ASCII character “@” (40h) as an autobaud detection byte and measure the period between the falling edge of the start bit and the falling edge after bit 6, as shown in Figure 16-12. Since this period enclosed 8 bits, apply the formula:

```
DIVISOR = TIMERx_PERIOD/(16 × 8)
```

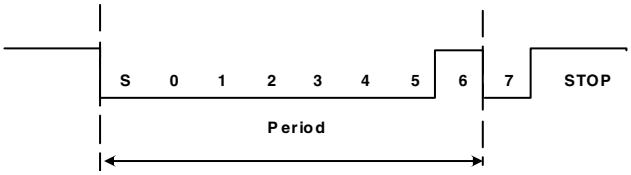


Figure 16-12. Autobaud Detection Character 0x40

External Event Counter Mode (EXT_CLK)

In EXT_CLK mode, the TMRx pin is an input. The timer works as a counter clocked by any external source, which can also be asynchronous to the core clock. Setting the MODE_FIELD bits in TIMERx_CONFIG to b#11 enables this mode. The TIMERx_PERIOD register is programmed with the value of the maximum timer external count.

After the timer is enabled, it waits for the first rising edge on the TMRx pin. This edge forces the Timer Count register to be loaded by the value (0xFFFF FFFF – period). Every subsequent rising edge increments the count register. After reaching the count value 0xFFFF FFFE, the IRQx bit is set and an interrupt is generated. The next rising edge reloads the count register again by (0xFFFF FFFF – period).

The PULSE_HI and the PERIOD_CNT configuration bits have no effect in this mode, and the OVF_ERRx bits are never set. The Timer Width register is not used.

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 MMRs, the Timer Control register (TCNTL), the Timer Count register (TCOUNT), the Timer Period register (TPERIOD), and the Timer Scale register (TSCALE).

Figure 16-13 provides a block diagram of the core timer.

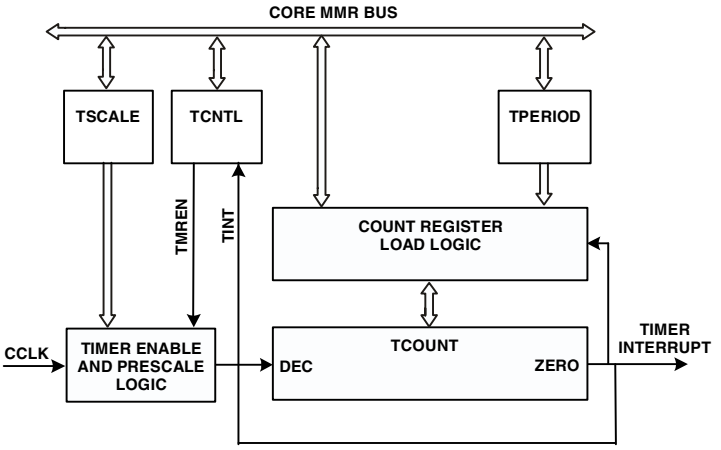


Figure 16-13. Core Timer Block Diagram

Core Timer Control Register (TCNTL)

When the timer is enabled by setting the TMREN bit in the TCNTL register, shown in Figure 16-14, 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 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.

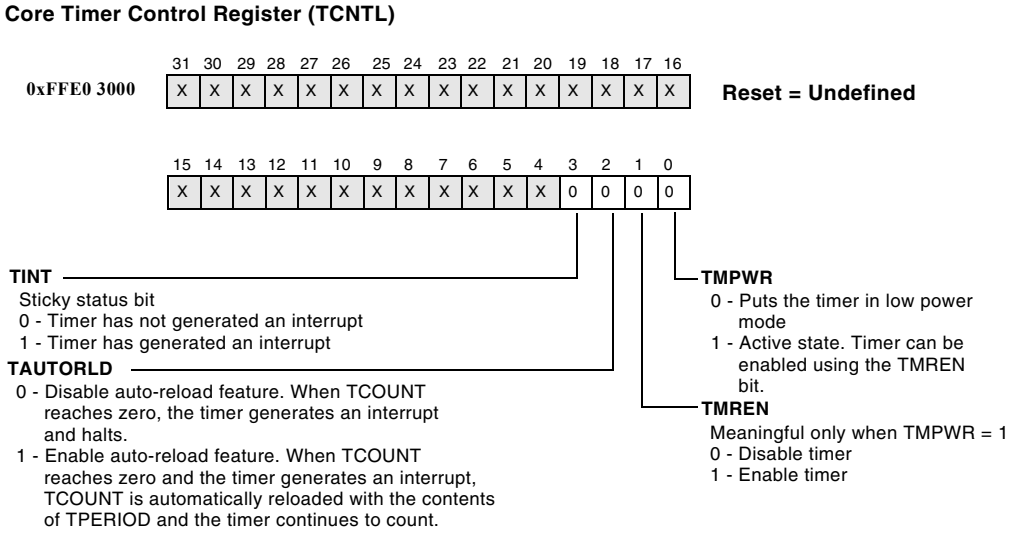


Figure 16-14. Core Timer Control Register

Core Timer Count Register (TCOUNT)

This register, shown in Figure 16-15, 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.

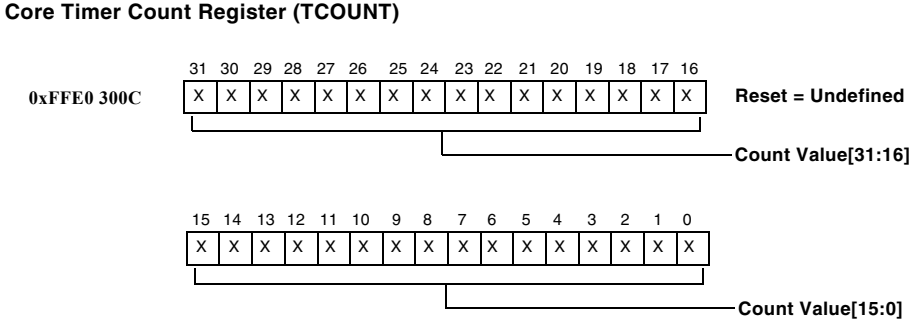


Figure 16-15. Core Timer Count Register

Core Timer Period Register (TPERIOD)

When auto-reload is enabled, the TCOUNT register is reloaded with the value of TPERIOD (shown in Figure 16-16) whenever TCOUNT reaches 0.

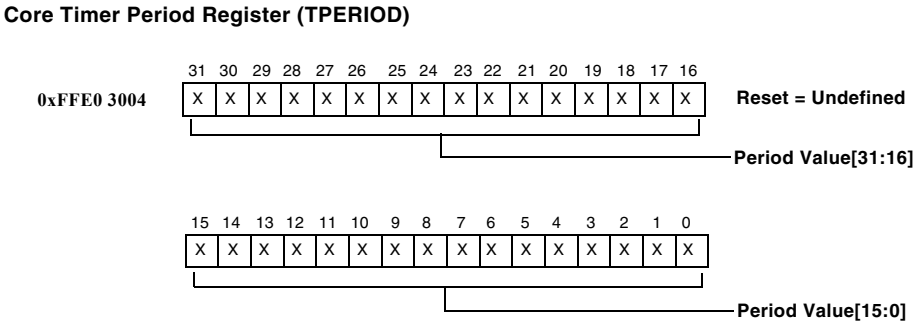


Figure 16-16. Core Timer Period Register

Core Timer Scale Register (TSCALE)

The TSCALE register, shown in Figure 16-17, 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 zero, the counter register decrements once every clock cycle. If TSCALE is 1, the counter decrements once every 2 cycles.

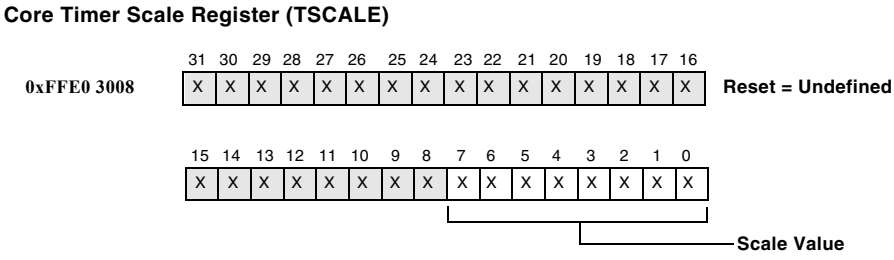


Figure 16-17. Core Timer Scale Register

Watchdog Timer

The ADSP-BF535 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 Blackfin 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 non-maskable 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).
2. Copy the count value from WDOG_CNT into the Watchdog Status register (WDOG_STAT) by executing a write to WDOG_STAT. For more information, see "Watchdog Status Register (WDOG_STAT)" on page 16-26.
3. In the Watchdog Control register (WDOG_CTL), select the event to be generated upon timeout.
4. Enable the watchdog timer in WDOG_CTL.
5. 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.

Watchdog Count Register (WDOG_CNT)

The Watchdog Count register (WDOG_CNT), shown in Figure 16-18, contains a 32-bit unsigned count value that is copied into WDOG_STAT. The WDOG_CNT register must be accessed with 32-bit read/writes only.

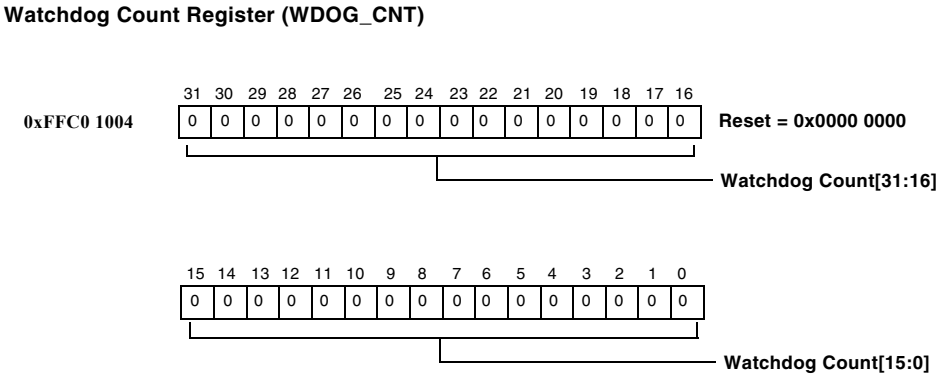


Figure 16-18. Watchdog Count Register

Watchdog Status Register (WDOG_STAT)

The Watchdog Status register (WDOG_STAT), shown in Figure 16-19, contains the current value of the watchdog timer. Reads to WDOG_STAT return this 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. 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 timer expires. This reloads the timer with the value from WDOG_CNT and prevents generation of the selected event.

The WDOG_STAT register is a 32-bit unsigned system MMR that must be accessed with 32-bit reads and writes.

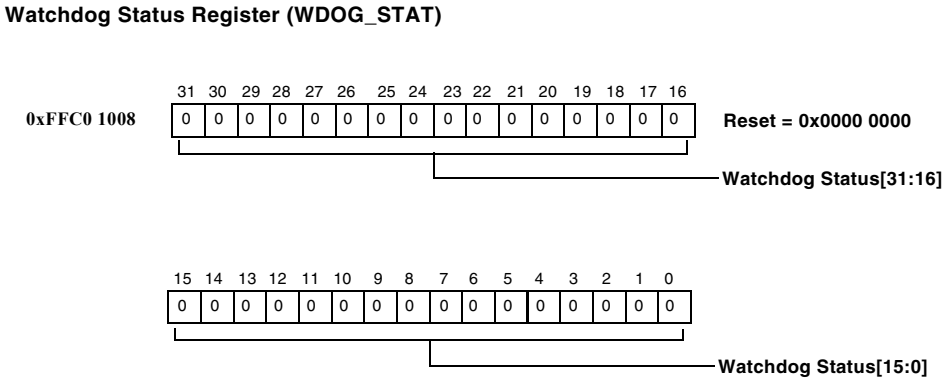


Figure 16-19. Watchdog Status Register

Watchdog Control Register (WDOG_CTL)

The Watchdog Control register (WDOG_CTL), shown in Figure 16-20, is a 16-bit system MMR used to control the watchdog timer.

The ICTL[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 TMR_EN[3:0] 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.

The TRO bit is a sticky bit that is set when the watchdog timer expires. Note that this bit is cleared by both hardware and software resets. If the reset event is selected in ICTL and subsequently generated by the watchdog timer, software must check the Software Watchdog Timer Source bit (SWTS) in the Software Reset register (SWRST) to determine if the reset was caused by the watchdog timer. For more information, see "Software Resets and Watchdog Timer" on page 3-14. The TRO bit can also be manually cleared by writing a 1 to it.

Watchdog Control Register (WDOG_CTL)

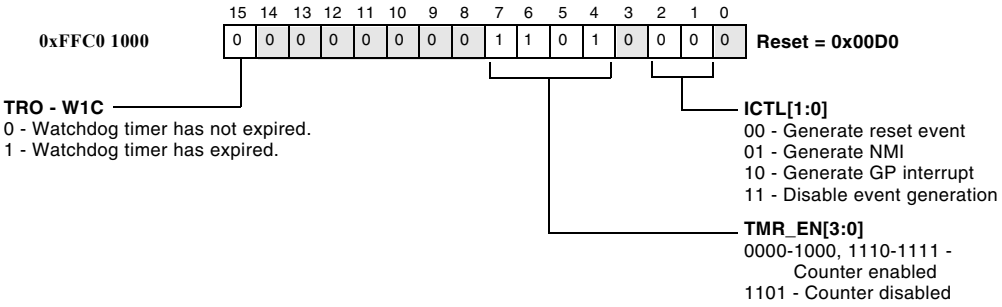


Figure 16-20. Watchdog Control Register

When the processor is in Emulation mode, the watchdog timer will not decrement even if it is enabled.

17 REAL-TIME CLOCK (RTC)

The Real-Time Clock (RTC) provides a set of digital watch features to the ADSP-BF535 processor, including time of day, alarm, and stopwatch countdown.

The RTC is clocked by a 32.768 kHz crystal external to the ADSP-BF535 processor. The RTC uses dedicated power supply pins, which enable it to maintain functionality even when the rest of the ADSP-BF535 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
- 255-day counter

The RTC increments the 60-second counter once per second and increments the other three counters when appropriate. The 255-day counter is incremented each day at midnight (0 hours, 0 minutes, 0 seconds). The RTC can generate an interrupt once per second, once per minute, and once per day.

The RTC provides two alarm features. The application can program an exact time (hour, minute, and second) into the RTC Alarm register (RTC_ALARM). When the alarm interrupt is enabled, the RTC generates an interrupt each day at the specified time. 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 by the application. The alarm interrupt and day-alarm interrupt can be enabled or disabled independently.

The RTC provides a stopwatch function that acts a countdown timer. The application can program a minute count into the RTC Stopwatch Count register (RTC_SWCNT). When the stopwatch interrupt is enabled and the specified number of minutes have elapsed, the RTC generates an interrupt.

The RTC can also generate an interrupt once every second, minute, and day. Each of these interrupts can be independently controlled.

Note that although each of the RTC counters operates continually and cannot be stopped by the application, no interrupts are generated unless enabled in the RTC Interrupt Control register (RTC_ICTL).

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. The RTC state can be read via the system MMR status registers at any time.

Writes to all RTC MMRs, except the RTC Interrupt Status register (RTC_ISTAT), are synchronized to the RTC clock. If the prescaler is enabled, writes are synchronized at a 1 Hz rate. If the prescaler is disabled, writes are synchronized to the 32.768 kHz crystal rate. The Write Pending Status bit in RTC_ISTAT indicates the progress of the write. The Write Pending Status bit is set when a write occurs and is cleared when the write is 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. There is no latency when reading RTC MMRs. 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.

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.

Do not read a register that has been written to until the Write Complete flag is set. Always check the Write Pending Status bit before attempting a read or write.

Writes to the RTC MMRs are synchronized to the RTC clock. When setting the time of day with the prescaler enabled, do not factor in the delay when writing to the RTC MMRs. The most accurate method of setting the Real-Time Clock when the prescaler is enabled is to monitor the Seconds (1 Hz) Event flag or to program an interrupt for this event and then write the current time plus two seconds to the RTC Status register (RTC_STAT) in the interrupt service routine (ISR).

The Seconds (1 Hz) Event flag is set on every positive edge of the RTC clock. Because of this, the write to RTC_STAT occurs almost immediately following the positive edge of the RTC clock. Since the value is registered into the actual RTC_STAT register on the second clock edge after the write occurs, adding two seconds to the real time before writing accounts for the synchronization delay.

Writes to clear bits in RTC_ISTAT take effect immediately.

The single active bit of the RTC Enable register (RTC_FAST) is set using a synchronization path. Clearing the bit is accomplished with a clear signal that is synchronized by the 32.768 kHz clock. This faster synchronization allows the module to be put into high-speed mode (bypassing the prescaler) without waiting the 1 to 2 seconds for the write to complete if the module is already running with the prescaler enabled. The first positive edge of the 1 Hz clock occurs 2 to 3 cycles of the 32.768 kHz clock after the prescaler is enabled.

Note the clear path does not clear the synchronization path for the bit that is set. Setting, then immediately clearing, the Prescaler Enable bit could result in its being set.

Clear all flags at power-up and when setting the RTC Status register (RTC_STAT), the RTC Alarm register (RTC_ALARM), or the RTC Stopwatch Count register (RTC_SWCNT). Wait for the write to complete for all of these registers before clearing the flags and enabling the Alarm interrupt, Day Alarm interrupt, or Stopwatch interrupt.

The unknown values in the registers at power-up can cause interrupts to occur before the correct value is written into each of the registers. By catching the slow 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 slow clock. This is valid as long as the RTC clock is running. Use this flag or interrupt to validate the other flags.
- Write Complete
    - Always valid, except before the first edge of the slow clock. Not valid for up to one second after power-up and at the start of the slow clock. Use the Seconds (1 Hz) Event flag to check for the first edge of the RTC clock. The Seconds (1 Hz) interrupt may also be used, but this causes a delay of up to two seconds.
- Write Pending Status
    - Same as Write Complete.
- Minutes 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 interrupt 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.

Wait for the RTC_STAT, RTC_ALARM, and RTC_SWCNT fields to be valid before enabling the alarm interrupt, so that no spurious interrupts occur before the values of these registers are updated.

Do not disable the prescaler by clearing the bit in RTC_FAST without making sure that there are no writes in progress. Do not switch between fast and slow mode during normal operation by setting and clearing this bit, because this introduces errors in the operation of the counters. To avoid these potential errors, initialize the RTC during startup and do not dynamically alter the state of the prescaler during normal operation.

Interrupts

The RTC should be programmed to provide interrupts at several programmable intervals, including:

- Per second
- Per minute
- Per day
- On countdown from a programmable value
- Daily at a specific time
- On a specific day and time

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_STAT`).

RTC Memory-Mapped Registers (MMRs)

This section describes the memory-mapped registers for the RTC.

RTC Status Register (RTC_STAT)

The RTC Status register, shown in Figure 17-1, is used to read or write the current time. Reads and writes to this register can occur at any time. To ensure that a valid read occurred, software should read the register twice and compare for equal results (within one second). 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.

RTC Status Register (RTC_STAT)

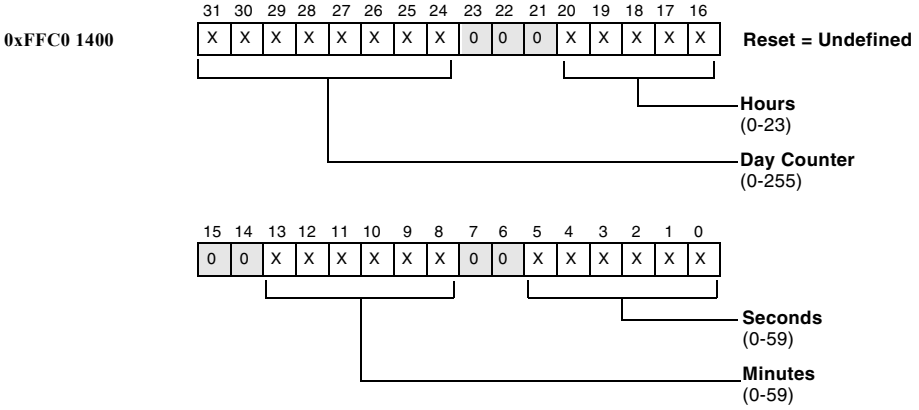


Figure 17-1. RTC Status Register

RTC Interrupt Control Register (RTC_ICTL)

The RTC provides up to six independently enabled (unmasked) interrupts in this register. At reset, all interrupts are disabled (masked). The stopwatch and alarm interrupts have their own dedicated control registers to determine count values. The seconds interrupt is generated on each 1 Hz clock tick, if enabled. The minutes interrupt is generated as the minute counter counts through sixty 1 Hz clock ticks. The 24-hour interrupt occurs once per 24-hour period (at midnight). Any of these interrupts can generate a wake-up request to the processor. All bits are read/write. This register, shown in Figure 17-2, is cleared at reset.

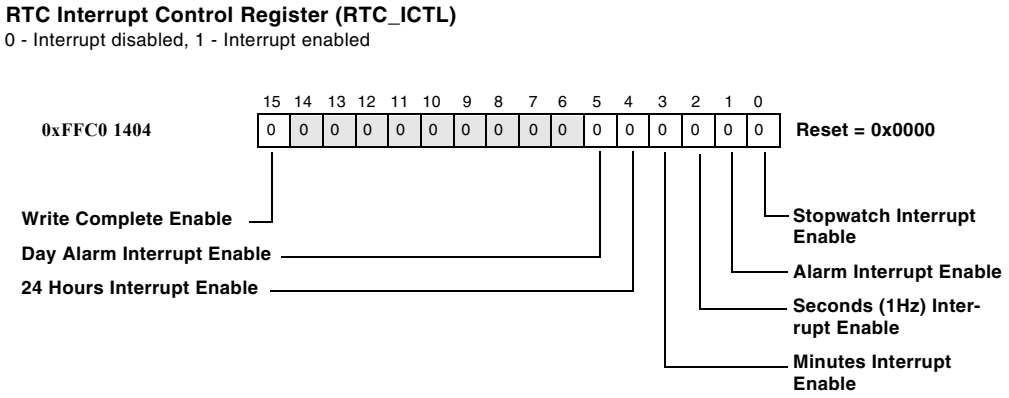


Figure 17-2. RTC Interrupt Control Register

RTC Interrupt Status Register (RTC_ISTAT)

The RTC Interrupt Status register, shown in Figure 17-3, provides the status of all RTC interrupts. These bits are sticky. Once set by the corresponding interrupt event, each bit remains set until cleared by a software write to this register. 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.

RTC Interrupt Status Register (RTC_ISTAT)
All bits are write-1-to-clear, except bit 14.

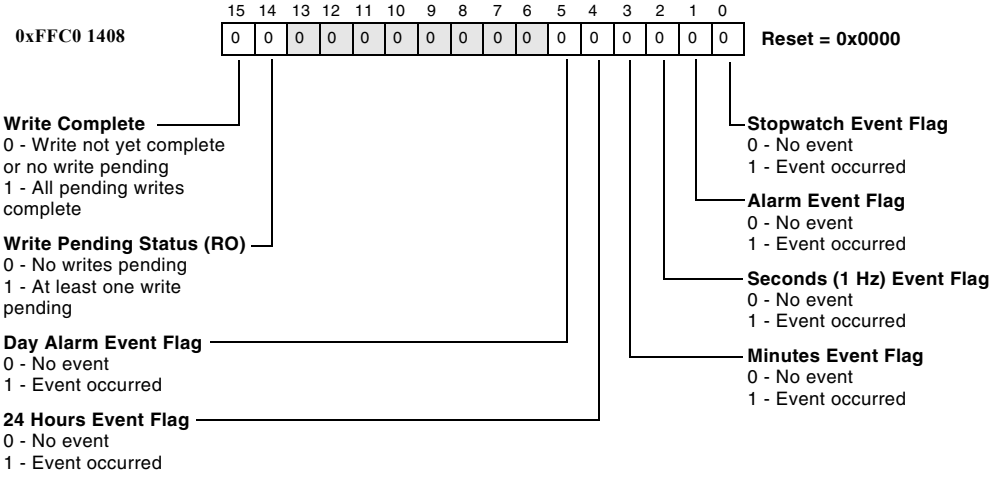


Figure 17-3. RTC Interrupt Status Register

RTC Stopwatch Count Register (RTC_SWCNT)

The RTC Stopwatch Count register, shown in Figure 17-4, contains the countdown value for the stopwatch. The stopwatch counts down minutes from the programmed value and generates an interrupt 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`. The register can be programmed to any value between 0 and 255 minutes.

RTC Stopwatch Count Register (RTC_SWCNT)

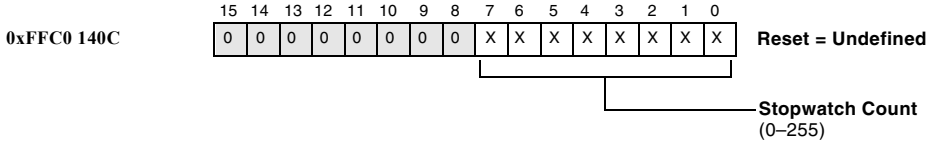


Figure 17-4. RTC Stopwatch Count Register

RTC Alarm Register (RTC_ALARM)

The RTC Alarm register, shown in Figure 17-5, 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 match those of the RTC Status register. The day interrupt occurs whenever the day, hour, minute, and second fields match those of the RTC Status register.

RTC Alarm Register (RTC_ALARM)

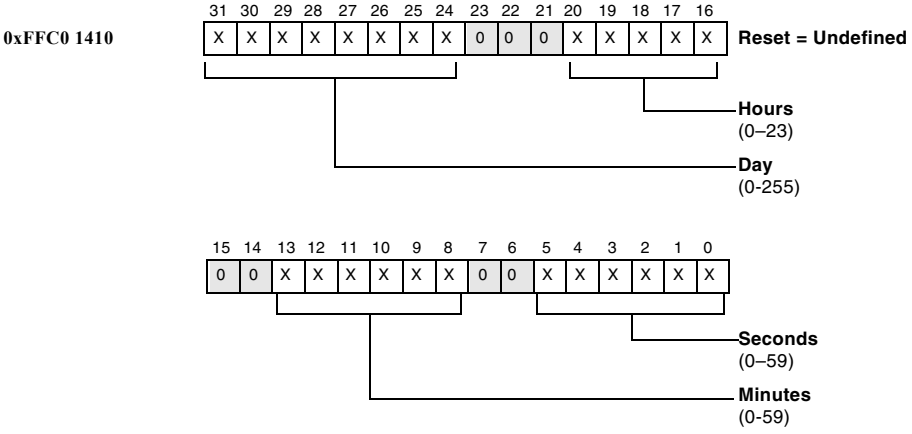


Figure 17-5. RTC Alarm Register

RTC Enable Register (RTC_FAST)

The RTC Enable register, shown in Figure 17-6, 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. Software must set this bit for the RTC to operate at the proper rate.

Do not disable the prescaler by clearing the bit in `RTC_FAST` without making sure that there are no writes in progress. Do not switch between fast and slow mode during normal operation by setting and clearing this bit, because this introduces errors in the operation of the counters. To avoid these potential errors, initialize the RTC during startup and do not dynamically alter the state of the prescaler during normal operation.

RTC Enable Register (RTC_FAST)

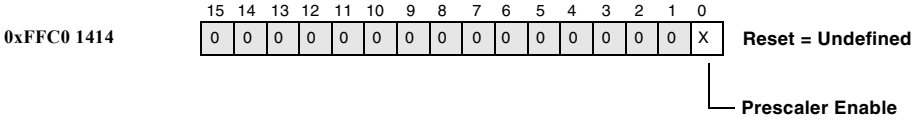


Figure 17-6. RTC Enable Register

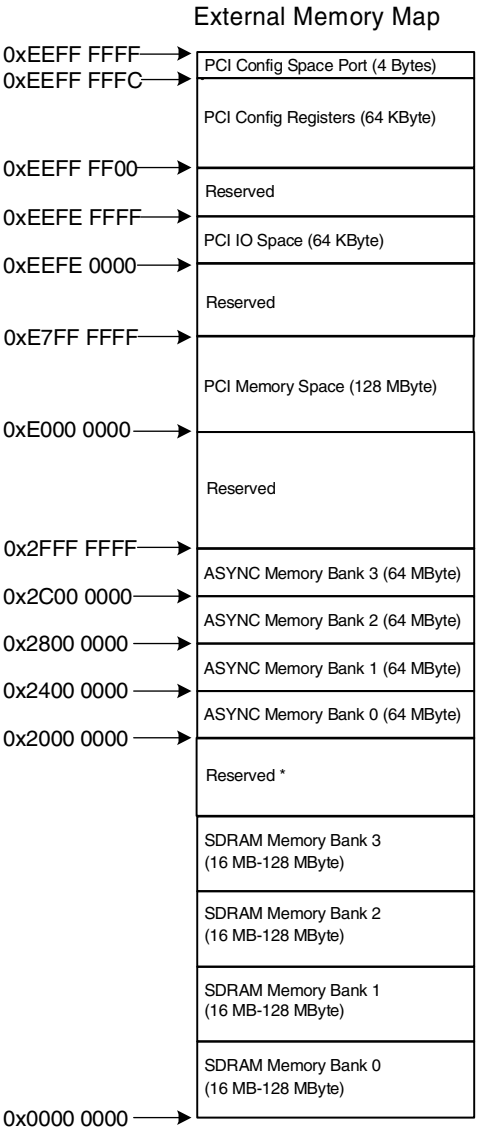
18 EXTERNAL BUS INTERFACE UNIT

The ADSP-BF535 processor's External Bus Interface Unit (EBIU) provides glueless interfaces to external memories. The ADSP-BF535 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.

The EBIU is clocked by the system clock (SCLK). All synchronous memories interfaced to the ADSP-BF535 processor operate at the SCLK frequency. The ratio between core frequency and SCLK frequency is programmable using a PLL system MMR. For more information, see "Core Clock/System Clock Ratio Control" on page 8-5.

The external memory space is shown in Figure 18-1. Four of the memory regions are dedicated to SDRAM support. SDRAM interface timing and the size of each SDRAM region are programmable. Each SDRAM bank can range in size from 16 MBytes to 128 MBytes. The start address of bank 0 is 0x0000 0000. The start addresses of banks 1, 2, and 3 follow contiguously from the previous bank. If all four SDRAM banks are not fully populated with 128 MBytes of SDRAM, the area from the end of bank 3 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.



* This reserved block does not exist if all four blocks of SDRAM are configured to their full 128 MByte size.

Figure 18-1. External Memory Map

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 PCI or a DMA channel, depending on where the access originated.

The PCI space shown at the top of the external memory map is supported by the PCI Controller, not the EBIU. For more information, refer to Chapter 15, “PCI.”

Block Diagram

Figure 18-2 is a conceptual block diagram of the EBIU and its interfaces. Since only one ADSP-BF535 processor 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.

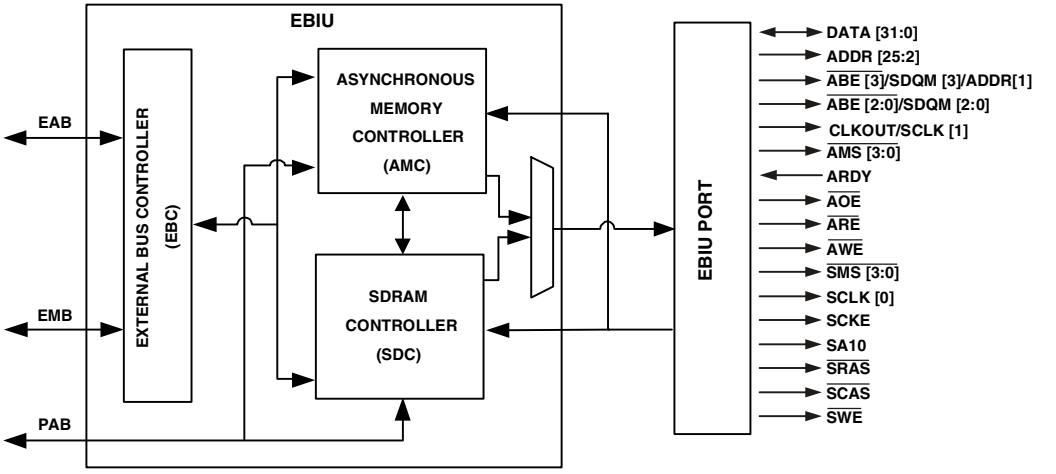


Figure 18-2. External Bus Interface Unit (EBIU)

Internal Memory Interfaces

The EBIU functions as a slave on three buses internal to the ADSP-BF535 processor:

- External Access Bus (EAB), mastered by the System Bus Interface Unit (SBIU) on behalf of external bus requests from the core or the DMA bus or mastered by the MemDMA controller.
- External Mastered Bus (EMB), mastered by the PCI bridge.
- Peripheral Access Bus (PAB), mastered by the SBIU on behalf of system MMR requests from the core or from the PCI master.

EBIU Arbitration

The SDC and the AMC can accept transaction requests in parallel, if queue capacity is available. However, the External Bus Controller (EBC) serializes requests from the EMB and the EAB internal buses. The EMB bus is strictly for PCI accesses. The EMB and EAB buses have equal priority. However, once the EBIU grants a bus request to either the EMB or the EAB bus, the arbitration algorithm favors the opposite bus for subsequent requests if both buses are requesting EBIU access.

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. Table 18-1 and Table 18-2 describe the signals associated with each interface. Pin types are abbreviated with "I" for input and "O" for output.

Table 18-1. Description of Shared EBIU Port Pins

EBIU Port Pins	Pin Type	Description
DATA[31:0]	I/O	Data Bus
ADDR[19:2]	O	Address Bus
$\overline{\text{ABE[3]}}$/SDQM[3]/ADDR[1]	O	AMC Byte Enable 3/SDC Data Mask 3/Address 1
$\overline{\text{ABE[2:0]}}$/SDQM[2:0]	O	AMC Byte Enables/SDC Data Masks
CLKOUT/SCLK[1]	O	AMC System Clock Output/SDC Clock 1

No other signals are multiplexed between the two controllers.

Table 18-2. Asynchronous Memory Interface Signals

EBIU Port Pins	Pin Type	Description
DATA[31:0]	I/O	External Data Bus
ADDR[25:2]	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 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. Refer to *ADSP-BF535 Blackfin 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.
ARDY	I	Asynchronous Memory Ready Response Note this is a synchronous input.
$\overline{\text{ABE[3]}}$/ADDR[1]	O	Byte Enables For 16 bit wide asynchronous memories ADDR[1] is output on $\overline{\text{ABE[3]}}$. Shared with SDC.
$\overline{\text{ABE[2:0]}}$	O	Byte Enables
CLKOUT	O	System Clock Output

Table 18-3. SDRAM Interface Signals

EBIU Port Pins	Pin Type	Description
DATA[31:0]	I/O	External Data Bus
ADDR[19:18], ADDR[16:2]	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.

Table 18-3. SDRAM Interface Signals (Cont'd)

EBIU Port Pins	Pin Type	Description
$\overline{\text{SCAS}}$	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.
SDQM[3]/ADDR[1] SDQM[2:0]	O	SDRAM Data Mask pins Connect to SDRAM's DQM pins. For 16-bit wide SDRAM configurations ADDR[1] is output on SDQM[3] and should be connected to SDRAM A[0] pin.
$\overline{\text{SMS[3:0]}}$	O	Memory select pin of external memory bank configured for SDRAM Connect to SDRAM's $\overline{\text{CS}}$ (chip select) pin.
SA10	O	SDRAM A10 pin SDRAM interface uses this pin to make refreshes possible while the AMC is using the bus. Connect to SDRAM's A[10] pin.
SCKE	O	SDRAM Clock Enable pin Connect to SDRAM's CKE pin.
SCLK[0]	O	SDRAM Clock output pin 0 Switches at system clock frequency. Connect to the SDRAM's CLK pin.
SCLK[1]	O	SDRAM Clock output pin 1 Same frequency and timing as SCLK[0]. Provided to reduce capacitive loading on SCLK[0]. Connect to the SDRAM's CLK pin.

EBIU Programming Model

This section describes the programming model of the EBIU. This model is based on system memory-mapped registers (MMRs) used to program the EBIU.

There are 6 control registers and 1 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.

In addition to the EBIU control and status registers (system MMRs) described in this chapter, the clock distributed to the EBIU has a dedicated enable, located in the IOCK register of the PLL control. For more information, see “Peripheral Clock Enable Register (PLL_IOCK)” on page 8-22.

Error Detection

The EBIU responds to any operation which addresses the range of 0x0000 0000 - 0xDFFF FFFF, even if that bus operation addresses reserved or disabled memory. It responds by completing the bus operation (asserting the appropriate number of acknowledges as specified by the bus master) and by asserting the EAB or EMB bus error signal for these error conditions:

- Any access to reserved memory space
- Any access to a disabled external memory bank

- Any access to an unpopulated area of an SDRAM memory bank
- Any access to SDRAM memory space before the SDC is enabled or before the power-up sequence has been initiated

If the core requested the faulting bus operation, the bus error response from the EBIU generates a hardware error interrupt (IVHW) internal to the core (this interrupt can be masked off in the core). If PCI requested the faulting bus operation, then the bus error is captured in that controller, and can optionally generate an interrupt to the core. For more information, refer to "PCI Bus Interface" on page 13-1. 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. For more information, refer to "Direct Memory Access" on page 9-1.

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 18-4.

Table 18-4. Asynchronous Memory Bank Address Range

Memory Bank Select	Address Start	Address End
$\overline{\text{AMS[3]}}$	2C00 0000	2FFF FFFF
$\overline{\text{AMS[2]}}$	2800 0000	2BFF FFFF
$\overline{\text{AMS[1]}}$	2400 0000	27FF FFFF
$\overline{\text{AMS[0]}}$	2000 0000	23FF FFFF

Asynchronous Memory Address Decode

The address range allocated to each asynchronous memory bank is fixed at 64 MB; however, not all of an enabled memory bank need be populated. 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 that 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 18-2 on page 18-6. The timing of these pins is programmable to allow a flexible interface to devices of different speeds. For example interfaces, see "System Design" on page 19-1.

Asynchronous Memory Global Control Register (EBIU_AMGCTL)

The Asynchronous Memory Global Control register 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 ADSP-BF535 processor to access external memory-mapped asynchronous devices. Figure 18-3 shows the Asynchronous Memory Global Control register.

Asynchronous Memory Global Control Register (EBIU_AMGCTL)

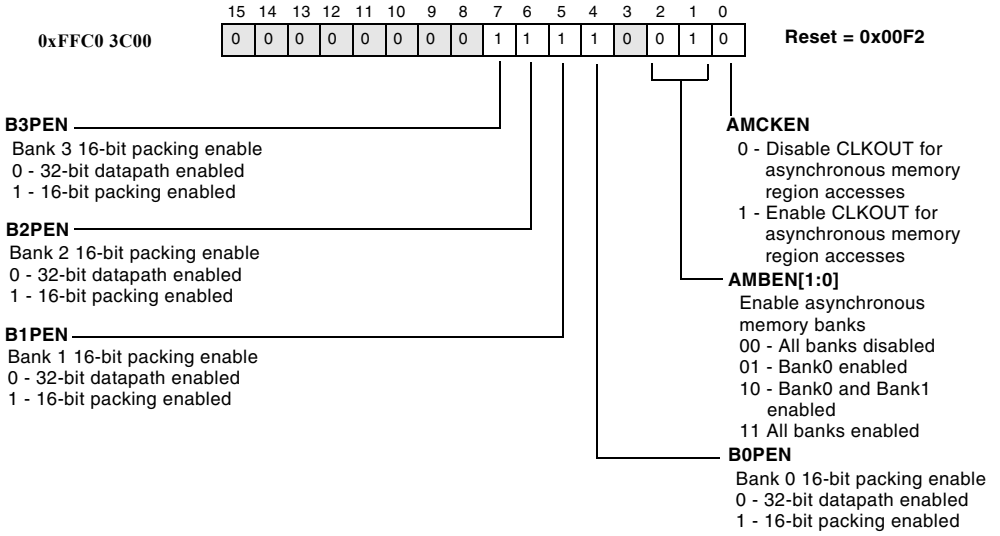


Figure 18-3. Asynchronous Memory Global Control Register

For external devices that need a clock, the CLKOUT pin 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 zero.

The BxPEN bits allow software to enable 16-bit packing mode for each bank independently. When 16-bit packing is enabled, $\overline{\text{ABE[3]}}$ becomes ADDR[1] and $\overline{\text{ABE[2]}}$ is held deasserted. Only the lower 16 bits of the external data bus are used, DATA[15:0].

In 16-bit packing mode, 32-bit transactions to AMC space that are requested on internal buses are converted to two sequential 16-bit accesses on the external bus. All 8-bit and 16-bit requests are still processed with a single external transaction. When 32-bit packing is enabled, all 32-bit transactions are processed with a single transfer.

Asynchronous Memory Bank Control Registers (EBIU_AMBCTL0, EBIU_AMBCTL1)

The EBIU Asynchronous Memory Controller has two memory bank control registers. 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.

For bit descriptions of the two registers, see Figure 18-4 (Asynchronous Memory Bank Control 0 Register) and Figure 18-5 (Asynchronous Memory Bank Control 1 Register).

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 that ARDY is *synchronously* sampled, therefore:

- Assertion and deassertion of ARDY to the ADSP-BF535 processor must meet the data sheet setup and hold times. Failure to meet these synchronous specifications could result in meta-stable behavior internally. The ADSP-BF535 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 zero. 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 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 18-26.

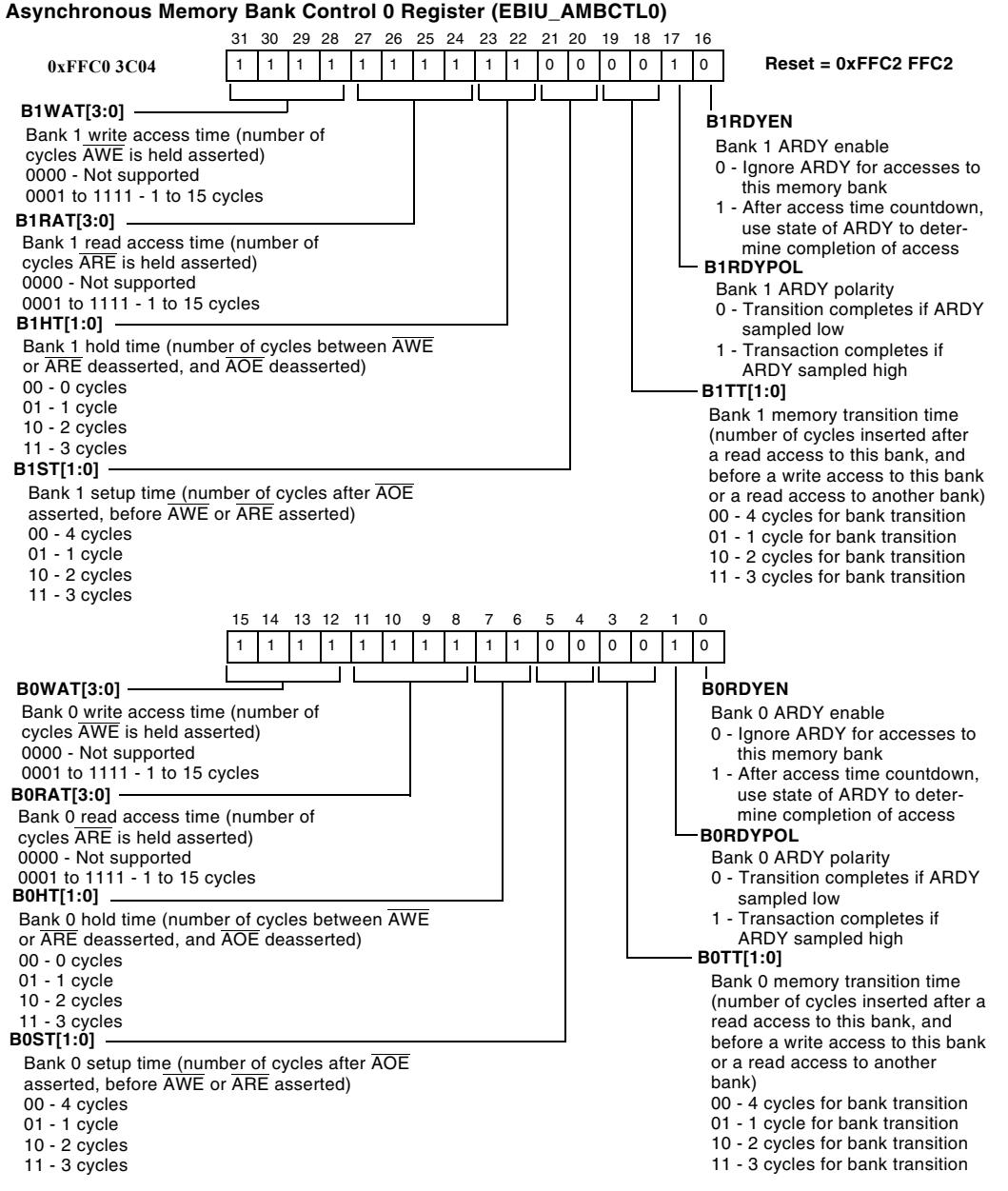


Figure 18-4. Asynchronous Memory Bank Control 0 Register

Asynchronous Memory Bank Control 1 Register (EBIU_AMBCTL1)

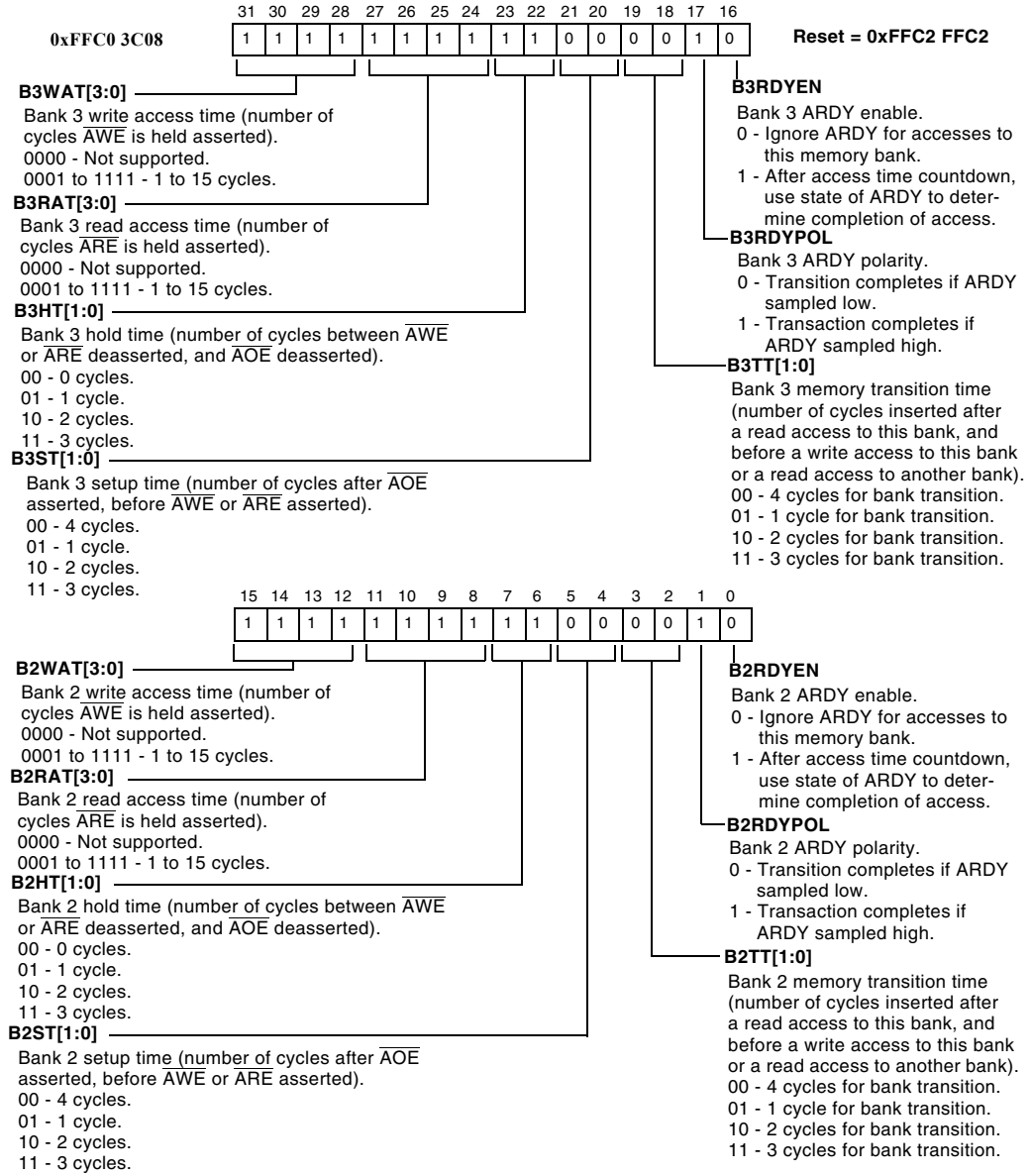


Figure 18-5. Asynchronous Memory Bank Control 1 Register

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 MemDMA, and the sequence of transactions (read followed by read, read followed by write, etc.).

Asynchronous Accesses by Core Instructions

Some external memory accesses are caused by core instructions of the type:

```
R0 = [P0++] ; /* Read from external memory, where P0 points to
a location in external memory */
```

or:

```
[P0++] = R0 ; /* Write to external memory */
```

Asynchronous Reads

Figure 18-6 shows two core-initiated asynchronous read bus cycles to the same bank, with timing programmed with setup = 1 cycle, read access = 3 cycles, hold = 2 cycles, and transition time = 1 cycle.

Asynchronous read bus cycles proceed as:

- At the start of the setup period, $\overline{\texttt{AMS[x]}}$, the address bus, and $\overline{\texttt{ABE[3:0]}}$ become valid, and $\overline{\texttt{AOE}}$ asserts.
- At the beginning of the read access period and after the setup cycle, $\overline{\texttt{ARE}}$ asserts.
- At the beginning of the hold period, read data is sampled on the rising edge of CLKOUT. The $\overline{\texttt{ARE}}$ pin deasserts after this rising edge.

- At the end of the hold period, $\overline{\text{AOE}}$ and $\overline{\text{AMS[x]}}$ deassert.
- Unless another read of the same memory bank is queued internally, the AMC appends the programmed number of memory transition time cycles.

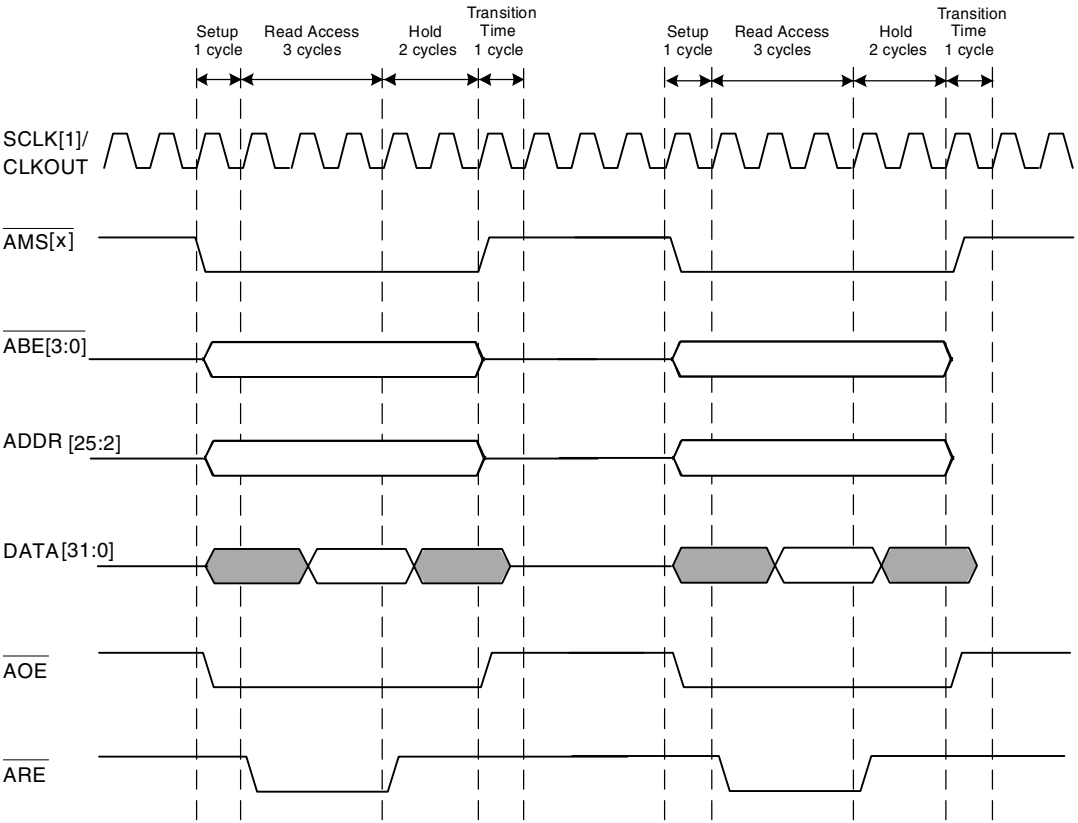


Figure 18-6. Core-Initiated Asynchronous Read Bus Cycles

Asynchronous Writes

Write accesses can only be initiated by the AMC every 5th SCLK cycle. If the setup period plus the access period plus the hold period is less than 5 cycles, AMS[x] deasserts between accesses. This is shown in the next two examples.

Figure 18-7 shows two core-initiated asynchronous write bus cycles to the same bank, with timing programmed with setup = 1 cycle, write access = 2 cycles, hold = 2 cycles, and transition time = 1 cycle.

The first asynchronous write bus cycle proceeds as:

- At the start of the setup period, AMS[x], the address bus, data buses, and ABE[3:0] become valid.
- At the beginning of the write access period, AWE asserts.
- At the beginning of the hold period, AWE deasserts.
- After the hold period, AMS[x] remains low for the next setup period of the next access.

The second asynchronous write bus cycle proceeds as:

- At the start of the setup period, AMS[x] is still asserted. The address and data buses, and ABE[3:0] become valid.
- At the beginning of the write access period, AWE asserts.
- At the beginning of the hold period, AWE deasserts.
- After the hold period, AMS[x] deasserts.

Figure 18-8 shows two higher-speed asynchronous write bus cycles to the same bank, with timing programmed with setup = 1 cycle, write access = 2 cycles, hold = 0 cycles, and transition time = 1 cycle.

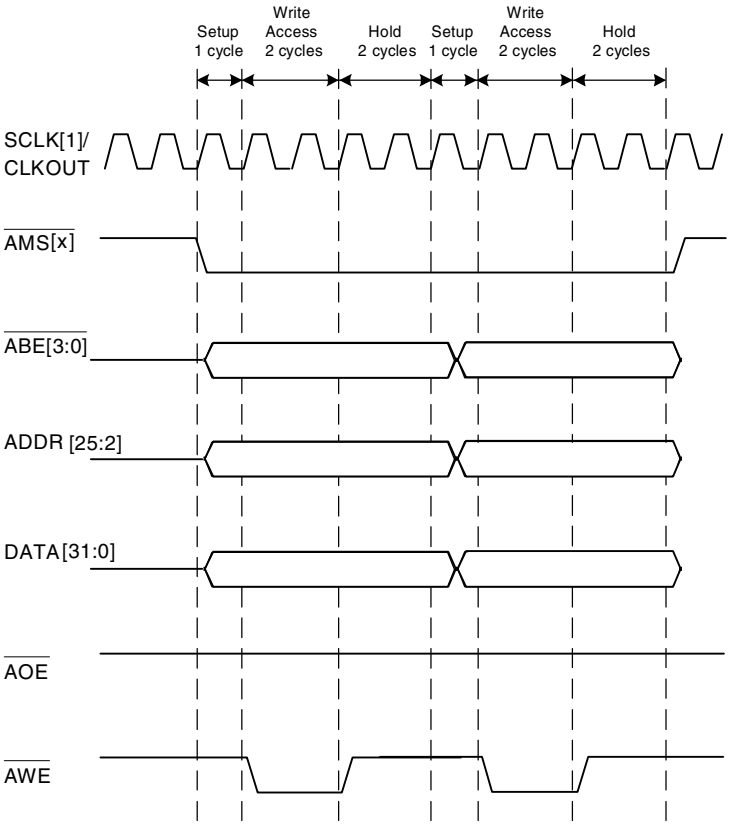


Figure 18-7. Core-Initiated Asynchronous Write Bus Cycles

The first asynchronous write bus cycle proceeds as:

- At the start of the setup period, $\overline{\texttt{AMS[x]}}$, the address bus, data buses, and $\overline{\texttt{ABE[3:0]}}$ become valid.
- At the beginning of the write access period, $\overline{\texttt{AWE}}$ asserts.
- At the beginning of the hold period, $\overline{\texttt{AWE}}$ deasserts.
- After the hold period, $\overline{\texttt{AMS[x]}}$ deasserts.

The second asynchronous write bus cycle proceeds as:

- At the start of the setup period, $\overline{\texttt{AMS[x]}}$, the address bus, data buses, and $\overline{\texttt{ABE[3:0]}}$ become valid.
- At the beginning of the write access period, $\overline{\texttt{AWE}}$ asserts.
- At the beginning of the hold period, $\overline{\texttt{AWE}}$ deasserts.
- After the hold period, $\overline{\texttt{AMS[x]}}$ deasserts.

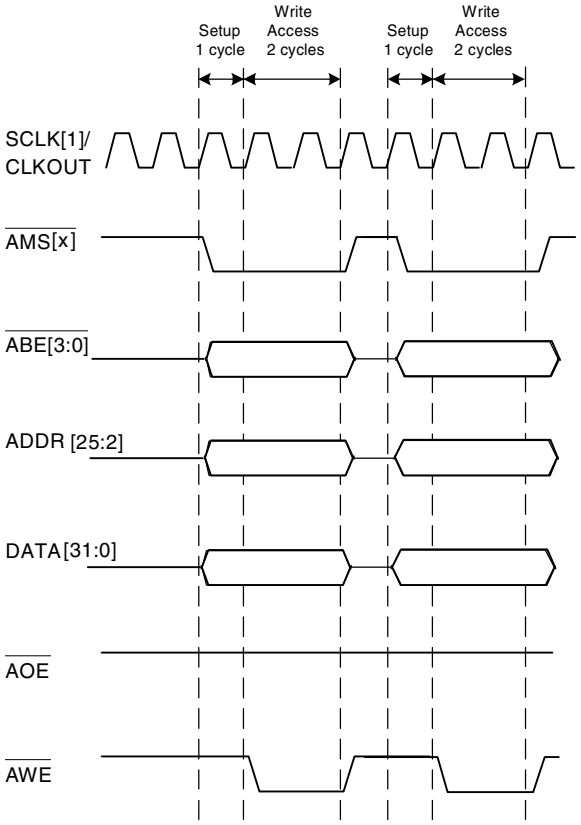


Figure 18-8. High Speed Core-Initiated Asynchronous Write Bus Cycles

Asynchronous Writes Followed by Reads

Figure 18-9 shows an asynchronous write bus cycle followed by two asynchronous read cycles to the same bank, with timing programmed with setup = 1 cycle, write access = 2 cycles, read access = 2 cycles, hold = 2 cycles, and transition time = 1 cycle.

The asynchronous write bus cycles proceed as:

- At the start of the setup period, $\overline{\text{AMS[x]}}$, the address bus, data buses, and $\overline{\text{ABE[3:0]}}$ become valid.
- At the beginning of the write access period, $\overline{\text{AWE}}$ asserts.
- At the beginning of the hold period, $\overline{\text{AWE}}$ deasserts and $\overline{\text{AMS[x]}}$ remains low for the setup period of the next access.

The first asynchronous read bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMS[x]}}$ is still asserted. The address bus, and $\overline{\text{ABE[3:0]}}$ become valid, and $\overline{\text{AOE}}$ asserts.
- At the beginning of the read access period, $\overline{\text{ARE}}$ asserts.
- At the beginning of the hold period, read data is sampled on the rising edge of the EBIU clock. The $\overline{\text{ARE}}$ pin deasserts after this rising edge.
- At the end of the hold period, $\overline{\text{AOE}}$ and $\overline{\text{AMS[x]}}$ deassert.

The second asynchronous read bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMS[x]}}$, the address bus, and $\overline{\text{ABE[3:0]}}$ become valid, and $\overline{\text{AOE}}$ asserts.
- At the beginning of the read access period, $\overline{\text{ARE}}$ asserts again.

- 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.
- At the end of the hold period, $\overline{\texttt{AOE}}$ and $\overline{\texttt{AMS[x]}}$ deassert.
- Unless another read of the same memory bank is queued internally, the AMC appends the programmed number of memory transition time cycles.

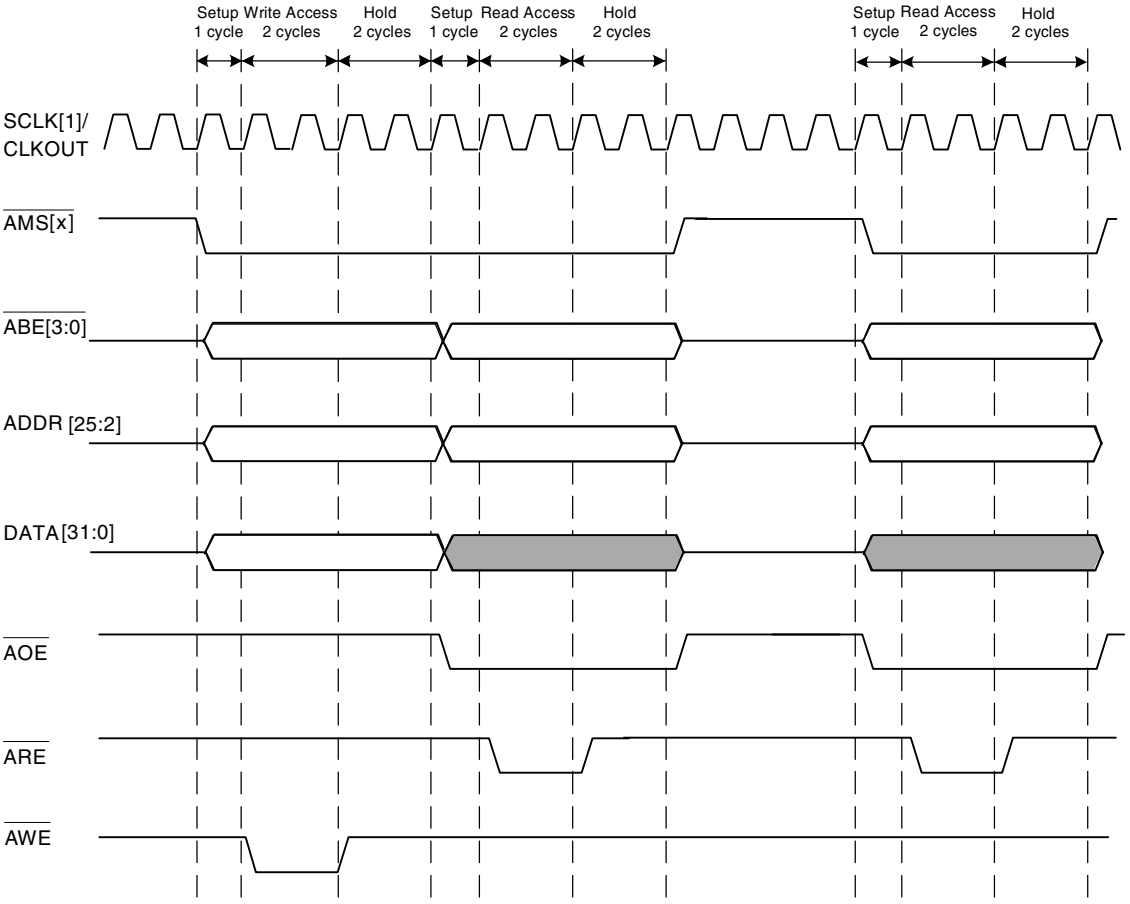


Figure 18-9. Core-Initiated Write and Read Bus Cycles

Asynchronous Accesses by MemDMA

External memory accesses are also caused by MemDMA. MemDMA transfers to external memory space occur in bursts of 8 accesses. There are 6 SCLK cycles inserted between bursts due to internal bus transactions.

Within each burst, the accesses may be back-to-back ($\overline{\text{AMS[x]}}$ continually asserted) or there may be separation between accesses ($\overline{\text{AMS[x]}}$ deasserted after each access), depending on the programming of the AMC.

Asynchronous Reads

Figure 18-10 shows a MemDMA read access of 16 words from external memory. The asynchronous memory controller is programmed with setup = 1 cycle, read access = 3 cycles, hold = 3 cycles, and transition time = 1 cycle.

The MemDMA access proceeds as:

- At the start of the setup period, $\overline{\text{AMS[x]}}$, the address bus, and $\overline{\text{ABE[3:0]}}$ become valid, and $\overline{\text{AOE}}$ asserts.
- At the beginning of the read access period and after the setup cycle, $\overline{\text{ARE}}$ asserts.
- At the beginning of the hold period, read data is sampled on the rising edge of the EBIU clock. The $\overline{\text{ARE}}$ pin deasserts after this rising edge.
- At the end of the hold period, $\overline{\text{AOE}}$ and $\overline{\text{AMS[x]}}$ deassert if the transaction is the last in the burst, otherwise $\overline{\text{AOE}}$ and $\overline{\text{AMS[x]}}$ remain asserted for the setup period of the next read.

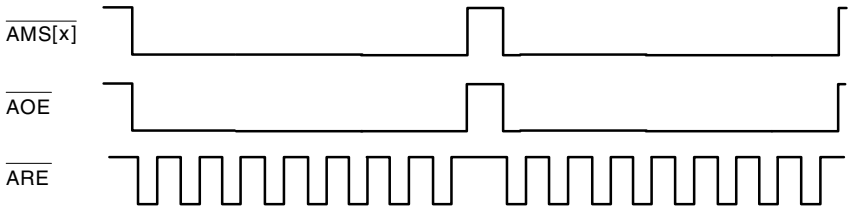


Figure 18-10. MemDMA Read With 3-Cycle Hold

For MemDMA accesses through AMC, 2 SCLK cycles are required from the time the read data is sampled until the next setup period begins. As a result, if the hold period programmed is less than 2 cycles, $\overline{\text{AMS[x]}}$ and $\overline{\text{AOE}}$ deassert between accesses within a burst. Regardless of hold period programming, if the access is the last in the burst, $\overline{\text{AMS[x]}}$ and $\overline{\text{AOE}}$ deassert for 6 cycles before the next burst begins.

Figure 18-11 also shows a MemDMA read access of 16 words from external memory. In this case, the asynchronous memory controller is programmed with setup = 1 cycle, read access = 3 cycles, hold = 1 cycle, and transition time = 1 cycle. This is a smaller number of hold cycles than programmed in Figure 18-10.

The MemDMA access proceeds as:

- At the start of the setup period, $\overline{\text{AMS[x]}}$, the address bus, and $\overline{\text{ABE[3:0]}}$ become valid, and $\overline{\text{AOE}}$ asserts.
- At the beginning of the read access period and after the setup cycle, $\overline{\text{ARE}}$ asserts.
- At the beginning of the hold period, read data is sampled on the rising edge of the EBIU clock. The $\overline{\text{ARE}}$ pin deasserts after this rising edge.
- At the end of the hold period, $\overline{\text{AOE}}$ and $\overline{\text{AMS[x]}}$ deassert.

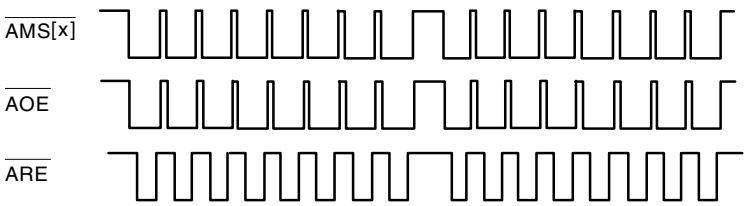


Figure 18-11. MemDMA Read With 1-Cycle Hold

Asynchronous Writes

Figure 18-12 shows a MemDMA write access of 16 words to external memory. The asynchronous memory controller is programmed with setup = 1 cycle, write access = 3 cycles, hold = 0 cycles, and transition time = 1 cycle.

The MemDMA access proceeds as:

- At the start of the setup period, $\overline{\text{AMS[x]}}$, the address bus, data buses, and $\overline{\text{ABE[3:0]}}$ become valid.
- At the beginning of the write access period and after the setup cycle, $\overline{\text{AWE}}$ asserts.
- At the beginning of the hold period, $\overline{\text{AWE}}$ deasserts.
- At the end of the hold period, $\overline{\text{AMS[x]}}$ deasserts if the transaction is the last in the burst, otherwise $\overline{\text{AMS[x]}}$ remains asserted for the setup period of the next write.

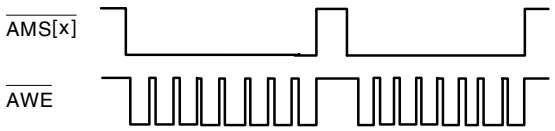


Figure 18-12. MemDMA Write

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 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 asserted. The read- or write-enable remains asserted for one clock cycle after ARDY is sampled asserted. An

example of this behavior is shown in Figure 18-13, where setup = 2 cycles, read access = 4 cycles, and hold = 1 cycle. Note that the read access period must be programmed to a minimum of two cycles to make use of the ARDY input.

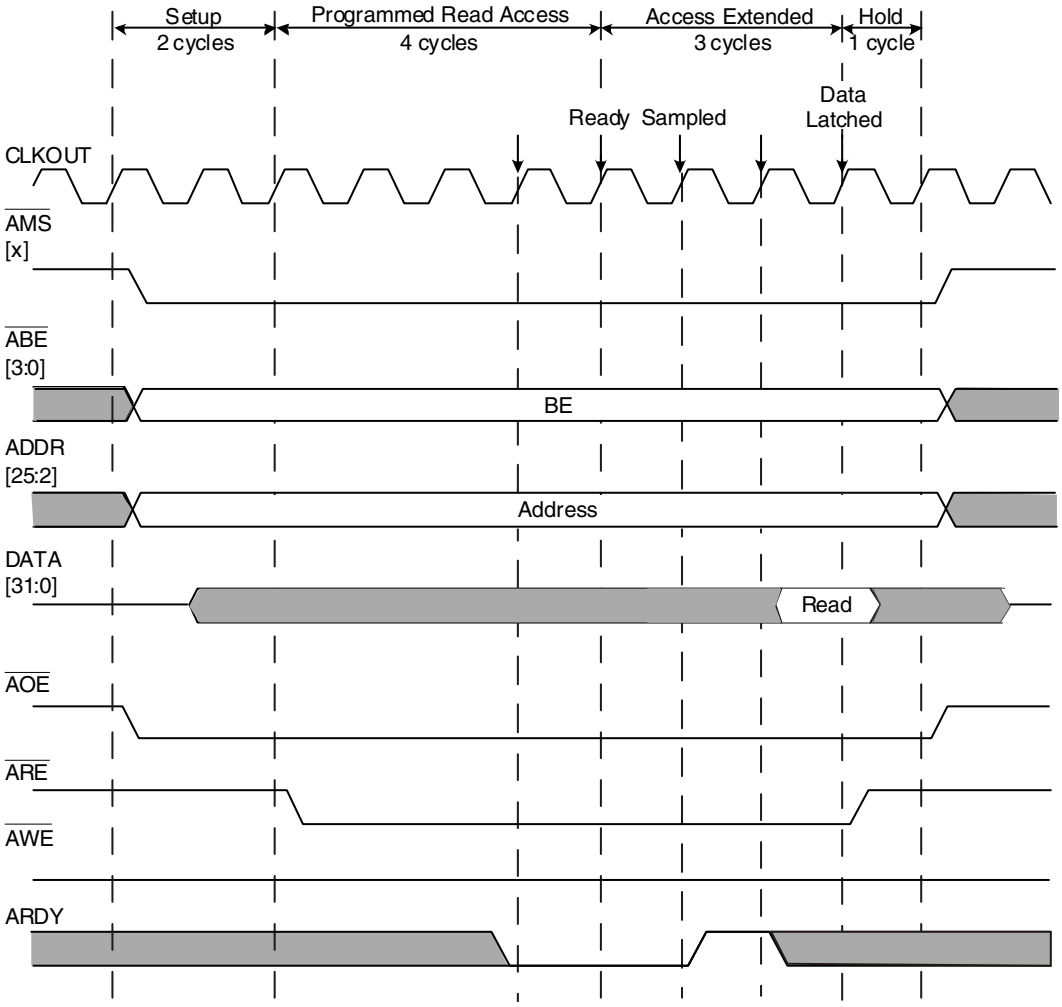


Figure 18-13. Inserting Wait States Using ARDY

SDRAM Controller (SDC)

The SDRAM Controller (SDC) enables the ADSP-BF535 processor to transfer data to and from Synchronous DRAM (SDRAM). The ADSP-BF535 processor supports a glueless interface with up to four banks of standard SDRAMs of 64 Mbit, 128 Mbit, 256 Mbit, and 512 Mbit with configurations x4, x8, and x16. Each bank supports up to 128 MByte, with a maximum total capacity of 512 MBytes (MB) of SDRAM. The interface can also support DIMMs and includes timing options to support additional buffers between the ADSP-BF535 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 `SCLK[0]` (or `SCLK[1]`, which has the same timing). The SDRAM interface is able to connect to as many as four external memory banks of SDRAM. All banks share the same SDRAM timing parameters.

Do not confuse the “SDRAM internal banks” which are internal to the SDRAM and are selected with the bank address, with the four “SDRAM banks,” or “external banks,” that are enabled by the $\overline{\texttt{SMS[3:0]}}$ pins.

The EBIU SDC provides a glueless interface with standard SDRAMs. The ADSP-BF535 processor’s SDRAM controller:

- Supports SDRAMs of 64 Mbit, 128 Mbit, 256 Mbit, and 512 Mbit with configurations of x4, x8, and x16.
- Supports up to two x64 DIMM modules, or up to four x32 DIMM modules.
- Supports up to 512 MB total array size with 128 MB of SDRAM on each of the four external SDRAM banks.

- Supports SDRAM page sizes of 1 KB, 2 KB, 4 KB, and 8 KB for 32-bit wide SDRAM banks and 1 KB, 2 KB, and 4 KB for 16-bit wide SDRAM banks.
- Supports four internal banks within the SDRAMs.
- 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 ADSP-BF535 processor and SDRAM. The same timing options are applied to all four external memory banks.
- Provides two clock output pins to drive capacitive loads of larger memory arrays.
- 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 mode.
- Provides two SDRAM power-up options.

Definition of Terms

These 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. Therefore, only one

internal bank is open at a time, which means the banks are accessed sequentially. The Bank Activate command must be applied before a read or write command.

The SDC does not interleave SDRAM accesses, so only one internal bank in a row is open at a time.

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 power-up sequence.

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. The Burst Length is independent from 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 power-up 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 power-up sequence.

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. The CAS latency is programmed in the SDRAM Mode register during the power-up sequence.

The speed grade of the SDRAM and the SCLK[0] frequency determine the value of the CAS latency. The SDC can support CAS latency of 2 or 3 clock cycles. The selected CAS latency value must be programmed into the SDRAM Memory Global Control register (EBIU_SDGCTL) before the SDRAM power-up sequence. See "SDRAM Memory Global Control Register (EBIU_SDGCTL)" on page 18-37.

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 SDRAM banks are precharged.

DQM Data I/O Mask Function.

The SDQM[3:0] pins provide a byte-masking capability on 8- or 16-bit writes to SDRAM. The DQM pins are used to block the output buffer of the SDRAM during precharge and certain write operations. The SDQM[3:0] pins are not used to mask data on read cycles. For 16-bit wide SDRAM, only SDQM[1:0] are needed for byte masking.

Internal Bank.

There are several internal memory banks on a given SDRAM row. An internal bank in a specific row cannot be activated (opened) until the previous internal bank in that row has been precharged.

The SDC does not support interleaved accesses. The bank address can be thought of as part of the row address. The SDC also assumes that all SDRAMs to which it interfaces have four internal banks. You cannot connect two-bank (16 Mbit) SDRAMs to the interface.

Do not confuse the “SDRAM internal banks” which are internal to the SDRAM and are selected with the bank address, with the four “SDRAM banks,” or “external banks,” that are enabled by the $\overline{\texttt{SMS[3:0]}}$ pins.

Mode Register.

SDRAM devices contain an internal configuration register which allows specification of the SDRAM device’s functionality. After power-up 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 power-up 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 power-up sequence and writing of the mode register needs to be done only once. Once the power-up sequence has completed, the PSSE bit should not be set again unless a change to the mode register is desired.

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 32-bit and 16-bit SDRAM banks with these formulas:

- 32-bit SDRAM banks: page size = $2^{(CAW + 2)}$

    - where CAW is the column address width of the SDRAM, plus 2 because the SDRAM bank is 32 bits wide (2 address bits = 4 bytes).

- 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. The SDC always does a Precharge All, closing all internal banks.

SDRAM banks.

The SDRAM banks are four regions of memory that can be configured to be 16 MB, 32 MB, 64 MB, or 128 MB and are selected by the $\overline{\text{SMS[3:0]}}$ pins. Each bank can be selected to be either all 32 bits wide or all 16 bits wide.

Do not confuse the “SDRAM internal banks” which are internal to the SDRAM and are selected with the bank address, with the “SDRAM banks,” or “external banks,” that are enabled by the $\overline{\text{SMS[3:0]}}$ pins.

SDRAM DIMMs.

Dual In-line Memory Modules, or DIMMs, are an industry-standard SDRAM packaging option. The DIMM consists of a small, standardized form factor board, populated with SDRAM devices on one or both sides. DIMMs are populated with sufficient SDRAM to provide either 32-bit or 64-bit data paths, with some configurations supporting additional data bits for parity protection.

Many DIMMs also include a serial presence detect port, which provides access to an on-DIMM serial ROM which includes information describing the type and characteristics of SDRAMs on the DIMM. This information can be read to determine the type, size, and timing parameters of installed DIMMs at boot time. One of the SPI or SPORT peripherals of the ADSP-BF535 processor can be used to interface to the DIMM serial-presence-detect port, if dynamic boot time determination of SDRAM configuration is required.

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 18-44.

t_{RAS}.

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. "Selecting the Bank Activate Command Delay (TRAS)" on page 18-47.

t_{RP}.

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 3 bits wide and can be programmed to be 1 to 7 clock cycles long. "Selecting the Precharge Delay (TRP)" on page 18-48.

t_{RCD}.

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 3 bits wide and can be programmed to be from 1 to 7 clock cycles long.

t_{WR}.

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 2 bits wide and can be programmed to be from 1 to 3 clock cycles long.

t_{RC}.

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_{RFC}.

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_{XSR}.

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 t_{RAS} and t_{RP} fields to ensure that $t_{RAS} + t_{RP} \geq t_{RC}$.

SDRAM Memory Global Control Register (EBIU_SDGCTL)

The SDRAM Memory Global Control register includes all programmable parameters associated with the SDRAM access timing and configuration.

Figure 18-14 shows the SDRAM Global Control register bit definitions.

SDRAM Memory Global Control Register (EBIU_SDGCTL)

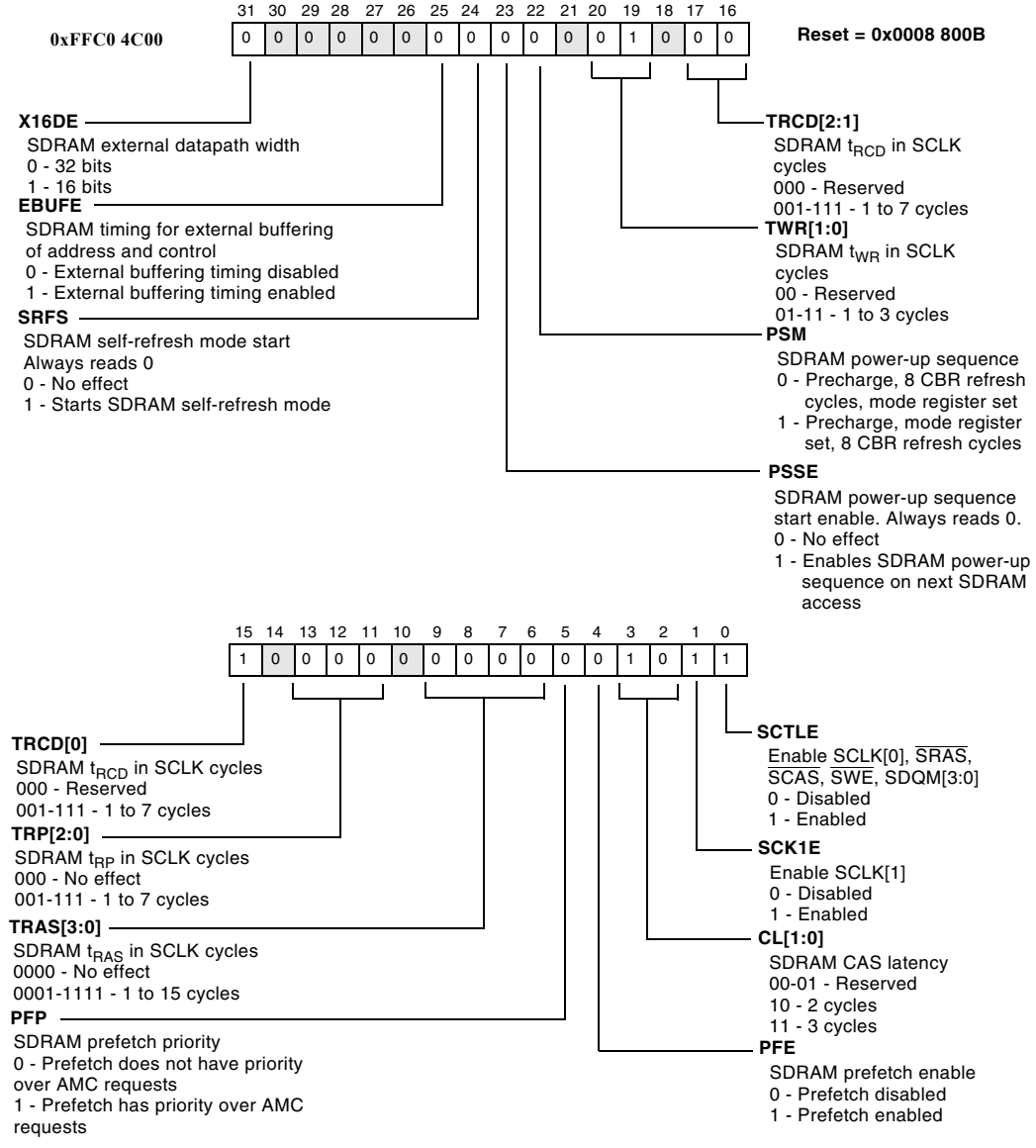


Figure 18-14. SDRAM Memory Global Control Register

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 18-8. 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.

To support higher clock load requirements, two SDRAM clock pins (SCLK[0] and SCLK[1]) are provided. The clock timing of these two pins is identical, and the clock load on SCLK[0] and SCLK[1] should be balanced. SCLK[0] runs whenever the SDC is enabled (SCTLE = 1). SCLK[1] runs whenever SCK1E is enabled. Both the SCTLE and SCK1E bits are enabled during reset, so SCLK[0] and SCLK[1] are running after reset deasserts. If the SDC will not be used, the SCTLE and SCK1E bits can be disabled to stop the clocks and reduce power dissipation. Even though the SDC is enabled (SCTLE = 1) at reset, the power-up sequence (PSSE = 1) must be executed before reading or writing to SDRAM address space.

Failure to execute the power-up sequence before reading or writing to SDRAM address space results in unpredictable operation.

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. Note that all SDRAM banks use the same timing. All timing parameters must be written with valid values based on the clock frequency and the timing specifications of the SDRAM before any access to SDRAM address space, including the power-up sequence.

The user must ensure that `tRAS + tRP >= max(tRC,tRFC,tXSR)`.

Note that these timing parameters should not be changed while the SDC is active. Therefore, since the SDC is enabled by default upon reset, the SDC must be disabled after reset to allow for modifications to the default `EBIU_SDGCTL` bit values.

The `PFE` bit is used to enable or disable the SDC read buffer. When `PFE = 1`, the SDC speculatively prefetches (reads) a subsequent cache line in order to improve performance on cache misses. Clearing the `PFE` bit causes any data in the read buffer to be invalidated and disables prefetching.

When the read buffer is enabled (`PFE = 1`), SDC prefetching can potentially take bus bandwidth away from the Asynchronous Memory Controller (AMC), because the external bus is shared between the SDC and the AMC. By clearing the `PFP` bit, the amount of bandwidth taken from the AMC is limited; when an AMC access needs the external bus, prefetching halts, and the AMC can use the pins immediately after the completion of prefetch reads that were already in progress. It should be noted that although clearing `PFP` limits the bandwidth taken from the AMC, it does not restore the AMC bandwidth to what it is when prefetching is disabled (`PFE = 0`). When `PFP` is set to 1, prefetching takes priority over AMC accesses. This means that the read buffer prefetches an entire cache line before releasing the external bus to the AMC for a pending AMC access.

Refer to "Read Buffer (Prefetch) Operation" on page 18-72 for more information.

The `PSM` and `PSSE` bits work together to specify and trigger an SDRAM power-up (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 power-up sequence:

- The PSSE bit must be set to 1 to enable the SDRAM power-up 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 power-up sequence may occur. The SDRAM power-up sequence is performed for all four SDRAM banks.

The SDRAM power-up sequence occurs and is followed immediately by the read or write transfer to SDRAM that was used to trigger the SDRAM power-up sequence. Note that there is a long latency for this first access to SDRAM because the SDRAM power-up sequence takes many cycles to complete.

Before executing the SDC power-up sequence, ensure that the SDRAM receives stable power and is clocked for the proper amount of time, as specified by the SDRAM specification.

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 18-44 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 18-45 for more information about the EBUFE bit.

The X16DE bit is used to select whether the SDRAM interface is 32 bits wide or 16 bits wide. If X16DE is 0, DATA[31:0] should be connected to the SDRAM. If X16DE is 1, DATA[15:0] should be connected to the SDRAM. Note that all SDRAM banks must be either all 32 bits wide or all 16 bits wide.

Setting the SDRAM Clock Enables (SCTLE and SCK1E)

To meet higher clock load requirements for systems with multiple SDRAM devices, the ADSP-BF535 processor provides two SDRAM clock control pins, SCLK[0] and SCLK[1]. These pins eliminate the need for off-chip clock buffers for most system memory configurations. The SCTLE and SCK1E bits in the SDRAM Memory Global Control register (EBIU_SDGCTL) provide control for the SDRAM clock control pins.

The SCTLE bit disables all of the SDRAM control pins: SDQM[3:0], $\overline{\text{SCAS}}$, $\overline{\text{SRAS}}$, $\overline{\text{SWE}}$, SCKE, and the SCLK[0].

SCTLE = 0	Disable all SDRAM control pins (control pins deasserted, SCLK[0] low)
SCTLE = 1	Enable all SDRAM control pins (SCLK[0] toggles)

The SCK1E bit disables the SCLK[1] pin independently:

SCK1E = 0	Disable SCLK[1] (SCLK[1] low)
SCK1E = 1	Enable SCLK[1] (SCLK[1] toggles)

Note the SCLK[1] function is also shared with the Asynchronous Memory Controller (AMC). Even if SCTLE and SCK1E are disabled, SCLK[1] can be enabled independently by the CLKOUT enable in the AMC (AMCKEN in the EBIU_AMGCTL register).

If the system does not use SDRAM, both SCTLE and SCK1E should be 0.

If the system uses SDRAM, but the clock load is minimal, SCTLE should be 1 and SCK1E should be 0. This setting enables the SCLK[0] pin and all related SDRAM control pins, but disables (holds low) the second clock pin SCLK[1].

If the system uses SDRAM and the clock loading is high, both SCTLE and SCK1E should be set to 1. This setting enables SCLK[0], SCLK[1], and all SDRAM control pins. In this configuration, SCLK[0] and SCLK[1] should each share half of the clock load.

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 18-8. With careful software control, the SCTLE and SCK1E bits can be used in conjunction with Self-Refresh mode to further lower the 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	No effect
SRFS = 1	Start Self-Refresh mode

When SRFS is set to 1, once the SDC enters an idle state it issues a Precharge 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 18-75 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 controller ignores another Self-Refresh request (SRFS = 1) when the SDRAM device is already in Self-Refresh mode.

The SDRAM device exits Self-Refresh mode only when the SDC receives a core, DMA, or PCI access request.

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 ADSP-BF535 processor and multiple SDRAM devices. This buffering generally consists of a register and driver.

To meet such timing requirements, 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 SCLK 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.

SDQM Operation

The SDQM[3:0] (Data I/O Mask) pins enable the SDC to mask off bytes during byte and half word (two-byte) write transfers. 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. The SDQM[x] pin should be asserted during Precharge.

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 Precharge 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.

Selecting the Bank Activate Command Delay (TRAS)

The t_{RAS} value (Bank Activate command delay) defines the required delay, in number of SCLK 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 a period of time of at least t_{RAS}. The t_{RP} and t_{RAS} values define the t_{RFC}, t_{RC}, and t_{XSR} values. See the t_{RFC}, t_{RC}, and t_{XSR} descriptions on page 18-36 for more information.

The t_{RAS} parameter allows the ADSP-BF535 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 SCLK 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 Precharge Delay (TRP)

The t_{RP} value (Precharge delay) defines the required delay, in number of SCLK 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 SCLK 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 SCLK 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 SCLK cycles may be selected. For example:

TWR = 00	Reserved
TWR = 01	1 clock cycle
TWR = 10	2 clock cycles
TWR = 11	3 clock cycles

SDRAM Memory Bank Control Register (EBIU_SDBCTL)

The SDRAM Memory Bank Control register (Figure 18-15) includes external bank-specific programmable parameters. It allows software to control some parameters of the SDRAM on a per-external-bank basis. Each external bank can be individually configured for a different size of SDRAM; however, note that all external banks use the same access timing parameters, as defined in the SDRAM Memory Global Control register (EBIU_SDGCTL). The EBIU_SDBCTL register should be programmed before power-up and should be changed only when the SDC is idle.

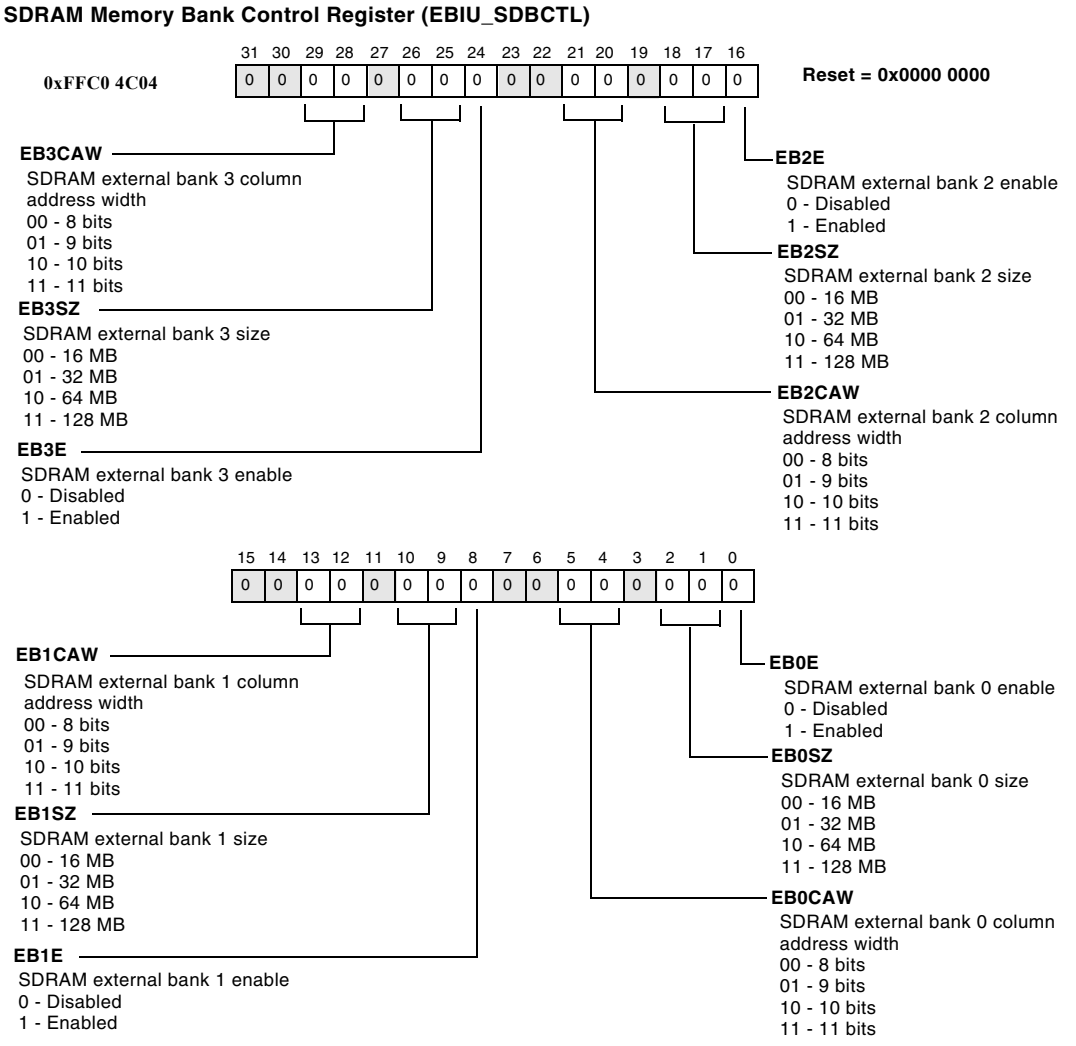


Figure 18-15. SDRAM Memory Bank Control Register

The EBIU_SDBCTL register stores the configuration information for each SDRAM bank interface. The EBIU supports 64 Mbit, 128 Mbit, 256 Mbit, and 512 Mbit SDRAM devices with x4, x8, x16 configurations. See "SDRAM External Bank Address Decode" on page 18-56 for more information on bank starting address decodes.

The SDC determines the internal SDRAM page size from the X16DE and EBxCAW parameters. Page sizes of 1 KB, 2 KB, 4 KB, and 8 KB are supported. Table 18-5 shows the page size and breakdown of the internal address (IA[31:0], as seen from the core, DMA, or PCI) into the row, bank, column, and byte address. The column address and the byte address together make up the address inside the page.

The bank address can be thought of as part of the row address. The combinations of external bank width (X16DE), external bank size (EBxSZ) and column address width (EBxCAW) which are not supported are also indicated in this table. Programming the SDC with non-supported values produces unpredictable results.

The EBxE bits in the EBIU_SDBCTL register are used to enable or disable a bank. If a bank is disabled, any access to the address space of that disabled bank generates an internal bus error, and the access does not occur externally. For more information, see "Error Detection" on page 18-8. Note that the size (EBxSZ) of all banks, regardless of whether they are enabled or disabled, is used in determining the bank starting addresses.

Table 18-5. Internal Address Mapping

Bank Width Bits	Bank Size Mbyte	Col Addr Width (CAW	Page Size kbyte	Row Address	Bank Address	Page		SDRAM Config		
						Column Address	Byte Address	Size Mbit	Width Bits	Num Chips
32	128	11	8	IA[26:15]	IA[14:13]	IA[12:2]	IA[1:0]	128	4	8
32	128	10	4	IA[26:14]	IA[13:12]	IA[11:2]	IA[1:0]	256	8	4
								512	16	2

Table 18-5. Internal Address Mapping (Cont'd)

Bank Width Bits	Bank Size Mbyte	Col Addr Width (CAW	Page Size kbyte	Row Address	Bank Address	Page		SDRAM Config		
						Column Address	Byte Address	Size Mbit	Width Bits	Num Chips
32	128	9	Not Supported - Reserved							
32	128	8								
32	64	11								
32	64	10	4	IA[25:14]	IA[13:12]	IA[11:2]	IA[1:0]	64	4	8
								128	8	4
32	64	9	2	IA[25:13]	IA[12:11]	IA[10:2]	IA[1:0]	256	16	2
32	64	8	Not Supported - Reserved							
32	32	11								
32	32	10								
32	32	9	2	IA[24:13]	IA[12:11]	IA[10:2]	IA[1:0]	64	8	4
								128	16	2
32	32	8	Not Supported - Reserved							
32	16	11								
32	16	10								
32	16	9								
32	16	8	1	IA[23:12]	IA[11:10]	IA[9:2]	IA[1:0]	64	16	2
16	128	11	4	IA[26:14]	IA[13:12]	IA[11:1]	IA[0]	256	4	4
								512	8	2
16	128	10	Not Supported - Reserved							
16	128	9								
16	128	8								
16	64	11	4	IA[25:14]	IA[13:12]	IA[11:1]	IA[0]	128	4	4
16	64	10	2	IA[25:13]	IA[12:11]	IA[10:1]	IA[0]	256	8	2
								512	16	1

Table 18-5. Internal Address Mapping (Cont'd)

Bank Width Bits	Bank Size Mbyte	Col Addr Width (CAW	Page Size kbyte	Row Address	Bank Address	Page		SDRAM Config		
						Column Address	Byte Address	Size Mbit	Width Bits	Num Chips
16	64	9	Not Supported - Reserved							
16	64	8								
16	32	11								
16	32	10	2	IA[24:13]	IA[12:11]	IA[10:1]	IA[0]	64	4	4
								128	8	2
16	32	9	1	IA[24:12]	IA[11:10]	IA[9:1]	IA[0]	256	16	1
16	32	8	Not Supported - Reserved							
16	16	11								
16	16	10								
16	16	9	1	IA[23:12]	IA[11:10]	IA[9:1]	IA[0]	64	8	2
								128	16	1
16	16	8	Not Supported - Reserved							

SDRAM Control Status Register (EBIU_SDSTAT)

The SDRAM Control Status register, shown in Figure 18-16, 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 as a debug aid. The error bits of this register are sticky. Once an error bit has been set, software must explicitly write a 1 to the bit to clear it.

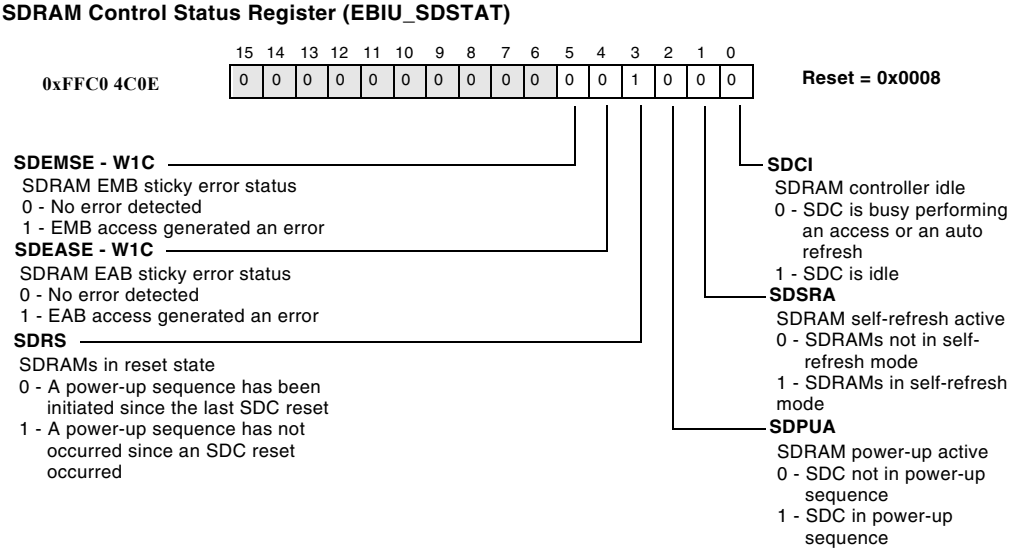


Figure 18-16. SDRAM Control Status Register

SDRAM Refresh Rate Control Register (EBIU_SDRRC)

The SDRAM Refresh Rate Control register, shown in Figure 18-17, provides a flexible mechanism for specifying the Auto-Refresh timing. The SDC provides a programmable refresh counter which has a period based on the value programmed into the RDIV field of this register, that coordinates the supplied clock rate with the SDRAM device's required refresh rate.

The delay (in number of SCLK cycles) desired 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 banks. Write the RDIV value to the EBIU_SDRRC register before the SDRAM power-up sequence is triggered. Change this value only when the SDC is idle.

SDRAM Refresh Rate Control Register (EBIU_SDRRC)

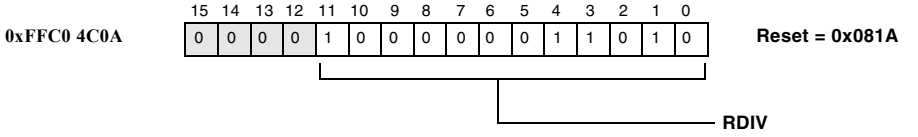


Figure 18-17. SDRAM Refresh Rate Control Register

To calculate the value that should be written to the EBIU_SDRRC register, use this equation:

$RDIV = ((f_{SCLK} \times t_{REF})/NRA) - (t_{RAS} + t_{RP})$

Where:

- f_{SCLK} = SCLK 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 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:

$$RDIV = ((133 \times 10^6 \times 64 \times 10^{-3}) / 4096) - (2 + 2) = 2074 \text{ clock cycles.}$$

This means RDIV is 0x81A (hex) and the SDRAM Refresh Rate Control register should be written with 0x081A.

Note that 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.

SDRAM External Bank Address Decode

The memory address space for each SDRAM bank is contiguous. Therefore, the starting addresses for each of the SDRAM banks and for the beginning of the unpopulated/reserved space is determined by adding up the sizes of each bank that precedes that bank. This can be done by simply summing the Bank Size Encodings listed in Table 18-6 for each bank as shown in Table 18-7. If every SDRAM bank is 128 MB, the SDRAM address space is fully populated. If any bank is less than 128 MB, the address space after bank3 is referred to as “unpopulated/reserved” space.

Any access to unpopulated/reserved space generates an internal bus error and the access does not occur externally. For more information, see “Error Detection” on page 18-8.

Table 18-6. Bank Size Encodings

EBxSZ	Bank Size (Mbyte)	bank_x_size[4:0]
00	16	00001
01	32	00010
10	64	00100
11	128	01000

Table 18-7. Start Address Calculation

SDRAM Bank	bank_x_start_addr[28:24]
0	00000
1	00000 + bank_0_size[4:0]
2	00000 + bank_0_size[4:0] + bank_1_size[4:0]
3	00000 + bank_0_size[4:0] + bank_1_size[4:0] + bank_2_size[4:0]
Unpopulated/ Reserved	00000 + bank_0_size[4:0] + bank_1_size[4:0] + bank_2_size[4:0] + bank_3_size[4:0]

Note:
1. bank_x_start_addr[31:29] = 000 and bank_x_start_addr[23:0] = all 0’s
2. If there is a carry out on the reserved space start address, then the SDRAM address space is fully populated and therefore, no bus error from unpopulated/reserved space is possible.

For example, if the SDRAM memory configuration is as follows:

- Bank0 and bank1 are connected to a 64 MByte SDRAM DIMM (EB0E=1, EB0SZ=01, EB1E=1, EB1SZ=01).
- Bank2 is unpopulated, but is set to be 16 MB (EB2E=0, EB2SZ=00).
- Bank3 is connected to a 64 MByte discrete SDRAM memory array (EB3E=0, EB3SZ=10).

```
bank_0_start_addr[28:24] =      00000

bank_1_start_addr[28:24] =      00000
                                00010      bank0
                                -----
                                00010

bank_2_start_addr[28:24] =      00000
                                00010      bank0
                                00010      bank1
                                -----
                                00100

bank_3_start_addr[28:24] =      00000
                                00010      bank0
                                00010      bank1
                                00001      bank2
                                -----
                                00101

unpopulated_start_addr[28:24] = 00000
                                00010      bank0
                                00010      bank1
                                00001      bank2
                                00100      bank3
                                -----
                                00101
```

Figure 18-18. Bank Start Address Calculation

The start addresses for the SDRAM banks and reserved spaces are calculated as:

For all banks, bank_x_start_addr[31:29] = 000 and bank_x_start_addr[32:0] = all zeros. Therefore, the bank size encodings are as shown in Table 18-8.

Table 18-8. Bank Size Encodings

SDRAM Bank	Bank Size (MB)	Start Address (hex)
0	32	0x00000000
1	32	0x02000000
2	16	0x04000000
3	64	0x05000000
Reserved	368	0x09000000

Note even though bank2 is disabled (EB2E=0), the bank size (EB2SZ=00) is used to determine the start addresses of successive banks and unpopulated/reserved space.

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, as shown in Figure 18-19. The lowest bit or bits are mapped to byte data masks, the next bits are mapped into the column address, the next 2 bits are mapped into the bank address, and the remaining bits are mapped into the row address. This mapping is based on the EBxSZ and EBxCAW parameters programmed into the SDRAM Memory Bank Control register, and the X16DE bit in the EBIU_SDGTL register.

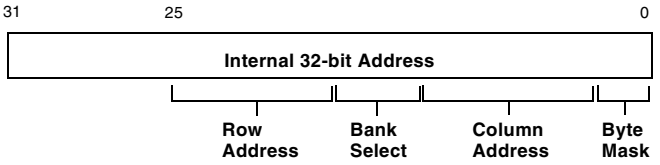


Figure 18-19. Multiplexed SDRAM Addressing Scheme

The following sections have tables which describe the muxing of the Internal Address (from the EAB or EMB buses) into the Row Address, Bank Address, and Column Address, for all the combinations of SDRAM bank width, and number of column address pins on the SDRAMs, and whether the SDRAM is 16- or 32-bits wide. This muxing assumes the SDRAMs have 4 internal banks.

All the possible configurations of SDRAM sizes and number of chips connected in parallel are given for each muxing table.

Note there is no separate ADDR[1] pin on the ADSP-BF535 processor. The ADDR[1] pin is muxed onto the pin $\overline{\text{ABE[3]}}$/$\overline{\text{SDQM[3]}}$/ADDR[1] for 16-bit SDRAM banks. This is further described in "Data Mask (SDQM[3:0]) Encodings" on page 18-67.

32-Bit Wide SDRAM Address Muxing

Table 18-9, Table 18-10, Table 18-11, and Table 18-12 show address muxing for 32-bit wide SDRAM banks using SDRAMs with 8, 9, 10, or 11 column address pins.

Table 18-9. 32-Bit SDRAM Bank, 8 Column Address Pins

ADSP-BF535 Processor Address Pin	Internal Address for RAS	Internal Address for CAS	SDRAM Address Pin
ADDR[16]	IA[26]		A[14]
ADDR[15]	IA[25]		A[13]
ADDR[14]	IA[24]		A[12]
ADDR[13]	IA[23]		A[11]
SA[10]	IA[22]		A[10]
ADDR[11]	IA[21]		A[9]
ADDR[10]	IA[20]		A[8]
ADDR[9]	IA[19]	IA[9]	A[7]
ADDR[8]	IA[18]	IA[8]	A[6]
ADDR[7]	IA[17]	IA[7]	A[5]
ADDR[6]	IA[16]	IA[6]	A[4]
ADDR[5]	IA[15]	IA[5]	A[3]
ADDR[4]	IA[14]	IA[4]	A[2]
ADDR[3]	IA[13]	IA[3]	A[1]
ADDR[2]	IA[12]	IA[2]	A[0]
ADDR[19]	IA[11]	IA[11]	BA[1]
ADDR[18]	IA[10]	IA[10]	BA[0]
Possible configurations: 2 chips: 64 Mb - 1 M × 16 × 4 banks (16 MB)			

Table 18-10. 32-Bit SDRAM Bank, 9 Column Address Pins

ADSP-BF535 Processor Address Pin	Internal Address for RAS	Internal Address for CAS	SDRAM Address Pin
ADDR[15]	IA[26]		A[13]
ADDR[14]	IA[25]		A[12]
ADDR[13]	IA[24]		A[11]
SA[10]	IA[23]		A[10]
ADDR[11]	IA[22]		A[9]
ADDR[10]	IA[21]	IA[10]	A[8]
ADDR[9]	IA[20]	IA[9]	A[7]
ADDR[8]	IA[19]	IA[8]	A[6]
ADDR[7]	IA[18]	IA[7]	A[5]
ADDR[6]	IA[17]	IA[6]	A[4]
ADDR[5]	IA[16]	IA[5]	A[3]
ADDR[4]	IA[15]	IA[4]	A[2]
ADDR[3]	IA[14]	IA[3]	A[1]
ADDR[2]	IA[13]	IA[2]	A[0]
ADDR[19]	IA[12]	IA[12]	BA[1]
ADDR[18]	IA[11]	IA[11]	BA[0]
Possible configurations: 2 chips: 256 Mb - 4 M × 16 × 4 banks (64 MB) 2 chips: 128 Mb - 2 M × 16 × 4 banks (32 MB) 4 chips: 64 Mb - 2 M × 8 × 4 banks (32 MB)			

Table 18-11. 32-Bit SDRAM Bank, 10 Column Address Pins

ADSP-BF535 Processor Address Pin	Internal Address for RAS	Internal Address for CAS	SDRAM Address Pin
ADDR[14]	IA[26]		A[12]
ADDR[13]	IA[25]		A[11]
SA[10]	IA[24]		A[10]
ADDR[11]	IA[23]	IA[11]	A[9]
ADDR[10]	IA[22]	IA[10]	A[8]
ADDR[9]	IA[21]	IA[9]	A[7]
ADDR[8]	IA[20]	IA[8]	A[6]
ADDR[7]	IA[19]	IA[7]	A[5]
ADDR[6]	IA[18]	IA[6]	A[4]
ADDR[5]	IA[17]	IA[5]	A[3]
ADDR[4]	IA[16]	IA[4]	A[2]
ADDR[3]	IA[15]	IA[3]	A[1]
ADDR[2]	IA[14]	IA[2]	A[0]
ADDR[19]	IA[13]	IA[13]	BA[1]
ADDR[18]	IA[12]	IA[12]	BA[0]
Possible configurations: 2 chips: 256 Mb - 8 M × 8 × 4 banks (128 MB) 2 chips: 512 Mb - 8 M × 16 × 4 banks (128 MB) 2 chips: 128 Mb - 4 M × 8 × 4 banks (64 MB) 2 chips: 64 Mb - 4 M × 4 × 4 banks (64 MB)			

Table 18-12. 32-Bit SDRAM Bank, 11 Column Address Pins

ADSP-BF535 Processor Address Pin	Internal Address for RAS	Internal Address for CAS	SDRAM Address Pin
ADDR[13]	IA[26]	IA[12]	A[11]
SA[10]	IA[25]		A[10]
ADDR[11]	IA[24]	IA[11]	A[9]
ADDR[10]	IA[23]	IA[10]	A[8]
ADDR[9]	IA[22]	IA[9]	A[7]
ADDR[8]	IA[21]	IA[8]	A[6]
ADDR[7]	IA[20]	IA[7]	A[5]
ADDR[6]	IA[19]	IA[6]	A[4]
ADDR[5]	IA[18]	IA[5]	A[3]
ADDR[4]	IA[17]	IA[4]	A[2]
ADDR[3]	IA[16]	IA[3]	A[1]
ADDR[2]	IA[15]	IA[2]	A[0]
ADDR[19]	IA[14]	IA[14]	BA[1]
ADDR[18]	IA[13]	IA[13]	BA[0]
Possible configurations: 8 chips: 128 Mb - 8 M × 4 × 4 banks (128 MB)			

16-Bit Wide SDRAM Address Muxing

Table 18-13, Table 18-14, and Table 18-15 show address muxing for 16-bit wide SDRAM banks using SDRAMs with 9, 10, or 11 column address pins.

Table 18-13. 16-Bit SDRAM Bank, 9 Column Address Pins

ADSP-BF535 Processor Address Pin	Internal Address for RAS	Internal Address for CAS	SDRAM Address Pin
ADDR[15]	IA[26]		A[14]
ADDR[14]	IA[25]		A[13]
ADDR[13]	IA[24]		A[12]
ADDR[12]	IA[23]		A[11]
SA[10]	IA[22]		A[10]
ADDR[10]	IA[21]		A[9]
ADDR[9]	IA[20]	IA[9]	A[8]
ADDR[8]	IA[19]	IA[8]	A[7]
ADDR[7]	IA[18]	IA[7]	A[6]
ADDR[6]	IA[17]	IA[6]	A[5]
ADDR[5]	IA[16]	IA[5]	A[4]
ADDR[4]	IA[15]	IA[4]	A[3]
ADDR[3]	IA[14]	IA[3]	A[2]
ADDR[2]	IA[13]	IA[2]	A[1]
ADDR[1]	IA[12]	IA[1]	A[0]
ADDR[19]	IA[11]	IA[11]	BA[1]
ADDR[18]	IA[10]	IA[10]	BA[0]
Possible configurations: 1 chip: 256 Mb - 4 M × 16 × 4 banks (32 MB) 1 chip: 128 Mb - 2 M × 16 × 4 banks (16 MB) 2 chips: 64 Mb - 2 M × 8 × 4 banks (16 MB)			

Table 18-14. 16-Bit SDRAM Bank, 10 Column Address Pins

ADSP-BF535 Processor Address Pin	Internal Address for RAS	Internal Address for CAS	SDRAM Address Pin
ADDR[14]	IA[26]		A[13]
ADDR[13]	IA[25]		A[12]
ADDR[12]	IA[24]		A[11]
SA[10]	IA[23]		A[10]
ADDR[10]	IA[22]	IA[10]	A[9]
ADDR[9]	IA[21]	IA[9]	A[8]
ADDR[8]	IA[20]	IA[8]	A[7]
ADDR[7]	IA[19]	IA[7]	A[6]
ADDR[6]	IA[18]	IA[6]	A[5]
ADDR[5]	IA[17]	IA[5]	A[4]
ADDR[4]	IA[16]	IA[4]	A[3]
ADDR[3]	IA[15]	IA[3]	A[2]
ADDR[2]	IA[14]	IA[2]	A[1]
ADDR[1]	IA[13]	IA[1]	A[0]
ADDR[19]	IA[12]	IA[12]	BA[1]
ADDR[18]	IA[11]	IA[11]	BA[0]
Possible configurations: 1 chip: 512 Mb - 8 M × 16 × 4 banks (64 MB) 2 chips: 256 Mb - 8 M × 8 × 4 banks (64 MB) 2 chips: 128 Mb - 4 M × 8 × 4 banks (32 MB) 4 chips: 64 Mb - 4 M × 4 × 4 banks (32 MB)			

Table 18-15. 16-Bit SDRAM Bank, 11 Column Address Pins

ADSP-BF535 Processor Address Pin	Internal Address for RAS	Internal Address for CAS	SDRAM Address Pin
ADDR[13]	IA[26]		A[12]
ADDR[12]	IA[25]	IA[11]	A[11]
SA[10]	IA[24]		A[10]
ADDR[10]	IA[23]	IA[10]	A[9]
ADDR[9]	IA[22]	IA[9]	A[8]
ADDR[8]	IA[21]	IA[8]	A[7]
ADDR[7]	IA[20]	IA[7]	A[6]
ADDR[6]	IA[19]	IA[6]	A[5]
ADDR[5]	IA[18]	IA[5]	A[4]
ADDR[4]	IA[17]	IA[4]	A[3]
ADDR[3]	IA[16]	IA[3]	A[2]
ADDR[2]	IA[15]	IA[2]	A[1]
ADDR[1]	IA[14]	IA[1]	A[0]
ADDR[19]	IA[13]	IA[13]	BA[1]
ADDR[18]	IA[12]	IA[12]	BA[0]
Possible configurations: 2 chips: 512 Mb - 16 M × 8 × 4 banks (128 MB) 4 chips: 256 Mb - 16 M × 4 × 4 banks (128 MB) 4 chips: 128 Mb - 8 M × 4 × 4 banks (64 MB)			

Data Mask (SDQM[3:0]) Encodings

During write transfers to SDRAM, the SDQM[3:0] pins are used to mask writes to bytes that are not accessed. Table 18-16 shows the SDQM[3:0] encodings for 32-bit wide SDRAM banks based on the internal transfer address bits IA[1:0] 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 32-bit SDRAM banks, SDQM[3:0] are all zeros and for 16-bit SDRAM banks, SDQM[1:0] are zeros, SDQM[2] is 1, and SDQM[3] outputs IA[1].

The only time that the SDQM[3:0] pins are high is when bytes are masked during write transfers to the SDRAM banks. At all other times, the SDQM[3:0] pins are held low (with the exception of SDQM[3] when the SDRAM banks are 16-bit wide). This means for 32-bit SDRAM banks, SDQM[3:0] are all zeros, and for 16-bit SDRAM banks SDQM[1:0] are zeros, SDQM[2] is 1, and SDQM[3] outputs IA[1].

The pin configurations in Table 18-16 are internally set up and not programmable.

Table 18-16. SDQM[3:0] Encodings During Writes for 32-bit SDRAM Banks

Internal Address IA[1:0]	Internal Transfer Size		
	byte	2 bytes	4 bytes
00	SDQM[3]=1 SDQM[2]=1 SDQM[1]=1 SDQM[0]=0	SDQM[3]=1 SDQM[2]=1 SDQM[1]=0 SDQM[0]=0	SDQM[3]=0 SDQM[2]=0 SDQM[1]=0 SDQM[0]=0
01	SDQM[3]=1 SDQM[2]=1 SDQM[1]=0 SDQM[0]=1		
10	SDQM[3]=1 SDQM[2]=0 SDQM[1]=1 SDQM[0]=1	SDQM[3]=0 SDQM[2]=0 SDQM[1]=1 SDQM[0]=1	
11	SDQM[3]=0 SDQM[2]=1 SDQM[1]=1 SDQM[0]=1		

Table 18-17 shows the SDQM[1:0] encodings during writes for 16-bit SDRAM banks.

Table 18-17. SDQM[1:0] Encodings During Writes for 16-bit SDRAM Banks

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		
Note: SDQM[3] outputs IA[1] and SDQM[2] always outputs 1 for 16-bit SDRAM banks.			

SDC Operation

The ADSP-BF535 processor's 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 the first read in a burst and for any single read transfer. Subsequent reads in a burst 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, and all external banks are refreshed at the same time. For more information, see "Executing a Parallel Refresh Command" on page 18-47.

SDC Configuration

After an ADSP-BF535 processor is reset, the SDC clocks are enabled; however, the SDC must be configured and initialized. Before programming the SDC and executing the power-up 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 power-up 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 power-up 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 (EBxSZ and EBxCAW) and to enable the external banks which are populated (EBxE). Note until the SDRAM power-up sequence has been started, any access to SDRAM address space, regardless of the state of the EBxE bits, generates an internal bus error, and the access does not occur externally. For more information, see "Error Detection" on page 18-8. After the SDRAM power-up sequence has completed, any transfer to a disabled external bank results in a hardware error interrupt, and the SDRAM transfer does not occur.

The EBIU_SDGCTL register should be written:

- to set the SDRAM cycle timing options (CL, TRAS, TRP, TRCD, TWR, EBUFE)
- to enable the SDRAM clocks (SCTLE, SCLK1)
- to set the prefetch functionality (PFE, PFP)
- to set the datapath width (X16DE)
- to select and enable the start of the SDRAM power-up 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 18-8.

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 power-up sequence. The exact sequence is determined by the PSM bit in the EBIU_SDGCTL register. The transfer that is used to trigger the SDRAM power-up sequence can be either a read or a write. This transfer occurs when the SDRAM power-up sequence has completed. This initial transfer takes many cycles to complete since the SDRAM power-up sequence must take place.

Read Buffer (Prefetch) Operation

When a cache line fill completes, the SDC launches speculative read accesses, or prefetches, in order to minimize the latency seen by L1 cache line fills. The SDC stores the read data from these prefetches into a read buffer. A cache line fill fetches 32 bytes, or 8 words, from memory. If a cache line fill hits (the address matches) at least 5 words (12 half words for 16-bit wide SDRAM), the maximum throughput of 1 word per cycle is achieved.

Prefetches start only after a cache line fill if all of these conditions are met:

- Prefetching is enabled (PFE = 1 in the SDRAM Memory Global Control register).
- The AMC does not have priority over prefetches (PFP = 1 in the SDRAM Memory Global Control register), or the AMC has priority over prefetches (PFP = 0), but the AMC is not requesting use of the shared external pins.
- The successive memory line is in the page which is currently open.
- No Auto-Refresh or Self-Refresh request is pending.

After the prefetch reads start, they continue until one of these conditions is met:

- The SDC read buffer is full.
- The AMC has a pending request and the AMC has priority over prefetches (`PFP = 0` in the SDRAM Memory Global Control register).
- An Auto-Refresh or Self-Refresh request occurs.
- Another SDC access occurs from the core, a DMA, or PCI.
- Prefetching is disabled (`PFE = 0` in the SDRAM Memory Global Control register).

If a cache line fill access starts to the address of the line stored in the read buffer, data from the read buffer starts being returned every cycle. At the same time, the SDC determines the address of the first word which is not in the read buffer and then starts this transfer from the SDRAM. As long as there are enough words that hit in the read buffer to cover the latency of reading the remaining addresses that did not hit in the read buffer, the maximum throughput can be achieved. If there are not enough hits to cover the latency, wait states are inserted in the middle of the cache line fill until read data for the remaining words can be returned. When the read accesses for the words that were not in the read buffer complete, the SDC begins launching prefetch reads of the next sequential line in memory.

If a cache line fill access starts to an address that does not match any of the addresses of the data in the read buffer (a read buffer miss), the read buffer data is invalidated, the accesses required to service the cache line fill are launched, and then prefetches of the next line begin.

The read buffer is invalidated if any of these things occur:

- A cache line fill misses the read buffer.
- A write is done to any word in the line that is stored in the read buffer.
- Prefetching is disabled (`PFE = 0` in the SDRAM Memory Global Control register).

Cache line fills always start at the address that missed in the cache. This word is referred to as the critical word. For a cache line fill to have a partial read buffer hit, the address of the critical word of the line being filled must be in the read buffer.

A prefetch may be interrupted; therefore, it is possible to have a partially filled read buffer. Words that are successfully prefetched into the read buffer are considered valid words. All valid words which follow the critical word in a line wrapping manner are counted as read buffer hits. For example, prefetch begins and stores words 7, 0, 1, 2, and 3 into the read buffer, and then it is interrupted. If the critical word of a cache line fill is word 3, then only word 3 is a read buffer hit, since word 4 is not a valid word in the read buffer. If the critical word of a cache line fill is word 7 in the same prefetch, than all five words in the read buffer (words 7, 0, 1, 2, 3) would be read-buffer hits. In this case, the maximum throughput of 1 word per cycle is achieved.

Prefetch accesses are started only after a cache line fill access occurs. The address of the first prefetch read is always the corresponding address of the current critical word in the next line. For example, if a cache line fill access starts with word 5, then the next prefetch would be from word 5 in the following line.

SDC Commands

This section provides a description of each of the commands that the SDC uses to manage the SDRAM interface. These commands are handled automatically by the SDC. A summary of the various commands used by the on-chip controller for the SDRAM interface is as follows:

- Precharge: Closes all internal banks.
- Activate: Activates a page in the required SDRAM internal bank.
- Load Mode Register: Initializes the SDRAM operation parameters during the power-up 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 18-18 shows the SDRAM pin state during SDC commands.

Table 18-18. Pin State During SDC Commands

Command	$\overline{\text{SMS[x]}}$	$\overline{\text{SCAS}}$	$\overline{\text{SRAS}}$	$\overline{\text{SWE}}$	SCKE	SA10
Precharge	low	high	low	low	high	high
Bank Activate	low	high	low	high	high	
Load Mode Register	low	low	low	low	high	
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 Command

The Precharge command is given to precharge an active internal bank. The SDC always issues the Precharge command to all internal SDRAM banks (this is controlled by asserting the SA10 pin high). The Precharge command is executed by the SDC if the address to be accessed falls in a different page in the same external bank or before an Auto-Refresh or Self-Refresh. A Precharge is not done if the address to be accessed falls in an open page in another external bank or to an external bank with no page open. For page miss reads or writes, only the external bank to be accessed by the read or write is precharged. For Auto-Refresh and Self-Refresh, all external SDRAM banks are precharged at one time.

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. Only one SDRAM internal bank in each external bank may be active at a time.

Load Mode Register Command

The Load Mode Register command initializes SDRAM operation parameters. This command is a part of the SDRAM power-up sequence. Load Mode Register uses the address bus of the SDRAM as data input. The power-up 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 power-up sequence. The exact order of the power-up 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 zero
- Wrap type = sequential, bit 3, always zero
- Ltmode = latency mode (CAS latency), bits 6-4, programmable in the EBIU_SDGCTL register
- Bits 14-7, always zero

While executing the Load Mode Register command, the unused address pins are set to zero. During the two SCLK cycles following Load Mode Register, the SDC issues only NOP commands.

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.

In cases where the internal bus transfer is an L1 cache line fill burst-read operation, the SDC attempts to service the line fill with data stored in the read buffer. For each part of the line fill not available in the read buffer, the SDC issues a Read command. The SDC then speculatively issues Read commands if certain conditions are met. For more information, see “Read Buffer (Prefetch) Operation” on page 18-72.

The SDC never speculatively issues a write command to the SDRAM.

In the case of a page miss, the SDRAM is precharged and activated before issuing the Read or Write command. If the internal refresh counter asserts a refresh request, any new access is delayed until the Auto-Refresh cycle completes.

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 all the external banks whether or not they are enabled (EBxE in the SDRAM Memory Global Control register). Before executing the Auto-Refresh command, the SDC executes a Precharge All command to all external banks. 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 power-up sequence and 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 external 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). 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 $2 \times (t_{RC} + t_{RP})$.

Note the SCLK[0] and SCLK[1] are not disabled by the SDC during Self-Refresh mode. However, software may disable the clocks by clearing the SCTLE and SCK1E bits 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 18-8.

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 any SDRAM external banks, the Command Inhibit command is given.

SDRAM Timing Specifications

To support key timing requirements and power-up sequences for different SDRAM vendors, the SDC provides programmability for t_{RAS}, t_{RP}, t_{RCD}, t_{WR} and the power-up sequence mode. (For more information, see “SDRAM Memory Global Control Register (EBIU_SDGCTL)” on page 18-37.) CAS latency should be programmed in the EBIU_SDGCTL register based on the frequency of operation. (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} = 2 SCLK cycles

SDRAM Performance

Table 18-19 lists the data throughput rates for the core, DMA, or PCI read/write accesses to 32-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)

Table 18-19. Throughput for Accesses to 32-Bit Wide SDRAM

Accesses	Read/Write	Page Hit/Miss[1]	Throughput
L1 Cache Line Fill (≥ 5 word Read Buffer Hit)	Read	Page Hit	8 words/8 cycles
L1 Cache Line Fill (N word Read Buffer Hit, where N < 5)	Read	Page Hit	N word hit: 8 words/(13-N) cycles
L1 Cache Line Fill (Read Buffer Miss)	Read	Page Hit	8 words/13 cycles
L1 Cache Copyback	Write	Page Hit	8 words/16 cycles
DMA Burst Read (4 words)	Read	Page Hit	4 words/9 cycles
DMA Burst Write (4 words)	Write	Page Hit	4 words/8 cycles
DMA Burst Read (8 words)	Read	Page Hit	8 words/13 cycles
DMA Burst Write (8 words)	Write	Page Hit	8 words/16 cycles
Single Read	Read	Page Hit	1 word/6 cycles
Single Write	Write	Page Hit	1 word/2 cycles
L1 Cache Line Fill (Read Buffer Miss)	Read	Page Miss	8 words/17 cycles
L1 Cache Copyback	Write	Page Miss	8 words/20 cycles
DMA Burst Read (4 words)	Read	Page Miss	4 words/13 cycles
DMA Burst Write (4 words)	Write	Page Miss	4 words/12 cycles
DMA Burst Read (8 words)	Read	Page Miss	8 words/17 cycles
DMA Burst Write (8 words)	Write	Page Miss	8 words/20 cycles
Single Read	Read	Page Miss	1 word/10 cycles
Single Write	Write	Page Miss	1 word/6 cycles
1 Page Miss Penalty = t_{RP} + t_{RCD}			

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. Accesses that hit in the Read Buffer follow these rules:

If either EBUFE = 1 or CL = 11:

- For N = 8 to 6, N half word hit: 8 words/8 cycles
- For N = 5 to 0, N half word hit: 8 words/(14 – N) cycles

If both EBUFE = 1 and CL = 11:

- For N = 8 to 7, N half word hit: 8 words/8 cycles
- For N = 6 to 0, N half word hit: 8 words/(15 – N) cycles

Table 18-20 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)

Table 18-20. Throughput for Accesses to 16-Bit Wide SDRAM

Accesses	Read/ Write	Page Hit/Miss[1]	Throughput
L1 Cache Line Fill (≥ 12 half word Read Buffer Hit)	Read	Page Hit	8 words/8 cycles
L1 Cache Line Fill (N half word Read Buffer Hit, where N < 12)	Read	Page Hit	N half word hit: 8 words/(21-N) cycles
L1 Cache Line Fill (Read Buffer Miss)	Read	Page Hit	8 words/21 cycles
L1 Cache Copyback	Write	Page Hit	8 words/24 cycles
DMA Burst Read (4 words)	Read	Page Hit	4 words/13 cycles
DMA Burst Write (4 words)	Write	Page Hit	4 words/12 cycles
DMA Burst Read (8 words)	Read	Page Hit	8 words/21 cycles
DMA Burst Write (8 words)	Write	Page Hit	8 words/20 cycles
Single Read	Read	Page Hit	1 word/7 cycles
Single Write	Write	Page Hit	1 word/3 cycles
L1 Cache Line Fill (Read Buffer Miss)	Read	Page Miss	8 words/25 cycles
L1 Cache Copyback	Write	Page Miss	8 words/28 cycles
DMA Burst Read (4 words)	Read	Page Miss	4 words/17 cycles
DMA Burst Write (4 words)	Write	Page Miss	4 words/16 cycles
DMA Burst Read (8 words)	Read	Page Miss	8 words/25 cycles
DMA Burst Write (8 words)	Write	Page Miss	8 words/24 cycles
Single Read	Read	Page Miss	1 words/11 cycles
Single Write	Write	Page Miss	1 words/7 cycles
1 Page-Miss Penalty = t_{RP} + t_{RCD}			

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. Accesses that hit in the Read Buffer follow these rules:

If either `EBUFE = 1` or `CL = 11`:

- For N = 16 to 13, N half word hit: 8 words/8 cycles
- For N = 12 to 0, N half word hit: 8 words/(22 – N) cycles

If both `EBUFE = 1` and `CL = 11`:

- For N = 16 to 14, N half word hit: 8 words/8 cycles
- For N = 13 to 0, N half word hit: 8 words/(23 – N) cycles

19 SYSTEM DESIGN

This chapter provides hardware, software and system design information to aid users in developing systems based on the ADSP-BF535 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 *ADSP-BF535 Blackfin Embedded Processor Data Sheet* for pin information, including pin numbers for the 260-Lead PBGA package.

Recommendations for Unused Pins

Refer to *ADSP-BF535 Blackfin Embedded Processor Data Sheet* for detailed pin descriptions.

Pins Requiring Termination

ADSP-BF535 Blackfin Embedded Processor Data Sheet provides guidelines on pins that require termination during normal operation of the ADSP-BF535 processor.

Resetting the Processor

In addition to the hardware reset mode provided via the $\overline{\texttt{RESET}}$ pin, the ADSP-BF535 processor supports several software reset modes. For detailed information on the various modes, see "System Reset and Power-up Configuration" on page 3-12.

The processor state of the ADSP-BF535 processor after reset is described in "Reset State" on page 3-10.

Booting the Processor

The ADSP-BF535 processor can be booted via a variety of methods. These include executing from external 16-bit memory, or booting from a ROM configured to load code from 8-bit FLASH memory or a serial ROM (8-bit or 16-bit address range). For more information on boot modes, see "Booting Methods" on page 3-17.

Managing Clocks

Systems can drive the ADSP-BF535 processor's 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 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 ADSP-BF535 processor produces a 1x-31x multiplication of the clock input provided on the CLKIN pin to generate the core clock (CCLK). Additionally, the system clock (SCLK) is derived from the core clock. This clock is based on a divider ratio that is programmed via the SSEL bit settings. For detailed information about how to set and change CCLK and SCLK frequencies, see "Dynamic Power Management" on page 8-1.

Designing for Multiplexed Clock Pins

The ADSP-BF535 processor's MSEL6-0 pins are multiplexed with the PF6-0 pins; SSEL1-0 pins are multiplexed with the PF9-8 pins. During reset, these pins act as multiplier selects when in multiplier mode and as programmable flags after reset. This multiplexing influences system design as follows.

- For systems selecting Bypass mode during reset, MSELx pin states do not need to be managed during reset. The multiplexed nature of these pins does not influence system design for the PFx pins.
- For systems using Multiplier mode during reset and not using PFx pins at runtime, use pull-up or pull-down resistors to select the MSELx and SSELx values. Do not leave these pins unconnected.
- For systems using Multiplier mode during reset and using the PFx pins at runtime, use pull-up or pull-down resistors to select the MSELx values during reset. Ensure that the system permits the MSELx and SSELx pins to stabilize to a valid multiplier value in compliance with the timing for $\overline{\text{RESET}}$ in the data sheet.

Figure 19-1 shows an example clock pin configuration.

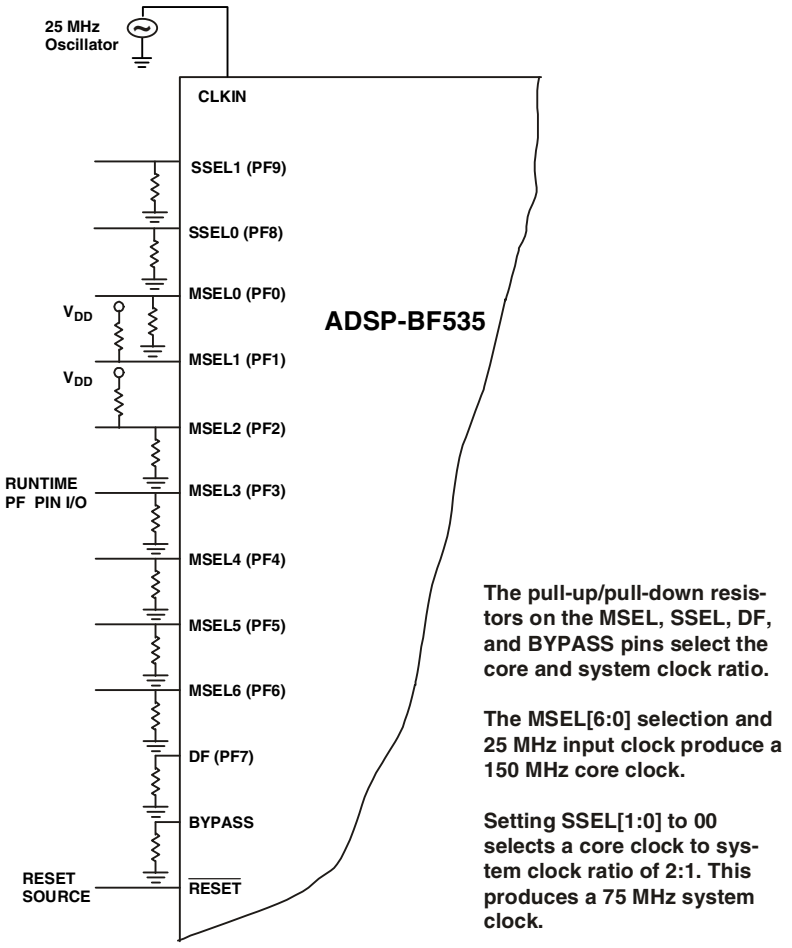


Figure 19-1. External Clock Connections

All of the core and peripheral clocks are enabled during reset. This results in a momentary power surge after reset. To avoid this condition, use the MSEL pins to select a lower processor clock (CCLK) speed upon reset. Adjust the clock to the desired operating speed in the software.

The timing for the SSELx, MSELx, BYPASS, and DF pins during reset is identical and has these features (as shown in Figure 19-2).

- t_{PFD}—Delay from RESET asserted to PFx input or output is terminated. From this point, the pin values begin stabilizing to a valid state.
- t_{MSD}—Delay from RESET asserted to the appearance of valid data values on the pin. The values can change from this point, but only from one valid value to another.
- t_{MSS}—Setup for pin value before RESET deasserted. The value must be held from this point until hold time completes.
- t_{MSH}—Hold for pin value after RESET deasserted.

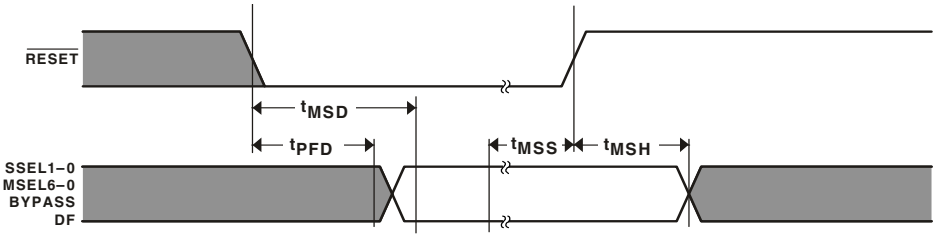


Figure 19-2. SSELx, MSELx, BYPASS, and DF Timing

Configuring and Servicing Interrupts

A variety of interrupts are available on the ADSP-BF535 processor. They include both core and peripheral interrupts. The ADSP-BF535 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 4-29.

The ADSP-BF535 processor supports nested and non-nested interrupts, as well as self-nested interrupts. For explanations of the various modes of servicing events, see "Interrupts With and Without Nesting" on page 4-48.

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 signaling 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 byte at memory location pointed to by a P-register.
- Sets `CC` if the value is equal to zero.
- Stores the value back in its original location (but with the most significant bit 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` immediately after semaphore release.

The TESTSET instruction can be used to implement binary semaphores or any other type of mutual exclusion method (for example, counting semaphores). TESTSET represents a system-level requirement for a multicycle bus lock mechanism.

The ADSP-BF535 processor restricts use of the TESTSET instruction to the L2 memory and external memory regions only. Use of the TESTSET instruction to address any other area of the ADSP-BF535 memory map may result in unreliable behavior.

Example Code for Query Semaphore

Listing 19-1 provides an example of a query semaphore that checks the availability of a shared resource.

Listing 19-1. Query Semaphore

```
/* Query semaphore. Denotes "Busy" if its value is non-zero. Wait
until free (or re-schedule 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 non-zero. 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/PCI buses, refer to "Chip Bus Hierarchy" on page 7-1.

Bus Priorities

For an explanation of prioritization between the various internal SBIU buses, refer to "Chip Bus Hierarchy" on page 7-1.

PCI Arbiter

If the ADSP-BF535 processor is intended for use as a PCI host in a given application, an external PCI arbiter chip is required to connect to the on-chip PCI interface. There is no additional circuitry necessary in order for the ADSP-BF535 processor to act as a slave on the PCI bus.

USB Device Connection

The ADSP-BF535 processor's USB peripheral supports USB device mode, but not USB host mode. For USB device operation, use an external USB buffer/driver chip and oscillator.

External Memory Design Issues

This section describes design issues related to external memory including supported SDRAM configurations, examples of SDRAM and SRAM interfaces, and how to avoid bus contention on the external memory bus.

Example Asynchronous Memory Interfaces

This section shows glueless connections to 32-bit and 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 19-3 shows the glueless interface to 32-bit SRAM. Note that this application requires the 32-bit datapath mode to be enabled for this bank of memory in the Asynchronous Memory Global Control Register.

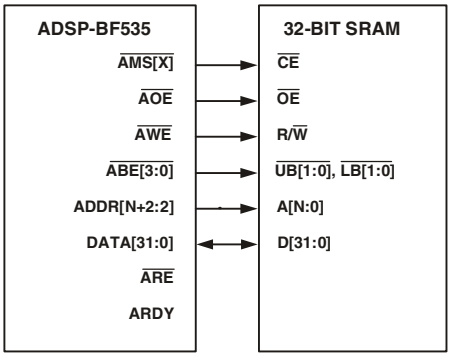


Figure 19-3. Interface to 32-bit SRAM

Figure 19-4 shows the system interconnect required to support 16-bit memories. Note this application requires that the 16-bit packing mode be enabled for this bank of memory. Otherwise, the programming model must ensure that every other 16-bit memory location is accessed starting on an even (byte address[1:0]=00) 16-bit address.

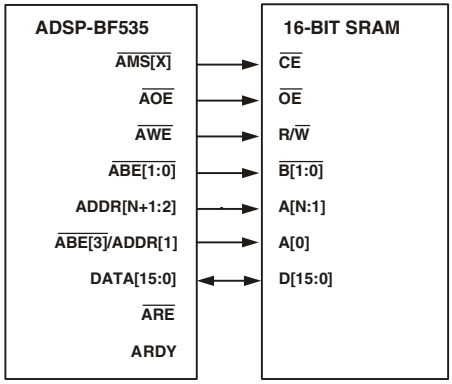


Figure 19-4. Interface to 16-bit SRAM

Avoiding Bus Contention

Because the tristatable 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 tristate 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 ADSP-BF535 processor's 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 EBIU provides one cycle for the transition to occur.

Supported SDRAM Configurations

Table 19-1 shows all possible bank sizes, bank widths and SDRAM discrete component configurations that can be gluelessly interfaced to the SDC.

Table 19-1. SDRAM Discrete Component Configurations Supported

Bank Size (MByte)	Bank Width (Bits)	SDRAM			
		Size (Mbit)	Configuration	Number of Chips	Number of Column Address Pins
16	32	64	1M × 16 × 4banks	2	8
32	32	64	2M × 8 × 4banks	4	9
32	32	128	2M × 16 × 4banks	2	9
64	32	64	4M × 4 × 4banks	8	10
64	32	128	4M × 8 × 4banks	4	10
64	32	256	4M × 16 × 4banks	2	9
128	32	128	8M × 4 × 4banks	8	11
128	32	256	8M × 8 × 4banks	4	10
128	32	512	8M × 16 × 4banks	2	10
16	16	64	2M × 8 × 4banks	2	9
16	16	128	2M × 16 × 4banks	1	9
32	16	64	4M × 4 × 4banks	4	10
32	16	128	4M × 8 × 4banks	2	10
32	16	256	4M × 16 × 4banks	1	9
64	16	128	8M × 4 × 4banks	4	11

Table 19-1. SDRAM Discrete Component Configurations Supported (Cont'd)

Bank Size (MByte)	Bank Width (Bits)	SDRAM			
		Size (Mbit)	Configuration	Number of Chips	Number of Column Address Pins
64	16	256	8M × 8 × 4banks	2	10
64	16	512	8M × 16 × 4banks	1	10
128	16	256	16M × 4 × 4banks	4	11
128	16	512	16M × 8 × 4banks	2	11

Example SDRAM Interfaces

Figure 19-5 shows a block diagram of the ADSP-BF535 processor's SDRAM interface. In this example, the SDRAM interface connects to four 64 Mbit (x8) SDRAM devices, to form one external bank of 32 Mbytes of memory. The same address and control bus feeds all four SDRAM devices.

The ADSP-BF535 processor's SDRAM controller supports industry standard 32-bit and 64-bit wide data bus DIMMs. These DIMMs generally have multiple chip select inputs with each connected to a portion of the chips on the DIMM. Most DIMMs can be interfaced to the ADSP-BF535 processor gluelessly by using the $\overline{\text{SMS}}$[3:0] bank selects, the SDQM[3:0] data masks, and careful connection of the data bus.

Divide the DIMM into multiple banks and connect it based on the SDRAM chip enables. Since the data bus is only 32 bits wide at the most, 64-bit DIMMs must be connected as two banks. Note the range of numbers for the DQM and DATA signals on the SDRAM DIMM may not match the corresponding range on the ADSP-BF535 processor.

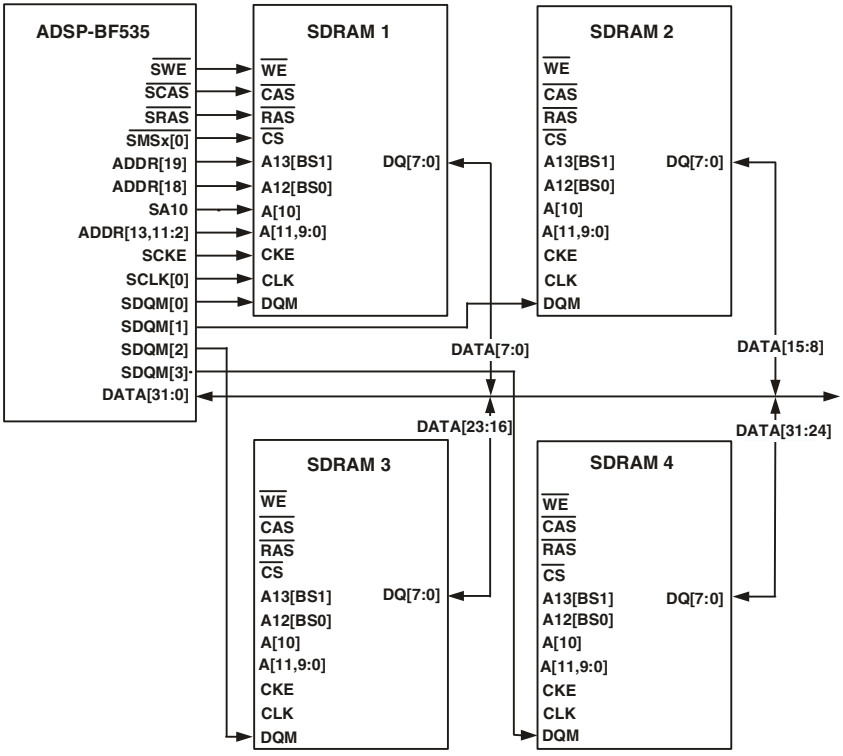


Figure 19-5. 64 MB SDRAM System Example

The SDRAM DIMM system example in Figure 19-6 shows a glueless interface to a 64 MB DIMM. The DIMM is connected to the ADSP-BF535 processor as bank 0 ($\overline{\text{SMS}}$[0]) and bank 1 ($\overline{\text{SMS}}$[1]) with each bank having 32 MB.

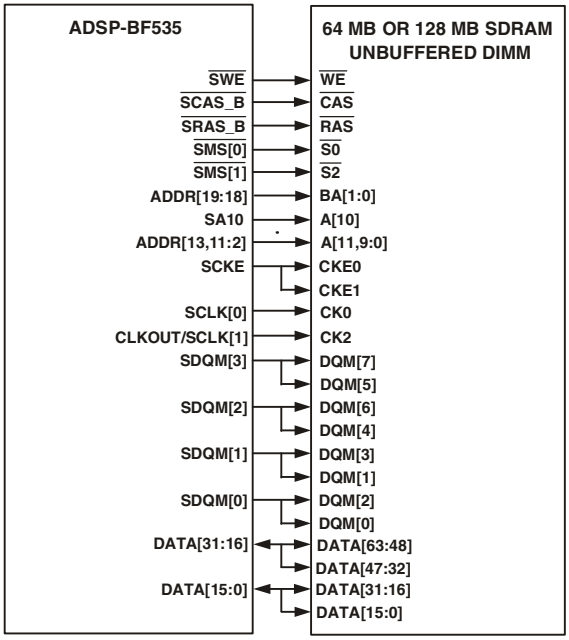


Figure 19-6. SDRAM DIMM System Example

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 processor systems.

Point-to-Point Connections on Serial Ports

Although the processor's serial ports may be operated at a slow rate, the output drivers still have fast edge rates and for longer distances 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 six inches. For details, see the reference source in "Recommended Reading" on page 19-17 for suggestions on transmission-line termination. Also, see *ADSP-BF535 Blackfin Embedded Processor Data Sheet* for output drivers' rise and fall time data.

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 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 19-7. 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 shown in Figure 19-6 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.

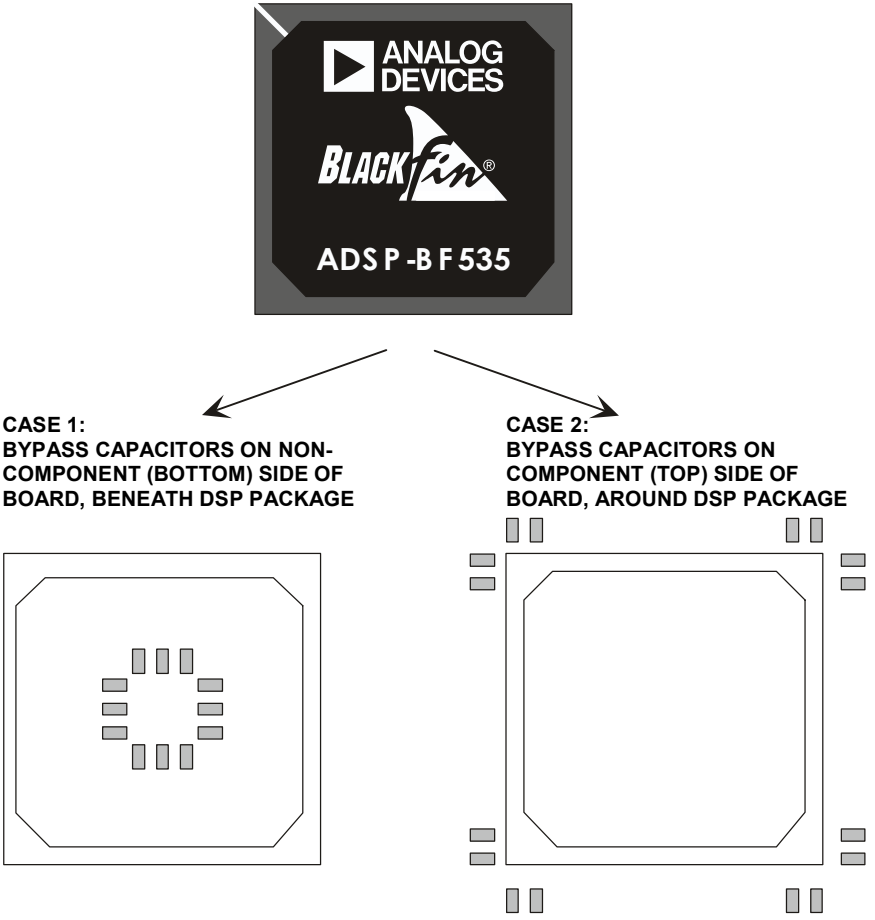


Figure 19-7. Bypass Capacitor Placement

Recommended Reading

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 & Layer Stacking
- Terminations
- Vias
- Power Systems
- Connectors
- Ribbon Cables
- Clock Distribution
- Clock Oscillators

20 BLACKFIN PROCESSOR'S DEBUG

The Blackfin processor's debug functionality can be used for software debugging. It also complements some services often found in an operating system (OS) kernel. The functionality is implemented in the processor hardware and is grouped into multiple levels.

A summary of available debug features is shown in Table 20-1.

Table 20-1. Blackfin Debug Features

Debug Feature	Description
Watchpoints	Specify address ranges and conditions that halt the processor when satisfied.
Trace History	Stores the last 16 discontinuous values of the Program Counter in an on-chip trace buffer.
Cycle Count	Provides functionality for all code profiling functions.
Performance Monitoring	Allows internal resources to be monitored and measured non-intrusively.

Watchpoint Unit

By monitoring the addresses on both the instruction bus and the data bus, the Watchpoint Unit provides several mechanisms for examining program behavior. After counting the number of times a particular address is matched, the unit schedules an event based on this count.

In addition, information that the Watchpoint Unit can provide helps optimize code. The unit can also make it easier to maintain executables through code patching.

The Watchpoint Unit contains these MMRs, which are accessible in Supervisor and Emulator modes:

- The Watchpoint Status register (WPSTAT)
- Six Watchpoint Instruction Address registers (WPIA[5:0])
- Six Watchpoint Instruction Address Count registers (WPIACNT[5:0])
- The Watchpoint Instruction Address Control register (WPIACTL)
- Two Watchpoint Data Address registers (WPDA[1:0])
- Two Watchpoint Data Address Count registers (WPDACNT[1:0])
- The Watchpoint Data Address Control register (WPDACTL)

Two operations implement instruction watchpoints:

- The values in the six Watchpoint Instruction Address registers, WPIA[5:0], are compared to the address on the instruction bus.
- Corresponding count values in the Watchpoint Instruction Address Count registers, WPIACNT[5:0], are decremented on each match.

The six Watchpoint Instruction Address registers may be further grouped into three ranges of instruction-address-range watchpoints. The ranges are identified by the addresses in WPIA0 to WPIA1, WPIA2 to WPIA3, and WPIA4 to WPIA5.

The address ranges stored in WPIA0, WPIA1, WPIA2, WPIA3, WPIA4 and WPIA5 must satisfy these conditions:

- WPIA0 <= WPIA1
- WPIA2 <= WPIA3
- WPIA4 <= WPIA5

Two operations implement data watchpoints:

- The values in the two Watchpoint Data Address registers, WPDA[1:0], are compared to the address on the data buses.
- Corresponding count values in the Watchpoint Data Address Count registers, WPDACNT[1:0], are decremented on each match.

The two Watchpoint Data Address registers may be further grouped together into one data-address-range watchpoint, WPDA[1:0].

The instruction and data count value registers must be loaded with one less than the number of times the watchpoint must match. After the count value reaches zero, the subsequent watchpoint match results in an exception or emulation event.

Note count values must be reinitialized after the event has occurred.

An event can also be triggered on a combination of the instruction and data watchpoints. If the WPAND bit in the WPIACTL register is set, then an event is triggered only when both an instruction address watchpoint matches *and* a data address watchpoint matches. If the WPAND bit is 0, then an event is triggered when any of the enabled watchpoints or watchpoint ranges match.

To enable the Watchpoint Unit, the WPPWR bit in the WPIACTL register must be set. If WPPWR=1, then the individual watchpoints and watchpoint ranges may be enabled using the specific enable bits in the WPIACTL and WPDACTL MMRs. If WPPWR=0, then all watchpoint activity is disabled.

Instruction Watchpoints

Each instruction watchpoint is controlled by three bits in the WPIACTL register, as shown in Table 20-2.

Table 20-2. WPIACTL Control Bits

Bit Names	Description
EMUSWx	Determines whether an instruction-address match causes either an emulation event or an exception event.
WPICNTENx	Enables the 16-bit counter that counts the number of address matches. If the counter is disabled, then every match causes an event.
WPIAENx	Enables the address watchpoint activity.

When two watchpoints are associated to form a range, two additional bits are used, as shown in Table 20-3.

Table 20-3. WPIACTL Watchpoint Range Control Bits

Bit Names	Description
WPIRENxy	Indicates the two watchpoints that are to be associated to form a range.
WPIRINVxy	Determines whether an event is caused by an address within the range identified or outside of the range identified.

Code patching allows software to replace sections of existing code with new code. The watchpoint registers are used to trigger an exception at the start addresses of the earlier code. The exception routine then vectors to the location in memory that contains the new code.

On the ADSP-BF535 processor, code patching can be achieved by writing the start address of the earlier code to one of the WPIAx registers and setting the corresponding EMUSWx bit to trigger an exception. In the exception service routine, the WPSTAT register is read to determine which watchpoint triggered the exception. Next, the code writes the start address of the new code in the RETX register and then returns from the exception to the new code. Because the exception mechanism is used for code patching, event service routines of the same or higher priority (exception, NMI, and reset routines) cannot be patched.

A write to the WPSTAT MMR clears all the sticky status bits. The data value written is ignored.

Watchpoint Instruction Address Registers (WPIAx)

When the Watchpoint Unit is enabled, the values in the WPIAx registers are compared to the address on the instruction bus. Corresponding count values in the Watchpoint Instruction Address Count registers (WPIACNTx) are decremented on each match.

Figure 20-1 shows the Watchpoint Instruction Address registers, WPIA[5:0].

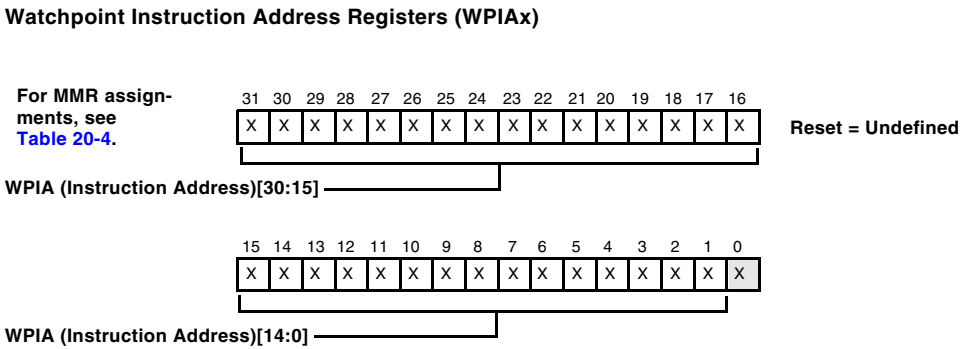


Figure 20-1. Watchpoint Instruction Address Registers

Table 20-4. Watchpoint Instruction Address Register MMR Assignments

Register Name	Memory-Mapped Address
WPIA0	0xFFE0 7040
WPIA1	0xFFE0 7044
WPIA2	0xFFE0 7048
WPIA3	0xFFE0 704C
WPIA4	0xFFE0 7050
WPIA5	0xFFE0 7054

Watchpoint Instruction Address Count Registers (WPIACNTx)

When the Watchpoint Unit is enabled, the count values in the Watchpoint Instruction Address Count registers (`WPIACNT[5:0]`) are decremented each time the address or the address bus matches a value in the `WPIAx` registers. Load the `WPIACNTx` register with a value that is one less than the number of times the watchpoint must match before triggering an event (see Figure 20-2).

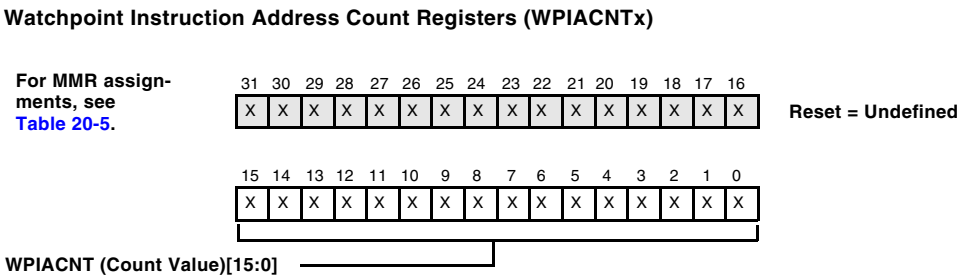


Figure 20-2. Watchpoint Instruction Address Count Registers

Table 20-5. Watchpoint Instruction Address Count Register MMR Assignments

Register Name	Memory-Mapped Address
WPIACNT0	0xFFE0 7080
WPIACNT1	0xFFE0 7084
WPIACNT2	0xFFE0 7088
WPIACNT3	0xFFE0 708C
WPIACNT4	0xFFE0 7090
WPIACNT5	0xFFE0 7094

Watchpoint Instruction Address Control Register (WPIACTL)

Three bits in the Watchpoint Instruction Address Control register (`WPIACTL`) control each instruction watchpoint.

The bits in the `WPIACTL` register have no effect unless the `WPPWR` bit is set.

Figure 20-3 shows the upper half of the register, and Figure 20-4 shows the lower half of the register. For more information about the bits in this register, see "Instruction Watchpoints" on page 20-4.

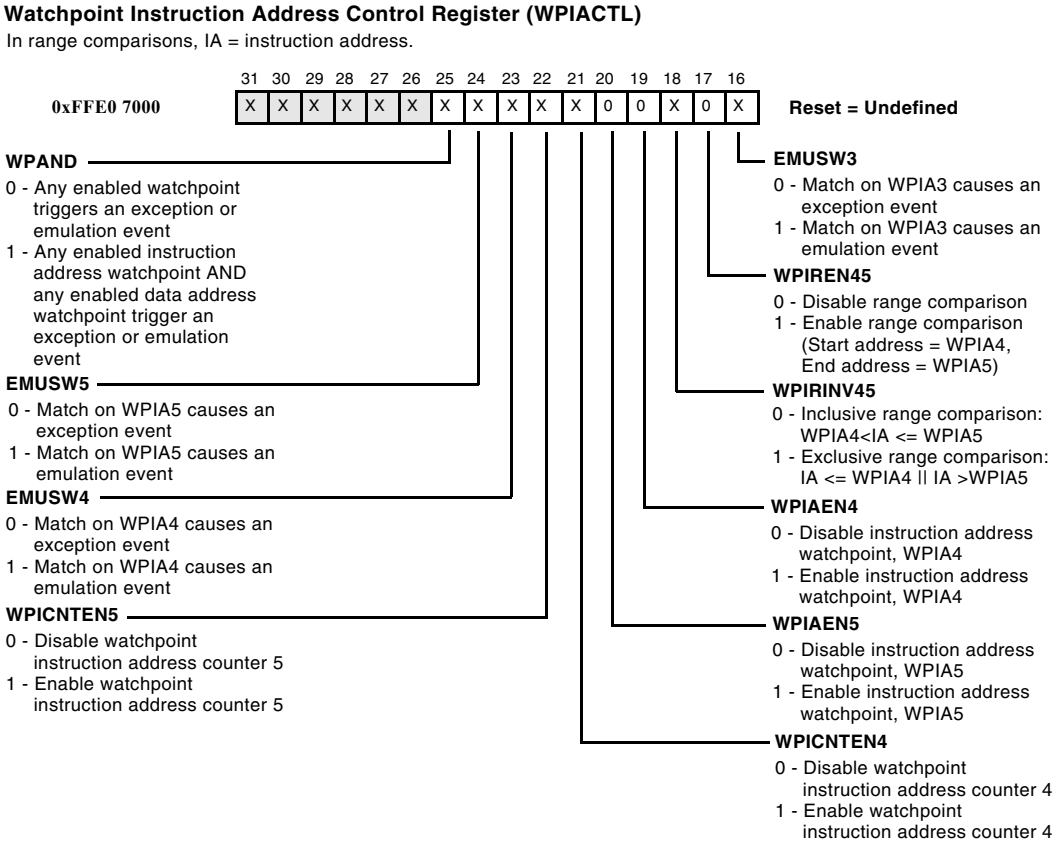


Figure 20-3. Watchpoint Instruction Address Control Register [31:16]

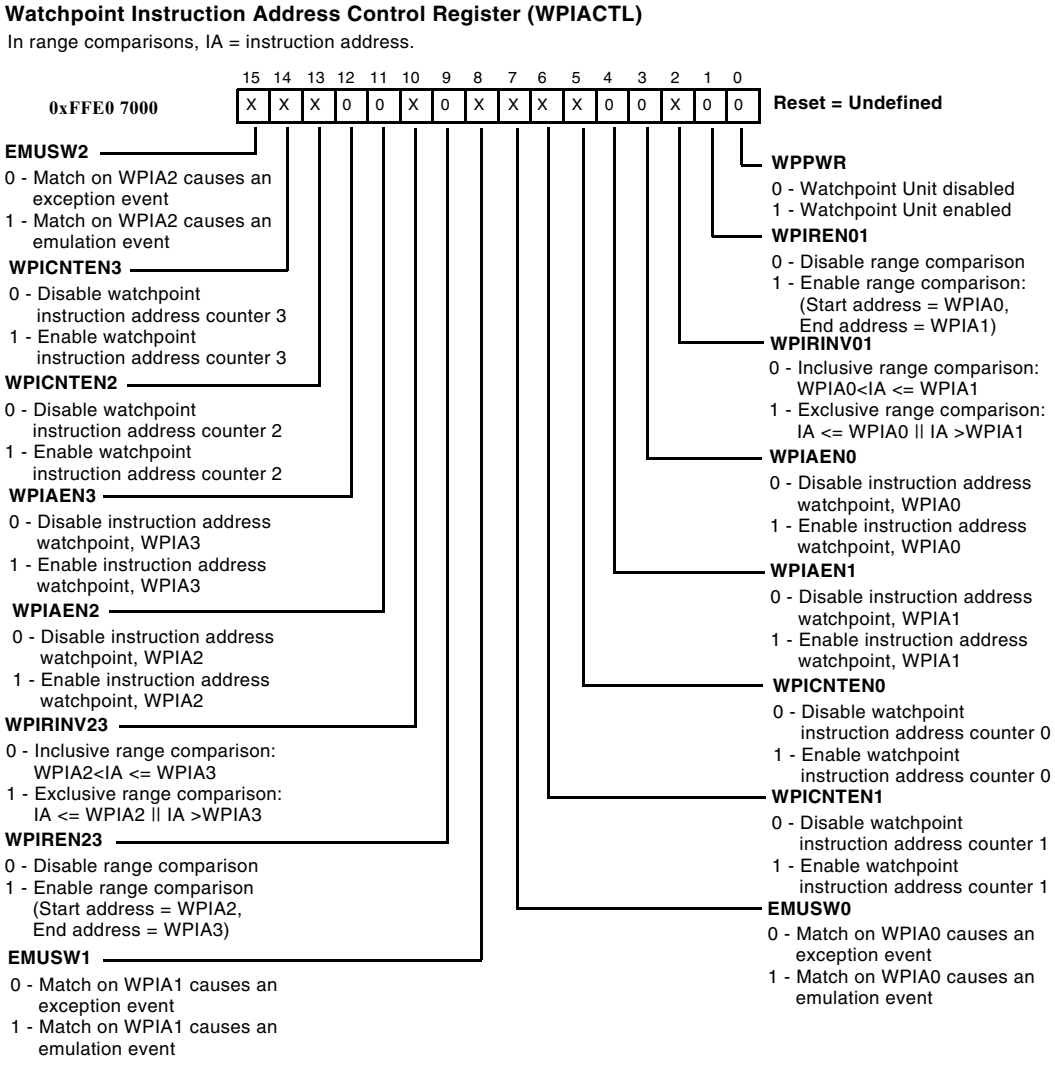


Figure 20-4. Watchpoint Instruction Address Control Register [15:0]

Data Address Watchpoints

Each data watchpoint is controlled by four bits in the WPDACTL register, as shown in Table 20-6.

Table 20-6. Data Address Watchpoints

Bit Names	Description
WPDACCx	Determines whether the match should be on a read or write access.
WPDSRCx	Determines which DAG the unit should monitor.
WPDCNTENx	Enables the counter that counts the number of address matches. If the counter is disabled, then every match causes an event.
WPDAENx	Enables the data watchpoint activity.

Alternatively, the watchpoint unit can be configured to monitor a range of data addresses. To enable this function, the WPDAEN0 and WPDAEN1 bits are not used and must be set to 0. Instead, the WPDREN01 and WPDRINV01 bits are used to configure the watchpoint unit, as described in Table 20-7.

Table 20-7. WPDACTL Watchpoint Control Bits

Bit Name	Description
WPDREN01	Indicates the two watchpoints that are to be associated to form a range.
WPDRINV01	Determines whether an event is caused by an address within the range identified or outside the range.

Note Data address watchpoints always trigger emulation events.

To enable the Data address watchpoints, the WPPWR bit of the WPIACTL register must be set to 1.

Watchpoint Data Address Registers (WPDAx)

When the Watchpoint Unit is enabled, the values in the WPDAx registers are compared to the address on the data buses. Corresponding count values in the Watchpoint Data Address Count registers (WPDACNTx) are decremented on each match.

Figure 20-5 shows the Watchpoint Data Address registers, WPDA[1:0].

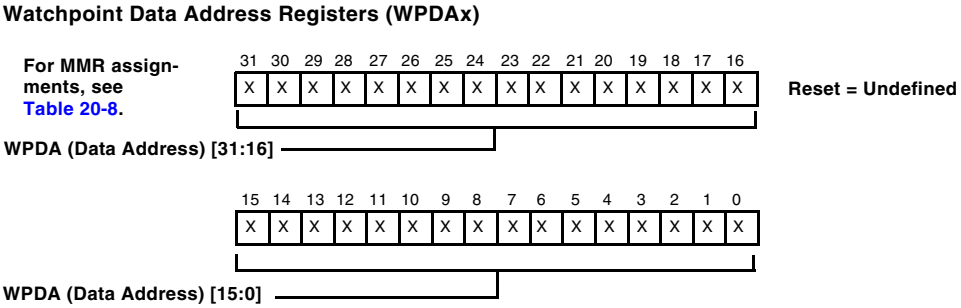


Figure 20-5. Watchpoint Data Address Registers

Table 20-8. Watchpoint Data Address Register MMR Assignments

Register Name	Memory-Mapped Address
WPDA0	0xFFE0 7140
WPDA1	0xFFE0 7144

Watchpoint Data Address Count Value Registers (WPDACNTx)

When the Watchpoint Unit is enabled, the count values in these registers are decremented each time the address or the address bus matches a value in the WPDAx registers. Load this WPDACNTx register with a value that is one less than the number of times the watchpoint must match before triggering an event.

Figure 20-6 shows the Watchpoint Data Address Count Value registers, `WPDACNT[1:0]`.

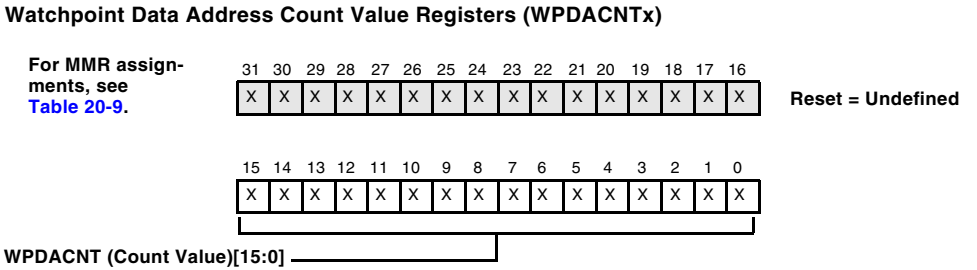


Figure 20-6. Watchpoint Data Address Count Value Registers

Table 20-9. Watchpoint Data Address Count Value Register MMR Assignments

Register Name	Memory-Mapped Address
WPDACNT0	0xFFE0 7180
WPDACNT1	0xFFE0 7184

Watchpoint Data Address Control Register (WPDACTL)

For more information about the bits in the `WPDACTL` register (Figure 20-7), see "Data Address Watchpoints" on page 20-10.

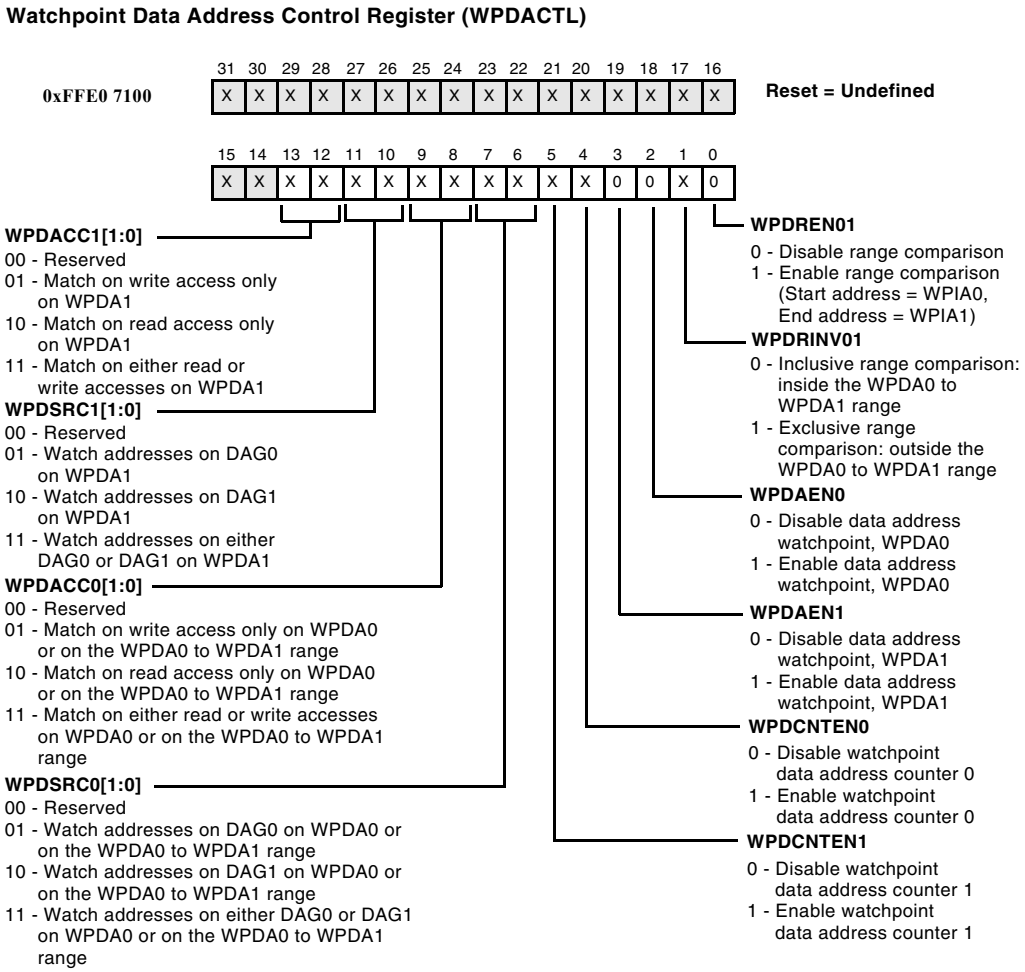


Figure 20-7. Watchpoint Data Address Control Register

Watchpoint Status Register (WPSTAT)

The WPSTAT register monitors the status of the watchpoints. It may be read and written in Supervisor or Emulator modes only. When a watchpoint or watchpoint range matches, this register reflects the source of the watchpoint. The status bits in the WPSTAT register are sticky, and all of them are cleared when any write, regardless of the value, is performed to the register.

Figure 20-8 shows the Watchpoint Status register.

Trace Unit

The Trace Unit stores a history of the last sixteen changes in program flow taken by the program sequencer. The history allows the user to recreate the program sequencer's recent path.

The trace buffer can be enabled to cause an exception when full. The exception service routine associated with the exception saves trace buffer entries to memory. Thus, the complete path of the program sequencer since the trace buffer was enabled can be recreated.

Changes in program flow because of zero-overhead loops are not stored in the trace buffer. For debugging code that is halted within a zero-overhead loop, the iteration count is available in the loop count registers, LC0 and LC1.

The trace buffer can be configured to omit recording of changes in program flow that match either the last entry or one of the last two entries. Omitting one of these entries from the record prevents the trace buffer from overflowing because of loops in the program. Because zero-overhead loops are not recorded in the trace buffer, this feature can be used to prevent trace overflow from loops that are nested four deep.

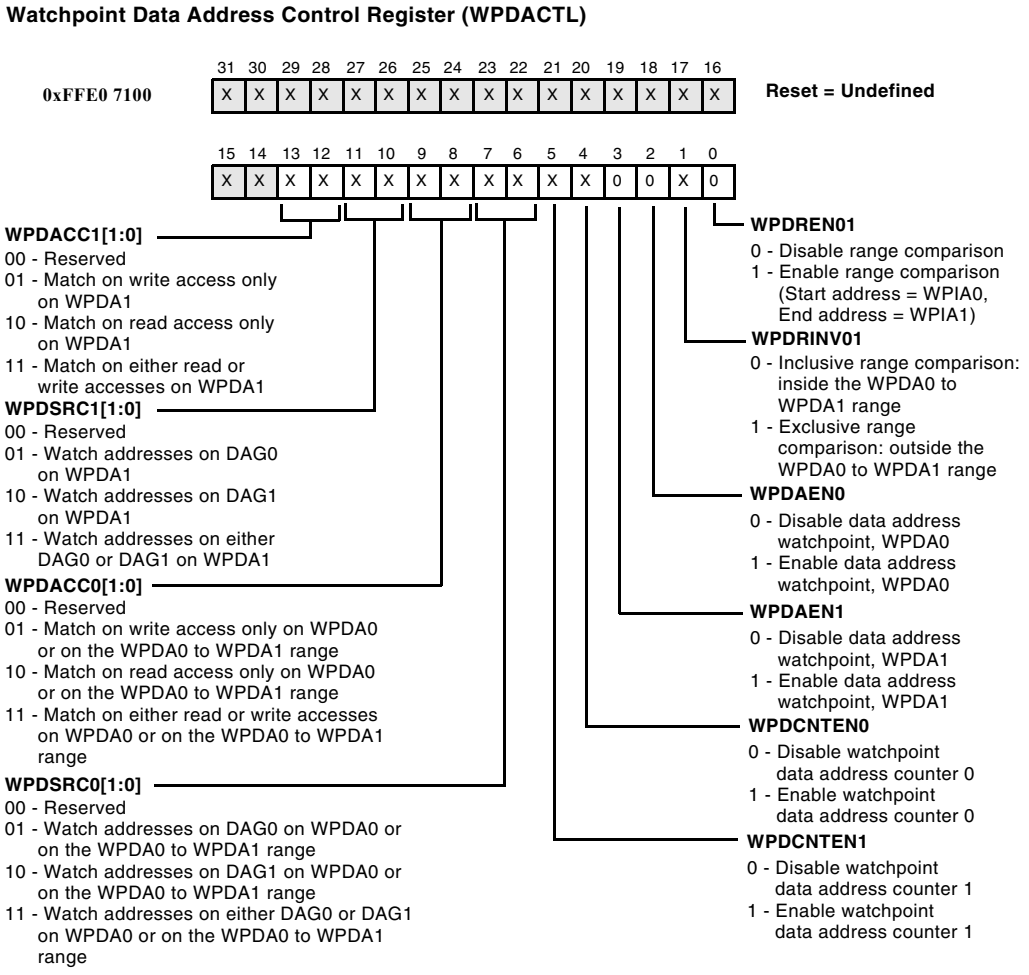


Figure 20-8. Watchpoint Status Register

When read, the Trace Buffer register (TBUF) returns the top value from the Trace Unit stack, which contains as many as sixteen entries. Each entry contains a pair of branch source and branch target addresses. A read of TBUF returns the newest entry first, starting with the branch destination. The next read provides the branch source address.

The number of valid entries in TBUF is held in the TBUFCNT field of the TBUFSTAT register. On every second read, TBUFCNT is decremented. Because each entry corresponds to two pieces of data, a total of 2 × TBUFCNT reads empties TBUF.

Discontinuities that are the same as either of the last two entries in the trace buffer are not recorded.

Because reading the trace buffer is a destructive operation, it is recommended that TBUF be read in a non-interruptible section of code.

Note if single-level compression has occurred, the LSB of the branch target address is set. If two-level compression has occurred, the LSB of the branch source address is set.

Trace Buffer Control Register (TBUFCTL)

The Trace Unit is enabled by two control bits in the Trace Buffer Control register (TBUFCTL) register. First, the Trace Unit must be activated by setting the TBUFPWR bit. If TBUFPWR=1, then setting TBUFEN to 1 enables the Trace Unit.

Figure 20-9 shows the Trace Buffer Control register (TBUFCTL). If TBUFOVF = 1, then the Trace Unit does not record discontinuities in the exception, NMI, and reset routines.

Trace Buffer Control Register (TBUFCTL)

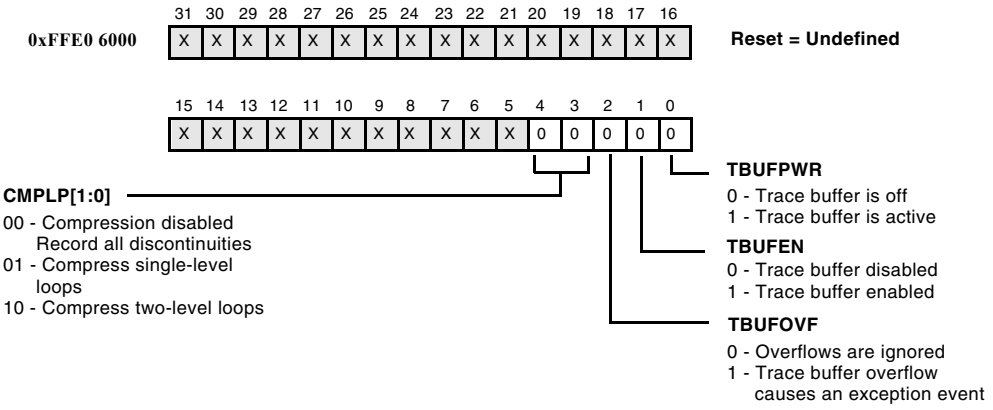


Figure 20-9. Trace Buffer Control Register

Trace Buffer Status Register (TBUFSTAT)

Figure 20-10 shows the Trace Buffer Status register (TBUFSTAT). Two reads from TBUF decrements TBUFCNT by 1.

Trace Buffer Status Register (TBUFSTAT)

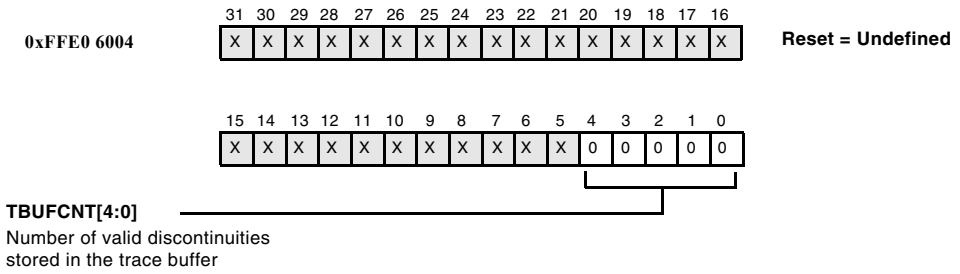


Figure 20-10. Trace Buffer Status Register

Trace Buffer Register (TBUF)

Figure 20-11 shows the Trace Buffer register (TBUF). The first read returns the latest branch target address. The second read returns the latest branch source address.

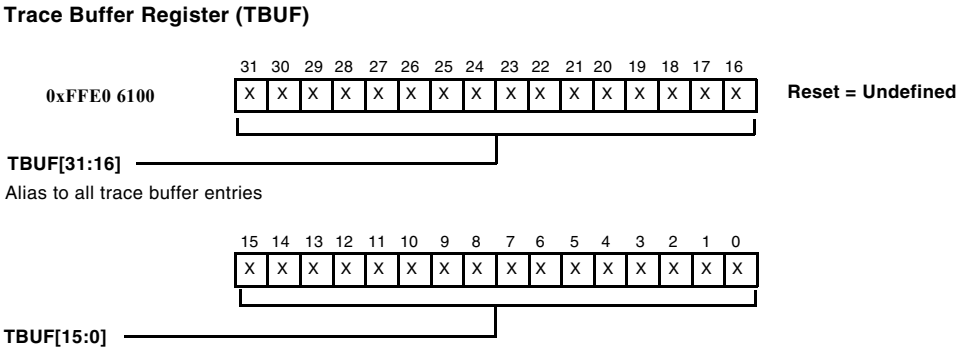


Figure 20-11. Trace Buffer Register

The Trace Unit does not record changes in program flow in:

- Emulator mode
- The reset service routine
- The exception or higher priority service routines (if TBUOVF = 1)
- In the exception service routine, the program flow discontinuities may be read from TBUF and stored in memory by using the following code.

While TBUF is being read, be sure to disable the trace buffer from recording new discontinuities.

Code to Recreate the Execution Trace in Memory

Listing 20-1 provides code that can be used to recreate the entire execution trace in memory.

Listing 20-1. Recreating the Execution Trace in Memory

```
[--sp] = (r7:7, p5:2);   /* save registers used in this routine
*/
p5 = 32;   /* 32 reads are needed to empty TBUF pointer to the
header (first location) of the software trace buffer */
p2.l = buf;
p2.h = buf;   /* the header stores the first available empty buf
location for subsequent trace dumps */

p4 = [p2++];   /* get the first available empty buf location from
the buf header */
p3.l = TBUF & 0xffff;   /* low 16-bits of TBUF */
p3.h = TBUF >> 16;   /* high 16-bits of TBUF */

lsetup(loop1_start, loop1_end) lc0 = p5;
loop1_start: r7 = [p3];   /* read from TBUF */
loop1_end: [p4++] = r7;   /* write to memory and increment */
[p2] = p4;   /* pointer to the next available buf location is
saved in the header of buf */
(r7:7, p5:3) = [sp++];   /* restore saved registers */
```

Performance Monitoring Unit

Two 32-bit counters, the Performance Monitor Counter registers (PFCNTR[1:0]) and the Performance Control register (PFCTL), count the number of occurrences of an event from within a processor core unit during a performance monitoring period. These registers provide feedback indicating the measure of load balancing between the various resources on the chip so that expected and actual usage can be compared and analyzed. In addition, events such as mispredictions and hold cycles can also be monitored.

Performance Monitor Counter Registers (PFCNTRx)

Figure 20-12 shows the Performance Monitor Counter registers, PFCNTR[1:0]. The PFCNTR0 register contains the count value of performance counter 0. The PFCNTR1 register contains the count value of performance counter 1.

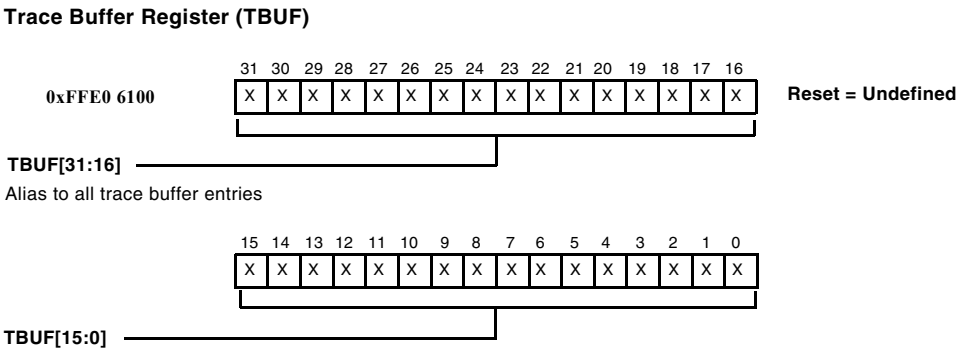


Figure 20-12. Performance Monitor Counter Registers

Table 20-10. Performance Monitor Counter Register MMR Assignments

Register Name	Memory-Mapped Address
PFCNTR0	0xFFE0 8100
PFCNTR1	0xFFE0 8104

Performance Monitor Control Register (PFCTL)

To enable the Performance Monitoring Unit, set the PFPWR bit in the PFCTL register (Figure 20-13). Once the unit is enabled, individual count-enable bits (PFCENx) take effect. Use the PFCENx bits to enable or disable the performance monitors in User mode, Supervisor mode, or both. Use the PEMUSWx bits to select the type of event triggered.

Performance Monitor Control Register (PFCTL)

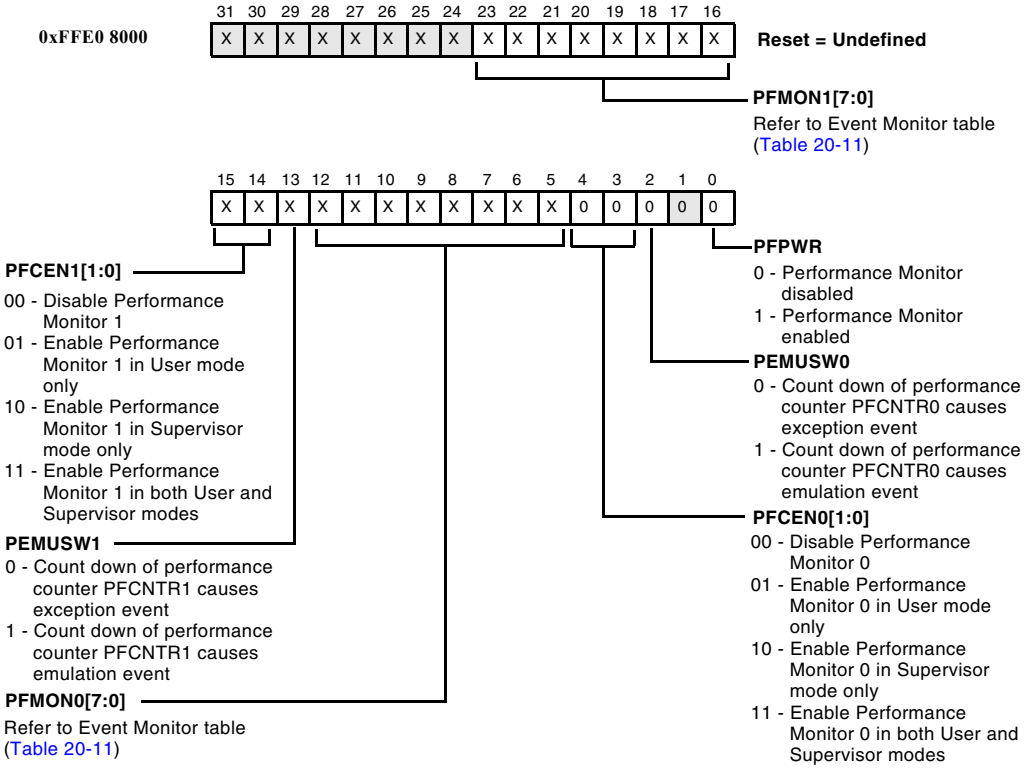


Figure 20-13. Performance Monitor Control Register

Event Monitor Table

Table 20-11 identifies events that cause the Performance Monitor Counter registers (PFMON[1:0]) to increment.

Table 20-11. Event Monitor Table

PFMONx Fields	Events That Cause the PFCNTRx Counter Registers to Increment
0x00	Stalls caused by L1 Instruction Cache (IC) misses
0x01	L1 IC misses
0x02	L1 IC hits
0x04	L1 IC bank conflicts
0x05	L1 IC fill buffer hits
0x20	Loop0 iterations
0x21	Loop1 iterations
0x22	Loop buffer 0 invalidates (because of branches, jumps, and similar operations)
0x23	Loop buffer 1 invalidates (because of branches, jumps, and similar operations)
0x24	PC-relative branches
0x25	Indirect branches
0x26	Mispredicted branches
0x27	Taken branches
0x28	Not taken branches
0x29	Total branches
0x2A	Stalls because of CSYNC, SSYNC
0x2B	EXCPT instructions
0x2C	CSYNC, SSYNC instructions
0x2D	Committed instructions
0x2E	Interrupts taken

Table 20-11. Event Monitor Table (Cont'd)

PFMONx Fields	Events That Cause the PFCNTRx Counter Registers to Increment
0x2F	Misaligned Address Violation exceptions
0x40	Stall cycles because of read-after-write hazards on DAG registers
0x41	Stall cycles because of write-after-write hazards on DAG registers
0x60	Load hits
0x61	Load misses
0x62	Store hits
0x63	Store misses
0x64	Data Bank A load hits
0x65	Data Bank A store hits
0x66	Data Bank B load hits
0x67	Data Bank B store hits
0x68	L1 data memory hits
0x69	L1 data memory misses
0x6A	Store buffer is full
0x6B	Write buffer is full
0x6C	Core/system bank conflicts
0x6D	DAG0 & DAG1 bank conflicts
0x6E	Scratch SRAM hits
0x6F	DMA reads
0x70	DMA writes
0x71	LMU stalls in EX2
0x72	LMU stalls in EX3
0x73	LMU exceptions
0x74	Stall cycles because of store buffer cancels

Table 20-11. Event Monitor Table (Cont'd)

PFMONx Fields	Events That Cause the PFCNTRx Counter Registers to Increment
0x75	Store to load forwarding
0x80	Stall cycles because of EX stage data hazards

Cycle Counter

The cycle counter counts CCLK cycles while the program is executing. All cycles, including execution, wait state, interrupts, and events, are counted while the processor is in User or Supervisor mode, but the cycle counter stops counting in Emulator mode.

The cycle counter is a 64-bit counter that increments every cycle. The count value is stored in two 32-bit registers, CYCLES and CYCLES2. The least significant 32 bits are stored in CYCLES. The most significant 32 bits are stored in CYCLES2.

To ensure the correct cycle count, a read from CYCLES must occur before reading from CYCLES2.

In User mode, these two registers may be read, but not written. In Supervisor and Emulator modes, they are read/write registers.

To enable the cycle counters, set the CCEN bit in the SYSCFG register.

This example shows how to turn on the cycle counter:

```
r2 = 0 ;
cycles = r2 ;
cycles2 = r2 ;
r2 = SYSCFG ;
bitset(r2, 1) ;
SYSCFG = r2 ;
/* Insert code to be benchmarked */
```

```
r2 = SYSCFG ;
bitclr(r2, 1) ;
SYSCFG = r2 ;
```

Execution Cycle Count Registers (CYCLES and CYCLES2)

The Execution Cycle Count registers, shown in Figure 20-14, consist of CYCLES and CYCLES2. This 64-bit counter increments every CCLK cycle. The CYCLES register contains the least significant 32 bits of the cycle counter's 64-bit count value. The CYCLES2 register contains the most significant 32 bits.

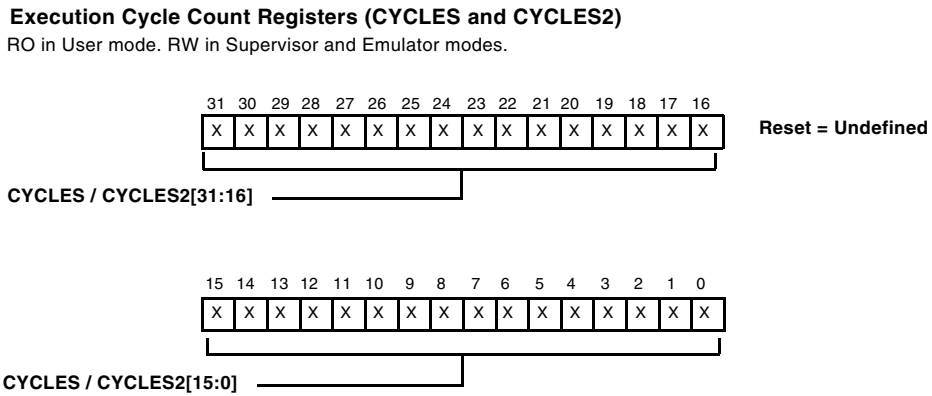


Figure 20-14. Execution Cycle Count Registers

Product Identification Registers

The 32-bit Chip ID register (CHIPID) is a system MMR that contains the product identification and revision fields for the ADSP-BF535 processor. The 32-bit DSP Device ID register (DSPID) is a core MMR that contains core identification and revision fields for the core.

Chip ID Register (CHIPID)

The Chip ID register (CHIPID), shown in Figure 20-15, is a read-only register. The definition of this register complies with Joint Electronic Device Engineering Council (JEDEC) standard JEP106, Standard Manufacturer's Identification Code:

- CHIPID[31:28]: silicon revision number
- CHIPID[27:12]: part number 0x4000
- CHIPID[11:1]: Analog Devices, Inc., ID number 0xE5 – drop the MSB (parity) and set MSBs to 0 (no ID extension) in accordance with the JTAG standard: 0x065.
- CHIPID[0]: set to 1

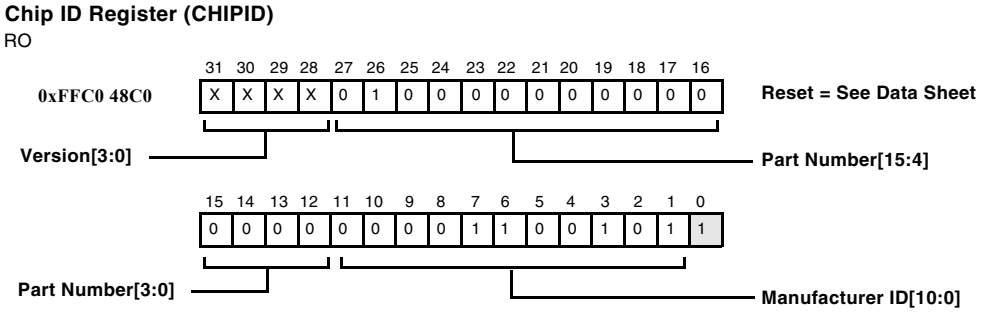


Figure 20-15. Chip ID Register

DSP Device ID Register (DSPID)

The DSP Device ID register (DSPID), shown in Figure 20-16, is read-only and part of the core. The DSPID value is an input to the core.

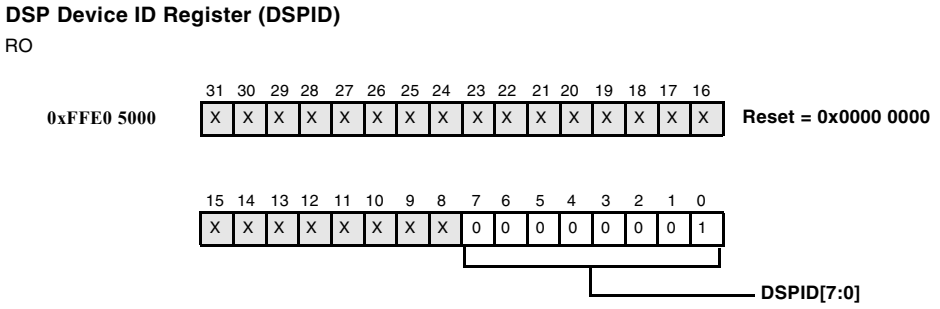


Figure 20-16. DSP Device ID Register

DMA Bus Debug Registers

During normal operation, the core has no direct visibility into the DMA bus. However, the SBIU includes two registers that provide visibility into the bus and control of it. Additional information about the SBIU is available in "Chip Bus Hierarchy" on page 7-1.

If a DMA channel does not behave as expected, these two registers provide a debug capability:

- DMA Bus Control Comparator register (DB_CCOMP)
- DMA Bus Address Comparator register (DB_ACOMP)

DMA Bus Control Comparator Register (DB_CCOMP)

The DMA Bus Control Comparator register (DB_CCOMP), shown in Figure 20-17, provides control for the debug register pair. Program this register with the type of bus operation that you wish to detect. Program the address associated with this control field into the DMA Bus Address Comparator register (DB_ACOMP). See "DMA Bus Address Comparator Register (DB_ACOMP)" on page 20-29.

When enabled, this function compares the address and control fields of each bus operation on the DMA bus to those programmed into the address and control comparator fields. If a match occurs, the function sets the Compare Hit (CH) status bit in DB_CCOMP. If the interrupt is unmasked, the hit also generates a Hardware Error interrupt to the core. The source of the interrupt can be determined by reading the core MMR HWE Cause field in SEQSTAT, and/or by reading DB_CCOMP status.

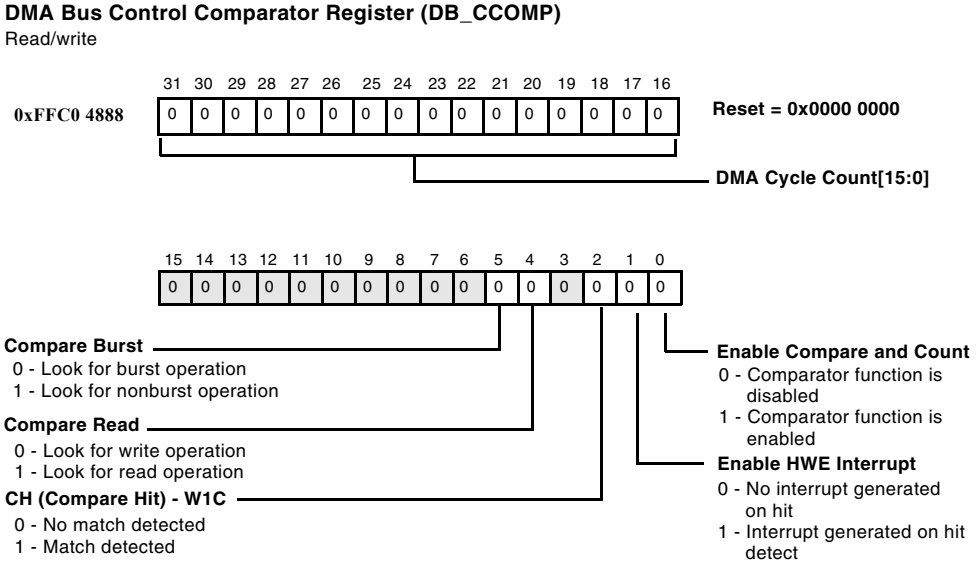


Figure 20-17. DMA Bus Control Comparator Register

The upper 16 bits of the DMA Bus Control Comparator register provide a cycle count function that helps determine when a given DMA access has occurred. The counter is clocked by the DMA bus clock.

The counter can be cleared in either of two ways:

- By a hardware reset
- By clearing the Enable Compare and Count bit (DB_CCOMP[0])

When enabled, the counter increments from 0 to a maximum value of 65,535 until a match is detected or the compare-and-count function is disabled. A comparison match freezes the counter within two cycles of the assertion of CH.

DMA Bus Address Comparator Register (DB_ACOMP)

Figure 20-18 shows the DMA Bus Address Comparator register (DB_ACOMP). Program into this register the DMA address that is associated with the type of bus information that you wish to detect.

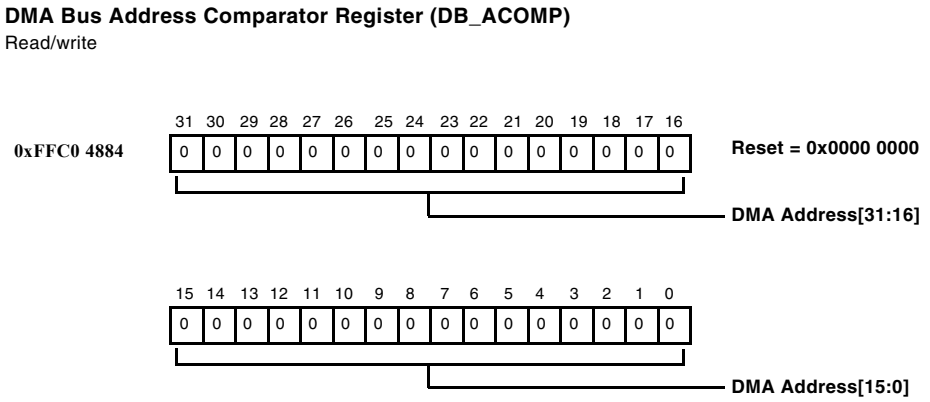


Figure 20-18. DMA Bus Address Comparator Register

A BLACKFIN PROCESSOR CORE MMR ASSIGNMENTS

The Blackfin processor's memory-mapped registers are in the address range 0xFFE0 0000–0xFFFF FFFF. All MMRs are 32-bits and can be accessed only with 32-bit aligned memory operations.

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 the additional information about the MMR.

L1 Data Memory Controller Registers

Table A-1 lists the MMR assignments for the L1 Data Memory Controller registers (0xFFE0 0000–0xFFE0 0040C).

Table A-1. L1 Data Memory Controller Registers

Memory- Mapped Address	Register Name	See Section
0xFFE0 0004	DMEM_CONTROL	"Data Memory Control Register (DMEM_CONTROL)" on page 6-12
0xFFE0 0008	DCPLB_STATUS	"DCPLB Status Register (DCPLB_STATUS)" on page 6-73
0xFFE0 000C	DCPLB_FAULT_ADDR	"DCPLB Fault Address Register (DCPLB_FAULT_ADDR)" on page 6-75
0xFFE0 0100	DCPLB_ADDR0	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69

Table A-1. L1 Data Memory Controller Registers (Cont'd)

Memory- Mapped Address	Register Name	See Section
0xFFE0 0104	DCPLB_ADDR1	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 0108	DCPLB_ADDR2	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 010C	DCPLB_ADDR3	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 0110	DCPLB_ADDR4	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 0114	DCPLB_ADDR5	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 0118	DCPLB_ADDR6	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 011C	DCPLB_ADDR7	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 0120	DCPLB_ADDR8	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 0124	DCPLB_ADDR9	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 0128	DCPLB_ADDR10	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 012C	DCPLB_ADDR11	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 0130	DCPLB_ADDR12	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 0134	DCPLB_ADDR13	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 0138	DCPLB_ADDR14	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69
0xFFE0 013C	DCPLB_ADDR15	"DCPLB Address Registers (DCPLB_ADDRx)" on page 6-69

Table A-1. L1 Data Memory Controller Registers (Cont'd)

Memory- Mapped Address	Register Name	See Section
0xFFE0 0200	DCPLB_DATA0	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 0204	DCPLB_DATA1	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0 xFFE0 0208	DCPLB_DATA2	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 020C	DCPLB_DATA3	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 0210	DCPLB_DATA4	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 0214	DCPLB_DATA5	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 0218	DCPLB_DATA6	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 021C	DCPLB_DATA7	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 0220	DCPLB_DATA8	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 0224	DCPLB_DATA9	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 0228	DCPLB_DATA10	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 022C	DCPLB_DATA11	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 0230	DCPLB_DATA12	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 0234	DCPLB_DATA13	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 0238	DCPLB_DATA14	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65

Table A-1. L1 Data Memory Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFE0 023C	DCPLB_DATA15	"DCPLB Data Registers (DCPLB_DATAx)" on page 6-65
0xFFE0 0300	DTEST_COMMAND	"Data Test Command Register (DTEST_COMMAND)" on page 6-49
0xFFE0 0400	DTEST_DATA0	"Data Test Data 0 Register (DTEST_DATA0)" on page 6-50
0xFFE0 0404	DTEST_DATA1	"Data Test Data 1 Register (DTEST_DATA1)" on page 6-49
0xFFE0 0408	DTEST_DATA2	"Data Test Data 0 Register (DTEST_DATA0)" on page 6-50
0xFFE0 040C	DTEST_DATA3	"Data Test Data 1 Register (DTEST_DATA1)" on page 6-49

L1 Instruction Memory Controller Registers

Table A-2 lists the MMR assignments for the L1 Instruction Memory Controller registers (0xFFE0 1004–0xFFE0 6100).

Table A-2. L1 Instruction Memory Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFE0 1004	IMEM_CONTROL	"Instruction Memory Control Register (IMEM_CONTROL)" on page 6-12
0xFFE0 1008	ICPLB_STATUS	"ICPLB Status Register (ICPLB_STATUS)" on page 6-74
0xFFE0 100C	ICPLB_FAULT_ADDR	"ICPLB Fault Address Register (ICPLB_FAULT_ADDR)" on page 6-76

Table A-2. L1 Instruction Memory Controller Registers (Cont'd)

Memory- Mapped Address	Register Name	See Section
0xFFE0 1100	ICPLB_ADDR0	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 1104	ICPLB_ADDR1	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 1108	ICPLB_ADDR2	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 110C	ICPLB_ADDR3	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 1110	ICPLB_ADDR4	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 1114	ICPLB_ADDR5	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 1118	ICPLB_ADDR6	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 111C	ICPLB_ADDR7	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 1120	ICPLB_ADDR8	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 1124	ICPLB_ADDR9	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 1128	ICPLB_ADDR10	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 112C	ICPLB_ADDR11	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 1130	ICPLB_ADDR12	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 1134	ICPLB_ADDR13	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 1138	ICPLB_ADDR14	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71

Table A-2. L1 Instruction Memory Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFE0 113C	ICPLB_ADDR15	"ICPLB Address Registers (ICPLB_ADDRx)" on page 6-71
0xFFE0 1200	ICPLB_DATA0	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 1204	ICPLB_DATA1	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 1208	ICPLB_DATA2	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 120C	ICPLB_DATA3	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 1210	ICPLB_DATA4	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 1214	ICPLB_DATA5	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 1218	ICPLB_DATA6	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 121C	ICPLB_DATA7	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 1220	ICPLB_DATA8	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 12245	ICPLB_DATA9	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 1228	ICPLB_DATA10	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 122C	ICPLB_DATA11	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 1230	ICPLB_DATA12	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 1234	ICPLB_DATA13	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67

Table A-2. L1 Instruction Memory Controller Registers (Cont'd)

Memory- Mapped Address	Register Name	See Section
0xFFE0 1238	ICPLB_DATA14	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 123C	ICPLB_DATA15	"ICPLB Data Registers (ICPLB_DATAx)" on page 6-67
0xFFE0 1300	ITEST_COMMAND	"Instruction Test Command Register (ITEST_COMMAND)" on page 6-27
0XFFE0 1400	ITEST_DATA0	"Instruction Test Data 0 Register (ITEST_DATA0)" on page 6-29
0XFFE0 1404	ITEST_DATA1	"Instruction Test Data 1 Register (ITEST_DATA1)" on page 6-28

Interrupt Controller Registers

Table A-3 lists the MMR assignments for the Interrupt Controller registers (0xFFE0 2000–0xFFE0 210C).

Table A-3. Interrupt Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFE0 2000	EVT0 (EMU)	"Core Event Vector Table" on page 4-35
0xFFE0 2004	EVT1 (RST)	"Core Event Vector Table" on page 4-35
0xFFE0 2008	EVT2 (NMI)	"Core Event Vector Table" on page 4-35
0xFFE0 200C	EVT3 (EVX)	"Core Event Vector Table" on page 4-35
0xFFE0 2010	EVT4	"Core Event Vector Table" on page 4-35

Table A-3. Interrupt Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFE0 2014	EVT5 (IVHW)	"Core Event Vector Table" on page 4-35
0xFFE0 2018	EVT6 (TMR)	"Core Event Vector Table" on page 4-35
0xFFE0 201C	EVT7 (IVG7)	"Core Event Vector Table" on page 4-35
0xFFE0 2020	EVT8 (IVG8)	"Core Event Vector Table" on page 4-35
0xFFE0 2024	EVT9 (IVG9)	"Core Event Vector Table" on page 4-35
0xFFE0 2028	EVT10 (IVG10)	"Core Event Vector Table" on page 4-35
0xFFE0 202C	EVT11 (IVG11)	"Core Event Vector Table" on page 4-35
0xFFE0 2030	EVT12 (IVG12)	"Core Event Vector Table" on page 4-35
0xFFE0 2034	EVT13 (IVG13)	"Core Event Vector Table" on page 4-35
0xFFE0 2038	EVT14 (IVG14)	"Core Event Vector Table" on page 4-35
0xFFE0 203C	EVT15 (IVG15)	"Core Event Vector Table" on page 4-35
0xFFE0 2104	IMASK (EVT_IMASK)	"Core Interrupt Mask Register" on page 4-32
0xFFE0 2108	IPEND (EVT_IPEND)	"Core Interrupt Pending Register" on page 4-34
0xFFE0 210C	ILAT (EVT_ILAT)	"Core Interrupt Latch Register" on page 4-33

Core Timer Registers

Table A-4 lists the MMR assignments for the Core Timer registers (0xFFE0 3000–0xFFE0 300C).

Table A-4. Core Timer Registers

Memory-Mapped Address	Register Name	See Section
0xFFE0 3000	TCNTL	"Core Timer Control Register (TCNTL)" on page 16-22
0xFFE0 3004	TPERIOD	"Core Timer Period Register (TPERIOD)" on page 16-24
0xFFE0 3008	TSCALE	"Core Timer Scale Register (TSCALE)" on page 16-24
0xFFE0 300C	TCOUNT	"Core Timer Count Register (TCOUNT)" on page 16-23

DSP Device ID Register

Table A-5 lists the MMR assignments for the DSP Device ID register (0xFFE0 5000–0xFFE0 500C).

Table A-5. DSPID Register

Memory- Mapped Address	Register Name	See Section
0xFFE0 5000	DSPID	"DSP Device ID Register (DSPID)" on page 20-27

Trace Unit Registers

Table A-6 lists the MMR assignments for the Trace Unit registers (0xFFE0 6000–0xFFE0 6100).

Table A-6. Trace Unit Registers

Memory-Mapped Address	Register Name	See Section
0xFFE0 6000	TBUFCTL	"Trace Buffer Control Register (TBUFCTL)" on page 20-16
0xFFE0 6004	TBUFSTAT	"Trace Buffer Status Register (TBUFSTAT)" on page 20-17
0xFFE0 6100	TBUF	"Trace Buffer Register (TBUF)" on page 20-18

Watchpoint and Patch Registers

Table A-7 lists the MMR assignments for the Watchpoint and Patch registers (0xFFE0 1C00–0xFFE0 7200).

Table A-7. Watchpoint and Patch Registers

Memory- Mapped Address	Register Name	See Section
0xFFE0 7000	WPIACTL	"Watchpoint Instruction Address Control Register (WPIACTL)" on page 20-7
0xFFE0 7040	WPIA0	"Watchpoint Instruction Address Registers (WPIAx)" on page 20-5
0xFFE0 7044	WPIA1	"Watchpoint Instruction Address Registers (WPIAx)" on page 20-5
0xFFE0 7048	WPIA2	"Watchpoint Instruction Address Registers (WPIAx)" on page 20-5

Table A-7. Watchpoint and Patch Registers (Cont'd)

Memory- Mapped Address	Register Name	See Section
0xFFE0 704C	WPIA3	"Watchpoint Instruction Address Registers (WPIAx)" on page 20-5
0xFFE0 7050	WPIA4	"Watchpoint Instruction Address Registers (WPIAx)" on page 20-5
0xFFE0 7054	WPIA5	"Watchpoint Instruction Address Registers (WPIAx)" on page 20-5
0xFFE0 7080	WPIACNT0	"Watchpoint Instruction Address Count Registers (WPIACNTx)" on page 20-6
0xFFE0 7084	WPIACNT1	"Watchpoint Instruction Address Count Registers (WPIACNTx)" on page 20-6
0xFFE0 7088	WPIACNT2	"Watchpoint Instruction Address Count Registers (WPIACNTx)" on page 20-6
0xFFE0 708C	WPIACNT3	"Watchpoint Instruction Address Count Registers (WPIACNTx)" on page 20-6
0xFFE0 7090	WPIACNT4	"Watchpoint Instruction Address Count Registers (WPIACNTx)" on page 20-6
0xFFE0 7094	WPIACNT5	"Watchpoint Instruction Address Count Registers (WPIACNTx)" on page 20-6
0xFFE0 7100	WPDACTL	"Watchpoint Instruction Address Registers (WPIAx)" on page 20-5
0xFFE0 7140	WPDA0	"Watchpoint Data Address Registers (WPDAx)" on page 20-11
0xFFE0 7144	WPDA1	"Watchpoint Data Address Registers (WPDAx)" on page 20-11
0xFFE0 7180	WPDACNT0	"Watchpoint Data Address Count Value Registers (WPDACNTx)" on page 20-11
0xFFE0 7184	WPDACNT1	"Watchpoint Data Address Count Value Registers (WPDACNTx)" on page 20-11
0xFFE0 7200	WPSTAT	"Trace Buffer Status Register (TBUFSTAT)" on page 20-17

Performance Monitor Registers

Table A-8 lists the MMR assignments for the Performance Monitor registers (0xFFE0 8000–0xFFE0 8104).

Table A-8. Performance Monitor Registers

Memory-Mapped Address	Register Name	See Section
0xFFE0 8000	PFCTL	"Performance Monitor Control Register (PFCTL)" on page 20-20
0xFFE0 8100	PFCNTR0	"Performance Monitor Control Register (PFCTL)" on page 20-20
0xFFE0 8104	PFCNTR1	"Performance Monitor Control Register (PFCTL)" on page 20-20

B SYSTEM MMR ASSIGNMENTS

These notes provide general information about the ADSP-BF535 system memory-mapped registers (MMRs):

- The system MMR address range is 0xFFC00000–0xFFDFFFFF.
- All system MMRs are either 16-bit, or 32-bit register widths.
- 32-bit system MMRs can only be accessed with 32-bit aligned memory operations.
- A dedicated set of PCI registers reside at 0xEEFFFF00–0xEEFFFFFF.
- 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 the additional information about the MMR.

Clock and System Control Registers

Table B-1 lists the MMR assignments for the Clock and System Control registers (0xFFC0 0400–0xFFC0 07FF).

Table B-1. Clock and System Control Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0400	PLL_CTL	"PLL Control Register (PLL_CTL)" on page 8-7
0xFFC0 0404	PLL_STAT	"PLL Status Register (PLL_STAT)" on page 8-9
0xFFC0 0406	PLL_LOCKCNT	"PLL Lock Count Register (PLL_LOCKCNT)" on page 8-10
0xFFC0 0408	PLL_IOCK	"Peripheral Clock Enable Register (PLL_IOCK)" on page 8-22
0xFFC0 0410	SWRST	"Software Reset Register (SWRST)" on page 3-16
0xFFC0 0414	SYSCR	"System Reset Configuration Register (SYSCR)" on page 3-14

Chip ID Register

Table B-2 lists the MMR assignment for the Chip ID register (0xFFC0 48C0).

Table B-2. Chip ID Register

Memory-Mapped Address	Register Name	See Section
0xFFC0 48C0	CHIPID	"Chip ID Register (CHIPID)" on page 20-26

System Interrupt Controller Registers

Table B-3 lists the MMR assignments for the System Interrupt Controller registers (0xFFC0 0C00–0xFFC0 0FFF).

Table B-3. System Interrupt Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 0C04	SIC_IAR0	"System Interrupt Assignment Registers (SIC_IARx)" on page 4-29
0xFFC0 0C08	SIC_IAR1	"System Interrupt Assignment Registers (SIC_IARx)" on page 4-29
0xFFC0 0C0C	SIC_IAR2	"System Interrupt Assignment Registers (SIC_IARx)" on page 4-29
0xFFC0 0C10	SIC_IMASK	"System Interrupt Status Register (SIC_ISR)" on page 4-25
0xFFC0 0C14	SIC_ISR	"System Interrupt Status Register (SIC_ISR)" on page 4-25
0xFFC0 0C18	SIC_IWR	"System Interrupt Wakeup-Enable Register (SIC_IWR)" on page 4-24

Watchdog Timer Registers

Table B-4 lists the MMR assignments for the Watchdog Timer registers (0xFFC0 1000–0xFFC0 13FF).

Table B-4. Watchdog Timer Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 1000	WDOG_CTL	"Watchdog Control Register (WDOG_CTL)" on page 16-27
0xFFC0 1004	WDOG_CNT	"Watchdog Count Register (WDOG_CNT)" on page 16-26
0xFFC0 1008	WDOG_STAT	"Watchdog Status Register (WDOG_STAT)" on page 16-26

Real-Time Clock Registers

Table B-5 lists the MMR assignments for the Real-Time Clock registers (0xFFC0 1400–0xFFC0 17FF).

Table B-5. Real-Time Clock Registers

Memory- Mapped Address	Register Name	See Section
0xFFC0 1400	RTC_STAT	"RTC Status Register (RTC_STAT)" on page 17-7
0xFFC0 1404	RTC_ICTL	"RTC Interrupt Control Register (RTC_ICTL)" on page 17-8
0xFFC0 1408	RTC_ISTAT	"RTC Interrupt Status Register (RTC_ISTAT)" on page 17-9
0xFFC0 140C	RTC_SWCNT	"RTC Stopwatch Count Register (RTC_SWCNT)" on page 17-10

Table B-5. Real-Time Clock Registers (Cont'd)

Memory- Mapped Address	Register Name	See Section
0xFFC0 1410	RTC_ALARM	"RTC Alarm Register (RTC_ALARM)" on page 17-11
0xFFC0 1414	RTC_FAST	"RTC Enable Register (RTC_FAST)" on page 17-12

UART0 Controller Registers

Table B-6 lists the MMR assignments for the UART0 Controller registers (0xFFC0 1800–0xFFC0 1BFF).

Table B-6. UART0 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 1800	UART0_THR	"UARTx Transmit Holding Registers (UARTx_THR)" on page 12-5
0xFFC0 1800	UART0_RBR	"UARTx Receive Buffer Registers (UARTx_RBR)" on page 12-6
0xFFC0 1800	UART0_DLL	"UARTx Divisor Latch Registers (UARTx_DLL, UARTx_DLH)" on page 12-10
0xFFC0 1802	UART0_DLH	"UARTx Divisor Latch Registers (UARTx_DLL, UARTx_DLH)" on page 12-10
0xFFC0 1802	UART0_IER	"UARTx Interrupt Enable Registers (UARTx_IER)" on page 12-7
0xFFC0 1804	UART0_IIR	"UARTx Interrupt Identification Registers (UARTx_IIR)" on page 12-9
0xFFC0 1806	UART0_LCR	"UARTx Line Control Registers (UARTx_LCR)" on page 12-3

Table B-6. UART0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 1808	UART0_MCR	"UARTx Modem Control Registers (UARTx_MCR)" on page 12-12
0xFFC0 180A	UART0_LSR	"UARTx Line Status Registers (UARTx_LSR)" on page 12-4
0xFFC0 180C	UART0_MSR	"UARTx Modem Status Registers (UARTx_MSR)" on page 12-14
0xFFC0 180E	UART0_SCR	"UARTx Scratch Registers (UARTx_SCR)" on page 12-15
0xFFC0 1810	UART0_IRCR	"UART0 Infrared Control Register (UART0_IRCR)" on page 12-36
0xFFC0 1A00	UART0_CURR_PTR_RX	"UARTx Receive DMA Current Descriptor Pointer Registers (UARTx_CURR_PTR_RX)" on page 12-19
0xFFC0 1A02	UART0_CONFIG_RX	"UARTx Receive DMA Configuration Registers (UARTx_CONFIG_RX)" on page 12-20
0xFFC0 1A04	UART0_START_ADDR_HI_RX	"UARTx Receive DMA Start Address High Registers (UARTx_START_ADDR_HI_RX)" on page 12-22
0xFFC0 1A06	UART0_START_ADDR_LO_RX	"UARTx Receive DMA Start Address Low Registers (UARTx_START_ADDR_LO_RX)" on page 12-23
0xFFC0 1A08	UART0_COUNT_RX	"UARTx Receive DMA Count Registers (UARTx_COUNT_RX)" on page 12-24
0xFFC0 1A0A	UART0_NEXT_DESCR_RX	"UARTx Receive DMA Next Descriptor Pointer Registers (UARTx_NEXT_DESCR_RX)" on page 12-25

Table B-6. UART0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 1A0C	UART0_DESCR_RDY_RX	"UARTx Receive DMA Descriptor Ready Registers (UARTx_DESCR_RDY_RX)" on page 12-26
0xFFC0 1A0E	UART0_IRQSTAT_RX	"UARTx Receive DMA IRQ Status Registers (UARTx_IRQSTAT_RX)" on page 12-27
0xFFC0 1B00	UART0_CURR_PTR_TX	"UARTx Transmit DMA Current Descriptor Pointer Registers (UARTx_CURR_PTR_TX)" on page 12-28
0xFFC0 1B02	UART0_CONFIG_TX	"UARTx Transmit DMA Configuration Registers (UARTx_CONFIG_TX)" on page 12-29
0xFFC0 1B04	UART0_START_ADDR_HI_TX	"UARTx Transmit DMA Start Address High Registers (UARTx_START_ADDR_HI_TX)" on page 12-30
0xFFC0 1B06	UART0_START_ADDR_LO_TX	"UARTx Transmit DMA Start Address Low Registers (UARTx_START_ADDR_LO_TX)" on page 12-31
0xFFC0 1B08	UART0_COUNT_TX	"UARTx Transmit DMA Count Registers (UARTx_COUNT_TX)" on page 12-32
0xFFC0 1B0A	UART0_NEXT_DESCR_TX	"UARTx Transmit DMA Next Descriptor Pointer Registers (UARTx_NEXT_DESCR_TX)" on page 12-33

Table B-6. UART0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 1B0C	UART0_DESCR_RDY_TX	"UARTx Transmit DMA Descriptor Ready Registers (UARTx_DESCR_RDY_TX)" on page 12-34
0xFFC0 1B0E	UART0_IRQSTAT_TX	"UARTx Transmit DMA IRQ Status Registers (UARTx_IRQSTAT_TX)" on page 12-35

UART1 Controller Registers

Table B-7 lists the MMR assignments for the UART1 Controller registers (0xFFC0 1C00–0xFFC0 1FFF).

Table B-7. UART1 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 1C00	UART1_THR	"UARTx Transmit Holding Registers (UARTx_THR)" on page 12-5
0xFFC0 1C00	UART1_RBR	"UARTx Receive Buffer Registers (UARTx_RBR)" on page 12-6
0xFFC0 1C00	UART1_DLL	"UARTx Divisor Latch Registers (UARTx_DLL, UARTx_DLH)" on page 12-10
0xFFC0 1C02	UART1_DLH	"UARTx Divisor Latch Registers (UARTx_DLL, UARTx_DLH)" on page 12-10
0xFFC0 1C02	UART1_IER	"UARTx Interrupt Enable Registers (UARTx_IER)" on page 12-7
0xFFC0 1C04	UART1_IIR	"UARTx Interrupt Identification Registers (UARTx_IIR)" on page 12-9

Table B-7. UART1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 1C06	UART1_LCR	"UARTx Line Control Registers (UARTx_LCR)" on page 12-3
0xFFC0 1C08	UART1_MCR	"UARTx Modem Control Registers (UARTx_MCR)" on page 12-12
0xFFC0 1C0A	UART1_LSR	"UARTx Line Status Registers (UARTx_LSR)" on page 12-4
0xFFC0 1C0C	UART1_MSR	"UARTx Modem Status Registers (UARTx_MSR)" on page 12-14
0xFFC0 1C0E	UART1_SCR	"UARTx Scratch Registers (UARTx_SCR)" on page 12-15
0xFFC0 1E00	UART1_CURR_PTR_RX	"UARTx Receive DMA Current Descriptor Pointer Registers (UARTx_CURR_PTR_RX)" on page 12-19
0xFFC0 1E02	UART1_CONFIG_RX	"UARTx Receive DMA Configuration Registers (UARTx_CONFIG_RX)" on page 12-20
0xFFC0 1E04	UART1_START_ADDR_HI_RX	"UARTx Receive DMA Start Address High Registers (UARTx_START_ADDR_HI_RX)" on page 12-22
0xFFC0 1E06	UART1_START_ADDR_LO_RX	"UARTx Receive DMA Start Address Low Registers (UARTx_START_ADDR_LO_RX)" on page 12-23
0xFFC0 1E08	UART1_COUNT_RX	"UARTx Receive DMA Count Registers (UARTx_COUNT_RX)" on page 12-24
0xFFC0 1E0A	UART1_NEXT_DESCR_RX	"UARTx Receive DMA Next Descriptor Pointer Registers (UARTx_NEXT_DESCR_RX)" on page 12-25

Table B-7. UART1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 1E0C	UART1_DESCR_RDY_RX	"UARTx Receive DMA Descriptor Ready Registers (UARTx_DESCR_RDY_RX)" on page 12-26
0xFFC0 1E0E	UART1_IRQSTAT_RX	"UARTx Receive DMA IRQ Status Registers (UARTx_IRQSTAT_RX)" on page 12-27
0xFFC0 1F00	UART1_CURR_PTR_TX	"UARTx Transmit DMA Current Descriptor Pointer Registers (UARTx_CURR_PTR_TX)" on page 12-28
0xFFC0 1F02	UART1_CONFIG_TX	"UARTx Transmit DMA Configuration Registers (UARTx_CONFIG_TX)" on page 12-29
0xFFC0 1F04	UART1_START_ADDR_HI_TX	"UARTx Transmit DMA Start Address High Registers (UARTx_START_ADDR_HI_TX)" on page 12-30
0xFFC0 1F06	UART1_START_ADDR_LO_TX	"UARTx Transmit DMA Start Address Low Registers (UARTx_START_ADDR_LO_TX)" on page 12-31
0xFFC0 1F08	UART1_COUNT_TX	"UARTx Transmit DMA Count Registers (UARTx_COUNT_TX)" on page 12-32
0xFFC0 1F0A	UART1_NEXT_DESCR_TX	"UARTx Transmit DMA Next Descriptor Pointer Registers (UARTx_NEXT_DESCR_TX)" on page 12-33

Table B-7. UART1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 1F0C	UART1_DESCR_RDY_TX	"UARTx Transmit DMA Descriptor Ready Registers (UARTx_DESCR_RDY_TX)" on page 12-34
0xFFC0 1F0E	UART1_IRQSTAT_TX	"UARTx Transmit DMA IRQ Status Registers (UARTx_IRQSTAT_TX)" on page 12-35

Timer Registers

Table B-8 lists the MMR assignments for the Timer registers (0xFFC0 2000–0xFFC0 23FF).

Table B-8. Timer Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 2000	TIMER0_STATUS	"Timer Status Registers (TIMERx_STATUS)" on page 16-4
0xFFC0 2002	TIMER0_CONFIG	"Timer Configuration Registers (TIMERx_CONFIG)" on page 16-7
0xFFC0 2004	TIMER0_COUNTER_LO	"Timer Counter Registers (TIMERx_COUNTER)" on page 16-11
0xFFC0 2006	TIMER0_COUNTER_HI	"Timer Counter Registers (TIMERx_COUNTER)" on page 16-11
0xFFC0 2008	TIMER0_PERIOD_LO	"Timer Period Registers (TIMERx_PERIOD)" on page 16-10
0xFFC0 200A	TIMER0_PERIOD_HI	"Timer Period Registers (TIMERx_PERIOD)" on page 16-10
0xFFC0 200C	TIMER0_WIDTH_LO	"Timer Width Registers (TIMERx_WIDTH)" on page 16-11

Table B-8. Timer Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 200E	TIMER0_WIDTH_HI	"Timer Width Registers (TIMERx_WIDTH)" on page 16-11
0xFFC0 2010	TIMER1_STATUS	"Timer Status Registers (TIMERx_STATUS)" on page 16-4
0xFFC0 2012	TIMER1_CONFIG	"Timer Configuration Registers (TIMERx_CONFIG)" on page 16-7
0xFFC0 2014	TIMER1_COUNTER_LO	"Timer Counter Registers (TIMERx_COUNTER)" on page 16-11
0xFFC0 2016	TIMER1_COUNTER_HI	"Timer Counter Registers (TIMERx_COUNTER)" on page 16-11
0xFFC0 2018	TIMER1_PERIOD_LO	"Timer Period Registers (TIMERx_PERIOD)" on page 16-10
0xFFC0 201A	TIMER1_PERIOD_HI	"Timer Period Registers (TIMERx_PERIOD)" on page 16-10
0xFFC0 201C	TIMER1_WIDTH_LO	"Timer Width Registers (TIMERx_WIDTH)" on page 16-11
0xFFC0 201E	TIMER1_WIDTH_HI	"Timer Width Registers (TIMERx_WIDTH)" on page 16-11
0xFFC0 2020	TIMER2_STATUS	"Timer Status Registers (TIMERx_STATUS)" on page 16-4
0xFFC0 2022	TIMER2_CONFIG	"Timer Configuration Registers (TIMERx_CONFIG)" on page 16-7
0xFFC0 2024	TIMER2_COUNTER_LO	"Timer Counter Registers (TIMERx_COUNTER)" on page 16-11
0xFFC0 2026	TIMER2_COUNTER_HI	"Timer Counter Registers (TIMERx_COUNTER)" on page 16-11
0xFFC0 2028	TIMER2_PERIOD_LO	"Timer Period Registers (TIMERx_PERIOD)" on page 16-10
0xFFC0 202A	TIMER2_PERIOD_HI	"Timer Period Registers (TIMERx_PERIOD)" on page 16-10

Table B-8. Timer Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 202C	TIMER2_WIDTH_LO	"Timer Width Registers (TIMERx_WIDTH)" on page 16-11
0xFFC0 202E	TIMER2_WIDTH_HI	"Timer Width Registers (TIMERx_WIDTH)" on page 16-11

Programmable Flag Registers

Table B-9 lists the MMR assignments for the Programmable Flag registers (0xFFC0 2400–0xFFC0 27FF).

Table B-9. Programmable Flags Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 2400	FIO_DIR	"Flag Direction Register (FIO_DIR)" on page 15-2
0xFFC0 2404	FIO_FLAG_C	"Flag Set (FIO_FLAG_S) and Flag Clear (FIO_FLAG_C) Registers" on page 15-2
0xFFC0 2406	FIO_FLAG_S	"Flag Set (FIO_FLAG_S) and Flag Clear (FIO_FLAG_C) Registers" on page 15-2
0xFFC0 2408	FIO_MASKA_C	"Flag Interrupt Mask Registers (FIO_MASKA_C, FIO_MASKA_S, FIO_MASKB_C, FIO_MASKB_S)" on page 15-5
0xFFC0 240A	FIO_MASKA_S	"Flag Interrupt Mask Registers (FIO_MASKA_C, FIO_MASKA_S, FIO_MASKB_C, FIO_MASKB_S)" on page 15-5
0xFFC0 240C	FIO_MASKB_C	"Flag Interrupt Mask Registers (FIO_MASKA_C, FIO_MASKA_S, FIO_MASKB_C, FIO_MASKB_S)" on page 15-5

Table B-9. Programmable Flags Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 240E	FIO_MASKB_S	"Flag Interrupt Mask Registers (FIO_MASKA_C, FIO_MASKA_S, FIO_MASKB_C, FIO_MASKB_S)" on page 15-5
0xFFC0 2410	FIO_POLAR	"Flag Polarity Register (FIO_POLAR)" on page 15-9
0xFFC0 2414	FIO_EDGE	"Flag Interrupt Sensitivity Register (FIO_EDGE)" on page 15-10
0xFFC0 2418	FIO_BOTH	"Flag Set on Both Edges Register (FIO_BOTH)" on page 15-10

SPORT0 Controller Registers

Table B-10 lists the MMR assignments for the SPORT0 Controller registers (0xFFC0 2800–0xFFC0 2BFF).

Table B-10. SPORT0 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 2800	SPORT0_TX_CONFIG	"Transmit and Receive Configuration Registers (SPORTx_TX_CONFIG, SPORTx_RX_CONFIG)" on page 11-9
0xFFC0 2802	SPORT0_RX_CONFIG	"Transmit and Receive Configuration Registers (SPORTx_TX_CONFIG, SPORTx_RX_CONFIG)" on page 11-9
0xFFC0 2804	SPORT0_TX	"SPORTx Transmit (SPORTx_TX) Registers" on page 11-17
0xFFC0 2806	SPORT0_RX	"SPORTx Receive (SPORTx_RX) Registers" on page 11-19

Table B-10. SPORT0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2808	SPORT0_TSCLKDIV	"SPORTx Transmit (SPORTx_TSCLKDIV) and Receive (SPORTx_RSCLKDIV) Serial Clock Divider Registers" on page 11-20
0xFFC0 280A	SPORT0_RSCLKDIV	"SPORTx Transmit (SPORTx_TSCLKDIV) and Receive (SPORTx_RSCLKDIV) Serial Clock Divider Registers" on page 11-20
0xFFC0 280C	SPORT0_TFSDIV	"SPORTx Transmit (SPORTx_TFSDIV) and Receive (SPORTx_RFSDIV) Frame Sync Divider Registers" on page 11-22
0xFFC0 280E	SPORT0_RFSDIV	"SPORTx Transmit (SPORTx_TFSDIV) and Receive (SPORTx_RFSDIV) Frame Sync Divider Registers" on page 11-22
0xFFC0 2810	SPORT0_STAT	"SPORTx Status (SPORTx_STAT) Registers" on page 11-23
0xFFC0 2812	SPORT0_MTCS0	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 2814	SPORT0_MTCS1	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 2816	SPORT0_MTCS2	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 2818	SPORT0_MTCS3	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 281A	SPORT0_MTCS4	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25

Table B-10. SPORT0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 281C	SPORT0_MTCS5	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 281E	SPORT0_MTCS6	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 2820	SPORT0_MTCS7	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 2822	SPORT0_MRCS0	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 2824	SPORT0_MRCS1	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 2826	SPORT0_MRCS2	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 2828	SPORT0_MRCS3	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 282A	SPORT0_MRCS4	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 282C	SPORT0_MRCS5	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 282E	SPORT0_MRCS6	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 2830	SPORT0_MRCS7	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27

Table B-10. SPORT0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2832	SPORT0_MCMC1	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 2834	SPORT0_MCMC2	"SPORTx Multichannel Configuration (SPORTx_MCMCx) Registers" on page 11-29
0xFFC0 2A00	SPORT0_CURR_PTR_RX	"SPORTx Receive DMA Current Descriptor Pointer (SPORTx_CURR_PTR_RX) Registers" on page 11-31
0xFFC0 2A02	SPORT0_CONFIG_DMA_RX	"SPORTx Receive DMA Configuration (SPORTx_CONFIG_DMA_RX) Registers" on page 11-31
0xFFC0 2A04	SPORT0_START_ADDR_HI_RX	"SPORTx Receive DMA Start Address High (SPORTx_START_ADDR_HI_RX) Registers" on page 11-34
0xFFC0 2A06	SPORT0_START_ADDR_LO_RX	"SPORTx Receive DMA Start Address Low (SPORTx_START_ADDR_LO_RX) Registers" on page 11-35
0xFFC0 2A08	SPORT0_COUNT_RX	"SPORTx Receive DMA Count (SPORTx_COUNT_RX) Registers" on page 11-36
0xFFC0 2A0A	SPORT0_NEXT_DESCR_RX	"SPORTx Receive DMA Next Descriptor Pointer (SPORTx_NEXT_DESCR_RX) Registers" on page 11-36
0xFFC0 2A0C	SPORT0_DESCR_RDY_RX	"SPORTx Receive DMA Descriptor Ready (SPORTx_DESCR_RDY_RX) Registers" on page 11-37

Table B-10. SPORT0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2A0E	SPORT0_IRQSTAT_RX	"SPORTx Receive DMA IRQ Status (SPORTx_IRQSTAT_RX) Registers" on page 11-38
0xFFC0 2B00	SPORT0_CURR_PTR_TX	"SPORTx Transmit DMA Current Descriptor Pointer (SPORTx_CURR_PTR_TX) Registers" on page 11-39
0xFFC0 2B02	SPORT0_CONFIG_DMA_TX	"SPORTx Transmit DMA Configuration (SPORTx_CONFIG_DMA_TX) Registers" on page 11-40
0xFFC0 2B04	SPORT0_START_ADDR_HI_TX	"SPORTx Transmit DMA Start Address High (SPORTx_START_ADDR_HI_TX) Registers" on page 11-43
0xFFC0 2B06	SPORT0_START_ADDR_LO_TX	"SPORTx Transmit DMA Start Address Low (SPORTx_START_ADDR_LO_TX) Registers" on page 11-44
0xFFC0 2B08	SPORT0_COUNT_TX	"SPORTx Transmit DMA Count (SPORTx_COUNT_TX) Registers" on page 11-45
0xFFC0 2B0A	SPORT0_NEXT_DESCR_TX	"SPORTx Transmit DMA Next Descriptor Pointer (SPORTx_NEXT_DESCR_TX) Registers" on page 11-45
0xFFC0 2B0C	SPORT0_DESCR_RDY_TX	"SPORTx Transmit DMA Descriptor Ready (SPORTx_DESCR_RDY_TX) Registers" on page 11-46
0xFFC0 2B0E	SPORT0_IRQSTAT_TX	"SPORTx Transmit DMA IRQ Status (SPORTx_IRQSTAT_TX) Registers" on page 11-47

SPORT1 Controller Registers

Table B-11 lists the MMR assignments for the SPORT1 Controller registers (0xFFC0 2C00–0xFFC0 2FFF).

Table B-11. SPORT1 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 2C00	SPORT1_TX_CONFIG	"Transmit and Receive Configuration Registers (SPORTx_TX_CONFIG, SPORTx_RX_CONFIG)" on page 11-9
0xFFC0 2C02	SPORT1_RX_CONFIG	"Transmit and Receive Configuration Registers (SPORTx_TX_CONFIG, SPORTx_RX_CONFIG)" on page 11-9
0xFFC0 2C04	SPORT1_TX	"SPORTx Transmit (SPORTx_TX) Registers" on page 11-17
0xFFC0 2C06	SPORT1_RX	"SPORTx Receive (SPORTx_RX) Registers" on page 11-19
0xFFC0 2C08	SPORT1_TSCLKDIV	"SPORTx Transmit (SPORTx_TSCLKDIV) and Receive (SPORTx_RSCLKDIV) Serial Clock Divider Registers" on page 11-20
0xFFC0 2C0A	SPORT1_RSCLKDIV	"SPORTx Transmit (SPORTx_TSCLKDIV) and Receive (SPORTx_RSCLKDIV) Serial Clock Divider Registers" on page 11-20
0xFFC0 2C0C	SPORT1_TFSDIV	"SPORTx Transmit (SPORTx_TFSDIV) and Receive (SPORTx_RFSDIV) Frame Sync Divider Registers" on page 11-22
0xFFC0 2C0E	SPORT1_RFSDIV	"SPORTx Transmit (SPORTx_TFSDIV) and Receive (SPORTx_RFSDIV) Frame Sync Divider Registers" on page 11-22

Table B-11. SPORT1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2C10	SPORT1_STAT	"SPORTx Status (SPORTx_STAT) Registers" on page 11-23
0xFFC0 2C12	SPORT1_MTCS0	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 2C14	SPORT1_MTCS1	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 2C16	SPORT1_MTCS2	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 2C18	SPORT1_MTCS3	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 2C1A	SPORT1_MTCS4	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 2C1C	SPORT1_MTCS5	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 2C1E	SPORT1_MTCS6	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 2C20	SPORT1_MTCS7	"SPORTx Multichannel Transmit Select (SPORTx_MTCSx) Registers" on page 11-25
0xFFC0 2C22	SPORT1_MRCS0	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 2C24	SPORT1_MRCS1	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27

Table B-11. SPORT1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2C26	SPORT1_MRCS2	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 2C28	SPORT1_MRCS3	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 2C2A	SPORT1_MRCS4	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 2C2C	SPORT1_MRCS5	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 2C2E	SPORT1_MRCS6	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 2C30	SPORT1_MRCS7	"SPORTx Multichannel Receive Select (SPORTx_MRCSx) Registers" on page 11-27
0xFFC0 2C32	SPORT1_MCMC1	"SPORTx Multichannel Configuration (SPORTx_MCMCx) Registers" on page 11-29
0xFFC0 2C34	SPORT1_MCMC2	"SPORTx Multichannel Configuration (SPORTx_MCMCx) Registers" on page 11-29
0xFFC0 2E00	SPORT1_CURR_PTR_RX	"SPORTx Receive DMA Current Descriptor Pointer (SPORTx_CURR_PTR_RX) Registers" on page 11-31
0xFFC0 2E02	SPORT1_CONFIG_DMA_RX	"SPORTx Receive DMA Configuration (SPORTx_CONFIG_DMA_RX) Registers" on page 11-31

Table B-11. SPORT1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2E04	SPORT1_START_ADDR_HI_RX	"SPORTx Receive DMA Start Address High (SPORTx_START_ADDR_HI_RX) Registers" on page 11-34
0xFFC0 2E06	SPORT1_START_ADDR_LO_RX	"SPORTx Receive DMA Start Address Low (SPORTx_START_ADDR_LO_RX) Registers" on page 11-35
0xFFC0 2E08	SPORT1_COUNT_RX	"SPORTx Receive DMA Count (SPORTx_COUNT_RX) Registers" on page 11-36
0xFFC0 2E0A	SPORT1_NEXT_DESCR_RX	"SPORTx Receive DMA Next Descriptor Pointer (SPORTx_NEXT_DESCR_RX) Registers" on page 11-36
0xFFC0 2E0C	SPORT1_DESCR_RDY_RX	"SPORTx Receive DMA Descriptor Ready (SPORTx_DESCR_RDY_RX) Registers" on page 11-37
0xFFC0 2E0E	SPORT1_IRQSTAT_RX	"SPORTx Receive DMA IRQ Status (SPORTx_IRQSTAT_RX) Registers" on page 11-38
0xFFC0 2F00	SPORT1_CURR_PTR_TX	"SPORTx Transmit DMA Current Descriptor Pointer (SPORTx_CURR_PTR_TX) Registers" on page 11-39
0xFFC0 2F02	SPORT1_CONFIG_DMA_TX	"SPORTx Transmit DMA Configuration (SPORTx_CONFIG_DMA_TX) Registers" on page 11-40
0xFFC0 2F04	SPORT1_START_ADDR_HI_TX	"SPORTx Transmit DMA Start Address High (SPORTx_START_ADDR_HI_TX) Registers" on page 11-43

Table B-11. SPORT1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 2F06	SPORT1_START_ADDR_LO_TX	"SPORTx Transmit DMA Start Address Low (SPORTx_START_ADDR_LO_TX) Registers" on page 11-44
0xFFC0 2F08	SPORT1_COUNT_TX	"SPORTx Transmit DMA Count (SPORTx_COUNT_TX) Registers" on page 11-45
0xFFC0 2F0A	SPORT1_NEXT_DESCR_TX	"SPORTx Transmit DMA Next Descriptor Pointer (SPORTx_NEXT_DESCR_TX) Registers" on page 11-45
0xFFC0 2F0C	SPORT1_DESCR_RDY_TX	"SPORTx Transmit DMA Descriptor Ready (SPORTx_DESCR_RDY_TX) Registers" on page 11-46
0xFFC0 2F0E	SPORT1_IRQSTAT_TX	"SPORTx Transmit DMA IRQ Status (SPORTx_IRQSTAT_TX) Registers" on page 11-47

SPI0 Controller Registers

Table B-12 lists the MMR assignments for the SPI0 Controller registers (0xFFC0 3000–0xFFC0 33FF).

Table B-12. SPI0 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 3000	SPI0_CTL	"SPIx Control Register (SPIx_CTL)" on page 10-8
0xFFC0 3002	SPI0_FLG	"SPIx Flag Register (SPIx_FLG)" on page 10-10

Table B-12. SPI0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 3004	SPI0_ST	"SPIx Status Register (SPIx_ST)" on page 10-15
0xFFC0 3006	SPI0_TDBR	"SPIx Transmit Data Buffer Register (SPIx_TDBR)" on page 10-17
0xFFC0 3008	SPI0_RDBR	"SPIx Receive Data Buffer Register (SPIx_RDBR)" on page 10-18
0xFFC0 300A	SPI0_BAUD	"SPIx Baud Rate Register (SPIx_BAUD)" on page 10-7
0xFFC0 300C	SPI0_SHADOW	"SPIx RDBR Shadow Register (SPIx_SHADOW)" on page 10-18
0xFFC0 3200	SPI0_CURR_PTR	"SPIx DMA Current Descriptor Pointer Register (SPIx_CURR_PTR)" on page 10-20
0xFFC0 3202	SPI0_CONFIG	"SPIx DMA Configuration Register (SPIx_CONFIG)" on page 10-21
0xFFC0 3204	SPI0_START_ADDR_HI	"SPIx DMA Start Address High Register (SPIx_START_ADDR_HI) and SPIx DMA Start Address Low Register (SPIx_START_ADDR_LO)" on page 10-22
0xFFC0 3206	SPI0_START_ADDR_LO	"SPIx DMA Start Address High Register (SPIx_START_ADDR_HI) and SPIx DMA Start Address Low Register (SPIx_START_ADDR_LO)" on page 10-22
0xFFC0 3208	SPI0_COUNT	"SPIx DMA Count Register (SPIx_COUNT)" on page 10-23
0xFFC0 320A	SPI0_NEXT_DESCR	"SPIx DMA Next Descriptor Pointer Register (SPIx_NEXT_DESCR)" on page 10-24
0xFFC0 320C	SPI0_DESCR_RDY	"SPIx DMA Descriptor Ready Register (SPIx_DESCR_RDY)" on page 10-25
0xFFC0 320E	SPI0_DMA_INT	"SPIx DMA Interrupt Register (SPIx_DMA_INT)" on page 10-26

SPI1 Controller Registers

Table B-13 lists the MMR assignments for the SPI1 Controller registers (0xFFC0 3400–0xFFC0 37FF).

Table B-13. SPI1 Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 3400	SPI1_CTL	"SPIx Control Register (SPIx_CTL)" on page 10-8
0xFFC0 3402	SPI1_FLG	"SPIx Flag Register (SPIx_FLG)" on page 10-10
0xFFC0 3404	SPI1_ST	"SPIx Status Register (SPIx_ST)" on page 10-15
0xFFC0 3406	SPI1_TDBR	"SPIx Transmit Data Buffer Register (SPIx_TDBR)" on page 10-17
0xFFC0 3408	SPI1_RDBR	"SPIx Receive Data Buffer Register (SPIx_RDBR)" on page 10-18
0xFFC0 340A	SPI1_BAUD	"SPIx Baud Rate Register (SPIx_BAUD)" on page 10-7
0xFFC0 340C	SPI1_SHADOW	"SPIx RDBR Shadow Register (SPIx_SHADOW)" on page 10-18
0xFFC0 3600	SPI1_CURR_PTR	"SPIx DMA Current Descriptor Pointer Register (SPIx_CURR_PTR)" on page 10-20
0xFFC0 3602	SPI1_CONFIG	"SPIx DMA Configuration Register (SPIx_CONFIG)" on page 10-21
0xFFC0 3604	SPI1_START_ADDR_HI	"SPIx DMA Start Address High Register (SPIx_START_ADDR_HI) and SPIx DMA Start Address Low Register (SPIx_START_ADDR_LO)" on page 10-22
0xFFC0 3606	SPI1_START_ADDR_LO	"SPIx DMA Start Address High Register (SPIx_START_ADDR_HI) and SPIx DMA Start Address Low Register (SPIx_START_ADDR_LO)" on page 10-22

Table B-13. SPI1 Controller Registers (Cont'd)

Memory- Mapped Address	Register Name	See Section
0xFFC0 3608	SPI1_COUNT	"SPIx DMA Count Register (SPIx_COUNT)" on page 10-23
0xFFC0 360A	SPI1_NEXT_DESCR	"SPIx DMA Next Descriptor Pointer Register (SPIx_NEXT_DESCR)" on page 10-24
0xFFC0 360C	SPI1_DESCR_RDY	"SPIx DMA Descriptor Ready Register (SPIx_DESCR_RDY)" on page 10-25
0xFFC0 360E	SPI1_DMA_INT	"SPIx DMA Interrupt Register (SPIx_DMA_INT)" on page 10-26

Memory DMA Controller Registers

Table B-14 lists the MMR assignments for the Memory DMA Controller registers (0xFFC0 3800–0xFFC0 3BFF).

Table B-14. Memory DMA Controller Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 3800	MDD_DCP	"Destination Memory DMA Current Descriptor Pointer Register (MDD_DCP)" on page 9-37
0xFFC0 3802	MDD_DCFG	"Destination Memory DMA Configuration Register (MDD_DCFG)" on page 9-33
0xFFC0 3804	MDD_DSAH	"Destination Memory DMA Start Address Registers (MDD_DSAH, MDD_DSAL)" on page 9-35
0xFFC0 3806	MDD_DSAL	"Destination Memory DMA Start Address Registers (MDD_DSAH, MDD_DSAL)" on page 9-35
0xFFC0 3808	MDD_DCT	"Source Memory DMA Transfer Count Register (MDD_DCT)" on page 9-40

Table B-14. Memory DMA Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 380A	MDD_DND	"Destination Memory DMA Next Descriptor Pointer Register (MDD_DND)" on page 9-36
0xFFC0 380C	MDD_DDR	"Destination Memory DMA Descriptor Ready Register (MDD_DDR)" on page 9-36
0xFFC0 380E	MDD_DI	"Source Memory DMA Interrupt Register (MDS_DI)" on page 9-43
0xFFC0 3900	MDS_DCP	"Source Memory DMA Current Descriptor Pointer Register (MDS_DCP)" on page 9-43
0xFFC0 3902	MDS_DCFG	"Source Memory DMA Configuration Register (MDS_DCFG)" on page 9-39
0xFFC0 3904	MDS_DSAH	"Source Memory DMA Start Address Registers (MDS_DSAH, MDS_DSAL)" on page 9-41
0xFFC0 3906	MDS_DSAL	"Source Memory DMA Start Address Registers (MDS_DSAH, MDS_DSAL)" on page 9-41
0xFFC0 3908	MDS_DCT	"Source Memory DMA Transfer Count Register (MDD_DCT)" on page 9-40
0xFFC0 390A	MDS_DND	"Destination Memory DMA Next Descriptor Pointer Register (MDD_DND)" on page 9-36
0xFFC0 390C	MDS_DDR	"Destination Memory DMA Descriptor Ready Register (MDD_DDR)" on page 9-36
0xFFC0 390E	MDS_DI	"Destination Memory DMA Interrupt Register (MDD_DI)" on page 9-38

Asynchronous Memory Controller—EBIU

Table B-15 lists the MMR assignments for the Asynchronous Memory Controller - External Bus Interface Unit registers (0xFFC0 3C00–0xFFC0 3FFF).

Table B-15. External Bus Interface Unit Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 3C00	EBIU_AMGCTL	"Asynchronous Memory Global Control Register (EBIU_AMGCTL)" on page 18-10
0xFFC0 3C04	EBIU_AMBCTL0	"Asynchronous Memory Bank Control Registers (EBIU_AMBCTL0, EBIU_AMBCTL1)" on page 18-12
0xFFC0 3C08	EBIU_AMBCTL1	"Asynchronous Memory Bank Control Registers (EBIU_AMBCTL0, EBIU_AMBCTL1)" on page 18-12

PCI Bridge Registers

Table B-16 lists the MMR assignments for the PCI Bridge registers (ranges 0xFFC0 4000–0xFFC0 43FF and 0xEEFF FF00–0xEEFF FFFF).

Table B-16. PCI Bridge Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 4000	PCI_CTL	"PCI Bridge Control Register (PCI_CTL)" on page 13-20
0xFFC0 4004	PCI_STAT	"PCI Status Register (PCI_STAT)" on page 13-21
0xFFC0 4008	PCI_ICTL	"PCI Interrupt Controller Register (PCI_ICTL)" on page 13-22

Table B-16. PCI Bridge Registers (Cont'd)

Memory- Mapped Address	Register Name	See Section
0xFFC0 400C	PCI_MBAP	"PCI Outbound Memory Base Address Register (PCI_MBAP)" on page 13-23
0xFFC0 4010	PCI_IBAP	"PCI Outbound I/O Base Address Register (PCI_IBAP)" on page 13-23
0xFFC0 4014	PCI_CBAP	"PCI Outbound I/O Configuration Address Register (PCI_CBAP)" on page 13-24
0xFFC0 4018	PCI_TMBAP	"PCI Inbound Memory Base Address Register (PCI_TMBAP)" on page 13-25
0xFFC0 401C	PCI_TIBAP	"PCI Inbound I/O Base Address Register (PCI_TIBAP)" on page 13-25
0xEEFF FF00	PCI_DMBARM	"PCI Device Memory BAR Mask Register (PCI_DMBARM)" on page 13-26
0xEEFF FF04	PCI_DIBARM	"PCI Device I/O BAR Mask Register (PCI_DIBARM)" on page 13-27
0xEEFF FF08	PCI_CFG_DIC	"PCI Configuration Device ID Register (PCI_CFG_DIC)" on page 13-29
0xEEFF FF0C	PCI_CFG_VIC	"PCI Configuration Vendor ID Register (PCI_CFG_VIC)" on page 13-30
0xEEFF FF10	PCI_CFG_STAT	"PCI Configuration Status Register (PCI_CFG_STAT)" on page 13-31
0xEEFF FF14	PCI_CFG_CMD	"PCI Configuration Command Register (PCI_CFG_CMD)" on page 13-32
0xEEFF FF18	PCI_CFG_CC	"PCI Configuration Class Code Register (PCI_CFG_CC)" on page 13-33
0xEEFF FF1C	PCI_CFG_RID	"PCI Configuration Revision ID Register (PCI_CFG_RID)" on page 13-34
0xEEFF FF20	PCI_CFG_BIST	"PCI Configuration BIST Register (PCI_CFG_BIST)" on page 13-35
0xEEFF FF24	PCI_CFG_HT	"PCI Configuration Header Type Register (PCI_CFG_HT)" on page 13-36

Table B-16. PCI Bridge Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xEEFF FF28	PCI_CFG_MLT	"PCI Configuration Memory Latency Timer Register (PCI_CFG_MLT)" on page 13-36
0xEEFF FF2C	PCI_CFG_CLS	"PCI Configuration Cache Line Size Register (PCI_CFG_CLS)" on page 13-37
0xEEFF FF30	PCI_CFG_MBAR	"PCI Configuration Memory Base Address Register (PCI_CFG_MBAR)" on page 13-38
0xEEFF FF34	PCI_CFG_IBAR	"PCI Configuration I/O Base Address Register (PCI_CFG_IBAR)" on page 13-39
0xEEFF FF38	PCI_CFG_SID	"PCI Configuration Subsystem ID Register (PCI_CFG_SID)" on page 13-40
0xEEFF FF3C	PCI_CFG_SVID	"PCI Configuration Subsystem Vendor ID Register (PCI_CFG_SVID)" on page 13-40
0xEEFF FF40	PCI_CFG_MAXL	"PCI Configuration Maximum Latency Register (PCI_CFG_MAXL)" on page 13-41
0xEEFF FF44	PCI_CFG_MING	"PCI Configuration Minimum Grant Register (PCI_CFG_MING)" on page 13-42
0xEEFF FF48	PCI_CFG_IP	"PCI Configuration Interrupt Pin Register (PCI_CFG_IP)" on page 13-42
0xEEFF FF4C	PCI_CFG_IL	"PCI Configuration Interrupt Line Register (PCI_CFG_IL)" on page 13-43
0xEEFF FF50	PCI_HMCTL	"PCI Host Memory Control Register (PCI_HMCTL)" on page 13-43

USB Device Registers

Table B-17 lists the MMR assignments for the USB Device registers (0xFFC0 4400–0xFFC0 47FF).

Table B-17. USB Device Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 4400	USBD_ID	"USB Device ID Register (USBD_ID)" on page 14-16
0xFFC0 4402	USBD_FRM	"Current USB Frame Number Register (USBD_FRM)" on page 14-17
0xFFC0 4404	USBD_FRMAT	"Match Value for USB Frame Number Register (USBD_FRMAT)" on page 14-17
0xFFC0 4406	USBD_EPBUF	"Enable Download of Configuration Into UDC Core Register (USBD_EPBUF)" on page 14-18
0xFFC0 4408	USBD_STAT	"USBD Module Status Register (USBD_STAT)" on page 14-19
0xFFC0 440A	USBD_CTRL	"USBD Module Configuration and Control Register (USBD_CTRL)" on page 14-21
0xFFC0 440C	USBD_GINTR	"Global Interrupt Register (USBD_GINTR)" on page 14-22
0xFFC0 440E	USBD_GMASK	"Global Interrupt Mask Register (USBD_GMASK)" on page 14-24
0xFFC0 4440	USBD_DMACFG	"DMA Master Channel Configuration Register (USBD_DMACFG)" on page 14-24
0xFFC0 4442	USBD_DMABL	"DMA Master Channel Base Address Low Register (USBD_DMABL)" on page 14-25
0xFFC0 4444	USBD_DMABH	"DMA Master Channel Base Address High Register (USBD_DMABH)" on page 14-26
0xFFC0 4446	USBD_DMACT	"DMA Master Channel Count Register (USBD_DMACT)" on page 14-26

Table B-17. USB Device Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 4448	USBD_DMAIRQ	"DMA Master Channel DMA Interrupt Register (USBD_DMAIRQ)" on page 14-27
0xFFC0 4480	USBD_INTR0	"USB Endpoint x Interrupt Registers (USBD_INTRx)" on page 14-27
0xFFC0 4482	USBD_MASK0	"USB Endpoint x Mask Registers (USBD_MASKx)" on page 14-29
0xFFC0 4484	USBD_EPCFG0	"USB Endpoint x Control Registers (USBD_EPCFGx)" on page 14-31
0xFFC0 4486	USBD_EPADR0	"USB Endpoint x Address Offset Registers (USBD_EPADRx)" on page 14-33
0xFFC0 4488	USBD_EPLEN0	"USB Endpoint x Buffer Length Registers (USBD_EPLENx)" on page 14-34
0xFFC0 448A	USBD_INTR1	"USB Endpoint x Interrupt Registers (USBD_INTRx)" on page 14-27
0xFFC0 448C	USBD_MASK1	"USB Endpoint x Mask Registers (USBD_MASKx)" on page 14-29
0xFFC0 448E	USBD_EPCFG1	"USB Endpoint x Control Registers (USBD_EPCFGx)" on page 14-31
0xFFC0 4490	USBD_EPADR1	"USB Endpoint x Address Offset Registers (USBD_EPADRx)" on page 14-33
0xFFC0 4492	USBD_EPLEN1	"USB Endpoint x Buffer Length Registers (USBD_EPLENx)" on page 14-34
0xFFC0 4494	USBD_INTR2	"USB Endpoint x Interrupt Registers (USBD_INTRx)" on page 14-27
0xFFC0 4496	USBD_MASK2	"USB Endpoint x Mask Registers (USBD_MASKx)" on page 14-29
0xFFC0 4498	USBD_EPCFG2	"USB Endpoint x Control Registers (USBD_EPCFGx)" on page 14-31
0xFFC0 449A	USBD_EPADR2	"USB Endpoint x Address Offset Registers (USBD_EPADRx)" on page 14-33

Table B-17. USB Device Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 449C	USBD_EPLEN2	"USB Endpoint x Buffer Length Registers (USBD_EPLENx)" on page 14-34
0xFFC0 449E	USBD_INTR3	"USB Endpoint x Interrupt Registers (USBD_INTRx)" on page 14-27
0xFFC0 44A0	USBD_MASK3	"USB Endpoint x Mask Registers (USBD_MASKx)" on page 14-29
0xFFC0 44A2	USBD_EPCFG3	"USB Endpoint x Control Registers (USBD_EPCFGx)" on page 14-31
0xFFC0 44A4	USBD_EPADR3	"USB Endpoint x Address Offset Registers (USBD_EPADRx)" on page 14-33
0xFFC0 44A6	USBD_EPLEN3	"USB Endpoint x Buffer Length Registers (USBD_EPLENx)" on page 14-34
0xFFC0 44A8	USBD_INTR4	"USB Endpoint x Interrupt Registers (USBD_INTRx)" on page 14-27
0xFFC0 44AA	USBD_MASK4	"USB Endpoint x Mask Registers (USBD_MASKx)" on page 14-29
0xFFC0 44AC	USBD_EPCFG4	"USB Endpoint x Control Registers (USBD_EPCFGx)" on page 14-31
0xFFC0 44AE	USBD_EPADR4	"USB Endpoint x Address Offset Registers (USBD_EPADRx)" on page 14-33
0xFFC0 44B0	USBD_EPLEN4	"USB Endpoint x Buffer Length Registers (USBD_EPLENx)" on page 14-34
0xFFC0 44B2	USBD_INTR5	"USB Endpoint x Interrupt Registers (USBD_INTRx)" on page 14-27
0xFFC0 44B4	USBD_MASK5	"USB Endpoint x Mask Registers (USBD_MASKx)" on page 14-29
0xFFC0 44B6	USBD_EPCFG5	"USB Endpoint x Control Registers (USBD_EPCFGx)" on page 14-31
0xFFC0 44B8	USBD_EPADR5	"USB Endpoint x Address Offset Registers (USBD_EPADRx)" on page 14-33

Table B-17. USB Device Registers (Cont'd)

Memory-Mapped Address	Register Name	See Section
0xFFC0 44BA	USBD_EPLEN5	"USB Endpoint x Buffer Length Registers (USBD_EPLENx)" on page 14-34
0xFFC0 44BC	USBD_INTR6	"USB Endpoint x Interrupt Registers (USBD_INTRx)" on page 14-27
0xFFC0 44BE	USBD_MASK6	"USB Endpoint x Mask Registers (USBD_MASKx)" on page 14-29
0xFFC0 44C0	USBD_EPCFG6	"USB Endpoint x Control Registers (USBD_EPCFGx)" on page 14-31
0xFFC0 44C2	USBD_EPADR6	"USB Endpoint x Address Offset Registers (USBD_EPADRx)" on page 14-33
0xFFC0 44C4	USBD_EPLEN6	"USB Endpoint x Buffer Length Registers (USBD_EPLENx)" on page 14-34
0xFFC0 44C6	USBD_INTR7	"USB Endpoint x Interrupt Registers (USBD_INTRx)" on page 14-27
0xFFC0 44C8	USBD_MASK7	"USB Endpoint x Mask Registers (USBD_MASKx)" on page 14-29
0xFFC0 44CA	USBD_EPCFG7	"USB Endpoint x Control Registers (USBD_EPCFGx)" on page 14-31
0xFFC0 44CC	USBD_EPADR7	"USB Endpoint x Address Offset Registers (USBD_EPADRx)" on page 14-33
0xFFC0 44CE	USBD_EPLEN7	"USB Endpoint x Buffer Length Registers (USBD_EPLENx)" on page 14-34

System DMA Control Registers

Table B-18 lists the MMR assignments for the System DMA Control registers (0xFFC0 4880–0xFFC0 488F).

Table B-18. System DMA Control Registers

Memory-Mapped Address	Register Name	See Section
0xFFC0 4880	DMA_DBP	"DMA Descriptor Base Pointer Register (DMA_DBP)" on page 9-24
0xFFC0 4884	DB_ACOMP	"DMA Bus Address Comparator Register (DB_ACOMP)" on page 20-29
0xFFC0 4888	DB_CCOMP	"DMA Bus Control Comparator Register (DB_CCOMP)" on page 20-28

SDRAM Controller External Bus Interface Unit

Table B-19 lists the MMR assignments for the SDRAM Controller External Bus Interface Unit registers (0xFFC0 4C00–0xFFC0 4FFF).

Table B-19. SDRAM Controller External Bus Interface Unit

Memory- Mapped Address	Register Name	See Section
0xFFC0 4C00	EBIU_SDGCTL	"SDRAM Memory Global Control Register (EBIU_SDGCTL)" on page 18-37
0xFFC0 4C04	EBIU_SDBCTL	"SDRAM Memory Bank Control Register (EBIU_SDBCTL)" on page 18-49
0xFFC0 4C0A	EBIU_SDRRC	"SDRAM Refresh Rate Control Register (EBIU_SDRRC)" on page 18-54
0xFFC0 4C0E	EBIU_SDSTAT	"SDRAM Control Status Register (EBIU_SDSTAT)" on page 18-53

C TEST FEATURES

The ADSP-BF535 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 and 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.

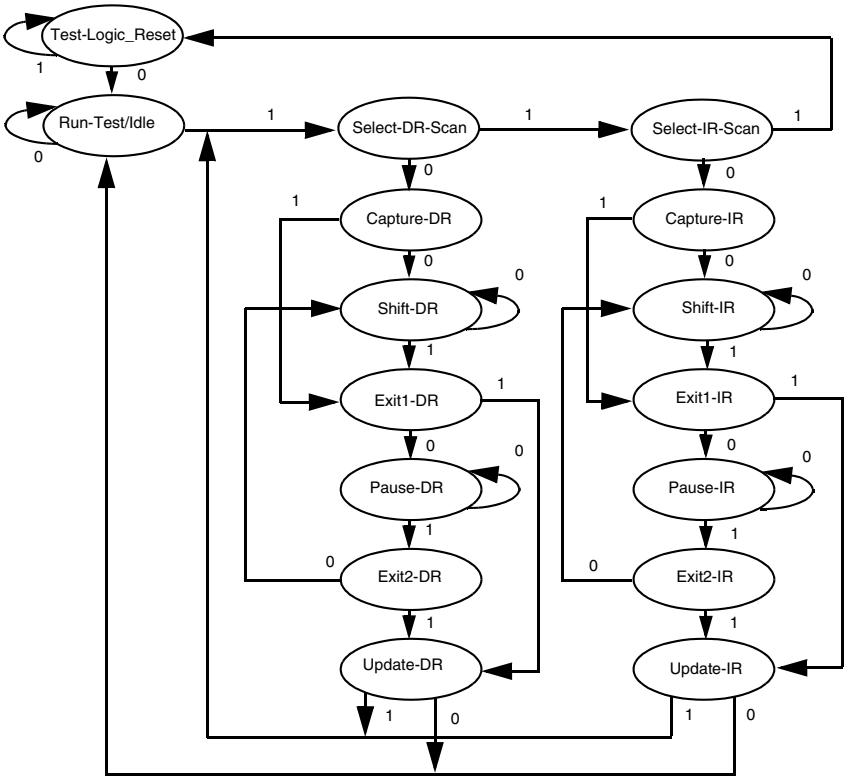


Figure C-1. TAP Controller State Diagram

Note:

- The TAP controller enters the Test-Logic-Reset state when TMS is held high after five (5) 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 shows the decode for the public instructions. The Register column shows the serial scan paths.

Table C-2. Decode for Public JTAG-Scan Instructions

Instruction Name	Binary Decode 01234	Register
EXTEST	00000	Boundary-Scan
MBIST_SCAN	00001	
IDCODE	00010	CHIPID
MEMTEST_SCAN	00011	
DBG_SCAN	00100	
WRAPPER_SCAN	00101	
RUNMBIST	00111	
EMUIR_SCAN	01000	
RESMBIST	01001	
DC_MBIST_SOF	01011	
IC_MBIST_SOF	01101	
WRAPPER_MODE	01111	
SAMPLE/PRELOAD	00001	Boundary-Scan
RS_WRAPPER_MODE	10010	
EMUDAT_SCAN	10100	

Table C-2. Decode for Public JTAG-Scan Instructions (Cont'd)

Instruction Name	Binary Decode 01234	Register
EMUPC_SCAN	10110	
BYPASS	11111	Bypass

Figure C-2 shows the instruction bit scan ordering for the paths shown in Table C-2.

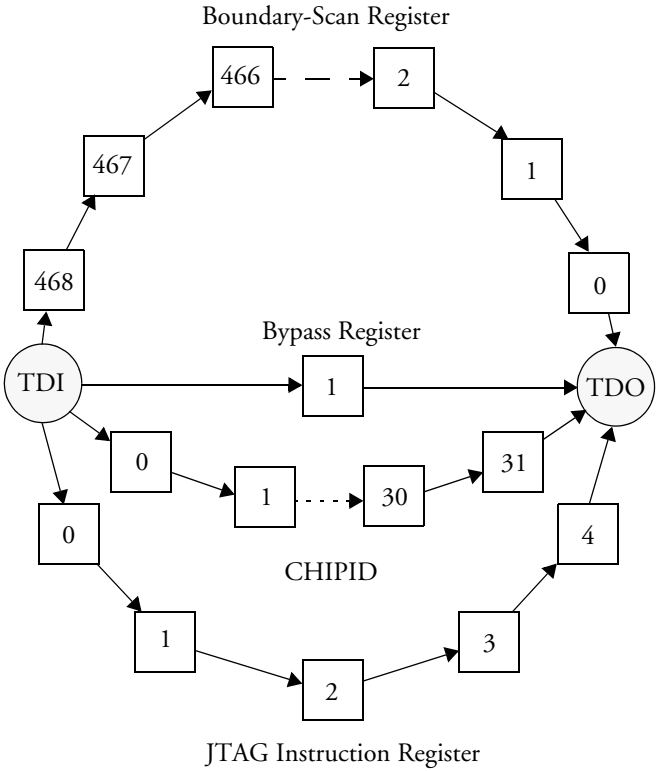


Figure C-2. Serial Scan Paths

Public Instructions

The following are descriptions of 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.

EXTEST 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 ADSP-BF535 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.

IDCODE – Binary Code 00010

To connect with TDI and TDO, the IDCODE instruction selects the Chip ID register (CHIPID), shown in Figure C-3. The instruction has no effect on the internal logic.

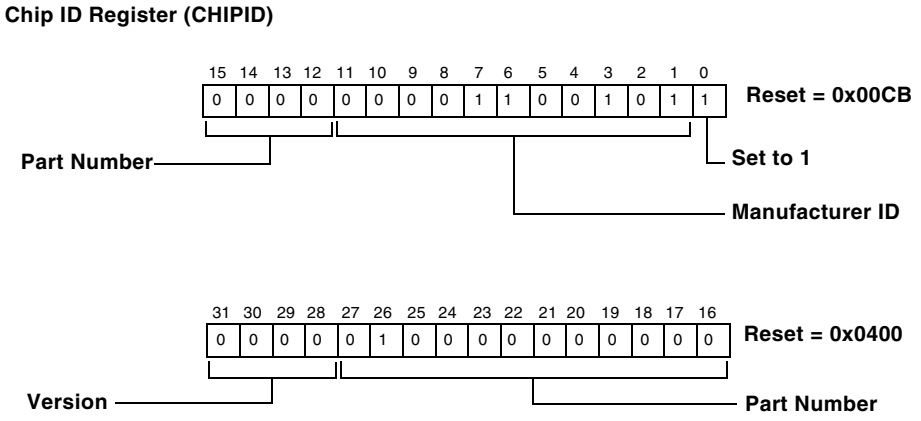


Figure C-3. Chip ID Register

As shown in Figure C-2, CHIPID is scanned out MSB first. For more information on CHIPID, see "Product Identification Registers" on page 20-25.

Per the JTAG standard, CHIPID is selected by default during the Test-Logic-Reset controller state.

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 469 bits long. Table C-3 lists and defines the latch type and function of each position in the scan path. The positions are numbered from 0 to 468. Bit 0 is the first bit output (closest to TDO) and bit 468 is the last bit output (closest to TDI).

Table C-3. Scan Path Position Definitions

Position	Type	Signal Name
0	I	NMI
1	O	SUSPEND
2	OE	SUSPEND OE
3	O	$\overline{\text{TXEN}}$
4	OE	$\overline{\text{TXEN}}$ OE
5	O	TXDMNS
6	OE	TXDMNS OE
7	O	TXDPLS
8	OE	TXDPLS OE
9	I	DMNS
10	I	DPLS
11	I	$\overline{\text{RESET}}$
12	I	XVER_DATA
13	I	USB_CLK
14	O	TX1
15	OE	TX1 OE
16	I	RX1
17	O	TX0
18	OE	TX0 OE
19	I	RX0

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
20	O	TMR(2)
21	OE	TMR(2) OE
22	I	TMR(2)
23	O	TMR(1)
24	OE	TMR(1) OE
25	I	TMR(1)
26	O	TMR(0)
27	OE	TMR(0) OE
28	I	TMR(0)
29	O	ADDR(2)
30	OE	ADDR(2) OE
31	O	ADDR(3)
32	OE	ADDR(3) OE
33	O	ADDR(4)
34	OE	ADDR(4) OE
35	O	ADDR(5)
36	OE	ADDR(5) OE
37	O	ADDR(6)
38	OE	ADDR(6) OE
39	O	ADDR(7)
40	OE	ADDR(7) OE
41	O	ADDR(8)
42	OE	ADDR(8) OE
43	O	ADDR(9)
44	OE	ADDR(9) OE
45	O	ADDR(10)

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
46	OE	ADDR(10) OE
47	O	ADDR(11)
48	OE	ADDR(11) OE
49	O	ADDR(12)
50	OE	ADDR(12) OE
51	O	ADDR(13)
52	OE	ADDR(13) OE
53	O	ADDR(14)
54	OE	ADDR(14) OE
55	O	ADDR(15)
56	OE	ADDR(15) OE
57	O	ADDR(16)
58	OE	ADDR(16) OE
59	O	ADDR(17)
60	OE	ADDR(17) OE
61	O	ADDR(18)
62	OE	ADDR(18) OE
63	O	ADDR(19)
64	OE	ADDR(19) OE
65	O	ADDR(20)
66	OE	ADDR(20) OE
67	O	ADDR(21)
68	OE	ADDR(21) OE
69	O	ADDR(22)
70	OE	ADDR(22) OE
71	O	ADDR(23)

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
72	OE	ADDR(23) OE
73	O	ADDR(24)
74	OE	ADDR(24) OE
75	O	ADDR(25)
76	OE	ADDR(25) OE
77	O	$\overline{\text{ABE\_SDQM(0)}}$
78	OE	$\overline{\text{ABE\_SDQM(0)}}$ OE
79	O	$\overline{\text{ABE\_SDQM(1)}}$
80	OE	$\overline{\text{ABE\_SDQM(1)}}$ OE
81	O	$\overline{\text{ABE\_SDQM(2)}}$
82	OE	$\overline{\text{ABE\_SDQM(2)}}$ OE
83	O	$\overline{\text{ABE\_SDQM(3)}}$
84	OE	$\overline{\text{ABE\_SDQM(3)}}$ OE
85	O	$\overline{\text{AMS(0)}}$
86	OE	$\overline{\text{AMS(0)}}$ OE
87	O	$\overline{\text{AMS(1)}}$
88	OE	$\overline{\text{AMS(1)}}$ OE
89	O	$\overline{\text{AMS(2)}}$
90	OE	$\overline{\text{AMS(2)}}$ OE
91	O	$\overline{\text{AMS(3)}}$
92	OE	$\overline{\text{AMS(3)}}$ OE
93	O	$\overline{\text{AOE}}$
94	OE	$\overline{\text{AOE}}$ OE
95	O	$\overline{\text{ARE}}$
96	OE	$\overline{\text{ARE}}$ OE
97	O	$\overline{\text{AWE}}$

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
98	OE	$\overline{\text{AWE}}$ OE
99	O	CLKOUT_SCLK1
100	--	Reserved
101	O	SCLK0
102	--	Reserved
103	O	SCKE
104	OE	SCKE OE
105	O	SA10
106	OE	SA10 OE
107	O	$\overline{\text{SRAS}}$
108	OE	$\overline{\text{SRAS}}$ OE
109	O	$\overline{\text{SWE}}$
110	OE	$\overline{\text{SWE}}$ OE
111	O	$\overline{\text{SMS(3)}}$
112	OE	$\overline{\text{SMS(3)}}$ OE
113	O	$\overline{\text{SMS(2)}}$
114	OE	$\overline{\text{SMS(2)}}$ OE
115	O	$\overline{\text{SMS(1)}}$
116	OE	$\overline{\text{SMS(1)}}$ OE
117	O	$\overline{\text{SMS(0)}}$
118	OE	$\overline{\text{SMS(0)}}$ OE
119	O	$\overline{\text{SCAS}}$
120	OE	$\overline{\text{SCAS}}$ OE
121	I	ARDY
122	O	DATA(0)
123	OE	DATA(0) OE

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
124	I	DATA(0)
125	O	DATA(1)
126	OE	DATA(1) OE
127	I	DATA(1)
128	O	DATA(2)
129	OE	DATA(2) OE
130	I	DATA(2)
131	O	DATA(3)
132	OE	DATA(3) OE
133	I	DATA(3)
134	O	DATA(4)
135	OE	DATA(4) OE
136	I	DATA(4)
137	O	DATA(5)
138	OE	DATA(5) OE
139	I	DATA(5)
140	O	DATA(6)
141	OE	DATA(6) OE
142	I	DATA(6)
143	O	DATA(7)
144	OE	DATA(7) OE
145	I	DATA(7)
146	O	DATA(8)
147	OE	DATA(8) OE
148	I	DATA(8)
149	O	DATA(9)

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
150	OE	DATA(9) OE
151	I	DATA(9)
152	O	DATA(10)
153	OE	DATA(10) OE
154	I	DATA(10)
155	O	DATA(11)
156	OE	DATA(11) OE
157	I	DATA(11)
158	O	DATA(12)
159	OE	DATA(12) OE
160	I	DATA(12)
161	O	DATA(13)
162	OE	DATA(13) OE
163	I	DATA(13)
164	O	DATA(14)
165	OE	DATA(14) OE
166	I	DATA(14)
167	O	DATA(15)
168	OE	DATA(15) OE
169	I	DATA(15)
170	O	DATA(16)
171	OE	DATA(16) OE
172	I	DATA(16)
173	O	DATA(17)
174	OE	DATA(17) OE
175	I	DATA(17)

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
176	O	DATA(18)
177	OE	DATA(18) OE
178	I	DATA(18)
179	O	DATA(19)
180	OE	DATA(19) OE
181	I	DATA(19)
182	O	DATA(20)
183	OE	DATA(20) OE
184	I	DATA(20)
185	O	DATA(21)
186	OE	DATA(21) OE
187	I	DATA(21)
188	O	DATA(22)
189	OE	DATA(22) OE
190	I	DATA(22)
191	O	DATA(23)
192	OE	DATA(23) OE
193	I	DATA(23)
194	O	DATA(24)
195	OE	DATA(24) OE
196	I	DATA(24)
197	O	DATA(25)
198	OE	DATA(25) OE
199	I	DATA(25)
200	O	DATA(26)
201	OE	DATA(26) OE

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
202	I	DATA(26)
203	O	DATA(27)
204	OE	DATA(27) OE
205	I	DATA(27)
206	O	DATA(28)
207	OE	DATA(28) OE
208	I	DATA(28)
209	O	DATA(29)
210	OE	DATA(29) OE
211	I	DATA(29)
212	O	DATA(30)
213	OE	DATA(30) OE
214	I	DATA(30)
215	O	DATA(31)
216	OE	DATA(31) OE
217	I	DATA(31)
218	O	PF(0)
219	OE	PF(0) OE
220	I	PF(0)
221	O	PF(1)
222	OE	PF(1) OE
223	I	PF(1)
224	O	PF(2)
225	OE	PF(2) OE
226	I	PF(2)
227	O	PF(3)

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
228	OE	PF(3) OE
229	I	PF(3)
230	O	PF(4)
231	OE	PF(4) OE
232	I	PF(4)
233	O	PF(5)
234	OE	PF(5) OE
235	I	PF(5)
236	O	PF(6)
237	OE	PF(6) OE
238	I	PF(6)
239	O	PF(7)
240	OE	PF(7) OE
241	I	PF(7)
242	O	PF(8)
243	OE	PF(8) OE
244	I	PF(8)
245	O	PF(9)
246	OE	PF(9) OE
247	I	PF(9)
248	O	PF(10)
249	OE	PF(10) OE
250	I	PF(10)
251	O	PF(11)
252	OE	PF(11) OE
253	I	PF(11)

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
254	O	PF(12)
255	OE	PF(12) OE
256	I	PF(12)
257	O	PF(13)
258	OE	PF(13) OE
259	I	PF(13)
260	O	PF(14)
261	OE	PF(14) OE
262	I	PF(14)
263	O	PF(15)
264	OE	PF(15) OE
265	I	PF(15)
266	O	RSCLK0
267	OE	RSCLK0 OE
268	I	RSCLK0
269	O	RFS0
270	OE	RFS0 OE
271	I	RFS0
272	I	DR0
273	O	TSCLK0
274	OE	TSCLK0 OE
275	I	TSCLK0
276	O	TFS0
277	OE	TFS0 OE
278	I	TFS0
279	O	DT0

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
280	OE	DT0 OE
281	O	RSCLK1
282	OE	RSCLK1 OE
283	I	RSCLK1
284	O	RFS1
285	OE	RFS1 OE
286	I	RFS1
287	I	DR1
288	O	TSCLK1
289	OE	TSCLK1 OE
290	I	TSCLK1
291	O	TFS1
292	OE	TFS1 OE
293	I	TFS1
294	O	DT1
295	OE	DT1 OE
296	O	MOSI0
297	OE	MOSI0 OE
298	I	MOSI0
299	O	MISO0
300	OE	MISO0 OE
301	I	MISO0
302	O	SCK0
303	OE	SCK0 OE
304	I	SCK0
305	O	MOSI1

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
306	OE	MOSI1 OE
307	I	MOSI1
308	O	MISO1
309	OE	MISO1 OE
310	I	MISO1
311	O	SCK1
312	OE	SCK1 OE
313	I	SCK1
314	O	PCI_AD(31)
315	OE	PCI_AD(31) OE
316	I	PCI_AD(31)
317	O	PCI_AD(30)
318	OE	PCI_AD(30) OE
319	I	PCI_AD(30)
320	O	PCI_AD(29)
321	OE	PCI_AD(29) OE
322	I	PCI_AD(29)
323	O	PCI_AD(28)
324	OE	PCI_AD(28) OE
325	I	PCI_AD(28)
326	O	PCI_AD(27)
327	OE	PCI_AD(27) OE
328	I	PCI_AD(27)
329	O	PCI_AD(26)
330	OE	PCI_AD(26) OE
331	I	PCI_AD(26)

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
332	O	PCI_AD(25)
333	OE	PCI_AD(25) OE
334	I	PCI_AD(25)
335	O	PCI_AD(24)
336	OE	PCI_AD(24) OE
337	I	PCI_AD(24)
338	O	PCI_AD(23)
339	OE	PCI_AD(23) OE
340	I	PCI_AD(23)
341	O	PCI_AD(22)
342	OE	PCI_AD(22) OE
343	I	PCI_AD(22)
344	O	PCI_AD(21)
345	OE	PCI_AD(21) OE
346	I	PCI_AD(21)
347	O	PCI_AD(20)
348	OE	PCI_AD(20) OE
349	I	PCI_AD(20)
350	O	PCI_AD(19)
351	OE	PCI_AD(19) OE
352	I	PCI_AD(19)
353	O	PCI_AD(18)
354	OE	PCI_AD(18) OE
355	I	PCI_AD(18)
356	O	PCI_AD(17)
357	OE	PCI_AD(17) OE

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
358	I	PCI_AD(17)
359	O	PCI_AD(16)
360	OE	PCI_AD(16) OE
361	I	PCI_AD(16)
362	O	PCI_AD(15)
363	OE	PCI_AD(15) OE
364	I	PCI_AD(15)
365	O	PCI_AD(14)
366	OE	PCI_AD(14) OE
367	I	PCI_AD(14)
368	O	PCI_AD(13)
369	OE	PCI_AD(13) OE
370	I	PCI_AD(13)
371	O	PCI_AD(12)
372	OE	PCI_AD(12) OE
373	I	PCI_AD(12)
374	O	PCI_AD(11)
375	OE	PCI_AD(11) OE
376	I	PCI_AD(11)
377	O	PCI_AD(10)
378	OE	PCI_AD(10) OE
379	I	PCI_AD(10)
380	O	PCI_AD(9)
381	OE	PCI_AD(9) OE
382	I	PCI_AD(9)
383	O	PCI_AD(8)

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
384	OE	PCI_AD(8) OE
385	I	PCI_AD(8)
386	O	PCI_AD(7)
387	OE	PCI_AD(7) OE
388	I	PCI_AD(7)
389	O	PCI_AD(6)
390	OE	PCI_AD(6) OE
391	I	PCI_AD(6)
392	O	PCI_AD(5)
393	OE	PCI_AD(5) OE
394	I	PCI_AD(5)
395	O	PCI_AD(4)
396	OE	PCI_AD(4) OE
397	I	PCI_AD(4)
398	O	PCI_AD(3)
399	OE	PCI_AD(3) OE
400	I	PCI_AD(3)
401	O	PCI_AD(2)
402	OE	PCI_AD(2) OE
403	I	PCI_AD(2)
404	O	PCI_AD(1)
405	OE	PCI_AD(1) OE
406	I	PCI_AD(1)
407	O	PCI_AD(0)
408	OE	PCI_AD(0) OE
409	I	PCI_AD(0)

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
410	O	$\overline{\text{PCI\_CBE(0)}}$
411	OE	$\overline{\text{PCI\_CBE(0)}}$ OE
412	I	$\overline{\text{PCI\_CBE(0)}}$
413	O	$\overline{\text{PCI\_CBE(1)}}$
414	OE	$\overline{\text{PCI\_CBE(1)}}$ OE
415	I	$\overline{\text{PCI\_CBE(1)}}$
416	O	$\overline{\text{PCI\_CBE(2)}}$
417	OE	$\overline{\text{PCI\_CBE(2)}}$ OE
418	I	$\overline{\text{PCI\_CBE(2)}}$
419	O	$\overline{\text{PCI\_CBE(3)}}$
420	OE	$\overline{\text{PCI\_CBE(3)}}$ OE
421	I	$\overline{\text{PCI\_CBE(3)}}$
422	O	$\overline{\text{PCI\_IRDY}}$
423	OE	$\overline{\text{PCI\_IRDY}}$ OE
424	I	$\overline{\text{PCI\_IRDY}}$
425	O	$\overline{\text{PCI\_RST}}$
426	OE	$\overline{\text{PCI\_RST}}$ OE
427	I	$\overline{\text{PCI\_RST}}$
428	O	$\overline{\text{PCI\_REQ}}$
429	OE	$\overline{\text{PCI\_REQ}}$ OE
430	I	$\overline{\text{PCI\_GNT}}$
431	I	PCI_IDSEL
432	O	$\overline{\text{PCI\_FRAME}}$
433	OE	$\overline{\text{PCI\_FRAME}}$ OE
434	I	$\overline{\text{PCI\_FRAME}}$
435	O	$\overline{\text{PCI\_TRDY}}$

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
436	OE	$\overline{\text{PCI\_TRDY}}$ OE
437	I	$\overline{\text{PCI\_TRDY}}$
438	O	$\overline{\text{PCI\_DEVSEL}}$
439	OE	$\overline{\text{PCI\_DEVSEL}}$ OE
440	I	$\overline{\text{PCI\_DEVSEL}}$
441	O	$\overline{\text{PCI\_STOP}}$
442	OE	$\overline{\text{PCI\_STOP}}$ OE
443	I	$\overline{\text{PCI\_STOP}}$
444	O	$\overline{\text{PCI\_PERR}}$
445	OE	$\overline{\text{PCI\_PERR}}$ OE
446	I	$\overline{\text{PCI\_PERR}}$
447	O	PCI_PAR
448	OE	PCI_PAR OE
449	I	PCI_PAR
450	O	$\overline{\text{PCI\_SERR}}$
451	OE	$\overline{\text{PCI\_SERR}}$ OE
452	I	$\overline{\text{PCI\_SERR}}$
453	I	$\overline{\text{PCI\_LOCK}}$
454	I	PCI_CLK
455	O	$\overline{\text{PCI\_INTA}}$
456	OE	$\overline{\text{PCI\_INTA}}$ OE
457	I	$\overline{\text{PCI\_INTA}}$
458	I	$\overline{\text{PCI\_INTB}}$
459	I	$\overline{\text{PCI\_INTC}}$
460	I	$\overline{\text{PCI\_INTD}}$
461	I	BMODE(0)

Table C-3. Scan Path Position Definitions (Cont'd)

Position	Type	Signal Name
462	I	BMODE(1)
463	I	BMODE(2)
464	O	SLEEP
465	OE	SLEEP OE
466	I	BYPASS
467	O	$\overline{\text{EMU}}$
468	OE	$\overline{\text{EMU}}$ OE

D NUMERIC FORMATS

Blackfin family processors support 8-, 16-, and 32-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 Blackfin family are in two's-complement format. Signed-magnitude, one's-complement, BCD or excess-n formats are not supported.

Integer or Fractional

The Blackfin family supports both fractional and integer data formats. In an integer, the radix point is assumed to lie to the right of the 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.

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 3 LSBs, and the bits have the weights indicated.

The native formats for the Blackfin 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.

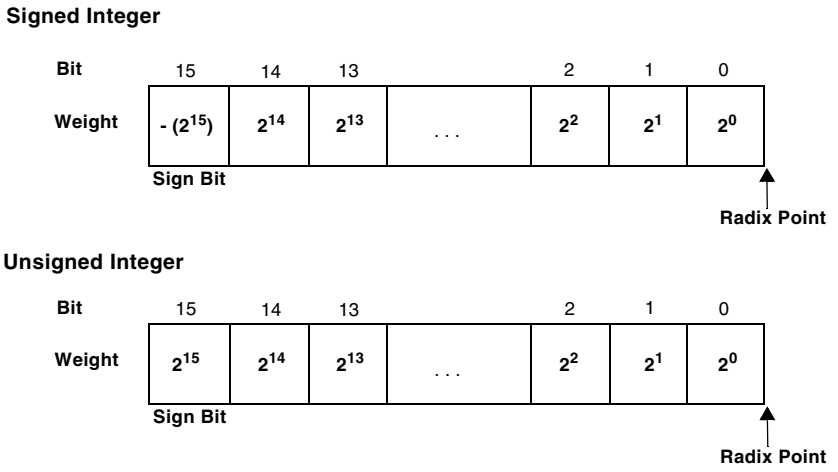


Figure D-1. Integer Format

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.

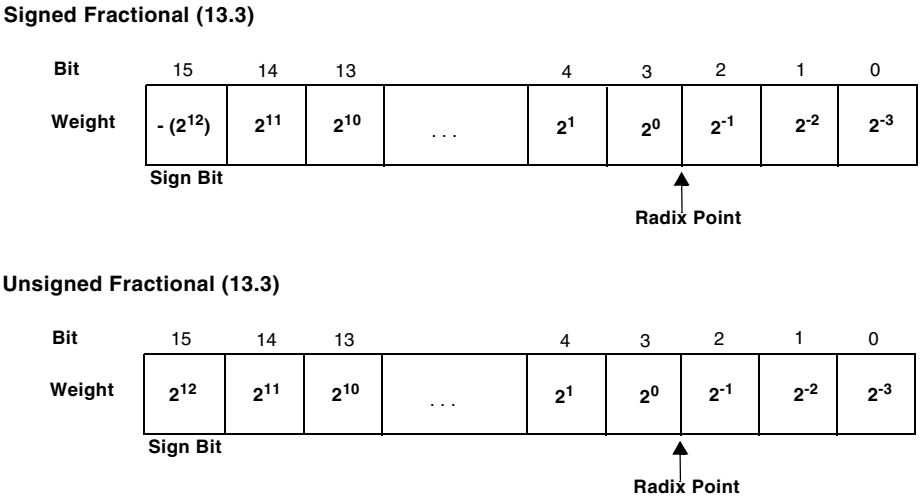


Figure D-2. Example of Fractional Format

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 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.

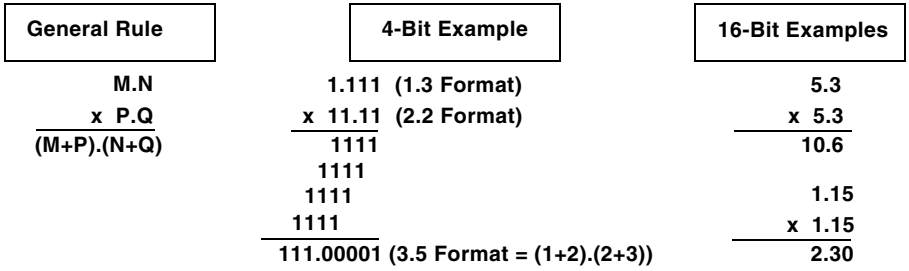


Figure D-3. Format of Multiplier Result

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 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 correctly formatted result.

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 2-11.

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. Some additional programming is required to maintain a block floating-point format, however.

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, increasing the precision available in each value. Block floating-point format can also be used to eliminate the possibility of a data value overflowing. Figure D-4 shows an example. In the example, each of the three data samples has at least two non-significant, redundant sign bits. Each data value can grow by these two bits (two orders of magnitude) before overflowing; thus, these bits are called guard bits.

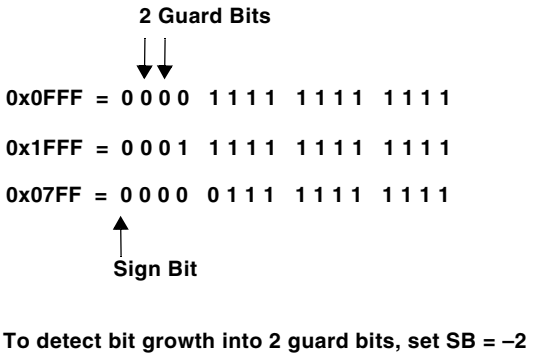


Figure D-4. Data With Guard Bits

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. Afterward, 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 that the output of the `SIGNBITS` instruction is `SB` and `SB` is used as an argument in the `EXPADJ` instruction (see *Blackfin Processor Programming Reference* for the usage and syntax of these instructions). Initially, the value of `SB` is –2, corresponding to the 2 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 2. In this example, `SB` = –1 after processing, indicating that the block of data must be shifted right one bit to maintain the 2 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.

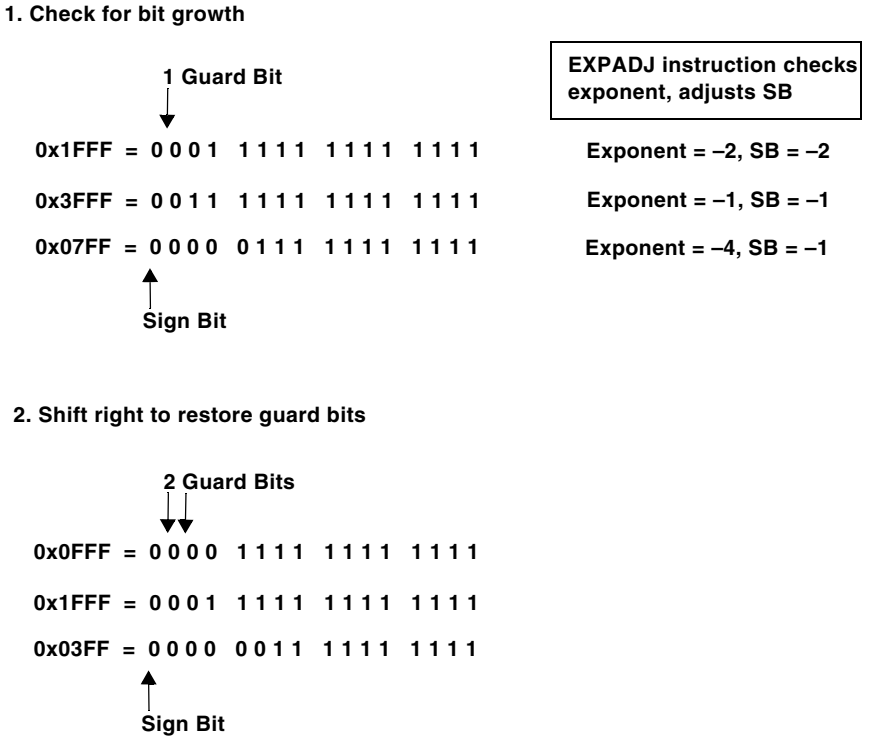


Figure D-5. Block Floating-Point Adjustment

G GLOSSARY

ALU.

See *Arithmetic/Logic Unit*

AMC (Asynchronous Memory Controller).

A configurable memory controller for separate banks of memory, 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.

asynchronous transfers.

Host accesses of the processor that occur intermittently rather than as a steady stream.

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 and the new bank address. 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. The SDC does not interleave SDRAM accesses, so only one internal bank in a row is open at a time.

base address.

The starting address of a circular buffer that the DAG wraps around.

Base register.

A DAG register that sets up the starting address for a circular buffer.

bit-reversed addressing.

The process in which the 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 loaded by the EPROM 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 power-up sequence.

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.

Burst Stop command.

The Burst Stop command is one of several ways to terminate a burst read or write operation.

Since the SDRAM burst length is always programmed 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 power-up 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 power-up 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 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.

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 datum is broadcast to all words in memory; the datum is compared with the stored values; and values that match are flagged.

CAS (Column Address Strobe).

A signal sent from the memory management unit (MMU) 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. The CAS latency is programmed in the SDRAM Mode register during the power-up sequence.

The speed grade of the SDRAM and the `SCLK[0]` frequency determine the value of the CAS latency. The SDC can support CAS latency of 2 or 3 clock cycles. The selected CAS latency value must be programmed into the SDRAM Memory Global Control register (`EBIU_SDGCTL`) before the SDRAM power-up sequence. See "SDRAM Memory Global Control Register (EBIU_SDGCTL)" on page 18-37.

CBR (CAS Before RAS) memory refresh.

DRAM devices have a built-in counter for the refresh row address. By activating CAS before activating RAS, this counter is selected to supply the row address instead of the address inputs.

CEC.

See *Core Event Controller*

circular buffer addressing.

The process by which the 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 testing an `if` condition.

core.

For the ADSP-BF535 processor, the processor, L1 memory, Event Controller, core timer, and Performance Monitoring registers.

Core Event Controller (CEC).

Part of the ADSP-BF535 processor's 2-stage event-control mechanism. 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 transfers data between the data buses and the computation units and provide local storage for operands.

data registers (Dreg).

Registers located in the computational units that hold operands for multiplier, ALU, or shifter operations.

delayed branch.

Jump or call/return instruction with the delayed branches modifier. In delayed branches, two (instead of four) instruction cycles are lost in the pipeline, because the processor executes the two instructions after the branch while the pipeline fills with instructions from the new branch.

descriptor block, DMA.

A set of parameters used by the DMA controller to describe a set of DMA sequences. When successive DMA sequences are needed, the descriptor blocks are chained together so that the completion of one DMA sequence autoinitiates and starts the next sequence.

descriptor loading, DMA.

The process in which the processor's DMA controller downloads a DMA descriptor from data memory and autoinitializes the DMA parameter registers.

DFT (Design For Testability).

A set of techniques that help designers of digital systems ensure that those system will be testable.

Digital Signal Processor.

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 an absolute address that does not change at runtime, such as a program label, or 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 are transferred through a DMA port without involving the processor (see *PIO*).

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 peripheral bus that allows DMA capable peripherals to gain access to memory.

DMA chaining.

The linking or chaining of multiple 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 descriptor registers.

Registers that hold the setup information for a DMA process.

DPMC (Dynamic Power Management Controller).

An ADSP-BF535 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[3:0] pins provide a byte-masking capability on 8- or 16-bit writes to SDRAM. The SDQM[3:0] pins are not used to mask data on read cycles. For 16-bit wide SDRAM, only SDQM[1:0] are needed for byte masking.

DRAM (Dynamic Random-Access Memory).

A type of semiconductor memory in which the data are 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*

EMB.

See *External Mastered 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 low 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 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.

External Bus Interface Unit (EBIU).

A component that provides glueless interfaces to external memories. It services requests for external memory from the core, from the PCI bridge, or from a DMA channel.

External Access Bus (EAB).

A line or bus that allows the processor core and Memory DMA controller to directly access off-chip memory and PCI memory space to perform instruction fetches, data loads, data stores, and high-throughput memory-to-memory DMA transfers.

External Bus Controller (EBC).

A component that provides arbitration between the EAB and EMB buses, granting at most one requester per cycle.

External Mastered Bus (EMB).

A line or bus that allows the PCI to directly access internal L2 memory, off-chip memory, and the system MMRs.

edge-sensitive interrupt.

A signal or interrupt that 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. The processor uses big endian format—moves data starting with most-significant-bit and finishing with least significant bit—in almost all instances. The two exceptions are bit order for data transfer through the serial port and word order for packing through the external port. For compatibility with little endian (least significant first) peripherals, the processor supports both big and little endian bit order data transfers. Also for compatibility little endian hosts, the processor supports both big and little endian word order data transfers.

external port.

A channel or port that extends the processors internal address and data buses off-chip, providing the processor's interface to off-chip memory and peripherals.

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. (For Jean Baptiste Joseph Fourier, 1768 – 1830, a French mathematician.)

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. Also known as a "shelf" from the analogy of pushing items onto one end of a shelf so that they fall off the other.

flag update.

A refresh from the processor to the status flags that occurs at the end of the cycle in which the status is generated. The flag update is available on the next cycle.

flash memory.

A type of single transistor cell, erasable memory in which erasing can only be done in blocks or for the entire chip.

Field Programmable Gate Array (FPGA).

A digital logic chip that can be programmed after production and that contains many thousands of gates.

FPGA.

See *Field Programmable Gate Array*.

fully associative.

Cache architecture where each line can be placed anywhere in the cache.

glueless.

No external hardware is required.

GSM (Global System for Mobile Communications).

A digital mobile telephone system that is widely used in Europe and other parts of the world. GSM uses a variation of time division multiple access (TDMA) and operates at either the 900 MHz or 1800 MHz frequency band.

Harvard architecture.

A memory architecture that uses separate buses for program and data storage. The two buses let the processor get a data word and an instruction 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.

IEEE (Institute of Electrical and Electronics Engineers).

An international technical professional society, based in the United States.

index.

Address portion that is used to select an array element (for example, line index)

Index registers.

A 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.

I/O processor.

DMA controllers, DMA channel arbitration, and peripheral-to-bus connections that handle input and output operations relieving the burden on the processor. The ADSP-BF535 processor uses a distributed DMA architecture with DMA controllers for each DMA capable peripheral. Most ports have some direct (non-DMA) access to internal memory and I/O memory.

I/O processor register.

One of the control, status, or data buffer registers of the processor's on-chip I/O processor.

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 PLL module.

internal memory bank.

There are several internal memory banks on a given SDRAM row. An internal bank in a specific row cannot be activated (opened) until the previous internal bank in that row has been precharged.

The SDC does not support interleaved accesses. The bank address can be thought of as part of the row address. The SDC also assumes that all SDRAMs to which it interfaces have four internal banks.

Do not confuse these “SDRAM internal banks” which are internal to the SDRAM and are selected with the bank address, with the four “SDRAM banks,” or “external banks,” that are enabled by the SMS[3:0] pins.

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*.

JEDEC.

(formerly known as the Joint Electronic Device Engineering Council, now called the JEDEC Solid State Technology Association) The semiconductor engineering standardization body of the Electronic Industries Alliance (EIA), an electronic industry trade association.

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 DAG register that sets up 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. Also known as a "stack" from the analogy with the stack of plates in a cafeteria in which the most recent plate placed on the stack is on top and thus will be the next plate removed. See *Stack*.

little endian.

The native data store format of the ADSP-BF535 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.

loop.

A sequence of instructions that executes several times with 0 overhead.

LRU (Least Recently Used algorithm).

A rule based on the observation that, in general, the cache entry that has not been accessed for longest is least likely to be accessed in the near future.

LSB.

See *Least Significant Bit*.

MAC (Multiply/Accumulate).

A floating-point operation that multiplies two numbers and then adds a third to get the result. See *Multiply Accumulator*.

memory bank.

A unit of the processor's external memory space, which may be addressed by either data address generator. External memory banks also may be configured for size and access waitstates.

memory block.

A unit of the processor's internal memory space, which is associated with a DAG.

Memory Management Unit (MMU).

A component of the ADSP-BF535 processor that supports protection and selective caching of memory by using CPLBs.

Mode Register.

SDRAM devices contain an internal configuration register which allows specification of the SDRAM device's functionality. After power-up 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 power-up 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 that for most applications the SDRAM power-up sequence and writing of the mode register needs to be done only once. Once the power-up sequence has completed, the PSSE bit should not be set again unless a change to the mode register is desired.

modified addressing.

The process whereby the DAG generates an address that is incremented by a value or a register.

Modify address.

The process by which a DAG increments the stored address without performing a data move.

Modify register.

A 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 location in main memory used by the processor as if it were a register. Registers are high-speed memory locations in the processor that can be addressed with just a few bits; memory-mapped registers must be addressed in the same way as other memory locations and the speed of MMR accesses is limited to the speed of the system's data bus.

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.

Multiply Accumulator.

An accumulator register in which operands are multiplied and added to the contents.

NMI (Non-maskable Interrupt).

A high priority interrupt that cannot be disabled by another interrupt.

NRZ (Non-return to Zero).

A binary encoding scheme in which a one (1) is represented by a change in the signal and a zero (0) by no change—there is no return to a reference (zero) 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 zero (0) is represented by a change in the signal and a one (1) by no change—there is no return to a reference (zero) 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. The page size can be calculated for 32-bit and 16-bit SDRAM banks with these formulas:

- 32-bit SDRAM banks: page size = $2^{(CAW + 2)}$
- where `CAW` is the column address width of the SDRAM, plus 2 because the SDRAM bank is 32 bits wide (2 address bits = 4 bytes).
- 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).

PC (Program Counter).

A register that contains the address of the next instruction to be executed. The PC is automatically incremented after each instruction is fetched to point to the following instruction.

PCI.

See *Peripheral Component Interconnect*

peripheral.

The ADSP-BF535 processor's peripherals include timers, Real-Time Clock, USB, programmable flags, UARTs, SPORTs, and SPIs.

Peripheral Access Bus (PAB).

A low and predictable latency bus that keeps core stalls to a minimum and allows for manageable interrupt latencies to time critical peripherals.

Peripheral Component Interconnect (PCI).

A bus system that provides a bus bridge between the EAB and the EMB internal to the ADSP-BF535 processor and an external PCI bus.

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.

PIO (Programmed Input/Output).

A way of moving data between system devices and memory in which all data must pass through the processor (see *Direct Memory Access*).

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 DAG provides an address during a data move and autoincrements the stored address for the next move.

Precharge command.

The Precharge command closes a specific internal bank in the active page or all internal banks in the page. The SDC always does a Precharge All, closing all internal banks.

pre-modify addressing.

The process in which the DAG provides an address during a data move without incrementing the stored address.

PWM (Pulse Width Modulation).

Also called Pulse Duration Modulation (PDM), 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 memory management unit (MMU) to a DRAM device to indicate that the row address lines are valid.

Real-Time Clock (RTC).

A component that generates timing pulses for the digital watch features of the ADSP-BF535 processor, including time of day, alarm, and stopwatch countdown features.

replacement policy.

The function used by the ADSP-BF535 processor to determine which line to replace on a cache miss. Often, a least recently used (LRU) algorithm is employed.

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 zero (0) is represented by a change from the low voltage level to the high voltage level; a one (1) is represented by a change from the high voltage level to the low voltage level. A return to a reference (zero) 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 one (1) is represented by a change from the low voltage level to the high voltage level; a zero (0) is represented by a change from the high voltage level to the low voltage level. A return to a reference (zero) 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.

SBIU.

See *System Bus Interface Unit*.

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 banks.

Four regions of memory that can be configured to be 16 MB, 32 MB, 64 MB, or 128 MB and are selected by the `SMS[3:0]` pins. Each bank can be selected to be either all 32 bits wide or all 16 bits wide.

Do not confuse the “SDRAM internal banks” which are internal to the SDRAM and are selected with the bank address, with the “SDRAM banks,” or “external banks,” that are enabled by the SMS[3:0] pins.

SDRAM DIMMs.

Dual In-line Memory Modules, or DIMMs, are an industry-standard SDRAM packaging option. The DIMM consists of a small, standardized form factor board, populated with SDRAM devices on one or both sides. DIMMs are populated with sufficient SDRAM to provide either 32-bit or 64-bit data paths, with some configurations supporting additional data bits for parity protection.

Many DIMMs also include a serial presence detect port, which provides access to an on-DIMM serial ROM that includes information describing the type and characteristics of SDRAMs on the DIMM. This information can be read to determine the type, size, and timing parameters of installed DIMMs at boot time. One of the SPI or SPORT peripherals of the ADSP-BF535 processor can be used to interface to the DIMM serial presence detect port, if dynamic boot time determination of SDRAM configuration is required.

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 18-44.

Serial Peripheral Interface (SPI).

A synchronous serial protocol used to connect integrated circuits.

serial ports.

An input/output location on the processor. The ADSP-BF535 processor uses three synchronous serial ports that provide an inexpensive interface 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 on 16-bit operands and derives exponents.

SIC (System Interrupt Controller).

Part of the ADSP-BF535 processor's 2-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 a collection of data is processed simultaneously under one instruction.

SP (Stack Pointer).

A register that points to the top of a stack. If the stack contains data, the SP points to the most recently pushed item; if the stack does not contain data, the SP points to the first empty location, where the next item will be pushed.

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 said to be "pushed"; when a data item is removed from the stack, it is "popped".

Static Random Access Memory (SRAM).

Very fast memory that does not require periodic refreshing.

synchronous process.

A process that runs only as a result of some other process being completed or a handing-off operation.

system.

For the ADSP-BF535 processor, the peripheral set (Timers, Real-Time Clock, USB, programmable flags, UARTs, SPORTs, and SPIs), the PCI controller, the external memory controller (EBIU), the Memory DMA controller, and the interfaces between these, the system, and the optional, external (off-chip) resources.

System Bus Interface Unit (SBIU).

A unit that performs bus bridging functions, clock domain conversion, routes requests, and provides data transfer capability.

System clock (SCLK).

A component that delivers pulses at a frequency determined by a programmable divider ratio from the core clock.

System Interrupt Controller (SIC).

Component that maps and routes events from peripheral interrupt sources to the prioritized, general-purpose interrupt inputs of the CIC.

subroutine.

A self-contained section of a program, with an entry point and an exit, that usually performs a single task. The main program flow branches to a subroutine when its task is required, and the subroutine returns (branches back) to the main flow when its task has been completed.

t_{RAS}.

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. "Selecting the Bank Activate Command Delay (TRAS)" on page 18-47.

t_{RP}.

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 3 bits wide and can be programmed to be 1 to 7 clock cycles long. "Selecting the Precharge Delay (TRP)" on page 18-48.

t_{RCD}.

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 3 bits wide and can be programmed to be from 1 to 7 clock cycles long.

t_{WR}.

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 2 bits wide and can be programmed to be from 1 to 3 clock cycles long.

t_{RC}.

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_{RFC}.

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_{XSR}

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 t_{RAS} and t_{RP} fields to ensure that $t_{RAS} + t_{RP} \geq t_{RC}$.

tag.

Upper address bits, stored along with the cached data line, to identify the specific address source in memory that the cached line represents.

TAP (Test Access Port).

See *JTAG port*

TDM.

See *Time Division Multiplexing*

three-state versus Tristate.

Analog Devices documentation uses the word *three-state* instead of *tristate*. Tristate™ is a trademarked term owned by National Semiconductor.

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 24 channels.

UART.

See *Universal Asynchronous Receiver Transmitter*

UDC (USB Device Controller).

Part of the USBD, a module that implements the low level USB protocol. It manages interaction with the USB host using the USB serial link, and presents data and command transactions to the application.

Universal Serial Bus (USB).

A module that handles USB protocol requirements. It also contains hardware to connect it to the processor's peripheral buses and to support an operating system.

Universal Asynchronous Receiver Transmitter (UART).

A module that contains both the receiving and transmitting circuits required for asynchronous serial communication.

USB.

See *Universal Serial Bus*

USBD (Universal Serial Bus Device).

The module in the ADSP-BF535 processor that controls the Universal Serial Bus. The USBD consists of the UDC core module and a front-end interface. The USBD connects the DAB and the PAB with and off-chip USB transceiver.

Valid.

A state bit, stored along with the tag, indicating 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. (For John von Neumann, 1903-1957, a Hungarian-born mathematician.)

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 a 1 to it. It cannot be cleared by writing a 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 the source memory. The modified cache line is *not* written to the source memory when it is replaced.

I INDEX

B

C

D

E

F

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z