ADSP-2106x SHARC® Processor User's Manual

Revision 2.1, March 2004

Part Number
82-000795-03

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

Copyright Information

Printed in the USA.

Disclaimer

Trademark and Service Mark Notice

Errata Correction Notice

This revision is published to incorporate corrections to errata in the Second Edition (May 1997). Please refer to Appendix H for more information.

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

APPENDIX B COMPUTE OPERATION REFERENCE

Contents

Contents

Contents

FIGURES

Contents

Contents

Contents

Contents

TABLES

Contents

Contents

LISTINGS

Introduction 1

1.1 OVERVIEW

The ADSP-2106x SHARC—Super Harvard Architecture Computer—is a high-performance 32-bit digital signal processor for speech, sound, graphics, and imaging applications. The SHARC builds on the ADSP-21000 Family DSP core to form a complete system-on-a-chip, adding a dual-ported on-chip SRAM and integrated I/O peripherals supported by a dedicated I/O bus. With its on-chip instruction cache, the processor can execute every instruction in a single cycle. Four independent buses for dual data, instructions, and I/O, plus crossbar switch memory connections, comprise the Super Harvard Architecture of the ADSP-2106x.

The ADSP-2106x SHARC represents a new standard of integration for digital signal processors, combining a high-performance floating-point DSP core with integrated, on-chip features including a host processor interface, DMA controller, serial ports, and link port and shared bus connectivity for glueless DSP multiprocessing.

Figure 1.1 illustrates the Super Harvard Architecture of the ADSP-2106x: a crossbar bus switch connecting the core numeric processor to an independent I/O processor, dual-ported memory, and parallel system bus port. Figure 1.2 shows a detailed block diagram of the processor, illustrating the following architectural features:

- 32-Bit IEEE Floating-Point Computation Units—Multiplier, ALU, and Shifter
- Data Register File
- Data Address Generators (DAG1, DAG2)
- Program Sequencer with Instruction Cache
- Interval Timer
- Dual-Ported SRAM
- External Port for Interfacing to Off-Chip Memory & Peripherals
- Host Port & Multiprocessor Interface
- DMA Controller
- Serial Ports
- Link Ports
- JTAG Test Access Port

Figure 1.2 also shows the three on-chip buses of the ADSP-2106x: the PM bus (program memory), DM bus (data memory), and I/O bus. The PM bus is used to access either instructions or data. During a single cycle the processor can access two data operands, one over the PM bus and one over the DM bus, an instruction (from the cache), and perform a DMA transfer.

The ADSP-2106x's external port provides the processor's interface to external memory, memory-mapped I/O, a host processor, and additional multiprocessing ADSP-2106xs. The external port performs internal and external bus arbitration as well as supplying control signals to shared, global memory and I/O devices.

Figure 1.3 illustrates a typical single-processor system. A multiprocessor system is shown in Chapter 7, *Multiprocessing*.

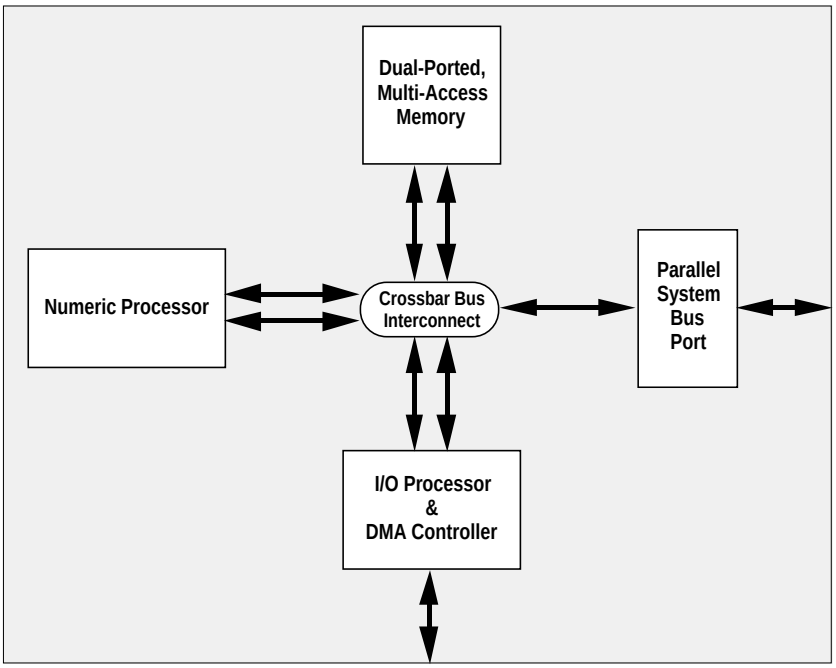
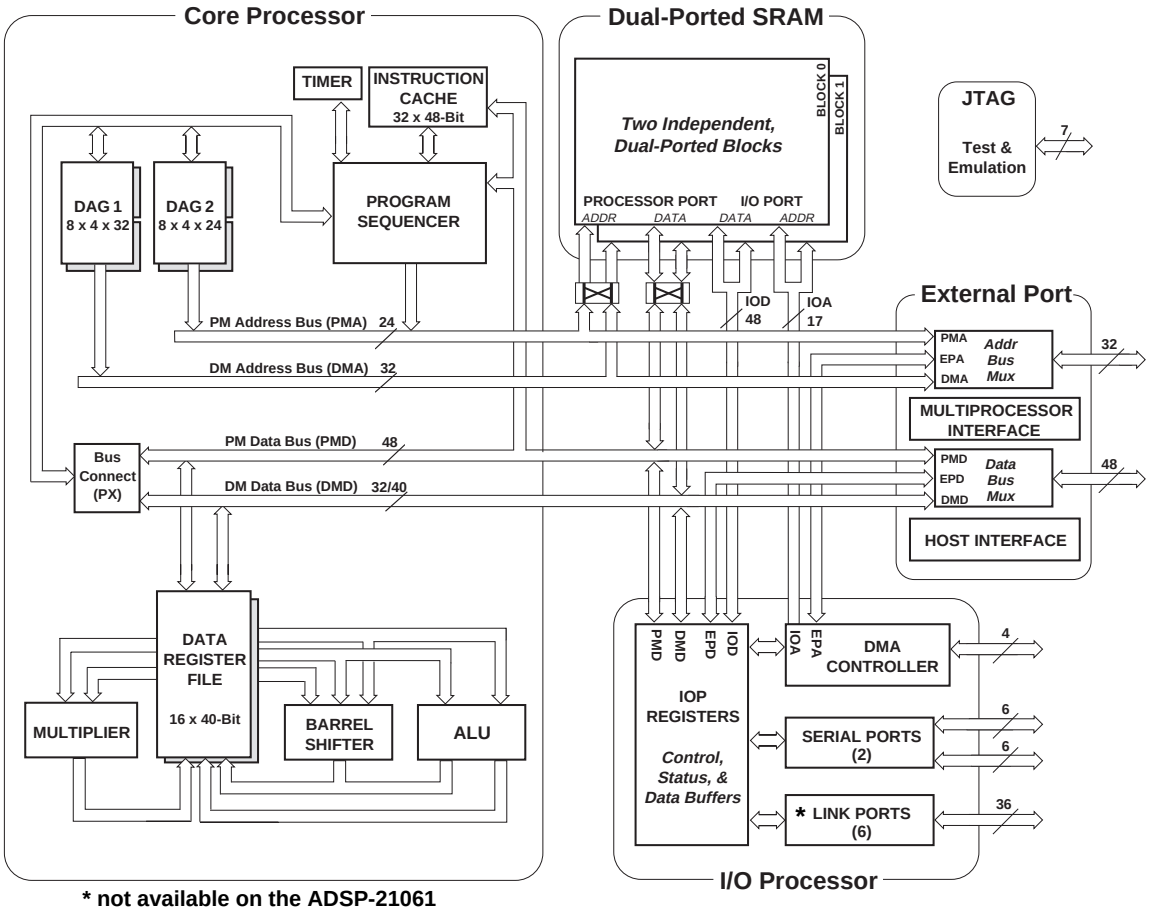


Figure 1.1 Super Harvard Architecture



*** not available on the ADSP-21061**

Figure 1.2 ADSP-2106x SHARC Block Diagram

This user's manual contains architectural information and an instruction set description required for the design and programming of ADSP-2106x-based systems. In addition to this manual, hardware designers should refer to the *ADSP-21060/62 Data Sheet* and the *ADSP-21061 Data Sheet* for timing, electrical, and package specifications.

1 Introduction

This manual covers three ADSP-2106x processors: the ADSP-21060, ADSP-21062, and ADSP-21061. The ADSP-21060 contains 4 megabits of on-chip SRAM, the ADSP-21062 contains 2 megabits, and the ADSP-21061 contains 1 megabit. The *Memory* chapter of this manual describes the differences in memory architecture and programming considerations of the three processors. All three processors are code- and function-compatible with the ADSP-21020 processor. With the exception of memory size, the ADSP-21060 and ADSP-21062 are identical in all other aspects as well. Besides memory size, there are four differences between these two processors and the ADSP-21061:

- No link ports on the ADSP-21061
- 6 DMA channels — 4 for serial port and 2 for external port (instead of 4)
- Additional features and changes in DMA for the serial port
- New `idle 16` instruction for a further reduced power mode

These differences are described in detail in the *DMA*, *Serial Port*, and *Program Sequencer* chapters.

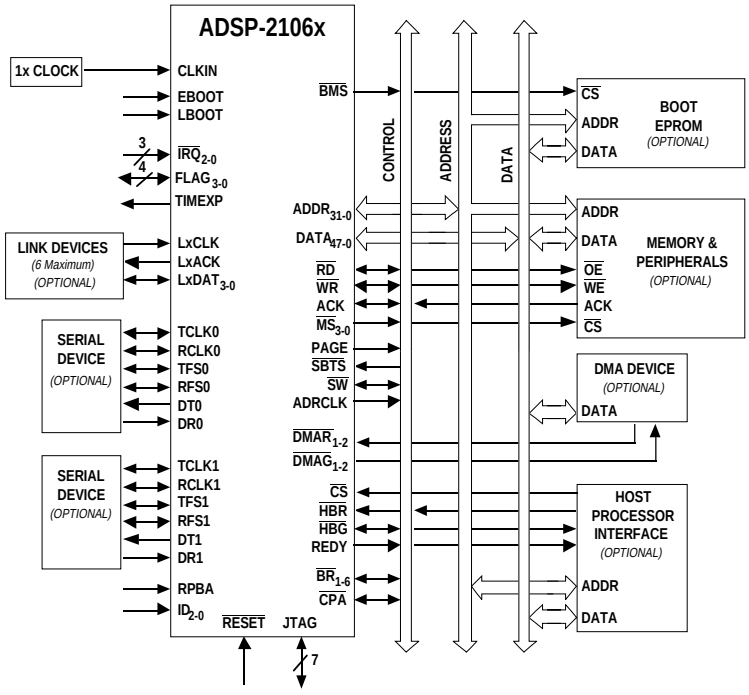


Figure 1.3 ADSP-2106x System

1.2 ADSP-21000 FAMILY FEATURES & BENEFITS

The ADSP-2106x SHARC processors belong to the ADSP-21000 Family of floating-point digital signal processors (DSPs). The ADSP-21000 Family architecture further addresses the five central requirements for DSPs established in the ADSP-2100 Family of 16-bit fixed-point DSPs:

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

Fast, Flexible Arithmetic. The ADSP-21000 Family processors execute all instructions in a single cycle. They provide both fast cycle times and a complete set of arithmetic operations including Seed 1/X, Seed $1/\sqrt{X}$, Min, Max, Clip, Shift, and Rotate, in addition to the traditional multiplication, addition, subtraction, and combined multiplication/addition. The processors are IEEE floating-point compatible and allow either interrupt on arithmetic exception or latched status exception handling.

Unconstrained Data Flow. The ADSP-2106x has an enhanced Harvard architecture combined with a 10-port data register file. In every cycle:

- Two operands can be read or written to or from the register file,
- Two operands can be supplied to the ALU,
- Two operands can be supplied to the multiplier, and
- Two results can be received from the ALU and multiplier.

The processor's 48-bit orthogonal instruction word supports fully parallel data transfer and arithmetic operations in the same instruction.

40-Bit Extended Precision. The ADSP-21000 Family processors handle 32-bit IEEE floating-point format, 32-bit integer and fractional formats (twos-complement and unsigned), and extended-precision 40-bit IEEE floating-point format. The processors carry extended precision throughout their computation units, limiting intermediate data truncation errors. When working with data on-chip, the extended-precision 32-bit mantissa can be transferred to and from all computation units. The 40-bit data bus may be extended off-chip if desired. The fixed-point formats have an 80-bit accumulator for true 32-bit fixed-point computations.

Dual Address Generators. The ADSP-21000 Family processors have two data address generators (DAGs) that provide immediate or indirect (pre- and post-modify) addressing. Modulus and bit-reverse operations are supported with no constraints on data buffer placement.

Efficient Program Sequencing. In addition to zero-overhead loops, the ADSP-21000 Family processors support single-cycle setup and exit for loops. Loops are both nestable (six levels in hardware) and interruptable. The processors support both delayed and non-delayed branches.

1.2.1 System-Level Enhancements

The ADSP-21000 Family processors include several enhancements that simplify system development. The enhancements occur in three key areas:

- Architectural features supporting high-level languages and operating systems
- IEEE 1149.1 JTAG serial scan path and on-chip emulation features
- Support of IEEE floating-point formats

High Level Languages. The ADSP-21000 Family architecture has several features that directly support high-level language compilers and operating systems:

- General purpose data and address register files
- 32-bit native data types
- Large address space
- Pre- and post-modify addressing
- Unconstrained circular data buffer placement
- On-chip program, loop, and interrupt stacks

Additionally, the ADSP-21000 Family architecture is designed specifically to support ANSI-standard Numerical C extensions—the first compiled language to support vector data types and operators for numeric and signal processing.

Serial Scan and Emulation Features. The ADSP-21000 Family processors support the IEEE standard P1149.1 Joint Test Action Group (JTAG) standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system. The JTAG serial port is also used by the ADSP-2106x EZ-ICE to gain access to the processor's on-chip emulation features.

IEEE Formats. The ADSP-21000 Family processors support IEEE floating-point data formats. This means that algorithms developed on IEEE-compatible processors and workstations are portable across processors without concern for possible instability introduced by biased rounding or inconsistent error handling.

1.2.2 Why Floating-Point DSP?

A digital signal processor's data format determines its ability to handle signals of differing precision, dynamic range, and signal-to-noise ratios. However, ease-of-use and time-to-market considerations are often equally important.

Precision. The number of bits of precision of A/D converters has continued to increase, and the trend is for both precision and sampling rates to increase.

Dynamic Range. Compression and decompression algorithms have traditionally operated on signals of known bandwidth. These algorithms were developed to behave regularly, to keep costs down and implementations easy. Increasingly, however, the trend in algorithm development is not to constrain the regularity and dynamic range of intermediate results. Adaptive filtering and imaging are two applications requiring wide dynamic range.

Signal-to-Noise Ratio. Radar, sonar and even commercial applications like speech recognition require wide dynamic range in order to discern selected signals from noisy environments.

Ease-of-Use. In general, 32-bit floating-point DSPs are easier to use and allow a quicker time-to-market than 16-bit fixed-point processors. The extent to which this is true depends on the floating-point processor's architecture. Consistency with IEEE workstation simulations and the elimination of scaling are two clear ease-of-use advantages. High-level language programmability, large address spaces, and wide dynamic range allow system development time to be spent on algorithms and signal processing concerns rather than assembly language coding, code paging, and error handling.

1 Introduction

1.3 ADSP-2106X ARCHITECTURE

The following sections summarize the features of the ADSP-2106x SHARC architecture. These features are described in greater detail in succeeding chapters.

1.3.1 Core Processor

The core processor of the ADSP-2106x consists of three computation units, a program sequencer, two data address generators, timer, instruction cache, and data register file.

1.3.1.1 Computation Units

The ADSP-2106x core processor contains three independent computation units: an ALU, a multiplier with a fixed-point accumulator, and a shifter. For meeting a wide variety of processing needs, the computation units process data in three formats: 32-bit fixed-point, 32-bit floating-point and 40-bit floating-point. The floating-point operations are single-precision IEEE-compatible. The 32-bit floating-point format is the standard IEEE format, whereas the 40-bit IEEE extended-precision format has eight additional LSBs of mantissa for greater accuracy.

The ALU performs a standard set of arithmetic and logic operations in both fixed-point and floating-point formats. The multiplier performs floating-point and fixed-point multiplication as well as fixed-point multiply / add and multiply / subtract operations. The shifter performs logical and arithmetic shifts, bit manipulation, field deposit and extraction and exponent derivation operations on 32-bit operands.

The computation units perform single-cycle operations; there is no computation pipeline. The units are connected in parallel rather than serially. The output of any unit may be the input of any unit on the next cycle. In a *multifunction* computation, the ALU and multiplier perform independent, simultaneous operations.

1.3.1.2 Data Register File

A general-purpose data register file is used for transferring data between the computation units and the data buses, and for storing intermediate results. The register file has two sets (primary and alternate) of sixteen registers each, for fast context switching. All of the registers are 40 bits wide. The register file, combined with the core processor's Harvard architecture, allows unconstrained data flow between computation units and internal memory.

1.3.1.3 Program Sequencer & Data Address Generators

Two dedicated address generators and a program sequencer supply addresses for memory accesses. Together the sequencer and data address generators allow computational operations to execute with maximum efficiency since the computation units can be devoted exclusively to processing data. With its instruction cache, the ADSP-2106x can simultaneously fetch an instruction (from the cache) and access two data operands (from memory). The data address generators implement circular data buffers in hardware.

The program sequencer supplies instruction addresses to program memory. It controls loop iterations and evaluates conditional instructions. With an internal loop counter and loop stack, the ADSP-2106x executes looped code with zero overhead. No explicit jump instructions are required to loop or to decrement and test the counter.

The ADSP-2106x achieves its fast execution rate by means of pipelined *fetch*, *decode* and *execute* cycles. If external memories are used, they are allowed more time to complete an access than if there were no decode cycle.

The data address generators (DAGs) provide memory addresses when data is transferred between memory and registers. Dual data address generators enable the processor to output simultaneous addresses for two operand reads or writes. DAG1 supplies 32-bit addresses to data memory. DAG2 supplies 24-bit addresses to program memory for program memory data accesses.

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

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

1.3.1.4 Instruction Cache

The program sequencer includes a 32-word instruction cache that enables three-bus operation for fetching an instruction and two data values. The cache is selective—only instructions whose fetches conflict with program memory data accesses are cached. This allows full-speed execution of core, looped operations such as digital filter multiply-accumulates and FFT butterfly processing.

1.3.1.5 Interrupts

The ADSP-2106x has four external hardware interrupts: three general-purpose interrupts, $\overline{IRQ}_{2\text{-}0}$, and a special interrupt for reset. The processor also has internally generated interrupts for the timer, DMA controller operations, circular buffer overflow, stack overflows, arithmetic exceptions, multiprocessor vector interrupts, and user-defined software interrupts.

For the general-purpose external interrupts and the internal timer interrupt, the ADSP-2106x automatically stacks the arithmetic status and mode (MODE1) registers in parallel with the interrupt servicing, allowing four nesting levels of very fast service for these interrupts.

1.3.1.6 Timer

The programmable interval timer provides periodic interrupt generation. When enabled, the timer decrements a 32-bit count register every cycle. When this count register reaches zero, the ADSP-2106x generates an interrupt and asserts its TIMEXP output. The count register is automatically reloaded from a 32-bit period register and the count resumes immediately.

1.3.1.7 Core Processor Buses

The processor core has four buses: Program Memory Address, Data Memory Address, Program Memory Data, and Data Memory Data. On the ADSP-2106x processors, data memory stores data operands while program memory is used to store both instructions and data (filter coefficients, for example)—this allows dual data fetches, when the instruction is supplied by the cache.

The PM Address bus and DM Address bus are used to transfer the addresses for instructions and data. The PM Data bus and DM Data bus are used to transfer the data or instructions stored in each type of memory. The PM Address bus is 24 bits wide allowing access of up to 16M words of mixed instructions and data. The PM Data bus is 48 bits wide to accommodate the 48-bit instruction width. Fixed-point and single-precision floating-point data is aligned to the upper 32 bits of the PM Data bus.

The DM Address bus is 32 bits wide allowing direct access of up to 4G words of data. The DM Data bus is 40 bits wide. Fixed-point and single-precision floating-point data is aligned to the upper 32 bits of the DM Data bus. The DM Data bus provides a path for the contents of any register in the processor to be transferred to any other register or to any data memory location in a single cycle. The data memory address comes from one of two sources: an absolute value specified in the instruction code *(direct addressing)* or the output of a data address generator *(indirect addressing)*.

1.3.1.8 Internal Data Transfers

Nearly every register in the core processor of the ADSP-2106x is classified as a *universal register*. Instructions are provided for transferring data between any two universal registers or between a universal register and memory. This includes control registers and status registers, as well as the data registers in the register file.

The PX bus connect registers permit data to be passed between the 48-bit PM Data bus and the 40-bit DM Data bus or between the 40-bit register file and the PM Data bus. These registers contain hardware to handle the 8-bit width difference.

1.3.1.9 Context Switching

Many of the processor's registers have alternate registers that can be activated during interrupt servicing to facilitate a fast context switch. The data registers in the register file, the DAG registers, and the multiplier result register all have alternates. Registers active at reset are called *primary* registers, while the others are called *alternate* (or *secondary*) registers. Control bits in a mode control register determine which set of registers is active at any particular time.

1.3.1.10 Instruction Set

The ADSP-21000 Family instruction set provides a wide variety of programming capabilities. *Multifunction* instructions enable computations in parallel with data transfers, as well as simultaneous multiplier and ALU operations. The addressing power of the ADSP-2106x gives you flexibility in moving data both internally and externally. Every instruction can be executed in a single processor cycle. The ADSP-21000 Family assembly language uses an algebraic syntax for ease of coding and readability. A comprehensive set of development tools supports program development.

1.3.2 Dual-Ported Internal Memory

The ADSP-21060 contains 4 megabits of on-chip SRAM, organized as two blocks of 2 Mbits each, which can be configured for different combinations of code and data storage. The ADSP-21062 includes a 2 Mbit SRAM, organized as two 1 Mbit blocks. Each memory block is dual-ported for single-cycle, independent accesses by the core processor and I/O processor or DMA controller. The dual-ported memory and separate on-chip buses allow two data transfers from the core and one from I/O, all in a single cycle.

All of the memory can be accessed as 16-bit, 32-bit, or 48-bit words. On the ADSP-21060, the memory can be configured as a maximum of 128K words of 32-bit data, 256K words of 16-bit data, 80K words of 48-bit instructions (and 40-bit data), or combinations of different word sizes up to 4 megabits. On the ADSP-21062, the memory can be configured as a maximum of 64K words of 32-bit data, 128K words of 16-bit data, 40K words of 48-bit instructions (and 40-bit data), or combinations of different word sizes up to 2 megabits. On the ADSP-21061, the memory can be configured as a maximum of 32K words of 32-bit data, 64K words of 16-bit data, 16K words of 48-bit instructions (and 40-bit data), or combinations of different word sizes up to 1 megabit.

A 16-bit floating-point storage format is supported which effectively doubles the amount of data that may be stored on chip. Conversion between the 32-bit floating-point and 16-bit floating-point formats is done in a single instruction.

While each memory block can store combinations of code and data, accesses are most efficient when one block stores data, using the DM bus for transfers, and the other block stores instructions and data, using the PM bus for transfers. Using the DM bus and PM bus in this way, with one dedicated to each memory block, assures single-cycle execution with two data transfers. In this case, the instruction must be available in the cache. Single-cycle execution is also maintained when one of the data operands is transferred to or from off-chip, via the ADSP-2106x's external port.

1.3.3 External Memory & Peripherals Interface

The ADSP-2106x's external port provides the processor's interface to off-chip memory and peripherals. The 4-gigaword off-chip address space is included in the ADSP-2106x's unified address space. The separate on-chip buses—for PM addresses, PM data, DM addresses, DM data, I/O addresses, and I/O data—are multiplexed at the external port to create an external system bus with a single 32-bit address bus and a single 48-bit data bus. External SRAM can be either 16, 32, or 48 bits wide; the ADSP-2106x's on-chip DMA controller automatically packs external data into the appropriate word width, either 48-bit instructions or 32-bit data.

Addressing of external memory devices is facilitated by on-chip decoding of high-order address lines to generate memory bank select signals. Separate control lines are also generated for simplified addressing of page-mode DRAM. The ADSP-2106x provides programmable memory wait states and external memory acknowledge controls to allow interfacing to DRAM and peripherals with variable access, hold, and disable time requirements.

1.3.4 Host Processor Interface

The ADSP-2106x's host interface allows easy connection to standard microprocessor buses, both 16-bit and 32-bit, with little additional hardware required. Asynchronous transfers at speeds up to the full clock rate of the ADSP-2106x are supported. The host interface is accessed through the ADSP-2106x's external port and is memory-mapped into the unified address space. Four channels of DMA are available for the host interface; code and data transfers are accomplished with low software overhead. The host can directly read and write the internal memory of the ADSP-2106x, and can access the DMA channel setup and mailbox registers. Vector interrupt support is provided for efficient execution of host commands.

1.3.5 Multiprocessing

The ADSP-2106x offers powerful features tailored to multiprocessing DSP systems. The unified address space allows direct interprocessor accesses of each ADSP-2106x's internal memory. Distributed bus arbitration logic is included on-chip for simple, glueless connection of systems containing up to six ADSP-2106xs and a host processor. Master processor changeover incurs only one cycle of overhead. Bus arbitration is selectable as either fixed or rotating priority. Processor bus lock allows indivisible *read-modify-write* sequences for semaphores. A vector interrupt capability is provided for interprocessor commands. Maximum throughput for interprocessor data transfer is 240 Mbytes/sec over the link ports or external port. *Broadcast writes* allow simultaneous transmission of data to all ADSP-2106xs and can be used to implement reflective semaphores.

1.3.6 I/O Processor

The ADSP-2106x's I/O Processor (IOP) includes two serial ports, six 4-bit link ports, and a DMA controller.

1.3.6.1 Serial Ports

The ADSP-2106x features two synchronous serial ports that provide an inexpensive interface to a wide variety of digital and mixed-signal peripheral devices. The serial ports can operate at the full clock rate of the processor, providing each with a maximum data rate of 40 Mbit/s. Independent transmit and receive functions provide greater flexibility for serial communications. Serial port data can be automatically transferred to and from on-chip memory via DMA. Each of the serial ports offers a TDM multichannel mode.

The serial ports can operate with little-endian or big-endian transmission formats, with word lengths selectable from 3 to 32 bits. They offer selectable synchronization and transmit modes as well as optional μ-law or A-law companding. Serial port clocks and frame syncs can be internally or externally generated.

1.3.6.2 Link Ports

The ADSP-21062 and ADSP-21060 feature six 4-bit link ports that provide additional I/O capabilities. The link ports can be clocked twice per cycle, allowing each to transfer 8 bits per cycle. Link port I/O is especially useful for point-to-point interprocessor communication in multiprocessing systems.

The link ports can operate independently and simultaneously, with a maximum data throughput of 240 Mbytes/s. Link port data is packed into 32-bit or 48-bit words, and can be directly read by the core processor or DMA-transferred to on-chip memory. Each link port has its own double-buffered input and output registers.
Clock/acknowledge handshaking controls link port transfers. Transfers are programmable as either transmit or receive.

There are no link ports on the ADSP-21061.

1.3.6.3 DMA Controller

The ADSP-2106x's on-chip DMA controller allows zero-overhead data transfers without processor intervention. The DMA controller operates independently and invisibly to the processor core, allowing DMA operations to occur while the core is simultaneously executing its program. Both code and data can be downloaded to the ADSP-2106x using DMA transfers.

DMA transfers can occur between the ADSP-2106x's internal memory and external memory, external peripherals, or a host processor. DMA transfers can also occur between the ADSP-2106x's internal memory and its serial ports or link ports. DMA transfers between external memory and external peripheral devices are another option. External bus packing to 16, 32, or 48-bit words is automatically performed during DMA transfers.

Ten channels of DMA are available on the ADSP-21060 and ADSP-21062—two via the link ports, four via the serial ports, and four via the processor's external port (for either host processor, other ADSP-2106xs, memory or I/O transfers). Four additional link port DMA channels are shared with serial port 1 and the external port. There are six channels of DMA available on the ADSP-21061—four via the serial ports and two via the external port. Asynchronous off-chip peripherals can control two DMA channels using DMA Request/Grant lines ($\overline{DMAR}_{1-2}$, $\overline{DMAG}_{1-2}$). Other DMA features include interrupt generation upon completion of DMA transfers and DMA chaining for automatic linked DMA transfers.

The ten DMA channels of the ADSP-21060 and ADSP-21062 are numbered as shown below:

DMA Channel#	*Data Buffer*	*Description*
DMA Channel 0	RX0	Serial Port 0 Receive
DMA Channel 1	RX1 *(or LBUF0)*	Serial Port 1 Receive *(or Link Buffer 0)*
DMA Channel 2	TX0	Serial Port 0 Transmit
DMA Channel 3	TX1 *(or LBUF1)*	Serial Port 1 Transmit *(or Link Buffer 1)*
DMA Channel 4	LBUF2	Link Buffer 2
DMA Channel 5	LBUF3	Link Buffer 3
DMA Channel 6	EPB0 *(or LBUF4)*	Ext. Port FIFO Buffer 0 *(or Link Buffer 4)*
DMA Channel 7 *	EPB1 *(or LBUF5)*	Ext. Port FIFO Buffer 1 *(or Link Buffer 5)*
DMA Channel 8 *	EPB2	Ext. Port FIFO Buffer 2
DMA Channel 9	EPB3	Ext. Port FIFO Buffer 3

* $\overline{\text{DMAR1}}$ and $\overline{\text{DMAG1}}$ are handshake controls for DMA Channel 7.
$\overline{\text{DMAR2}}$ and $\overline{\text{DMAG2}}$ are handshake controls for DMA Channel 8.

1.3.6.4 Booting

The internal memory of the ADSP-2106x can be booted at system powerup from an 8-bit EPROM or a host processor. Additionally, the ADSP-21060 and the ADSP-21062 can also be booted through one of the link ports. Selection of the boot source is controlled by the $\overline{\text{BMS}}$, EBOOT, and LBOOT pins. Both 32-bit and 16-bit host processors can be used for booting.

1.4 DEVELOPMENT TOOLS

The ADSP-2106x is supported with a complete set of software and hardware development tools, including an EZ-LAB® Evaluation Board, EZ-ICE® In-Circuit Emulator, and development software. The development software provides tools for programming and debugging applications in both assembly language and C. The EZ-ICE emulator allows system integration and hardware / software debugging. Figure 1.4 shows the process of developing an application using the development tools.

The development software includes an ANSI C Compiler. The compiler includes Numerical C extensions based on the work of the ANSI NCEG committee (Numerical C Extensions Group).

Numerical C provides extensions to the C language for array selection, vector math operations, complex data types, circular pointers, and variably-dimensioned arrays. Other components of the development software include a C Runtime Library with custom DSP functions, C and assembly language Debugger, Assembler, Assembly Library / Librarian, Linker, and Simulator.

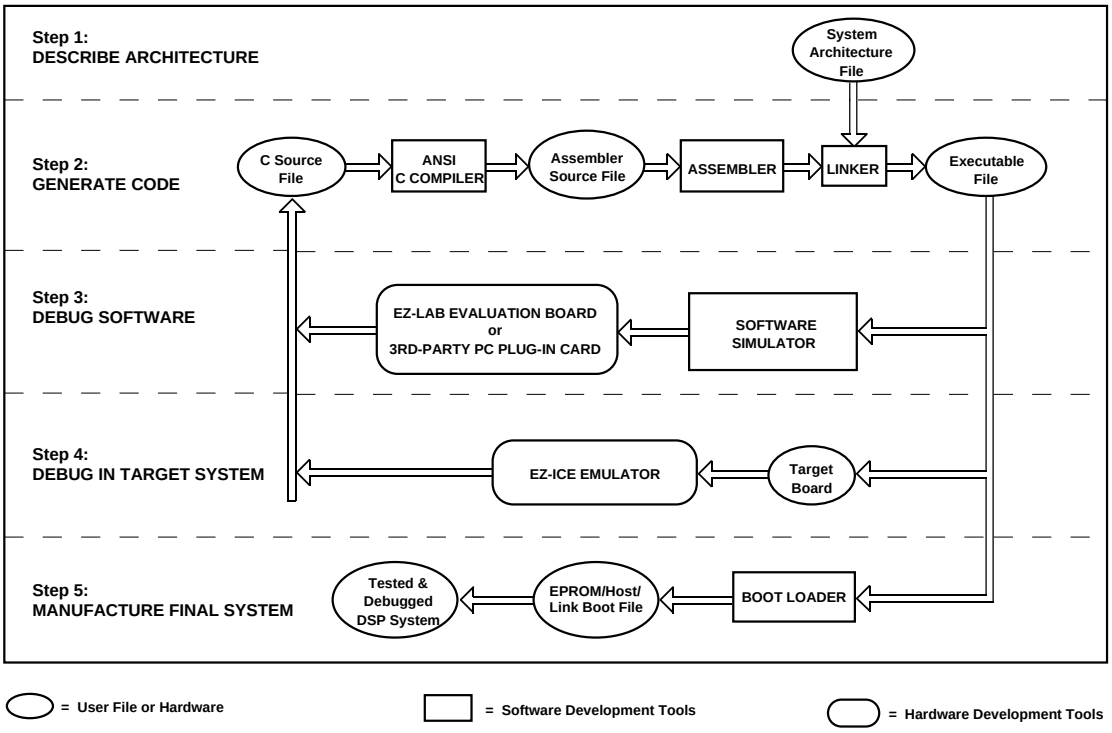


Figure 1.4 System Design and Development Process

1 Introduction

The ADSP-2106x EZ-ICE Emulator uses the IEEE 1149.1 JTAG test access port of the ADSP-2106x processor to monitor and control the target board processor during emulation. The EZ-ICE provides full-speed emulation, allowing inspection and modification of memory, registers, and processor stacks. Non-intrusive in-circuit emulation is assured by the use of the processor's JTAG interface—the emulator does not affect target system loading or timing.

Further details and ordering information are available in the *ADSP-21000 Family Hardware & Software Development Tools* data sheet. This data sheet can be requested from any Analog Devices sales office or distributor.

1.5 MESH MULTIPROCESSING

Mesh multiprocessing is a parallel processing system architecture that offers high throughput, system flexibility, and software simplicity. The ADSP-21060 and ADSP-21062 SHARC processors include features which specifically support this system architecture. Mesh multiprocessing systems are suited to a wide variety of applications including wide-area airborne radar systems, interactive medical imaging, virtual reality, high-speed engineering simulations, neural networks, and solutions of large systems of linear equations.

1.6 ADDITIONAL LITERATURE

The following publications can be ordered from any Analog Devices sales office.

ADSP-21060/62 SHARC Data Sheet
ADSP-21061 SHARC Data Sheet
ADSP-21000 Family Hardware & Software Development Tools Data Sheet
ADSP-21000 Family Assembler Tools & Simulator Manual
ADSP-21000 Family C Tools Manual
ADSP-21000 Family C Runtime Library Manual
ADSP-21000 Family Applications Handbook, Vol. 1

Computation Units 2

2.1 OVERVIEW

The computation units of the ADSP-2106x provide the numeric processing power for performing DSP algorithms. The ADSP-2106x contains three computation units: an arithmetic/logic unit (ALU), a multiplier and a shifter. Both fixed-point and floating-point operations are supported by the processor. Each computation unit executes instructions in a single cycle.

The ALU performs a standard set of arithmetic and logic operations in both fixed-point and floating-point formats. The multiplier performs floating-point and fixed-point multiplication as well as fixed-point multiply/add and multiply/subtract operations. The shifter performs logical and arithmetic shifts, bit manipulation, field deposit and extraction operations on 32-bit operands and can derive exponents as well.

The computation units are architecturally arranged in parallel, as shown in Figure 2.1 on the next page. The output of any computation unit may be the input of any computation unit on the next cycle. The computation units input data from and output data to a 10-port register file that consists of sixteen primary registers and sixteen alternate registers. The register file is accessible to the ADSP-2106x program and data memory data buses for transferring data between the computation units and external memory or other parts of the processor.

The individual registers of the register file are prefixed with an "F" when used in floating-point computations (in assembly language source code). The registers are prefixed with an "R" when used in fixed-point computations. The following instructions, for example, use the same registers:

```
F0=F1 * F2;        floating-point multiply
R0=R1 * R2;        fixed-point multiply
```

The F and R prefixes do not affect the 32-bit (or 40-bit) data transfer; they only determine how the ALU, multiplier, or shifter treat the data. The F or R may be either uppercase or lowercase; the assembler is case-insensitive.

2 Computation Units

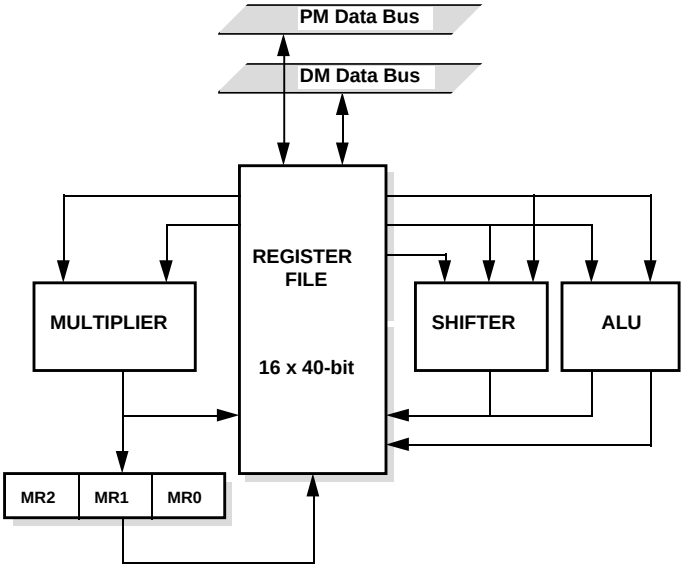


Figure 2.1 Computation Units

This chapter covers the following topics:

- Data Formats and Rounding
- ALU Architecture and Functions
- Multiplier Architecture and Functions
- Shifter Architecture and Functions
- Multifunction Computations
- Register File and Data Transfers

2.2 IEEE FLOATING-POINT OPERATIONS

The ADSP-2106x multiplier and ALU support the single-precision floating-point format specified in the IEEE 754/854 standard. This standard is described in Appendix C, *Numeric Formats*. The ADSP-2106x is IEEE 754/854 compatible for single-precision floating-point operations in all respects except that:

- The ADSP-2106x does not provide inexact flags.

- NAN ("Not-A-Number") inputs generate an invalid exception and return a quiet NAN (all 1s).

- Denormal operands are flushed to zero when input to a computation unit and do not generate an underflow exception. Any denormal or underflow result from an arithmetic operation is flushed to zero and an underflow exception is generated.

- Round-to-nearest and round-toward-zero modes are supported. Rounding to +Infinity and rounding to –Infinity are not supported.

In addition, the ADSP-2106x supports a 40-bit extended precision floating-point mode, which has eight additional LSBs of the mantissa and is compliant with the 754/854 standards; however, results in this format are more precise than the IEEE single-precision standard specifies.

2.2.1 Extended Floating-Point Precision

Floating-point data can be either 32 or 40 bits wide on the ADSP-2106x. Extended precision floating-point format (8 bits of exponent and 32 bits of mantissa) is selected if the RND32 bit in the MODE1 register is cleared (0). If this bit is set (1), then normal IEEE precision is used (8 bits exponent and 24 bits of mantissa). In this case, the computation unit sets the eight LSBs of floating-point inputs to zeros before performing the operation. The mantissa of a result is rounded to 23 bits (not including the hidden bit) and the 8 LSBs of the 40-bit result are set to zeros to form a 32-bit number that is equivalent to the IEEE standard result.

2.2.2 Short Word Floating-Point Format

The ADSP-2106x supports a 16-bit floating-point data type and provides conversion instructions for it. The short float data format has an 11-bit mantissa with a four-bit exponent plus sign bit. The 16-bit floating-point numbers reside in the lower 16 bits of the 32-bit floating-point field.

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

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

2.2.3 Floating-Point Exceptions

The multiplier and ALU each provide exception information when executing floating-point operations. Each unit updates overflow, underflow and invalid operation flags in the arithmetic status (ASTAT) register and in the sticky status (STKY) register. An underflow, overflow or invalid operation from any unit also generates a maskable interrupt. Thus, there are three ways to handle floating-point exceptions:

- **Interrupts.** The exception condition is handled immediately in an interrupt service routine. You would use this method if it was important to correct all exceptions as they happen.

- **ASTAT register.** The exception flags in the ASTAT register pertaining to a particular arithmetic operation are tested after the operation is performed. You would use this method to monitor a particular floating-point operation.

- **STKY register.** Exception flags in the STKY register are examined at the end of a series of operations. If any flags are set, some of the results are incorrect. You would use this method if exception handling was not critical.

2.3 FIXED-POINT OPERATIONS

Fixed-point numbers are always represented in 32 bits and are left-justified (occupy the 32 MSBs) in the 40-bit data fields of the ADSP-2106x. They may be treated as fractional or integer numbers and as unsigned or twos-complement. Each computation unit has its own limitations on how these formats may be mixed for a given operation. The computation units read 32-bit operands from 40-bit registers, ignoring the 8 LSBs, and write 32-bit results, zeroing the 8 LSBs.

2.4 ROUNDING

Two modes of rounding are supported in the ADSP-2106x: round-toward-zero and round-toward-nearest. The rounding modes follow the IEEE 754 standard definitions, which are briefly stated as follows:

Round-Toward-Zero. If the result before rounding is not exactly representable in the destination format, the rounded result is that number which is nearer to zero. This is equivalent to truncation.

Round-Toward-Nearest. If the result before rounding is not exactly representable in the destination format, the rounded result is that number which is nearer to the result before rounding. If the result before rounding is exactly halfway between two numbers in the destination format (differing by an LSB), the rounded result is that number which has an LSB equal to zero. Statistically, rounding up occurs as often as rounding down, so there is no large sample bias. Because the maximum floating-point value is one LSB less than the value that represents Infinity, a result that is halfway between the maximum floating-point value and Infinity rounds to Infinity in this mode.

The rounding mode for all ALU operations and for floating-point multiplier operations is determined by the TRUNC bit in the MODE1 register. If the TRUNC bit is set, the round-to-zero mode is selected; otherwise, the round-to-nearest mode is used.

For fixed-point multiplier operations on fractional data, the same two rounding modes are supported, but only the round-to-nearest operation is actually performed by the multiplier. Because the multiplier has a local result register for fixed-point operations, rounding-to-zero is accomplished implicitly by reading only the upper bits of the result and discarding the lower bits.

2.5 ALU

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

ALU instructions include:

- Floating-point addition, subtraction, add/subtract, average
- Fixed-point addition, subtraction, add/subtract, average
- Floating-point manipulation: binary log, scale, mantissa
- Fixed-point add with carry, subtract with borrow, increment, decrement
- Logical AND, OR, XOR, NOT
- Functions: Absolute value, pass, min, max, clip, compare
- Format conversion
- Reciprocal and reciprocal square root primitives

Dual add/subtract and parallel ALU and multiplier operations are described under “Multifunction Computations,” later in this chapter.

2.5.1 ALU Operation

The ALU takes one or two input operands, called the X input and the Y input, which can be any data registers in the register file. It usually returns one result; in add / subtract operations it returns two results, and in compare operations it returns no result (only flags are updated). ALU results can be returned to any location in the register file.

Input operands are transferred from the register file during the first half of the cycle. Results are transferred to the register file during the second half of the cycle. Thus the ALU can read and write the same register file location in a single cycle.

If the ALU operation is fixed-point, the X input and Y input are each treated as a 32-bit fixed-point operand. The upper 32 bits from the source location in the register file are transferred. For fixed-point operations, the result(s) are always 32-bit fixed-point values. Some floating-point operations (LOGB, MANT and FIX) can also yield fixed-point results. Fixed-point results are transferred to the upper 32 bits of register file. The lower eight bits of the register file destination are cleared.

The format of fixed-point operands and results depends on the operation. In most arithmetic operations, there is no need to distinguish between integer and fractional formats. Fixed-point inputs to operations such as scaling a floating-point value are treated as integers. For purposes of determining status such as overflow, fixed-point arithmetic operands and results are treated as twos-complement numbers.

2.5.2 ALU Operating Modes

The ALU is affected by three bits in the MODE1 register; the ALU saturation bit affects ALU operations that yield fixed-point results, and the rounding mode and rounding boundary bits affect floating-point operations in both the ALU and multiplier.

MODE1

Bit	*Name*	*Function*
13	ALUSAT	1=Enable ALU saturation (full scale in fixed-point) 0=Disable ALU saturation
15	TRUNC	1=Truncation; 0=Round to nearest
16	RND32	1=Round to 32 bits; 0=Round to 40 bits

2.5.2.1 Saturation Mode

In saturation mode, all positive fixed-point overflows cause the maximum positive fixed-point number (0x7FFF FFFF) to be returned, and all negative overflows cause the maximum negative number (0x8000 0000) to be returned. If the ALUSAT bit is set, fixed-point results that overflow are saturated. If the ALUSAT bit is cleared, fixed-point results that overflow are not saturated; the upper 32 bits of the result are returned unaltered. The ALU overflow flag reflects the ALU result before saturation.

2.5.2.2 Floating-Point Rounding Modes

The ALU supports two IEEE rounding modes. If the TRUNC bit is set, the ALU rounds a result to zero (truncation). If the TRUNC bit is cleared, the ALU rounds to nearest.

2.5.2.3 Floating-Point Rounding Boundary

The results of floating-point ALU operations can be either 32-bit or 40-bit floating-point data on the ADSP-2106x. If the RND32 bit is set, the eight LSBs of each input operand are flushed to zeros before the ALU operation is performed (except for the RND operation), and ALU floating-point results are output in the 32-bit IEEE format. The lower eight bits of the result are cleared. If the RND32 bit is cleared, the ALU inputs 40-bit operands unchanged and outputs 40-bit results from floating-point operations, and all 40 bits are written to the specified register file location.

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

2.5.3 ALU Status Flags

The ALU updates seven status flags in the ASTAT register, shown below, at the end of each operation. The states of these flags reflect the result of the most recent ALU operation. The ALU updates the Compare Accumulation bits in ASTAT at the end of every Compare operation. The ALU also updates four “sticky” status flags in the STKY register. Once set, a sticky flag remains high until explicitly cleared.

ASTAT

Bit	*Name*	*Definition*
0	AZ	ALU result zero or floating-point underflow
1	AV	ALU overflow
2	AN	ALU result negative
3	AC	ALU fixed-point carry
4	AS	ALU X input sign (ABS, MANT operations)
5	AI	ALU floating-point invalid operation
10	AF	Last ALU operation was a floating-point operation
31-24	CACC	Compare Accumulation register (results of last 8 compare operations)

STKY Bit	*Name*	*Definition*
0	AUS	ALU floating-point underflow
1	AVS	ALU floating-point overflow
2	AOS	ALU fixed-point overflow
5	AIS	ALU floating-point invalid operation

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

2.5.3.1 ALU Zero Flag (AZ)

The zero flag is determined for all fixed-point and floating-point ALU operations. AZ is set whenever the result of an ALU operation is zero. AZ also signifies floating-point underflow; see the next section. It is otherwise cleared.

2.5.3.2 ALU Underflow Flag (AZ, AUS)

Underflow is determined for all ALU operations that return a floating-point result and for floating-point to fixed-point conversion. AUS is set whenever the result of an ALU operation is smaller than the smallest number representable in the output format. AZ is set whenever a floating-point result is smaller than the smallest number representable in the output format.

2.5.3.3 ALU Negative Flag (AN)

The negative flag is determined for all ALU operations. It is set whenever the result of an ALU operation is negative. It is otherwise cleared.

2.5.3.4 ALU Overflow Flag (AV, AOS, AVS)

Overflow is determined for all fixed-point and floating-point ALU operations. For fixed-point results, AV and AOS are set whenever the XOR of the two most significant bits is a 1; otherwise AV is cleared. For floating-point results AV and AVS are set whenever the post-rounded result overflows (unbiased exponent > 127); otherwise AV is cleared.

2.5.3.5 ALU Fixed-Point Carry Flag (AC)

The carry flag is determined for all fixed-point ALU operations. For fixed-point arithmetic operations, AC is set if there is a carry out of most significant bit of the result, and is otherwise cleared. AC is cleared for fixed-point logic, PASS, MIN, MAX, COMP, ABS, and CLIP operations. The ALU reads the AC flag in fixed-point addition with carry and fixed-point subtraction with carry operations.

2.5.3.6 ALU Sign Flag (AS)

The sign flag is determined for only the fixed-point and floating-point ABS operations and the MANT operation. AS is set if the input operand is negative. It is otherwise cleared. The ALU clears AS for all operations other than ABS and MANT operations; this is different from the operation of ADSP-2100 family processors, which do not update the AS flag on operations other than ABS.

2.5.3.7 ALU Invalid Flag (AI)

The invalid flag is determined for all floating-point ALU operations. AI and AIS are set whenever

- an input operand is a NAN
- an addition of opposite-signed Infinities is attempted
- a subtraction of like-signed Infinities is attempted
- when saturation mode is not set, a floating-point to fixed-point conversion results in an overflow or operates on an Infinity.

AI is otherwise cleared.

2.5.3.8 ALU Floating-Point Flag (AF)

AF is determined for all fixed-point and floating-point ALU operations. It is set if the last operation was a floating-point operation; it is otherwise cleared.

2.5.3.9 Compare Accumulation

Bits 31-24 in the ASTAT register store the flag results of up to eight ALU compare operations. These bits form a right-shift register. When an ALU compare operation is executed, the eight bits are shifted toward the LSB (bit 24 is lost). The MSB, bit 31, is then written with the result of the compare operation. If the X operand is greater than the Y operand in the compare instruction, bit 31 is set; it is cleared otherwise. The accumulated compare flags can be used to implement 2- and 3-dimensional clipping operations for graphics applications.

2.5.4 ALU Instruction Summary

	Instruction	*ASTAT Status Flags*								*STKY Status Flags*			
Fixed-point:		AZ	AV	AN	AC	AS	AI	AF	CACC	AUS	AVS	AOS	AIS
c	Rn = Rx + Ry	*	*	*	*	0	0	0	–	–	–	**	–
c	Rn = Rx – Ry	*	*	*	*	0	0	0	–	–	–	**	–
c	Rn = Rx + Ry + CI	*	*	*	*	0	0	0	–	–	–	**	–
c	Rn = Rx – Ry + CI – 1	*	*	*	*	0	0	0	–	–	–	**	–
	Rn = (Rx + Ry) / 2	*	0	*	*	0	0	0	–	–	–	–	–
	COMP(Rx, Ry)	*	0	*	0	0	0	0	*	–	–	–	–
	Rn = Rx + CI	*	*	*	*	0	0	0	–	–	–	**	–
	Rn = Rx + CI – 1	*	*	*	*	0	0	0	–	–	–	**	–
	Rn = Rx + 1	*	*	*	*	0	0	0	–	–	–	**	–
	Rn = Rx – 1	*	*	*	*	0	0	0	–	–	–	**	–
c	Rn = –Rx	*	*	*	*	0	0	0	–	–	–	**	–
c	Rn = ABS Rx	*	*	0	0	*	0	0	–	–	–	**	–
	Rn = PASS Rx	*	0	*	0	0	0	0	–	–	–	–	–
c	Rn = Rx AND Ry	*	0	*	0	0	0	0	–	–	–	–	–
c	Rn = Rx OR Ry	*	0	*	0	0	0	0	–	–	–	–	–
c	Rn = Rx XOR Ry	*	0	*	0	0	0	0	–	–	–	–	–
c	Rn = NOT Rx	*	0	*	0	0	0	0	–	–	–	–	–
	Rn = MIN(Rx, Ry)	*	0	*	0	0	0	0	–	–	–	–	–
	Rn = MAX(Rx, Ry)	*	0	*	0	0	0	0	–	–	–	–	–
	Rn = CLIP Rx BY Ry	*	0	*	0	0	0	0	–	–	–	–	–
Floating–point:													
	Fn = Fx + Fy	*	*	*	0	0	*	1	–	**	**	–	**
	Fn = Fx – Fy	*	*	*	0	0	*	1	–	**	**	–	**
	Fn = ABS (Fx + Fy)	*	*	0	0	0	*	1	–	**	**	–	**
	Fn = ABS (Fx – Fy)	*	*	0	0	0	*	1	–	**	**	–	**
	Fn = (Fx + Fy) / 2	*	0	*	0	0	*	1	–	**	–	–	**
	COMP(Fx, Fy)	*	0	*	0	0	*	1	*	–	–	–	**
	Fn = –Fx	*	*	*	0	0	*	1	–	–	**	–	**
	Fn = ABS Fx	*	*	0	0	*	*	1	–	–	**	–	**
	Fn = PASS Fx	*	0	*	0	0	*	1	–	–	–	–	**
	Fn = RND Fx	*	*	*	0	0	*	1	–	–	**	–	**
	Fn = SCALB Fx BY Ry	*	*	*	0	0	*	1	–	**	**	–	**
	Rn = MANT Fx	*	*	0	0	*	*	1	–	–	**	–	**
	Rn = LOGB Fx	*	*	*	0	0	*	1	–	–	**	–	**
	Rn = FIX Fx BY Ry	*	*	*	0	0	*	1	–	**	**	–	**
	Rn = FIX Fx	*	*	*	0	0	*	1	–	**	**	–	**
	Fn = FLOAT Rx BY Ry	*	*	*	0	0	0	1	–	**	**	–	–
	Fn = FLOAT Rx	*	0	*	0	0	0	1	–	–	–	–	–
	Fn = RECIPS Fx	*	*	*	0	0	*	1	–	**	**	–	**
	Fn = RSQRTS Fx	*	*	*	0	0	*	1	–	–	**	–	**
	Fn = Fx COPYSIGN Fy	*	0	*	0	0	*	1	–	–	–	–	**
	Fn = MIN(Fx, Fy)	*	0	*	0	0	*	1	–	–	–	–	**
	Fn = MAX(Fx, Fy)	*	0	*	0	0	*	1	–	–	–	–	**
	Fn = CLIP Fx BY Fy	*	0	*	0	0	*	1	–	–	–	–	**

Rn, Rx, Ry = Any register file location; treated as fixed-point
Fn, Fx, Fy = Any register file location; treated as floating-point
c = ADSP-21xx-compatible instruction

* set or cleared, depending on results of instruction
** may be set (but not cleared), depending on results of instruction
– no effect

2.6 MULTIPLIER

The multiplier performs fixed-point or floating-point multiplication and fixed-point multiply/accumulate operations. Fixed-point multiply/accumulates may be performed with either cumulative addition or cumulative subtraction. Floating-point multiply/accumulates can be accomplished through parallel operation of the ALU and multiplier, using multifunction instructions. See "Multifunction Computations" later in this chapter.

Multiplier floating-point instructions operate on 32-bit or 40-bit floating-point operands and output 32-bit or 40-bit floating-point results. Multiplier fixed-point instructions operate on 32-bit fixed-point data and produce 80-bit results. Inputs are treated as fractional or integer, unsigned or twos-complement.

Multiplier instructions include:

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

2.6.1 Multiplier Operation

The multiplier takes two input operands, called the X input and the Y input, which can be any data registers in the register file. Fixed-point operations can accumulate fixed-point results in either of two local multiplier result registers (MR) or write results back to the register file. Results stored in the MR registers can also be rounded or saturated in separate operations. Floating-point operations yield floating-point results, which are always written directly back to the register file.

Input operands are transferred during the first half of the cycle. Results are transferred during the second half of the cycle. Thus the multiplier can read and write the same register file location in a single cycle.

If the multiplier operation is fixed-point, inputs taken from the register file are read from the upper 32 bits of the source location. Fixed-point operands may be treated as both in integer format or both in fractional format. The format of the result is the same as the format of the inputs. Each fixed-point operand may be treated as either an unsigned or a twos-complement number. If both inputs are fractional and signed, the multiplier automatically shifts the result left one bit to remove the redundant sign bit. The input data type is specified within the multiplier instruction.

2.6.2 Fixed-Point Results

Fixed-point operations yield 80-bit results in the MR register. The location of a result in the 80-bit field depends on whether the result is in fractional or integer format, as shown in Figure 2.2. If the result is sent directly to the register file, the 32 bits that have the same format as the input data are transferred, i.e. bits 63-32 for a fractional result or bits 31-0 for an integer result. The eight LSBs of the 40-bit register file location are zero-filled. Fractional results can be rounded-to-nearest before being sent to the register file, as explained later in this chapter. If rounding is not specified, discarding bits 31-0 effectively truncates a fractional result (rounds to zero).

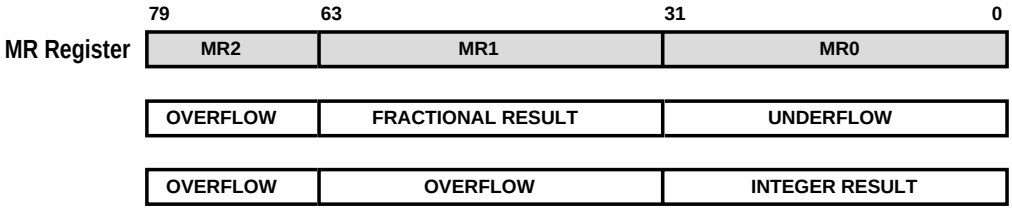


Figure 2.2 Multiplier Fixed-Point Result Placement

2.6.2.1 MR Registers

The entire result can be sent to one of two dedicated 80-bit result registers (MR). The MR registers have identical format; each is divided into MR2, MR1 and MR0 registers that can be individually read from or written to the register file. When data is read from MR2, it is sign-extended to 32 bits (see Figure 2.3). The eight LSBs of the 40-bit register file location are zero-filled when data is read from MR2, MR1 or MR0 to the register file. Data is written into MR2, MR1 or MR0 from the 32 MSBs of a register file location; the eight LSBs are ignored. Data written to MR1 is sign-extended to MR2, i.e. the MSB of MR1 is repeated in the 16 bits of MR2. Data written to MR0, however, is not sign-extended.

The two MR registers are designated MRF (foreground) and MRB (background); *foreground* refers to those registers currently activated by the SRCU bit in the MODE1 register, and *background* refers to those that are not. In the case that only one MR register is used at a time, the SRCU bit activates one or the other to facilitate context switching. However, unlike other registers for which alternate sets exist, both MR register sets are accessible at the same time. All (fixed-point) accumulation instructions

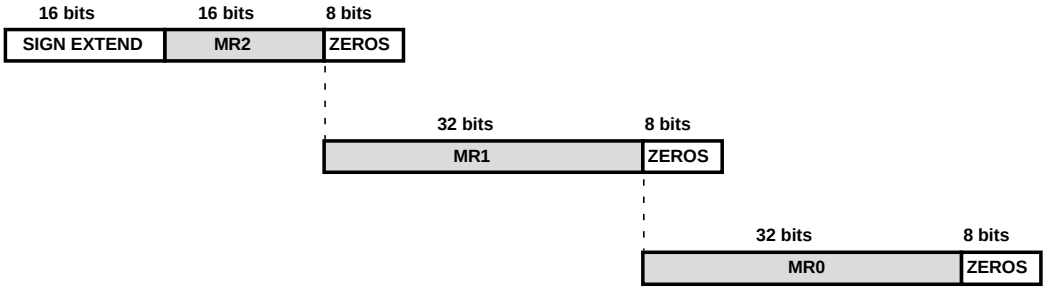


Figure 2.3 MR Transfer Formats

may specify either result register for accumulation, regardless of the state of the SRCU bit. Thus, instead of using the MR registers as a primary and an alternate, you can use them as two parallel accumulators. This feature facilitates complex math.

Transfers between MR registers and the register file are considered computation unit operations, since they involve the multiplier. Thus, although the syntax for the transfer is the same as for any other transfer to or from the register file, an MR transfer is placed in an instruction where a computation is normally specified. For example, the ADSP-2106x can perform a multiply/accumulate in parallel with a read of data memory, as in:

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

or it can perform an MR transfer instead of the computation, as in:

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

2.6.3 Fixed-Point Operations

In addition to multiplication, fixed-point operations include accumulation, rounding and saturation of fixed-point data. There are three MR register operations: Clear, Round and Saturate.

2.6.3.1 Clear MR Register

The clear operation resets the specified MR register to zero. This operation is performed at the start of a multiply/accumulate operation to remove results left over from the previous operation.

2.6.3.2 Round MR Register

Rounding of a fixed-point result occurs either as part of a multiply or multiply / accumulate operation or as an explicit operation on the MR register. The rounding operation applies only to fractional results (integer results are not affected) and rounds the 80-bit MR value to nearest at bit 32, i.e. at the MR1-MR0 boundary. The rounded result in MR1 can be sent either to the register file or back to the same MR register. To round a fractional result to zero (truncation) instead of to nearest, you would simply transfer the unrounded result from MR1, discarding the lower 32 bits in MR0.

2.6.3.3 Saturate MR Register On Overflow

The saturate operation sets MR to a maximum value if the MR value has overflowed. Overflow occurs when the MR value is greater than the maximum value for the data format (unsigned or twos-complement and integer or fractional) that is specified in the saturate instruction. There are six possible maximum values (shown in hexadecimal):

MR2	**MR1**	**MR0**	
Maximum twos-complement fractional number			
0000	7FFF FFFF	FFFF FFFF	positive
FFFF	8000 0000	0000 0000	negative
Maximum twos-complement integer number			
0000	0000 0000	7FFF FFFF	positive
FFFF	FFFF FFFF	8000 0000	negative
Maximum unsigned fractional number			
0000	FFFF FFFF	FFFF FFFF	
Maximum unsigned integer number			
0000	0000 0000	FFFF FFFF	

The result from MR saturation can be sent either to the register file or back to the same MR register.

2.6.4 Floating-Point Operating Modes

The multiplier is affected by two mode status bits in the MODE1 register: the rounding mode and rounding boundary bits, which affect operations in both the multiplier and the ALU.

MODE1

Bit	*Name*	*Function*
15	TRUNC	1=Truncation; 0=Round to nearest
16	RND32	1=Round to 32 bits; 0=Round to 40 bits

2.6.4.1 Floating-Point Rounding Modes

The multiplier supports two IEEE rounding modes for floating-point operations. If the TRUNC bit is set, the multiplier rounds a floating-point result to zero (truncation). If the TRUNC bit is cleared, the multiplier rounds to nearest.

2.6.4.2 Floating-Point Rounding Boundary

Floating-point multiplier inputs and results can be either 32-bit or 40-bit floating-point data on the ADSP-2106x. If the RND32 bit is set, the eight LSBs of each input operand are flushed to zeros before multiplication, and floating-point results are output in the 32-bit IEEE format, with the lower eight bits of the 40-bit register file location cleared. The mantissa of the result is rounded to 23 bits (not including the hidden bit). If the RND32 bit is cleared, the multiplier inputs full 40-bit values from the register file and outputs results in the 40-bit extended IEEE format, with the mantissa rounded to 31 bits not including the hidden bit.

2.6.5 Multiplier Status Flags

The multiplier updates four status flags at the end of each operation. All of these flags appear in the ASTAT register. The states of these flags reflect the result of the most recent multiplier operation. The multiplier also updates four “sticky” status flags in the STKY register. Once set, a sticky flag remains high until explicitly cleared.

ASTAT

Bit	*Name*	*Definition*
6	MN	Multiplier result negative
7	MV	Multiplier overflow
8	MU	Multiplier underflow
9	MI	Multiplier floating-point invalid operation

STKY

Bit	*Name*	*Definition*
6	MOS	Multiplier fixed-point overflow
7	MVS	Multiplier floating-point overflow
8	MUS	Multiplier underflow
9	MIS	Multiplier floating-point invalid operation

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

2.6.5.1 Multiplier Underflow Flag (MU)

Underflow is determined for all fixed-point and floating-point multiplier operations. It is set whenever the result of a multiplier operation is smaller than the smallest number representable in the output format. It is otherwise cleared.

For floating-point results, MU and MUS are set whenever the post-rounded result underflows (unbiased exponent < –126). Denormal operands are treated as Zeros, therefore they never cause underflows.

For fixed-point results, MU and MUS depend on the data format and are set under the following conditions:

Twos-complement:
Fractional: upper 48 bits all zeros or all ones, lower 32 bits not all zeros
Integer: not possible

Unsigned:
Fractional: upper 48 bits all zeros, lower 32 bits not all zeros
Integer: not possible

If the fixed-point result is sent to an MR register, the underflowed portion of the result is available in MR0 (fractional result only).

2.6.5.2 Multiplier Negative Flag (MN)

The negative flag is determined for all multiplier operations. MN is set whenever the result of a multiplier operation is negative. It is otherwise cleared.

2.6.5.3 Multiplier Overflow Flag (MV)

Overflow is determined for all fixed-point and floating-point multiplier operations.

For floating-point results, MV and MVS are set whenever the post-rounded result overflows (unbiased exponent > 127).

For fixed-point results, MV and MOS depend on the data format and are set under the following conditions:

Twos-complement:

Fractional:	upper 17 bits of MR not all zeros or all ones
Integer:	upper 49 bits of MR not all zeros or all ones

Unsigned:

Fractional:	upper 16 bits of MR not all zeros
Integer:	upper 48 bits of MR not all zeros

If the fixed-point result is sent to an MR register, the overflowed portion of the result is available in MR1 and MR2 (integer result) or MR2 only (fractional result).

2.6.5.4 Multiplier Invalid Flag (MI)

The invalid flag is determined for floating-point multiplication. MI is set whenever:

- an input operand is a NAN.
- the inputs are Infinity and Zero (note: denormal inputs are treated as Zeros.)

MI is otherwise cleared.

2.6.6 Multiplier Instruction Summary

Instruction	*ASTAT Flags*				*STKY Flags*			
	MU	MN	MV	MI	MUS	MOS	MVS	MIS
Fixed-Point:								
Rn / MRF / MRB = Rx * Ry (S / U , S / U , F / I / FR)	*	*	*	0	–	**	–	–
Rn = MRF + Rx * Ry (S / U , S / U , F / I / FR)	*	*	*	0	–	**	–	–
Rn = MRB + Rx * Ry (S / U , S / U , F / I / FR)	*	*	*	0	–	**	–	–
MRF = MRF + Rx * Ry (S / U , S / U , F / I / FR)	*	*	*	0	–	**	–	–
MRB = MRB + Rx * Ry (S / U , S / U , F / I / FR)	*	*	*	0	–	**	–	–
Rn = MRF – Rx * Ry (S / U , S / U , F / I / FR)	*	*	*	0	–	**	–	–
Rn = MRB – Rx * Ry (S / U , S / U , F / I / FR)	*	*	*	0	–	**	–	–
MRF = MRF – Rx * Ry (S / U , S / U , F / I / FR)	*	*	*	0	–	**	–	–
MRB = MRB – Rx * Ry (S / U , S / U , F / I / FR)	*	*	*	0	–	**	–	–
Rn = SAT MRF (SI)	*	*	*	0	–	**	–	–
Rn = SAT MRB (UI)	*	*	*	0	–	**	–	–
MRF = SAT MRF (SF)	*	*	*	0	–	**	–	–
MRB = SAT MRB (UF)	*	*	*	0	–	**	–	–
Rn = RND MRF (SF)	*	*	*	0	–	**	–	–
Rn = RND MRB (UF)	*	*	*	0	–	**	–	–
MRF = RND MRF	*	*	*	0	–	**	–	–
MRB = RND MRB	*	*	*	0	–	**	–	–
MRF / MRB = 0	0	0	0	0	–	–	–	–
MRxF / MRxB = Rn	0	0	0	0	–	–	–	–
Rn = MRxF / MRxB	0	0	0	0	–	–	–	–
Floating-Point:								
Fn = Fx * Fy	*	*	*	*	**	–	**	**

Note: For floating-point multiply/accumulates, see "Multifunction Computations"

* set or cleared, depending on results of instruction
** may be set (but not cleared), depending on results of instruction
– no effect

Rn, Rx, Ry = R15-R0; register file location, treated as fixed-point
Fn, Fx, Fy = F15-F0; register file location, treated as floating-point
MRxF = MR2F, MR1F, MR0F; multiplier result accumulators, foreground
MRxB = MR2B, MR1B, MR0B; multiplier result accumulators, background

Multiplier Instruction Summary, cont.

Optional Modifiers for Fixed-Point:

(X-input ❑ | Y-input ❑ | Data format, rounding ❑)

S	Signed input
U	Unsigned input
I	Integer input(s)
F	Fractional input(s)
FR	Fractional inputs, Rounded output
(SF)	Default format for 1-input operations
(SSF)	Default format for 2-input operations

2.7 SHIFTER

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

- shifts and rotates from off-scale left to off-scale right
- bit manipulation operations, including bit set, clear, toggle, and test
- bit field manipulation operations including extract and deposit
- support for ADSP-2100 family compatible fixed-point/floating-point conversion operations (exponent extract, number of leading 1s or 0s)

2.7.1 Shifter Operation

The shifter takes from one to three input operands: the X-input, which is operated upon; the Y-input, which specifies shift magnitudes, bit field lengths or bit positions; and the Z-input, which is operated on and updated (as in, for example, Rn = Rn OR LSHIFT Rx BY Ry). The shifter returns one output to the register file.

Input operands are fetched from the upper 32 bits of a register file location (bits 39-8, as shown in Figure 2.4 on the following page) or from an immediate value in the instruction. The operands are transferred during the first half of the cycle. The result is transferred to the upper 32 bits of a register (with the eight LSBs zero-filled) during the second half of the cycle. Thus the shifter can read and write the same register file location in a single cycle.

The X-input and Z-input are always 32-bit fixed-point values. The Y-input is a 32-bit fixed-point value or an 8-bit field (*shf8*), positioned in the register file as shown in Figure 2.4 below.

Some shifter operations produce 8-bit or 6-bit results. These results are placed in either the *shf8* field or the *bit6* field (see Figure 2.5) and are sign-extended to 32 bits. Thus the shifter always returns a 32-bit result.

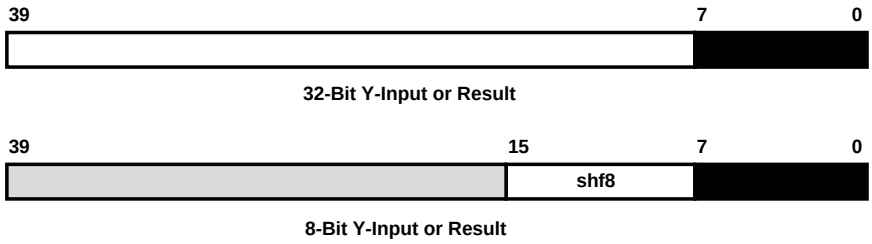


Figure 2.4 Register File Fields For Shifter Instructions

2.7.2 Bit Field Deposit & Extract Instructions

The shifter's bit field deposit and bit field extract instructions allow the manipulation of groups of bits within a 32-bit fixed-point integer word.

The Y-input for these instructions specifies two 6-bit values, *bit6* and *len6*, positioned in the Ry register as shown in Figure 2.5. Bit6 and len6 are interpreted as positive integers. Bit6 is the starting bit position for the deposit or extract. Len6 is the bit field length, which specifies how many bits are deposited or extracted.

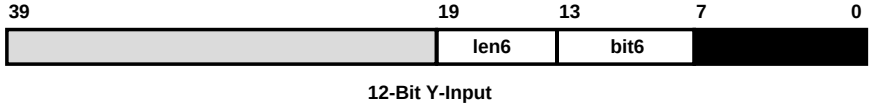


Figure 2.5 Register File Fields For FDEP, FEXT Instructions

The FDEP (field deposit) instructions take a group of bits from the input register Rx (starting at the LSB of the 32-bit integer field) and deposit them anywhere within the result register Rn. The bit6 value specifies the starting bit position for the deposit. See Figure 2.6.

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

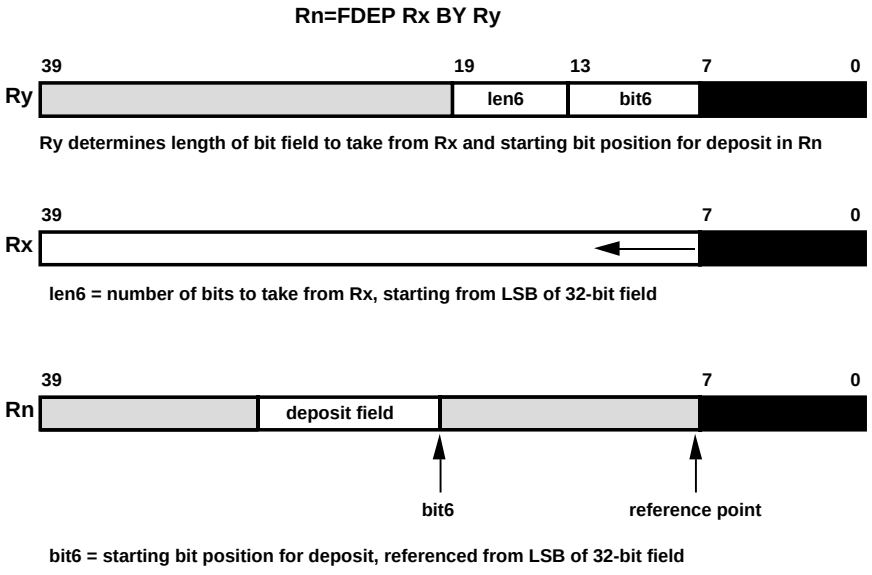


Figure 2.6 Bit Field Deposit Instruction

2 Computation Units

The following field deposit instruction example is pictured in Figure 2.7:

```
R0=FDEP R1 BY R2;
```

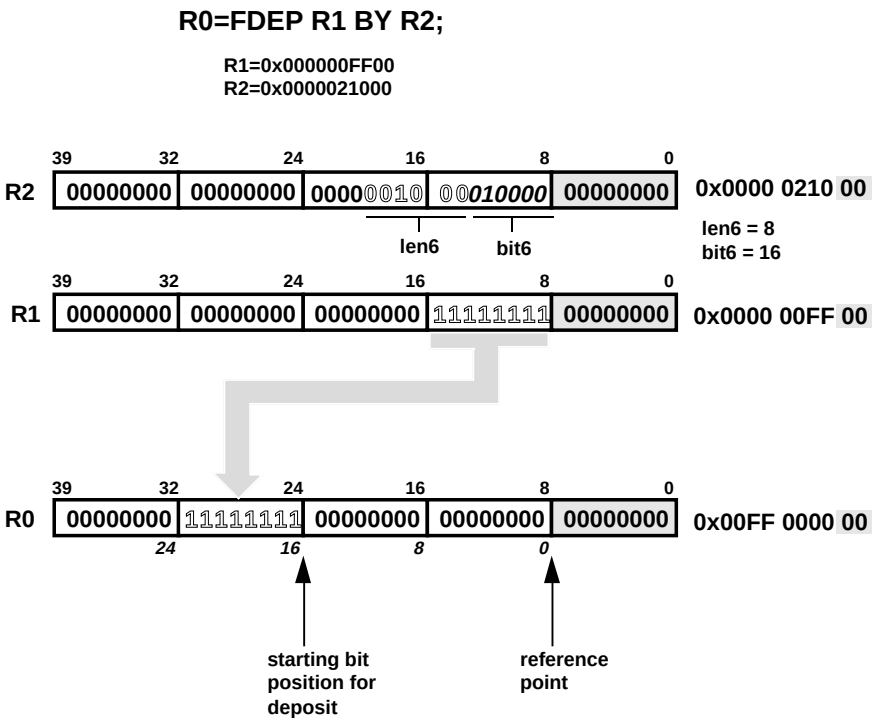


Figure 2.7 Bit Field Deposit Example

The following field extract instruction example is pictured in Figure 2.8:

```
R3=FEXT R4 BY R5;
```

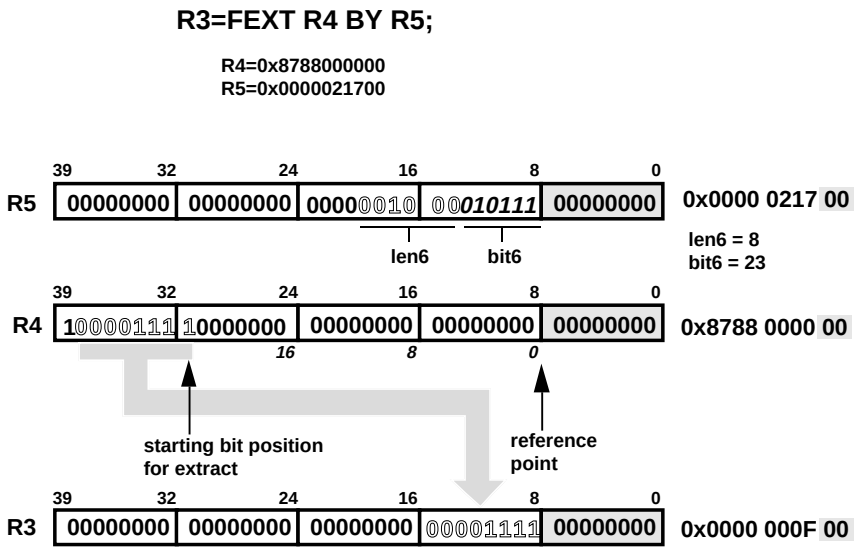


Figure 2.8 Bit Field Extract Example

2.7.3 Shifter Status Flags

The shifter returns three status flags at the end of the operation. All of these flags appear in the ASTAT register. The SZ flag indicates if the output is zero, the SV flag indicates an overflow, and the SS flag indicates the sign bit in exponent extract operations.

ASTAT

Bit	*Name*	*Definition*
11	SV	Shifter overflow of bits to left of MSB
12	SZ	Shifter result zero
13	SS	Shifter input sign (for exponent extract only)

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

2.7.3.1 Shifter Zero Flag (SZ)

SZ is affected by all shifter operations. It is set whenever:

- the result of a shifter operation is zero, or
- a bit test instruction specifies a bit outside of the 32-bit fixed-point field.

SZ is otherwise cleared.

2.7.3.2 Shifter Overflow Flag (SV)

SV is affected by all shifter operations. It is set whenever:

- significant bits are shifted to the left of the 32-bit fixed-point field,
- a bit outside of the 32-bit fixed-point field is tested, set or cleared,
- a field that is partially or wholly to the left of the 32-bit fixed-point field is extracted, or
- a LEFTZ or LEFTO operation returns a result of 32.

SV is otherwise cleared.

2.7.3.3 Shifter Sign Flag (SS)

SS is affected by all shifter operations. For the two EXP (exponent extract) operations, it is set if the fixed-point input operand is negative and cleared if it is positive. For all other shifter operations, SS is cleared.

2.7.4 Shifter Instruction Summary

	Instruction	*Flags* SZ	SV	SS
c	Rn = LSHIFT Rx BY Ry	*	*	0
c	Rn = LSHIFT Rx BY <data8>	*	*	0
c	Rn = Rn OR LSHIFT Rx BY Ry	*	*	0
c	Rn = Rn OR LSHIFT Rx BY <data8>	*	*	0
c	Rn = ASHIFT Rx BY Ry	*	*	0
c	Rn = ASHIFT Rx BY<data8>	*	*	0
c	Rn = Rn OR ASHIFT Rx BY Ry	*	*	0
c	Rn = Rn OR ASHIFT Rx BY <data8>	*	*	0
	Rn = ROT Rx BY RY	*	0	0
	Rn = ROT Rx BY <data8>	*	0	0
	Rn = BCLR Rx BY Ry	*	*	0
	Rn = BCLR Rx BY <data8>	*	*	0
	Rn = BSET Rx BY Ry	*	*	0
	Rn = BSET Rx BY <data8>	*	*	0
	Rn = BTGL Rx BY Ry	*	*	0
	Rn = BTGL Rx BY <data8>	*	*	0
	BTST Rx BY Ry	*	*	0
	BTST Rx BY <data8>	*	*	0
	Rn = FDEP Rx BY Ry	*	*	0
	Rn = FDEP Rx BY <bit6>:<len6>	*	*	0
	Rn = Rn OR FDEP Rx BY Ry	*	*	0
	Rn = Rn OR FDEP Rx BY <bit6>:<len6>	*	*	0
	Rn = FDEP Rx BY Ry (SE)	*	*	0
	Rn = FDEP Rx BY <bit6>:<len6> (SE)	*	*	0
	Rn = Rn OR FDEP Rx BY Ry (SE)	*	*	0
	Rn = Rn OR FDEP Rx BY <bit6>:<len6> (SE)	*	*	0
	Rn = FEXT Rx BY Ry	*	*	0
	Rn = FEXT Rx BY <bit6>:<len6>	*	*	0
	Rn = FEXT Rx BY Ry (SE)	*	*	0
	Rn = FEXT Rx BY <bit6>:<len6> (SE)	*	*	0
c	Rn = EXP Rx (EX)	*	0	*
c	Rn = EXP Rx	*	0	*
	Rn = LEFTZ Rx	*	*	0
	Rn = LEFTO Rx	*	*	0
	Rn = FPACK Fx	0	*	0
	Fn = FUNPACK Rx	0	0	0

* = Depends on data
Rn, Rx, Ry = Any register file location; bit fields used depend on instruction
Fn, Fx = Any register file location; floating-point word
c = ADSP-2100-compatible instruction

2.8 MULTIFUNCTION COMPUTATIONS

In addition to the computations performed by each computation unit, the ADSP-2106x also provides multifunction computations that combine parallel operation of the multiplier and the ALU, or dual functions in the ALU. The two operations are performed in the same way as they are in corresponding single-function computations. Flags are also determined in the same way as for the same single-function computations, except that in the dual add/subtract computation the ALU flags from the two operations are ORed together.

Each of the four input operands for computations that use both the ALU and multiplier are constrained to a different set of four register file locations, as summarized below and shown in Figure 2.9. For example, the X-input to the ALU can only be R8, R9, R10 or R11. In all other operations, the input operands may be any register file locations.

Dual Add/Subtract

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

Fixed-Point Multiply/Accumulate and Add, Subtract or Average

Rm=R3-0 * R7-4 (SSFR)	,	Ra=R11-8 + R15-12
MRF=MRF + R3-0 * R7-4 (SSF)	,	Ra=R11-8 – R15-12
Rm=MRF + R3-0 * R7-4 (SSFR)	,	Ra=(R11-8 + R15-12)/2
MRF=MRF – R3-0 * R7-4 (SSF)	,	
Rm=MRF – R3-0 * R7-4 (SSFR)	,	

Floating-Point Multiplication and ALU Operation

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

Multiplication and Dual Add/Subtract

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

Rm, Ra, Rs, Rx, Ry –Any register file location; fixed-point
Fm, Fa, Fs, Fx, Fy –Any register file location; floating-point

R3-0	–R3, R2, R1, R0	F3-0	–F3, F2, F1, F0
R7-4	–R7, R6, R5, R4	F7-4	–F7, F6, F5, F4
R11-8	–R11, R10, R9, R8	F11-8	–F11, F10, F9, F8
R15-12	–R15, R14, R13, R12	F15-12	–F15, F14, F13, F12

SSFR –X-input signed, Y-input signed, Fractional input, Rounded-to-nearest output
SSF –X-input signed, Y-input signed, Fractional input

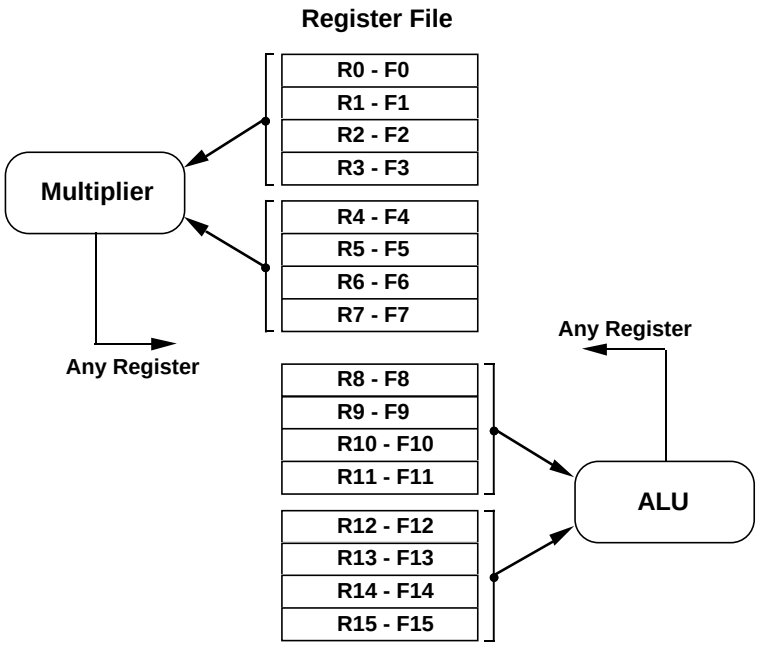


Figure 2.9 Input Registers For Multifunction Computations (ALU & Multiplier)

2.9 REGISTER FILE

The register file provides the interface between the processor's internal data buses and the computation units. It also provides local storage for operands and results. The register file consists of 16 primary registers and 16 alternate (secondary) registers. All of the data registers are 40 bits wide. 32-bit data from the computation units is always left-justified; on register reads, the eight LSBs are ignored, and on writes, the eight LSBs are written with zeros.

Program memory data accesses and data memory accesses to the register file occur on the PM Data bus and DM Data bus, respectively. One PM Data bus and / or one DM Data bus access can occur in one cycle. Transfers between the register file and the 40-bit DM Data bus are always 40 bits wide. The register file transfers data to and from the 48-bit PM Data bus in the most significant 40 bits, writing zeros in the lower eight bits on transfers to the PM Data bus.

If the same register file location is specified as both the source of an operand and the destination of a result or memory fetch, the read occurs in the first half of the cycle and the write in the second half. Thus the old data is used as the operand before the location is updated with the new result data. If writes to the same location take place in the same cycle, only the write with higher precedence actually occurs. Precedence is determined by the source of the data being written; from highest to lowest, the precedence is:

- Data memory or universal register
- Program memory
- ALU
- Multiplier
- Shifter

The individual registers of the register file are prefixed with an “F” when used in floating-point computations (in assembly language source code). The registers are prefixed with an “R” when used in fixed-point computations. The following instructions, for example, use the same registers:

```
F0=F1*F2;        floating-point multiply
R0=R1*R2;        fixed-point multiply
```

The F and R prefixes do not affect the 32-bit (or 40-bit) data transfer; they only determine how the ALU, multiplier, or shifter treat the data. The F or R may be either uppercase or lowercase; the assembler is case-insensitive.

2.9.1 Alternate (Secondary) Registers

To facilitate fast context switching, the register file has an alternate register set. Each half of the register file—the lower half, R0 through R7, and the upper half, R8 through R15—can independently activate its alternate register set. Two bits in the MODE1 register select the active sets. Data can be shared between contexts by placing the data to be shared in one half of the register file and activating the alternate register set of the other half.

MODE1

Bit	*Name*	*Definition*
7	SRRFH	Register file alternate select for R15-R8 (F15-F8)
10	SRRFL	Register file alternate select for R7-R0 (F7-F0)

Note that there is one cycle of effect latency from the instruction setting the bit in MODE1 to when the alternate registers may be accessed. For example,

```
BIT SET MODE1 SRRFL; /* activate alternate registers */
NOP;                 /* wait until alternate registers activate */
R0=7;
```

2 Computation Units

Program Sequencing 3

3.1 OVERVIEW

Program flow in the ADSP-2106x is most often linear; the processor executes program instructions sequentially. Variations in this linear flow are provided by the following program structures, illustrated in Figure 3.1 on the following page:

- *Loops.* One sequence of instructions is executed several times with zero overhead.
- *Subroutines.* The processor temporarily interrupts sequential flow to execute instructions from another part of program memory.
- *Jumps.* Program flow is permanently transferred to another part of program memory.
- *Interrupts.* A special case of subroutines in which the execution of the routine is triggered by an event that happens at run time, not by a program instruction.
- *Idle.* A special instruction that causes the processor to cease operations, holding its current state. When an interrupt occurs, the processor services the interrupt and continues normal execution.

Managing these program structures is the job of the ADSP-2106x's program sequencer. The program sequencer selects the address of the next instruction, generating most of those addresses itself. It also performs a wide range of related functions, such as

- incrementing the fetch address,
- maintaining stacks,
- evaluating conditions,
- decrementing the loop counter,
- calculating new addresses,
- maintaining an instruction cache, and
- handling interrupts.

3 Program Sequencing

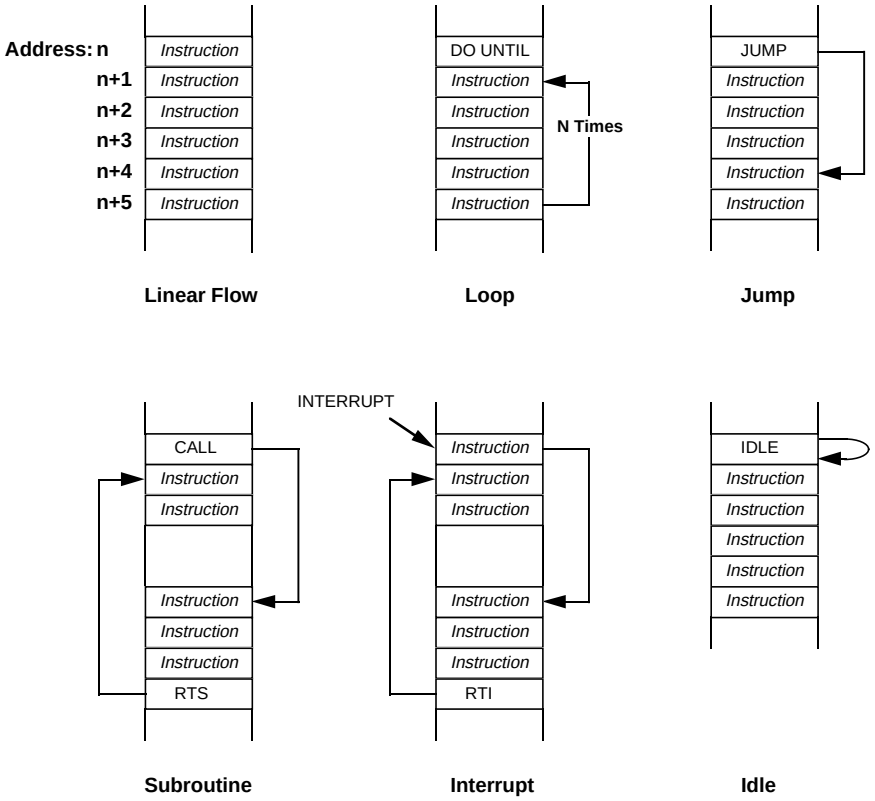


Figure 3.1 Program Flow Variations

3.1.1 Instruction Cycle

The ADSP-2106x processes instructions in three clock cycles:

- In the *fetch* cycle, the ADSP-2106x reads the instruction from either the on-chip instruction cache or from program memory.
- During the *decode* cycle, the instruction is decoded, generating conditions that control instruction execution.
- In the *execute* cycle, the ADSP-2106x executes the instruction; the operations specified by the instruction are completed.

These cycles are overlapping, or pipelined, as shown in Figure 3.2. In sequential program flow, when one instruction is being fetched, the instruction fetched in the previous cycle is being decoded, and the instruction fetched two cycles before is being executed. Thus, the throughput is one instruction per cycle.

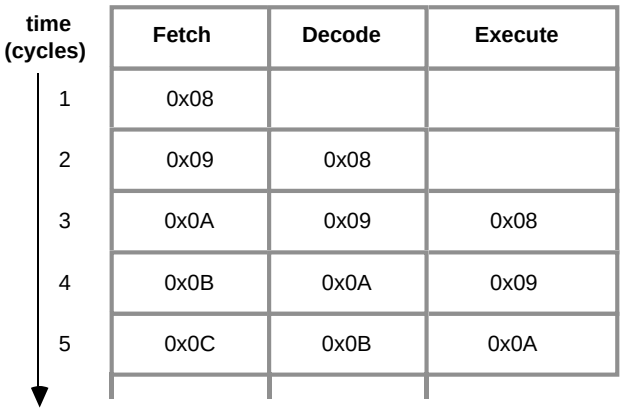


Figure 3.2 Pipelined Execution Cycles

Any non-sequential program flow can potentially decrease the ADSP-2106x's instruction throughput. Non-sequential program operations include:

- Program memory data accesses that conflict with instruction fetches
- Jumps
- Subroutine Calls and Returns
- Interrupts and Returns
- Loops

3.1.2 Program Sequencer Architecture

Figure 3.3, on the next page, shows a block diagram of the program sequencer. The sequencer selects the value of the next fetch address from several possible sources.

The fetch address register, decode address register and program counter (PC) contain, respectively, the addresses of the instructions currently being fetched, decoded and executed. The PC is coupled with the PC stack, which is used to store return addresses and top-of-loop addresses.

3 Program Sequencing

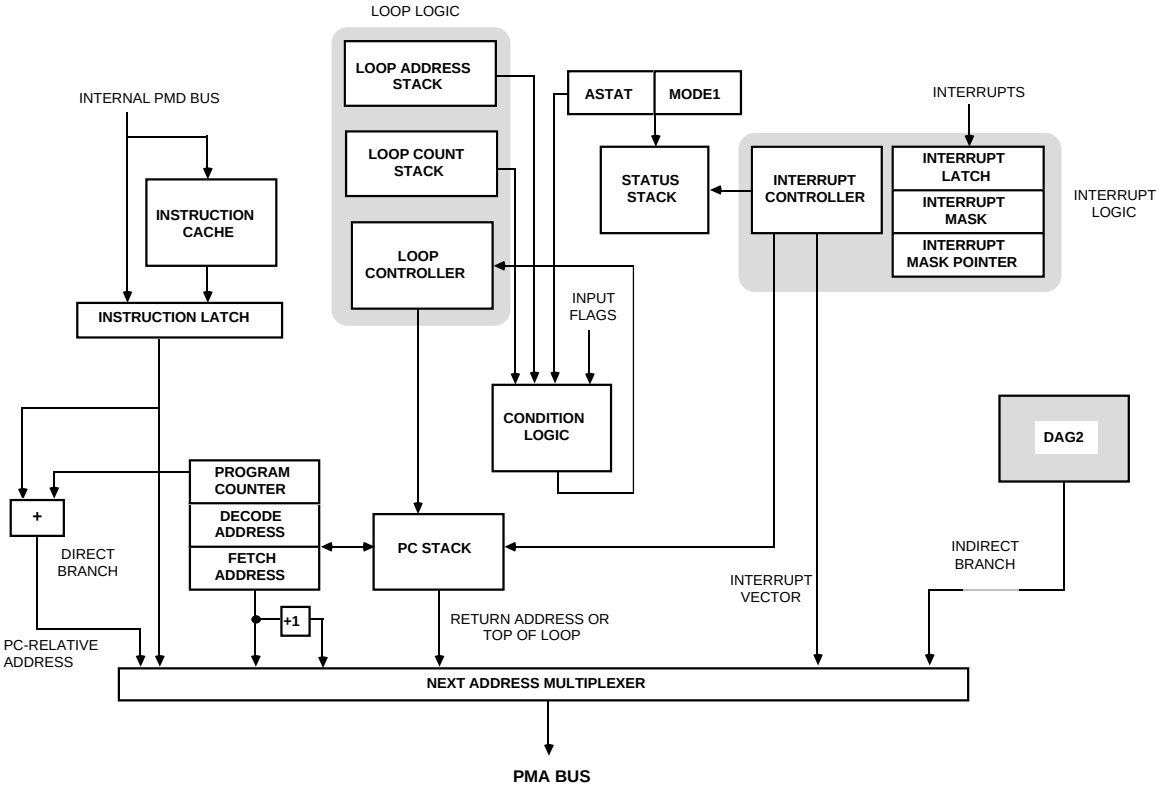


Figure 3.3 Program Sequencer Block Diagram

The interrupt controller performs all functions related to interrupt processing, such as determining whether an interrupt is masked and generating the appropriate interrupt vector address.

The instruction cache provides the means by which the ADSP-2106x can access data in program memory and fetch an instruction (from the cache) in the same cycle. The DAG2 data address generator (described in the next chapter) outputs program memory data addresses.

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

3.1.2.1 Program Sequencer Registers & System Registers

Table 3.1 lists the registers located in the program sequencer. The functions of these registers are described in subsequent sections of this chapter. All registers in the program sequencer are universal registers and are thus accessible to other universal registers as well as to data memory. All registers and the tops of stacks are readable; all registers except the fetch address, decode address and PC are writeable. The PC stack can be pushed and popped by writing the PC stack pointer, which is readable and writeable. The loop address stack and status stack are pushed and popped by explicit instructions.

The *System Register Bit Manipulation* instruction can be used to set, clear, toggle or test specific bits in the system registers. This instruction is described in Appendix A, Group IV–Miscellaneous Instructions.

Due to pipelining, writes to some of these registers do not take effect on the next cycle; for example, if you write the MODE1 register to enable ALU saturation mode, the change will not occur until two cycles after the write. Also, some registers are not updated on the cycle immediately following a write; it takes an extra cycle before a read of the register yields the new value. Table 3.1 summarizes the number of extra cycles for a write to take effect (*effect latency*) and for a new value to appear in the register (*read latency*). A "0" indicates that the write takes effect or appears in the register on the next cycle after the write instruction is executed. A "1" indicates one extra cycle.

Program Sequencer Registers	*Contents*	*Bits*	*Read Latency*	*Effect Latency*
FADDR*	fetch address	24	–	–
DADDR*	decode address	24	–	–
PC*	execute address	24	–	–
PCSTK	top of PC stack	24	0	0
PCSTKP	PC stack pointer	5	1	1
LADDR	top of loop address stack	32	0	0
CURLCNTR	top of loop count stack (current loop count)	32	0	0
LCNTR	loop count for next DO UNTIL loop	32	0	0
System Registers				
MODE1	mode control bits	32	0	1
MODE2	mode control bits	32	0	1
IRPTL	interrupt latch	32	0	1
IMASK	interrupt mask	32	0	1
IMASKP	interrupt mask pointer (for nesting)	32	1	1
ASTAT	arithmetic status flags	32	0	1
STKY	sticky status flags	32	0	1
USTAT1	user-defined status flags	32	0	0
USTAT2	user-defined status flags	32	0	0

Table 3.1 Program Sequencer Registers & System Registers

* read-only

3 Program Sequencing

3.2 PROGRAM SEQUENCER OPERATIONS

This section gives an overview of the operation of the program sequencer. The various kinds of program flow are defined here and described in detail in subsequent sections.

3.2.1 Sequential Instruction Flow

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 ADSP-2106x executes instructions from program memory in sequential order by simply incrementing the fetch address.

3.2.2 Program Memory Data Accesses

Usually, the ADSP-2106x fetches an instruction from memory on each cycle. When the ADSP-2106x executes an instruction which requires data to be read from or written to the same memory block in which the instruction is stored, there is a conflict for access to that block. The ADSP-2106x uses its instruction cache to reduce delays caused by this type of conflict.

The first time the ADSP-2106x encounters an instruction fetch that conflicts with a program memory data access, it must wait to fetch the instruction on the following cycle, causing a delay. The ADSP-2106x automatically writes the fetched instruction to the cache to prevent the same delay from happening again. The ADSP-2106x checks the instruction cache on every program memory data access. If the instruction needed is in the cache, the instruction fetch from the cache happens in parallel with the program memory data access, without incurring a delay.

3.2.3 Branches

A branch occurs when the fetch address is not the next sequential address following the previous fetch address. Jumps, calls and returns are the types of branches which the ADSP-2106x supports. In the program sequencer, the only difference between a jump and a call is that upon execution of a call, a return address is pushed onto the PC stack so that it is available when a return instruction is later executed. Jumps branch to a new location without allowing return.

3.2.4 Loops

The ADSP-2106x supports program loops with the DO UNTIL instruction. The DO UNTIL instruction causes the ADSP-2106x to repeat a sequence of instructions until a specified condition tests true.

3.3 CONDITIONAL INSTRUCTION EXECUTION

The program sequencer evaluates conditions to determine whether to execute a conditional instruction and when to terminate a loop. The conditions are based on information from the arithmetic status (ASTAT) register, mode control 1 (MODE1) register, flag inputs and loop counter. The arithmetic ASTAT bits are described in the previous chapter, *Computation Units*.

Each condition that the ADSP-2106x evaluates has an assembler mnemonic and a unique code which is used in a conditional instruction's opcode. For most conditions, the program sequencer can test both true and false states, e.g., equal to zero and not equal to zero. Table 3.2, on the following page, defines the 32 condition and termination codes.

The bit test flag (BTF) is bit 18 of the ASTAT register. This flag is set (or cleared) by the results of the BIT TST and BIT XOR forms of the *System Register Bit Manipulation* instruction, which can be used to test the contents of the ADSP-2106x's system registers. This instruction is described in Appendix A, Group IV–Miscellaneous instructions. After BTF is set by this instruction, it can be used as the condition in a conditional instruction (with the mnemonic TF; see Table 3.2).

The two conditions that do not have complements are LCE / NOT LCE (loop counter expired / not expired) and TRUE / FOREVER. The interpretation of these condition codes is determined by context; TRUE and NOT LCE are used in conditional instructions, FOREVER and LCE in loop termination. The IF TRUE construct creates an unconditional instruction (the same effect as leaving out the condition entirely). A DO FOREVER instruction executes a loop indefinitely, until an interrupt or reset intervenes.

The LCE condition (loop counter expired) is most commonly used in a DO UNTIL instruction. Because the LCE condition checks the value of the loop counter (CURLCNTR), an IF NOT LCE conditional instruction should not follow a write to CURLCNTR from memory. Otherwise, because the write occurs after the NOT LCE test, the condition is based on the old CURLCNTR value.

The bus master condition (BM) indicates whether the ADSP-2106x is the current bus master in a multiprocessor system. To enable the use of this condition, bits 17 and 18 of the MODE1 register must both be zeros; otherwise the condition is always evaluated as false.

No.	*Mnemonic*	*Description*	*True If*
0	EQ	ALU equal zero	AZ = 1
1	LT	ALU less than zero	*See Note 1 below*
2	LE	ALU less than or equal zero	*See Note 2 below*
3	AC	ALU carry	AC = 1
4	AV	ALU overflow	AV = 1
5	MV	Multiplier overflow	MV = 1
6	MS	Multiplier sign	MN = 1
7	SV	Shifter overflow	SV = 1
8	SZ	Shifter zero	SZ = 1
9	FLAG0_IN	Flag 0 input	FI0 = 1
10	FLAG1_IN	Flag 1 input	FI1 = 1
11	FLAG2_IN	Flag 2 input	FI2 = 1
12	FLAG3_IN	Flag 3 input	FI3 = 1
13	TF	Bit test flag	BTF = 1
14	BM	Bus Master	
15	LCE	Loop counter expired (DO UNTIL term)	CURLCNTR = 1
15	NOT LCE	Loop counter not expired (IF cond)	CURLCNTR ≠ 1
Bits 16-30 are the complements of bits 0-14			
16	NE	ALU not equal to zero	AZ = 0
17	GE	ALU greater than or equal zero	*See Note 3 below*
18	GT	ALU greater than zero	*See Note 4 below*
19	NOT AC	Not ALU carry	AC = 0
20	NOT AV	Not ALU overflow	AV = 0
21	NOT MV	Not multiplier overflow	MV = 0
22	NOT MS	Not multiplier sign	MN = 0
23	NOT SV	Not shifter overflow	SV = 0
24	NOT SZ	Not shifter zero	SZ = 0
25	NOT FLAG0_IN	Not Flag 0 input	FI0 = 0
26	NOT FLAG1_IN	Not Flag 1 input	FI1 = 0
27	NOT FLAG2_IN	Not Flag 2 input	FI2 = 0
28	NOT FLAG3_IN	Not Flag 3 input	FI3 = 0
29	NOT TF	Not bit test flag	BTF = 0
30	NBM	Not Bus Master	
31	FOREVER	Always False (DO UNTIL)	always
31	TRUE	Always True (IF)	always

Table 3.2 Condition & Loop Termination Codes

Notes:
1. [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN and $\overline{AZ}$)] = 1
2. [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN)] or AZ = 1
3. [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN and $\overline{AZ}$)] = 0
4. [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN)] or AZ = 0

3.4 BRANCHES (CALL, JUMP, RTS, RTI)

The CALL instruction initiates a subroutine. Both jumps and calls transfer program flow to another memory location, but a call also pushes a return address onto the PC stack so that it is available when a return from subroutine instruction is later executed. Jumps branch to a new location without allowing return.

A return causes the processor to branch to the address stored at the top of the PC stack. There are two types of returns: return from subroutine (RTS) and return from interrupt (RTI). The difference between the two is that the RTI instruction not only pops the return address off the PC stack, but also: 1) pops the status stack if the ASTAT and MODE1 status registers have been pushed (if the interrupt was $\overline{IRQ}_{2\text{-}0}$, the timer interrupt, or the VIRPT vector interrupt), and 2) clears the appropriate bit in the interrupt latch register (IRPTL) and the interrupt mask pointer (IMASKP).

There are a number of parameters you can specify for branches:

- Jumps, calls and returns can be conditional. The program sequencer can evaluate any one of several status conditions to decide whether the branch should be taken. If no condition is specified, the branch is always taken.

- Jumps and calls can be indirect, direct, or PC-relative. An *indirect* branch goes to an address supplied by one of the data address generators, DAG2. *Direct* branches jump to the 24-bit address specified in an immediate field in the branch instruction. *PC-relative* branches also use a value specified in the instruction, but the sequencer adds this value to the current PC value to compute the destination address.

- Jumps, calls and returns can be delayed or nondelayed. In a *delayed* branch, the two instructions immediately after the branch instruction are executed; in a *nondelayed* branch, the program sequencer suppresses the execution of those two instructions (NOPs are performed instead).

- The JUMP (LA) instruction causes an automatic loop abort if it occurs inside a loop. When the loop is aborted, the PC and loop address stacks are popped once, so that if the loop was nested, the stacks still contain the correct values for the outer loop. JUMP (LA) is similar to the *break* instruction of the C programming language used to prematurely terminate execution of a loop. (Note: JUMP (LA) may not be used in the last three instructions of a loop.)

3.4.1 Delayed & Nondelayed Branches

An instruction modifier (DB) indicates that a branch is delayed; otherwise, it is nondelayed. If the branch is nondelayed, the two instructions after the branch, which are in the fetch and decode stages, are not executed (see Figure 3.4); for a call, the decode address (the address of the instruction after the call) is the return address. During the two no-operation cycles, the first instruction at the branch address is fetched and decoded.

NON-DELAYED JUMP OR CALL

CLOCK CYCLES ⟶

Execute Instruction	**n**	**nop**	**nop**	**j**
Decode Instruction	**n+1->nop**	**n+2->nop**	**j**	**j+1**
Fetch Instruction	**n+2**	**j**	**j+1**	**j+2**
	n+1 suppressed	**n+2 suppressed; for call, n+1 pushed on PC stack**		

NON-DELAYED RETURN

CLOCK CYCLES ⟶

Execute Instruction	**n**	**nop**	**nop**	**r**
Decode Instruction	**n+1->nop**	**n+2->nop**	**r**	**r+1**
Fetch Instruction	**n+2**	**r**	**r+1**	**r+2**
	n+1 suppressed	**n+2 suppressed; r popped from PC stack**		

n = Branch instruction

j = Instruction at Jump or Call address

r = Instruction at Return address

Figure 3.4 Nondelayed Branches

In a delayed branch, the processor continues to execute two more instructions while the instruction at the branch address is fetched and decoded (see Figure 3.5); in the case of a call, the return address is the third address after the branch instruction. A delayed branch is more efficient, but it makes the code harder to understand because of the instructions between the branch instruction and the actual branch.

DELAYED JUMP OR CALL

CLOCK CYCLES →

Execute Instruction	**n**	**n+1**	**n+2**	**j**
Decode Instruction	**n+1**	**n+2**	**j**	**j+1**
Fetch Instruction	**n+2**	**j**	**j+1**	**j+2**
		for call, n+3 pushed on PC stack		

DELAYED RETURN

CLOCK CYCLES →

Execute Instruction	**n**	**n+1**	**n+2**	**r**
Decode Instruction	**n+1**	**n+2**	**r**	**r+1**
Fetch Instruction	**n+2**	**r**	**r+1**	**r+2**
		r popped from PC stack		

n = Branch instruction

j = Instruction at Jump or Call address

r = Instruction at Return address

Figure 3.5 Delayed Branches

Because of the instruction pipeline, a delayed branch instruction and the two instructions that follow it must be executed sequentially. Instructions in the two locations immediately following a delayed branch instruction may not be any of the following:

- Other Jumps, Calls or Returns
- Pushes or Pops of the PC stack
- Writes to the PC stack or PC stack pointer
- DO UNTIL instruction
- IDLE or IDLE16 instruction

These exceptions are checked by the ADSP-21000 Family assembler.

The ADSP-2106x does not process an interrupt in between a delayed branch instruction and either of the two instructions that follow, since these three instructions must be executed sequentially. Any interrupt that occurs during these instructions is latched but not processed until the branch is complete.

A read of the PC stack or PC stack pointer immediately after a delayed call or return is permitted, but it will show that the return address on the PC stack has already been pushed or popped, even though the branch has not occurred yet.

3.4.2 PC Stack

The PC stack holds return addresses for subroutines and interrupt service routines and top-of-loop addresses for loops. The PC stack is 30 locations deep by 24 bits wide.

The PC stack is popped during returns from interrupts (RTI), returns from subroutines (RTS) and terminations of loops. The stack is full when all entries are occupied, empty when no entries are occupied, and overflowed if a call occurs when the stack is already full. The full and empty flags are stored in the sticky status register (STKY). The full flag causes a maskable interrupt.

A PC stack interrupt occurs when 29 locations of the PC stack are filled (the *almost full* state). Entering the interrupt service routine then immediately causes a push on the PC stack, making it full. Thus the interrupt is a *stack full* interrupt, even though the condition that triggers it is the *almost full* condition. The other stacks in the sequencer, the loop address stack, loop counter stack and status stack, are provided with overflow interrupts that are activated when a push occurs while the stack is in a full state.

The program counter stack pointer (PCSTKP) is a readable and writeable register that contains the address of the top of the PC stack. The value of PCSTKP is zero when the PC stack is empty, 1, 2, ..., 30 when the stack contains data, and 31 when the stack is overflowed. A write to PCSTKP takes effect after a one-cycle delay. If the PC stack is overflowed, a write to PCSTKP has no effect.

3.5 LOOPS (DO UNTIL)

The DO UNTIL instruction provides for efficient software loops, without the overhead of additional instructions to branch, test a condition, or decrement a counter. Here is a simple example of an ADSP-2106x loop:

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

When the ADSP-2106x executes a DO UNTIL instruction, the program sequencer pushes the address of the last loop instruction and the termination condition for exiting the loop (both specified in the instruction) onto the loop address stack. It also pushes the top-of-loop address, which is the address of the instruction following the DO UNTIL instruction, on the PC stack.

Because of the instruction pipeline (fetch, decode and execute cycles), the processor tests the termination condition (and, if the loop is counter-based, decrements the counter) before the end of the loop so that the next fetch either exits the loop or returns to the top based on the test condition. Specifically, the condition is tested when the instruction two locations before the last instruction in the loop (at location $e - 2$, where e is the end-of-loop address) is executed. If the termination condition is not satisfied, the processor fetches the instruction from the top-of-loop address stored on the top of the PC stack. If the termination condition is true, the sequencer fetches the next instruction after the end of the loop and pops the loop stack and PC stack. Loop operation is shown in Figure 3.6, on the next page.

LOOP-BACK

CLOCK CYCLES ⟶

Execute Instruction	**e-2**	**e-1**	**e**	**b**
Decode Instruction	**e-1**	**e**	**b**	**b+1**
Fetch Instruction	**e**	**b**	**b+1**	**b+2**
	termination condition tests false	**loop start address is top of PC stack**		

LOOP TERMINATION

CLOCK CYCLES ⟶

Execute Instruction	**e-2**	**e-1**	**e**	**e+1**
Decode Instruction	**e-1**	**e**	**e+1**	**e+2**
Fetch Instruction	**e**	**e+1**	**e+2**	**e+3**
	termination condition tests true	**loop-back aborts; PC and loop stacks popped**		

e = Loop end instruction
b = Loop start instruction

Figure 3.6 Loop Operation

3.5.1 Restrictions & Short Loops

This section describes several programming restrictions for loops. It also explains restrictions applying to short (one- and two-instruction) loops, which require special consideration because of the three-instruction fetch-decode-execute pipeline.

3.5.1.1 General Restrictions

- Nested loops cannot terminate on the same instruction.

- The last three instructions of a loop cannot be any branch (jump, call, or return); otherwise, the loop may not be executed correctly. This also applies to one-instruction loops and two-instruction loops with only one iteration. There is one exception to this rule, a non-delayed CALL (no DB modifier) paired with an RTS (LR), return from subroutine with loop reentry modifier. The non-delayed CALL may be used as one of the last three instructions of a loop (but not in a one-instruction loop or a two-instruction, single-iteration loop.)

  The RTS (LR) instruction ensures proper reentry into a loop. In counter-based loops, for example, the termination condition is checked by decrementing the current loop counter (CURLCNTR) during execution of the instruction two locations before the end of the loop. A non-delayed call may then be used in one of the last two locations, providing an RTS (LR) instruction is used to return from the subroutine. The loop reentry (LR) modifier assures proper reentry into the loop, by preventing the loop counter from being decremented again (i.e. twice for the same loop iteration).

3.5.1.2 Counter-Based Loops

The third-to-last instruction of a counter-based loop (at $e - 2$, where e is the end-of-loop address) cannot be a write to the counter from memory.

Short loops terminate in a special way because of the instruction (fetch-decode-execute) pipeline. Counter-based loops of one or two instructions are not long enough for the sequencer to check the termination condition two instructions from the end of the loop. In these short loops, the sequencer has already looped back when the termination condition is tested. The sequencer provides special handling to avoid overhead (NOP) cycles if the loop is iterated a minimum number of times. The detailed operation is shown in Figures 3.7 and 3.8 (on the following page). For no overhead, a loop of length one must be executed at least three times and a loop of length two must be executed at least twice.

Loops of length one that iterate only once or twice and loops of length two that iterate only once incur two cycles of overhead because there are two aborted instructions after the last iteration to clear the instruction pipeline.

Processing of an interrupt that occurs during the last iteration of a one-instruction loop that executes once or twice, a two-instruction loop that executes once, or the cycle following one of these loops (which is a NOP) is delayed by one cycle. Similarly, in a one-instruction loop that iterates at least three times, processing is delayed by one cycle if the interrupt occurs during the third-to-last iteration.

3.5.1.3 Non-Counter-Based Loops

A non-counter-based loop is one in which the loop termination condition is something other than LCE. When a non-counter-based loop is the outer loop of a series of nested loops, the end address of the outer loop must be located at least two addresses after the end address of the inner loop.

The JUMP (LA) instruction is used to prematurely abort execution of a loop. When this instruction is located in the inner loop of a series of nested loops and the outer loop is non-counter-based, the address jumped to cannot be the last instruction of the outer loop. The address jumped to may, however, be the next-to-last instruction (or any earlier).

ONE-INSTRUCTION LOOP, THREE ITERATIONS

CLOCK CYCLES →

Execute Instruction	n	n+1 first iteration	n+1 second iteration	n+1 third iteration	n+2
Decode Instruction	n+1	n+1	n+1	n+2	n+3
Fetch Instruction	n+2	n+1	n+2	n+3	n+4
	LCNTR <– 3	opcode latch not updated; fetch address not updated; count expired tests true	loop-back aborts; PC & loop stacks popped		

ONE-INSTRUCTION LOOP, TWO ITERATIONS (Two Cycles of Overhead)

CLOCK CYCLES →

Execute Instruction	n	n+1 first iteration	n+1 second iteration	nop	nop	n+2
Decode Instruction	n+1	n+1	n+1 -> nop	n+1 –> nop	n+2	n+3
Fetch Instruction	n+2	n+1	n+1	n+2	n+3	n+4
	LCNTR <– 2	opcode latch not updated; fetch address not updated	count expired tests true	loop-back aborts; PC & loop stacks popped		

Figure 3.7 One-Instruction Counter-Based Loops

n = DO UNTIL instruction
n+2 = instruction after loop

Non-counter-based short loops terminate in a special way because of the *fetch-decode-execute* instruction pipeline:

- In a three-instruction loop, the termination condition is tested when the top of loop instruction is executed. When the condition becomes true, the sequencer completes one full pass of the loop before exiting.
- In a two-instruction loop, the termination condition is checked during the last (second) instruction. If the condition becomes true when the first instruction is executed, it tests true during the second and one more full pass is completed before exiting. If the condition becomes true during the second instruction, however, two more full passes occur before the loop exit.
- In a one-instruction loop, the termination condition is checked every cycle. When the condition becomes true, the loop executes three more times before exiting.

TWO-INSTRUCTION LOOP, TWO ITERATIONS

CLOCK CYCLES →

Execute Instruction	**n**	**n+1 first iteration**	**n+2 first iteration**	**n+1 second iteration**	**n+2 second iteration**	**n+3**
Decode Instruction	**n+1**	**n+2**	**n+1**	**n+2**	**n+3**	**n+4**
Fetch Instruction	**n+2**	**n+1**	**n+2**	**n+3**	**n+4**	**n+5**
	LCNTR <- 2	**PC stack supplies loop start address**	**last instruction fetched, causes condition test; tests true**	**loop-back aborts; PC & loop stacks popped**		

TWO-INSTRUCTION LOOP, ONE ITERATION (Two Cycles of Overhead)

CLOCK CYCLES →

Execute Instruction	**n**	**n+1 first iteration**	**n+2 first iteration**	**nop**	**nop**	**n+3**
Decode Instruction	**n+1**	**n+2**	**n+1->nop**	**n+2->nop**	**n+3**	**n+4**
Fetch Instruction	**n+2**	**n+1**	**n+2**	**n+3**	**n+4**	**n+5**
	LCNTR <- 1	**PC stack supplies loop start address**	**last instruction fetched, causes condition test; tests true**	**loop-back aborts; PC & loop stacks popped**		

Figure 3.8 Two-Instruction Counter-Based Loops

n = DO UNTIL instruction
n+3 = instruction after loop

3.5.2 Loop Address Stack

The loop address stack is six levels deep by 32 bits wide. The 32-bit word of each level consists of a 24-bit loop termination address, a 5-bit termination code, and a 2-bit loop type code:

Bits	*Value*
0-23	Loop termination address
24-28	Termination code
29	*reserved (always reads 0)*
30-31	Loop type code:

00	arithmetic condition-based (not LCE)
01	counter-based, length 1
10	counter-based, length 2
11	counter-based, length > 2

The loop termination address, termination code and loop type code are stacked when a DO UNTIL or PUSH LOOP instruction is executed. The stack is popped two instructions before the end of the last loop iteration or when a POP LOOP instruction is issued. A stack overflows if a push occurs when all entries in the loop stack are occupied. The stack is empty when no entries are occupied. The overflow and empty flags are in the sticky status register (STKY). Overflow causes a maskable interrupt.

The LADDR register contains the top of the loop address stack. It is readable and writeable over the DM Data bus. Reading and writing LADDR does not move the loop address stack pointer; a stack push or pop, performed with explicit instructions, moves the stack pointer. LADDR contains the value 0xFFFF FFFF when the loop address stack is empty.

Because the termination condition is checked two instructions before the end of the loop, the loop stack is popped before the end of the loop on the final iteration. If LADDR is read at either of these instructions, the value will no longer be the termination address for the loop.

A jump out of a loop pops the loop address stack (and the loop count stack if the loop is counter-based) if the Loop Abort (LA) modifier is specified for the jump. This allows the loop mechanism to continue to function correctly. Only one pop is performed, however, so the loop abort cannot be used to jump more than one level of loop nesting.

3.5.3 Loop Counters And Stack

The loop counter stack is six levels deep by 32 bits wide. The loop counter stack works in synchronization with the loop address stack; both stacks always have the same number of locations occupied. Thus, the same empty and overflow status flags apply to both stacks.

The ADSP-2106x program sequencer operates two separate loop counters: the current loop counter (CURLCNTR), which tracks iterations for a loop being executed, and the loop counter (LCNTR), which holds the count value before the loop is executed. Two counters are needed to maintain the count for an outer loop while setting up the count for an inner loop.

3.5.3.1 CURLCNTR

The top entry in the loop counter stack always contains the loop count currently in effect. This entry is the CURLCNTR register, which is readable and writeable over the DM Data bus. A read of CURLCNTR when the loop counter stack is empty gives the value 0xFFFF FFFF.

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

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

Writing CURLCNTR does not cause a stack push. Thus, if you write a new value to CURLCNTR, you change the count value of the loop currently executing. A write to CURLCNTR when no DO UNTIL LCE loop is executing has no effect.

Because the processor must use CURLCNTR to perform counter-based loops, there are some restrictions on when you can write CURLCNTR. As mentioned under “Loop Restrictions,” the third-to-last instruction of a DO UNTIL LCE loop cannot be a write to CURLCNTR from memory. The instruction that follows a write to CURLCNTR from memory cannot be an IF NOT LCE instruction.

3.5.3.2 LCNTR

LCNTR is the value of the top of the loop counter stack *plus one*, i.e., it is the location on the stack which will take effect on the next loop stack push. To set up a count value for a nested loop without affecting the count value of the loop currently executing, you write the count value to LCNTR. A value of zero in LCNTR causes a loop to execute 2^{32} times.

The DO UNTIL LCE instruction pushes the value of LCNTR on the loop count stack, so that it becomes the new CURLCNTR value. This process is illustrated in Figure 3.9. The previous CURLCNTR value is preserved one location down in the stack.

A read of LCNTR when the loop counter stack is full results in invalid data. When the loop counter stack is full, any data written to LCNTR is discarded.

If you read LCNTR during the last two instructions of a terminating loop, its value is the last CURLCNTR value for the loop.

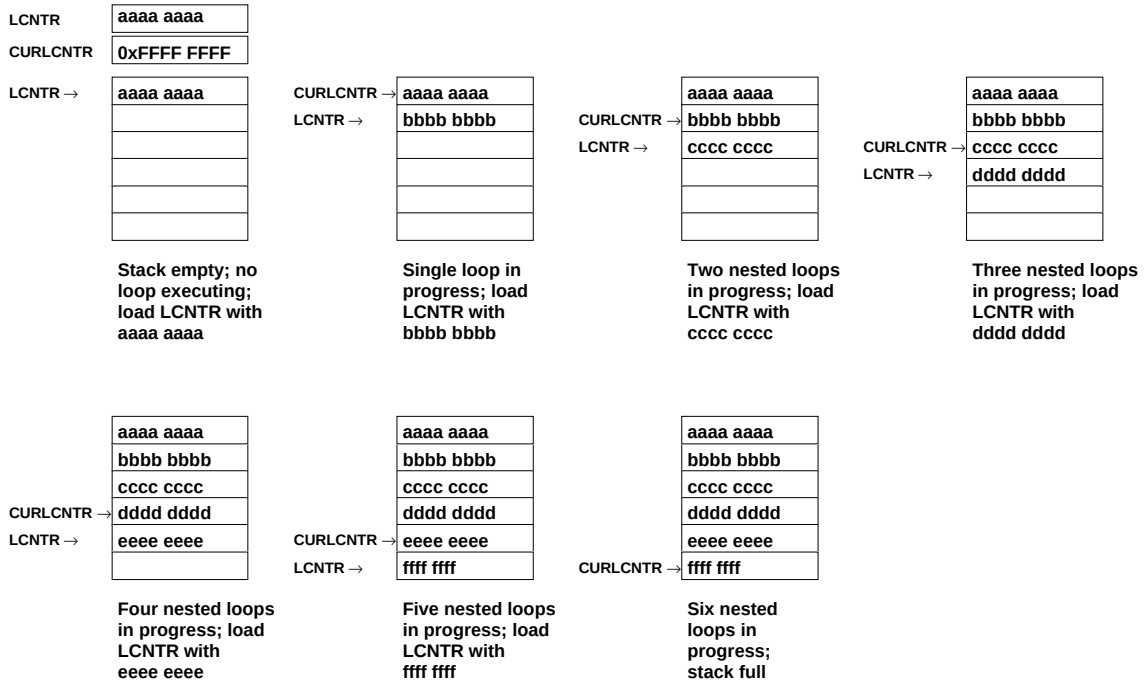


Figure 3.9 Pushing The Loop Counter Stack For Nested Loops

3.6 INTERRUPTS

Interrupts are caused by a variety of conditions, both internal and external to the processor. An interrupt forces a subroutine call to a predefined address, the interrupt vector. The ADSP-2106x assigns a unique vector to each type of interrupt.

Externally, the ADSP-2106x supports three prioritized, individually maskable interrupts, each of which can be either level or edge-triggered. These interrupts are caused by an external device asserting one of the ADSP-2106x's interrupt inputs ($\overline{IRQ}_{2-0}$). Among the internally generated interrupts are arithmetic exceptions, stack overflows, and circular data buffer overflows.

An interrupt request is deemed valid if it is not masked, if interrupts are globally enabled (if bit 12 in MODE1 is set), and if a higher priority request is not pending. Valid requests invoke an interrupt service sequence that branches to the address reserved for that interrupt. Interrupt vectors are spaced at 8-instruction intervals; longer service routines can be accommodated by branching to another region of memory. Program execution returns to normal sequencing when an RTI (return from interrupt) instruction is executed.

The ADSP-2106x core processor cannot service an interrupt unless it is executing instructions or is in the IDLE state. IDLE and IDLE16 are a special instructions that halt the processor core until an external interrupt or the timer interrupt occurs.

To process an interrupt, the ADSP-2106x's program sequencer performs the following actions:

1. Outputs the appropriate interrupt vector address.

2. Pushes the current PC value (the return address) on the PC stack.

3. If the interrupt is either an external interrupt ($\overline{IRQ}_{2-0}$), the internal timer interrupt, or the VIRPT multiprocessor vector interrupt, the program sequencer pushes the current value of the ASTAT and MODE1 registers onto the status stack.

4. Sets the appropriate bit in the interrupt latch register (IRPTL).

5. Alters the interrupt mask pointer (IMASKP) to reflect the current interrupt nesting state. The nesting mode (NESTM) bit in the MODE1 register determines whether all interrupts or only lower priority interrupts are masked during the service routine.

At the end of the interrupt service routine, the RTI instruction causes the following actions:

1. Returns to the address stored at the top of the PC stack.

2. Pops this value off of the PC stack.

3. Pops the status stack if the ASTAT and MODE1 status registers were pushed (for the $\overline{IRQ}_{2-0}$ external interrupts, timer interrupt, or VIRPT vector interrupt).

4. Clears the appropriate bit in the interrupt latch register (IRPTL) and interrupt mask pointer (IMASKP).

All interrupt service routines, except for reset, should end with a return-from-interrupt (RTI) instruction. After reset, the PC stack is empty, so there is no return address—the last instruction of the reset service routine should be a jump to the start of your program.

3.6.1 Interrupt Latency

The ADSP-2106x responds to interrupts in three stages: synchronization and latching (1 cycle), recognition (1 cycle), and branching to the interrupt vector (2 cycles). See Figure 3.10. If an interrupt is forced in software by a write to a bit in IRPTL, it is recognized in the following cycle, and the two cycles of branching to the interrupt vector follow that.

For most interrupts, internal and external, only one instruction is executed after the interrupt occurs (and before the two instructions aborted) while the processor fetches and decodes the first instruction of the service routine. Because of the one-cycle delay between an arithmetic exception and the STKY register update, however, there are two cycles after an arithmetic exception occurs before interrupt processing starts.

The standard latency associated with the $\overline{IRQ}_{2-0}$ interrupts and the multiprocessor vector interrupt are:

Interrupt	*Latency (minimum)*
$\overline{IRQ}_{2-0}$ interrupts	3 cycles
Multiprocessor vector interrupt (VIRPT register)	6 cycles

INTERRUPT, SINGLE-CYCLE INSTRUCTION n = Single-cycle instruction

CLOCK CYCLES →

Execute Instruction	n-1	n	nop	nop	v
Decode Instruction	n	n+1->nop	n+2->nop	v	v+1
Fetch Instruction	n+1	n+2	v	v+1	v+2
	interrupt occurs	interrupt recognized	n+1 pushed onto PC stack; interrupt vector output		

INTERRUPT, PROGRAM MEMORY DATA ACCESS WITH CACHE MISS n = Instruction coinciding with program memory data access, cache miss

CLOCK CYCLES →

Execute Instruction	n-1	n	nop	nop	nop	v
Decode Instruction	n	n+1->nop	n+1->nop	n+2->nop	v	v+1
Fetch Instruction	n+1	-	n+2	v	v+1	v+2
	interrupt occurs	interrupt recognized, but not processed; program memory data access	interrupt processed	n+1 pushed onto PC stack; interrupt vector output		

INTERRUPT, DELAYED BRANCH n = Delayed branch instruction

CLOCK CYCLES →

Execute Instruction	n-1	n	n+1	n+2	nop	nop	v
Decode Instruction	n	n+1	n+2	j->nop	j+1 ->nop	v	v+1
Fetch Instruction	n+1	n+2	j	j + 1	v	v+1	v+2
	interrupt occurs	interrupt recognized, but not processed		for a call, n+3 pushed onto PC stack; interrupt processed	j pushed onto PC stack; interrupt vector output		

Figure 3.10 Interrupt Handling

v = instruction at interrupt vector
j = instruction at branch address

If nesting is enabled and a higher priority interrupt occurs immediately after a lower priority interrupt, the service routine of the higher priority interrupt is delayed by one additional cycle. (See “Interrupt Nesting & IMASKP”.) This allows the first instruction of the lower priority interrupt routine to be executed before it is interrupted.

Certain ADSP-2106x operations that span more than one cycle will hold off interrupt processing. If an interrupt occurs during one of these operations, it is synchronized and latched, but its processing is delayed. The operations that delay interrupt processing in this way are as follows:

- a branch (call, jump, or return) and the following cycle, whether it is an instruction (in a delayed branch) or a NOP (in a non-delayed branch)
- the first of the two cycles needed to perform a program memory data access and an instruction fetch (when there is an instruction cache miss).
- the third-to-last iteration of a one-instruction loop
- the last iteration of a one-instruction loop executed once or twice or of a two-instruction loop executed once, and the following cycle (which is a NOP)
- the first of the two cycles needed to fetch and decode the first instruction of an interrupt service routine
- waitstates for external memory accesses
- when an external memory access is required and the ADSP-2106x does not have control of the external bus (during a host bus grant or when the ADSP-2106x is a bus slave in a multiprocessing system)

3.6.2 Interrupt Vector Table

Table 3.3 shows all ADSP-2106x interrupts, listed according their bit position in the IRPTL and IMASK registers (see “Interrupt Latch Register”). Also shown is the address of the interrupt vector; each vector is separated by eight memory locations. The addresses in the vector table represent offsets from a base address. For an interrupt vector table in internal memory, the base address is 0x0002 0000; for an interrupt vector table in external memory, the base address is 0x0040 0000. The third column in Table 3.3 lists a mnemonic name for each interrupt. These names are provided for convenience, and are not required by the assembler.

IRPTL/ IMASK Bit #	*Vector Address**	*Interrupt Name***	*Function*	
0	0x00	–	*reserved*	
1	0x04	RSTI	Reset (read-only, non-maskable)	**HIGHEST PRIORITY**
2	0x08	–	*reserved*	
3	0x0C	SOVFI	Status stack or loop stack overflow or PC stack full	
4	0x10	TMZHI	Timer=0 (high priority option)	
5	0x14	VIRPTI	Vector Interrupt	
6	0x18	IRQ2I	$\overline{\text{IRQ2}}$ asserted	
7	0x1C	IRQ1I	$\overline{\text{IRQ1}}$ asserted	
8	0x20	IRQ0I	$\overline{\text{IRQ0}}$ asserted	
9	0x24	–	*reserved*	
10	0x28	SPR0I	DMA Channel 0 – SPORT0 Receive	
11	0x2C	SPR1I	DMA Channel 1 – SPORT1 Receive (or Link Buffer 0)	
12	0x30	SPT0I	DMA Channel 2 – SPORT0 Transmit	
13	0x34	SPT1I	DMA Channel 3 – SPORT1 Transmit (or Link Buffer 1)	
14	0x38	LP2I	DMA Channel 4 – Link Buffer 2	
15	0x3C	LP3I	DMA Channel 5 – Link Buffer 3	
16	0x40	EP0I	DMA Channel 6 – Ext. Port Buffer 0 (or Link Buffer 4)	
17	0x44	EP1I	DMA Channel 7 – Ext. Port Buffer 1 (or Link Buffer 5)	
18	0x48	EP2I	DMA Channel 8 – Ext. Port Buffer 2	
19	0x4C	EP3I	DMA Channel 9 – Ext. Port Buffer 3	
20	0x50	LSRQ	Link Port Service Request	
21	0x54	CB7I	Circular Buffer 7 overflow	
22	0x58	CB15I	Circular Buffer 15 overflow	
23	0x5C	TMZLI	Timer=0 (low priority option)	
24	0x60	FIXI	Fixed-point overflow	
25	0x64	FLTOI	Floating-point overflow exception	
26	0x68	FLTUI	Floating-point underflow exception	
27	0x6C	FLTII	Floating-point invalid exception	
28	0x70	SFT0I	User software interrupt 0	
29	0x74	SFT1I	User software interrupt 1	
30	0x78	SFT2I	User software interrupt 2	
31	0x7C	SFT3I	User software interrupt 3	**LOWEST PRIORITY**

Table 3.3 Interrupt Vectors & Priority

* Offset from base address: 0x0002 0000 for interrupt vector table in internal memory, 0x0040 0000 for interrupt vector table in external memory

** These IRPTL / IMASK bit names are defined in the `def21060.h` include file supplied with the ADSP-21000 Family Development Software.

The interrupt vector table may be located in internal memory, at address 0x0002 0000 (the beginning of Block 0), or in external memory at address 0x0040 0000. If the ADSP-2106x's on-chip memory is booted from an external source, the interrupt vector table will be located in internal memory. If, however, the ADSP-2106x is not booted (because it will execute from off-chip memory), the vector table must be located in the off-chip memory. See "Booting" in the *System Design* chapter for details on booting mode selection.

Also, if booting is from an external EPROM or host processor, bit 16 of IMASK (the EP0I interrupt for external port DMA Channel 6) will automatically be set to 1 following reset—this enables the *DMA done* interrupt for booting on Channel 6. IRPTL is initialized to all zeros following reset.

The IIVT bit in the SYSCON control register can be used to override the booting mode in determining where the interrupt vector table is located. If the ADSP-2106x is not booted (*no boot* mode), setting IIVT to 1 selects an internal vector table while IIVT=0 selects an external vector table. If the ADSP-2106x is booted from an external source (any mode other than *no boot* mode), then IIVT has no effect.

3.6.3 Interrupt Latch Register (IRPTL)

The interrupt latch (IRPTL) register is a 32-bit register that latches interrupts. It indicates all interrupts currently being serviced as well as any which are pending. Because this register is readable and writeable, any interrupt (except reset) can be set or cleared in software. Do not write to the reset bit (bit 1) in IRPTL because this puts the processor into an illegal state.

When an interrupt occurs, the corresponding bit in IRPTL is set. During execution of the interrupt's service routine, this bit is kept cleared—the ADSP-2106x clears the bit during every cycle, preventing the same interrupt from being latched while its service routine is already executing.

A special method is provided, however, to allow the reuse of an interrupt while it is being serviced. This method is provided by the clear interrupt (CI) modifier of the JUMP instruction. See Section 3.6.8, "Clearing The Current Interrupt For Reuse."

IRPTL is cleared by a processor reset.

(Note: The bits in the IMASK register correspond exactly to those in IRPTL.)

3.6.4 Interrupt Priority

The interrupt bits in IRPTL are ordered by priority. The interrupt priority is from 0 (highest) to 31 (lowest). Interrupt priority determines which interrupt is serviced first when more than one occurs in the same cycle. It also determines which interrupts are nested when nesting is enabled (see “Interrupt Nesting and IMASKP”).

The arithmetic interrupts—fixed-point overflow and floating-point overflow, underflow, and invalid operation—are determined from flags in the sticky status register (STKY). By reading these flags, the service routine for one of these interrupts can determine which condition caused the interrupt. The routine also has to clear the appropriate STKY bit so that the interrupt is not still active after the service routine is done.

The timer decrementing to zero causes both interrupt 4 and interrupt 14. This feature allows you to choose the priority of the timer interrupt. Unmask the timer interrupt that has the priority you want, and leave the other one masked. Unmasking both interrupts results in two interrupts when the timer reaches zero. In this case the processor services the higher priority interrupt first, then the lower priority interrupt.

3.6.5 Interrupt Masking & Control

All interrupts except for reset can be enabled and disabled by the global interrupt enable bit, IRPTEN, bit 12 in the MODE1 register. This bit is cleared at reset. You must set this bit for interrupts to be enabled.

3.6.5.1 Interrupt Mask Register (IMASK)

All interrupts except for reset can be masked. Masked means the interrupt is disabled. Interrupts that are masked are still latched (in IRPTL), so that if the interrupt is later unmasked, it is processed.

The IMASK register controls interrupt masking. The bits in IMASK correspond exactly to the bits in the IRPTL register. For example, bit 10 in IMASK masks or unmasks the same interrupt latched by bit 10 in IRPTL.

– *If a bit in IMASK is set to 1, its interrupt is unmasked (enabled).*
– *If the bit is cleared (to 0), the interrupt is masked (disabled).*

After reset, all interrupts except for the reset interrupt and the EP0I interrupt for external port DMA Channel 6 (bit 16 of IMASK) are masked. The reset interrupt is always non-maskable. The EP0I interrupt is automatically unmasked after reset if the ADSP-2106x is booting from EPROM or from a host.

3.6.5.2 Interrupt Nesting & IMASKP

The ADSP-2106x supports the nesting of one interrupt service routine inside another; that is, a service routine can be interrupted by a higher priority interrupt. This feature is controlled by the nesting mode bit (NESTM) in the MODE1 register.

When the NESTM bit is a 0, an interrupt service routine cannot be interrupted; any interrupt that occurs will be processed only after the routine finishes. When NESTM is a 1, higher priority interrupts can interrupt if they are not masked; lower or equal priority interrupts cannot. The NESTM bit should only be changed outside of an interrupt service routine or during the reset service routine; otherwise, interrupt nesting may not work correctly.

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

In nesting mode, the ADSP-2106x uses the interrupt mask pointer (IMASKP) to create a temporary interrupt mask for each level of interrupt nesting; the IMASK value is not affected. The ADSP-2106x changes IMASKP each time a higher priority interrupt interrupts a lower priority service routine.

The bits in IMASKP correspond to the interrupts in order of priority, the same as in IRPTL and IMASK. When an interrupt occurs, its bit is set in IMASKP. If nesting is enabled, a new temporary interrupt mask is generated by masking all interrupts of equal or lower priority to the highest priority bit set in IMASKP (and keeping higher priority interrupts the same as in IMASK). When a return from an interrupt service routine (RTI) is executed, the highest priority bit set in IMASKP is cleared, and again a new temporary interrupt mask is generated by masking all interrupts of equal or lower priority to the highest priority bit set in IMASKP. The bit set in IMASKP that has the highest priority always corresponds to the priority of the interrupt being serviced.

If nesting is not enabled, the processor masks out all interrupts and IMASKP is not used, although IMASKP is still updated to create a temporary interrupt mask.

IRPTL is updated, but the ADSP-2106x does not vector to an interrupt that occurs while its service routine is already executing. It waits until the RTI completes before vectoring to the service routine again.

3.6.6 Status Stack Save & Restore

For low-overhead interrupt servicing, the ADSP-2106x automatically saves and restores the status and mode contexts of the interrupted program. The three external interrupts ($\overline{IRQ}_{2-0}$), the timer interrupt, and the VIRPT vector interrupt cause an automatic push of ASTAT and MODE1 onto the status stack, which is five levels deep. These registers are automatically popped from the status stack by the return from interrupt instruction, RTI (and by the JUMP (CI) instruction, described below in "Clearing The Current Interrupt For Reuse").

- Only $\overline{IRQ}_{2-0}$, timer, and VIRPT interrupts cause a push of the status stack. All other interrupts require an explicit save and restore of the appropriate registers to memory.

Pushing ASTAT and MODE1 preserves the status and control bit settings so that if the service routine alters these bits, the original settings are automatically restored upon the return from interrupt.

Note, however, that the $FLAG_{3-0}$ bits in ASTAT are not affected by status stack pushes and pops; the values of these bits carry over from the main program to the service routine and from the service routine back to the main program.

The top of the status stack contains the current values of ASTAT and MODE1. Reading and writing these registers does not move the stack pointer. The stack pointer is moved, however, by explicit PUSH and POP instructions.

3.6.7 Software Interrupts

The ADSP-2106x provides software interrupts that emulate interrupt behavior but are activated through software instead of hardware. Setting one of bits 28-31 in IRPTL, with either a BIT SET instruction or a write to IRPTL, activates a software interrupt. The ADSP-2106x branches to the corresponding interrupt routine if that interrupt is not masked and interrupts are enabled.

3.6.8 Clearing The Current Interrupt For Reuse

Normally the ADSP-2106x ignores and does not latch an interrupt that reoccurs while its service routine is already executing. When the interrupt initially occurs, the corresponding bit in IRPTL is set. During execution of the service routine, this bit is kept cleared—the ADSP-2106x clears the bit during every cycle, preventing the same interrupt from being latched while its service routine is already executing.

The clear interrupt (CI) modifier of the JUMP instruction, however, allows the reuse of an interrupt while it is being serviced. This can be useful in systems that require fast interrupt response and low interrupt latency. The JUMP (CI) instruction should be located within the interrupt service routine. JUMP (CI) clears the status of the current interrupt without leaving the interrupt service routine, reducing the interrupt routine to a normal subroutine—this allows the interrupt to occur again, as a result of a different event or task in the ADSP-2106x system.

The JUMP (CI) instruction reduces an interrupt service routine to a normal subroutine by clearing the appropriate bit in the interrupt latch register (IRPTL) and interrupt mask pointer (IMASKP) and popping the status stack. The ADSP-2106x then stops automatically clearing the interrupt's latch bit (in IRPTL) in every cycle, allowing the interrupt to occur again.

When returning from a subroutine which has been reduced from an interrupt service routine with a JUMP (CI) instruction, the (LR) modifier of the RTS instruction must be used (in case the interrupt occurred during the last two instructions of a loop). Refer to "General Restrictions" in Section 3.5, "Loops", for a description of the RTS (LR) instruction.

The following example shows an interrupt service routine that is reduced to a subroutine with the (CI) modifier:

```
instr1;               {interrupt entry from main program}
JUMP(PC,3) (DB,CI);   {clear interrupt status}
instr3;
instr4;
instr5;
RTS (LR);      {use LR modifier with return from subroutine}
```

Note that the `JUMP(PC,3)(DB,CI)` instruction actually only continues linear execution flow by jumping to the location PC + 3 (instr5), with the two intervening instructions (instr3, instr4) being executed because of the delayed branch (DB). This JUMP instruction is only an example—a JUMP (CI) can be to any location.

3.6.9 External Interrupt Timing & Sensitivity

Each of the ADSP-2106x's three external interrupts, $\overline{IRQ}_{2\text{-}0}$, can be either level- or edge-triggered.

The ADSP-2106x samples interrupts once every CLKIN cycle. Level-sensitive interrupts are considered valid if sampled active (low). A level-sensitive interrupt must go inactive (high) before the processor returns from the interrupt service routine. If a level-sensitive interrupt is still active when the processor samples it, the processor treats it as a new request, repeating the same interrupt routine without returning to the main program (assuming no higher priority interrupts are active).

Edge-triggered interrupt requests are considered valid if sampled high in one cycle and low in the next. The interrupt can stay active indefinitely. To request another interrupt, the signal must go high, then low again.

Edge-triggered interrupts require less external hardware compared to level-sensitive requests since there is never a need to negate the request. However, multiple interrupting devices may share a single level-sensitive request line on a wired-OR basis, which allows for easy system expansion.

A bit for each interrupt in the MODE2 register indicates the sensitivity mode of each interrupt.

MODE2

Bit	*Name*	*Definition*
0	IRQ0E	1=edge-sensitive; 0=level-sensitive
1	IRQ1E	1=edge-sensitive; 0=level-sensitive
2	IRQ2E	1=edge-sensitive; 0=level-sensitive

3.6.9.1 Asynchronous External Interrupts

The processor accepts interrupts that are asynchronous to the ADSP-2106x clock; that is, an interrupt signal may change at any time. An asynchronous interrupt must be held low at least one CLKIN cycle to guarantee that it is sampled. Synchronous interrupts need only meet the setup and hold time requirements relative to the rising edge of CLKIN, as specified in the *ADSP-2106x Data Sheet*.

3.6.10 Multiprocessor Vector Interrupts (VIRPT)

Vector interrupts are used for interprocessor commands in multiple-processor systems. When an external processor writes an address to the VIRPT register a vector interrupt is caused. The external processor may be either another ADSP-2106x or a host.

When the vector interrupt is serviced, the ADSP-2106x automatically pushes the status stack and begins executing the service routine located at the address specified in VIRPT. The lower 24 bits of VIRPT contain the address; the upper 8 bits may be optionally used as data to be read by the interrupt service routine. At reset, VIRPT is initialized to its standard address in the ADSP-2106x's interrupt vector table.

The minimum latency for vector interrupts is six cycles, five of which are NOPs. When the RTI (return from interrupt) instruction is reached in the service routine, the ADSP-2106x automatically pops the status stack.

The VIPD bit in SYSTAT reflects the status of the VIRPT register. If VIRPT is written while a previous vector interrupt is pending, the new vector address replaces the pending one. If VIRPT is written while a previous vector interrupt is being serviced, the new vector address is ignored and no new interrupt is triggered. If the ADSP-2106x writes to its own VIRPT register it is ignored.

To use the ADSP-2106x's vector interrupt feature, external processors can take the following sequence of actions:

1. Poll the VIRPT register until it reads a certain token value (i.e. zero).
2. Write the vector interrupt service routine address to VIRPT.
3. When the service routine is finished, it writes the token back into VIRPT to indicate that it is finished and that another vector interrupt can be initiated.

3.7 TIMER

The ADSP-2106x includes a programmable interval timer which can generate periodic interrupts. You program the timer by writing values to its two registers and you control timer operation through a bit in the MODE2 register. An external output, TIMEXP, signals to other devices that the timer count has expired.

Figure 3.11 shows a block diagram of the timer. Two universal registers, TPERIOD and TCOUNT, control the timer interval.

Register	*Function*	*Width*
TPERIOD	Timer Period Register	32 bits
TCOUNT	Timer Counter Register	32 bits

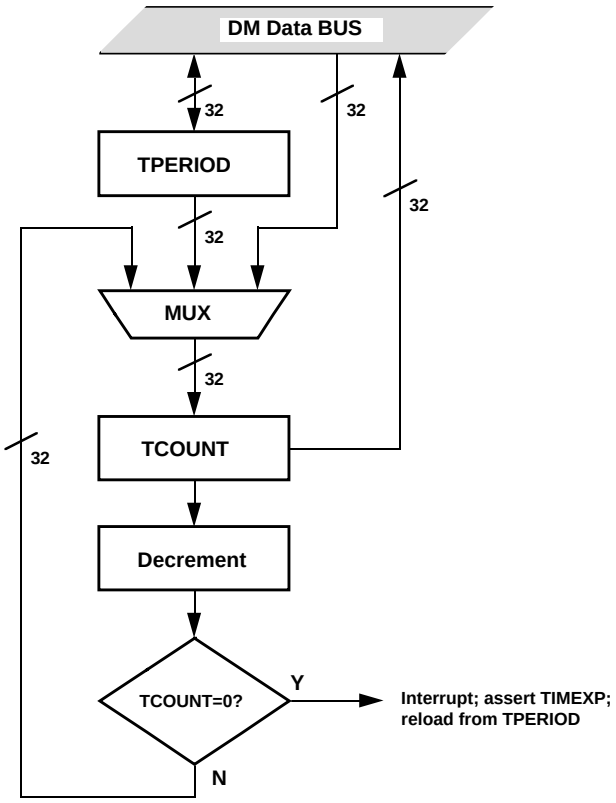


Figure 3.11 Timer Block Diagram

The TCOUNT register contains the timer counter. The timer decrements the TCOUNT register each clock cycle. When the TCOUNT value reaches zero, the timer generates an interrupt and asserts the TIMEXP output high for 4 cycles (when the timer is enabled). See Figure 3.12. On the clock cycle after TCOUNT reaches zero, the timer automatically reloads TCOUNT from the TPERIOD register.

The TPERIOD value specifies the frequency of timer interrupts. The number of cycles between interrupts is TPERIOD + 1. The maximum value of TPERIOD is $2^{32} - 1$, so if the clock cycle is 50 ns, the maximum interval between interrupts is 214.75 seconds.

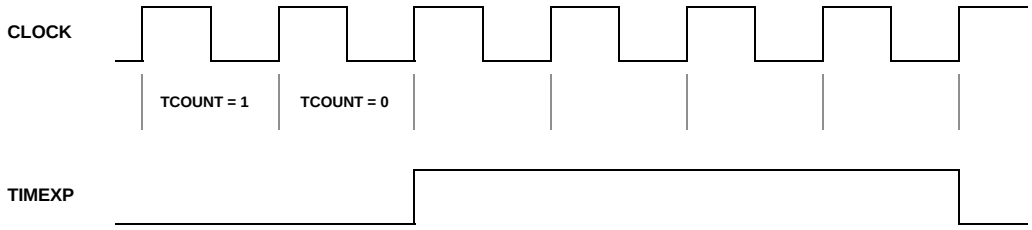


Figure 3.12 TIMEXP Signal

3.7.1 Timer Enable/Disable

To start and stop the timer, you enable and disable it with the TIMEN bit in the MODE2 register. With the timer disabled, you load TCOUNT with an initial count value and TPERIOD with the number of cycles for the interval you want. Then you enable the timer when you want to begin the count.

At reset, the timer enable bit in the MODE2 register is cleared, so the timer is disabled. When the timer is disabled, it does not decrement the TCOUNT register and it generates no interrupts. When the timer enable bit is set, the timer starts decrementing the TCOUNT register at the end of the next clock cycle. If the bit is subsequently cleared, the timer is disabled and stops decrementing TCOUNT after the next clock cycle (see Figure 3.13).

MODE2

Bit	*Name*	*Definition*
5	TIMEN	Timer enable

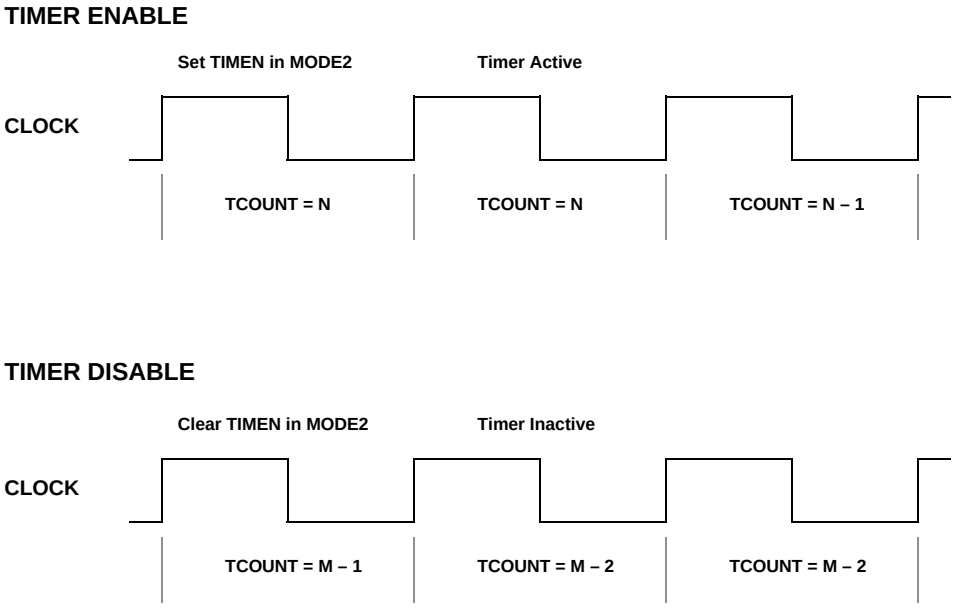


Figure 3.13 Timer Enable & Disable

3.7.2 Timer Interrupts

When the value of TCOUNT reaches zero, the timer generates two interrupts, one with a relatively high priority, the other with a relatively low priority. At reset, both are masked. You should unmask only the timer interrupt that has the priority you want, and leave the other masked.

IRPTL	*Interrupt*	*Vector*	
Bit	*Name*	*Address*	*Function*
4	TMZHI	0x10	Timer =0 (high priority option)
23	TMZLI	0x5C	Timer=0 (low priority option)

Interrupt priority determines which interrupt is serviced first when two occur in the same cycle. It also affects interrupt nesting—when nesting is enabled, only higher priority interrupts can interrupt a service routine in progress.

Like other interrupts, the timer interrupt requires two cycles to fetch and decode the first instruction of the service routine. The service routine begins executing four cycles after the timer count reaches zero, as shown in Figure 3.14.

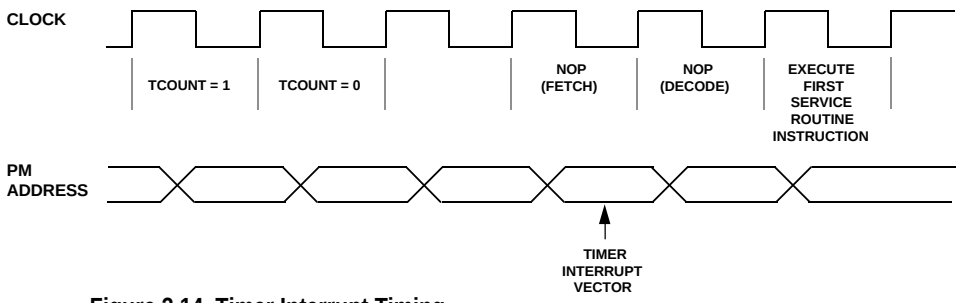


Figure 3.14 Timer Interrupt Timing

3.7.3 Timer Registers

Both the TPERIOD and TCOUNT registers can be read and written by universal register transfers. Reading the registers has no effect on the timer. An explicit write to TCOUNT has priority over both the loading of TCOUNT from TPERIOD and the decrementing of TCOUNT.

Neither TCOUNT nor TPERIOD are affected by a reset, so you should initialize both registers after reset, before enabling the timer.

3.8 STACK FLAGS

The STKY status register maintains stack full and stack empty flags for the PC stack as well as overflow and empty flags for the status stack and loop stack. Unlike other bits in STKY, several of these flag bits are not “sticky.” They are set by the occurrence of the corresponding condition and are cleared when the condition is changed (by a push, pop, or processor reset).

STKY *Bit*	*Name*	*Definition*	*Not Sticky*	*Sticky/ Cleared By*
21	PCFL	PC stack full	Not sticky	Pop
22	PCEM	PC stack empty	Not sticky	Push
23	SSOV	Status stack overflow	Sticky	RESET
24	SSEM	Status stack empty	Not sticky	Push
25	LSOV	Loop stacks* overflow	Sticky	RESET
26	LSEM	Loop stacks* empty	Not sticky	Push

* Loop address stack and loop counter stack

The status stack flags are read-only. Writes to the STKY register have no effect on these bits.

The overflow and full flags are provided for diagnostic aid only and are not intended to allow recovery from overflow. Status stack or loop stack overflow or PC stack full causes an interrupt.

The empty flags facilitate stack saves to memory. You monitor the empty flag when saving a stack to memory to know when all values have been transferred. The empty flags do not cause interrupts because an empty stack is an acceptable condition.

3.9 IDLE & IDLE16

IDLE and IDLE16 are special instructions that halt the ADSP-2106x core processor in a low-power state until an external interrupt ($\overline{IRQ}_{2-0}$), timer interrupt, DMA interrupt, or VIRPT vector interrupt occurs. When the processor executes an IDLE instruction, it fetches one more instruction at the current fetch address and then suspends operation. The ADSP-2106x's I/O processor is unaffected by the IDLE instruction—any DMA transfers to or from internal memory will continue uninterrupted.

The processor's internal clock continues to run during IDLE, as well as the timer (if it is enabled). When an external interrupt ($\overline{IRQ}_{2-0}$), timer interrupt, DMA interrupt, or VIRPT vector interrupt occurs, the processor responds normally. After two cycles needed to fetch and decode the first instruction of the interrupt service routine, the processor continues executing instructions normally.

On the ADSP-21061 only, the IDLE16 instruction executes a NOP and puts the processor in a low power state. IDLE16 is a lower power version of the IDLE instruction. This instruction halts the processor like the IDLE instruction; in this case, the internal clock runs at 1/16th the rate of CLKIN. The ADSP-21061's I/O processor continues to function, but all operations occur at 1/16th the rate. All internal memory transfers require an extra 15 cycles. The serial clocks and frame syncs (if being sourced by the ADSP-21061) are divided down by a factor of 16 during IDLE16. Similarly, all Host accesses take 16 times longer to complete. The processor remains in the low power state until an interrupt occurs.

After returning from the interrupt, execution continues at the instruction following the IDLE or IDLE16 instruction.

3.10 INSTRUCTION CACHE

The ADSP-2106x's on-chip instruction cache is a 2-way, set-associative cache with entries for 32 instructions. Operation of the cache is transparent to the programmer. The ADSP-2106x caches only instructions that conflict with program memory data accesses (over the PM Data Bus, with the address generated by DAG2 on the PM Address Bus). This feature makes the cache considerably more efficient than a cache that loads every instruction, since typically only a few instructions must access data from a block of program memory.

Because of the three-stage instruction pipeline, if the instruction at address *n* requires a program memory data access, there is a conflict with the instruction fetch at address *n*+2, assuming sequential execution. It is this fetched instruction (*n*+2) that is stored in the instruction cache, not the instruction requiring the program memory data access.

If the instruction needed is in the cache, a "cache hit" occurs—the cache provides the instruction while the program memory data access is performed. If the instruction needed is not in the cache, a "cache miss" occurs, and the instruction fetch (from memory) takes place in the cycle following the program memory data access, incurring one cycle of overhead. This instruction is loaded into the cache, if the cache is enabled and not frozen, so that it is available the next time the same instruction (requiring program memory data) is executed.

3.10.1 Cache Architecture

Figure 3.15 shows a block diagram of the instruction cache. The cache contains 32 entries. An entry consists of a register pair containing an instruction and its address. Each entry has a "valid" bit which is set if the entry contains a valid instruction.

The entries are divided into 16 sets (numbered 15-0) of two entries each, entry 0 and entry 1. Each set has an LRU (Least Recently Used) bit whose value indicates which of the two entries contains the least recently used instruction (1=entry 1, 0=entry 0).

Every possible instruction address is mapped to a set in the cache by its 4 LSBs. When the processor needs to fetch an instruction from the cache, it uses the 4 address LSBs as an index to a particular set. Within

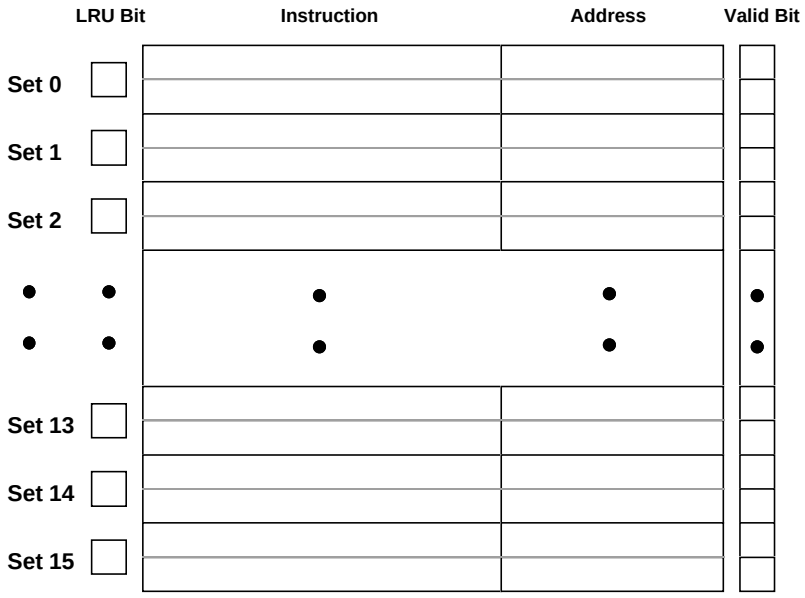


Figure 3.15 Instruction Cache Architecture

that set, it checks the addresses of the two entries to see whether either contains the needed instruction. A cache hit occurs if the instruction is found, and the LRU bit is updated if necessary to indicate the entry that did not contain the needed instruction.

A cache miss occurs if neither entry in the set contains the needed instruction. In this case, a new instruction and its address are loaded into the least recently used entry of the set that matches the 4 LSBs of the address. The LRU bit is toggled to indicate that the other entry in the set is now the least recently used.

Because instructions are mapped to sets by their 4 address LSBs, there is no need to store these bits in the cache; the 4 LSBs are implied by the set in which the instruction has been stored. Only bits 23-4 are actually stored in a cache entry.

3.10.2 Cache Efficiency

Usually, cache operation and its efficiency is not a concern. However, there are some situations that can degrade cache efficiency and can be remedied easily in your program.

When a cache miss occurs, the needed instruction is loaded into the cache so that if the same instruction is needed again, it will be there (i.e. a cache hit will occur). However, if another instruction whose address is mapped to the same set displaces this instruction, there will be a cache miss instead. The LRU bits help to reduce this possibility since at least two other instructions mapped to the same set must be needed before an instruction is displaced. If three instructions mapped to the same set are all needed repeatedly, cache efficiency (i.e. "hit rate") can go to zero. The solution is to move one or more of the instructions to a new address, one that is mapped to a different set.

An example of cache-inefficient code is shown in Figure 3.16. The program memory data access at address 0x101 in the *tight* loop causes the instruction at 0x103 to be cached (in set 3). Each time the subroutine *sub* is called, the program memory data accesses at 0x201 and 0x211 displace this instruction by loading the instructions at 0x203 and 0x213 into set 3. If the subroutine is called only rarely during the loop execution, the impact will be minimal. If the subroutine is called frequently, the effect will be noticeable. If the execution of the loop is time-critical, it would be advisable to move the subroutine up one location (starting at 0x201), so that the two cached instructions end up in set 4 instead of 3.

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

Figure 3.16 Cache-Inefficient Code

3.10.3 Cache Disable & Cache Freeze

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

MODE2

Bit	*Name*	*Function*
4	CADIS	Cache Disable
19	CAFRZ	Cache Freeze

After reset the cache is cleared, containing no instructions, and is unfrozen and enabled.

An instruction containing a program memory data access must not be placed directly after a cache enable or cache disable instruction—the ADSP-2106x must wait at least one cycle before executing the PM data access. A NOP may be inserted to accomplish this.

3 Program Sequencing

Data Addressing 4

4.1 OVERVIEW

The ADSP-2106x's two data address generators (DAGs) simplify the task of organizing data by maintaining pointers into memory. The DAGs allow the processor to address memory *indirectly*; that is, an instruction specifies a DAG register containing an address instead of the address value itself.

Data address generator 1 (DAG1) generates 32-bit addresses on the DM Address Bus. Data address generator 2 (DAG2) generates 24-bit addresses on the PM Address Bus. The basic architecture for both DAGs is shown in Figure 4.1 on the following page.

The DAGs also support in hardware some functions commonly used in digital signal processing algorithms. Both DAGs support circular data buffers, which require advancing a pointer repetitively through a range of memory. Both DAGs can also perform a bit-reversing operation, which outputs the bits of an address in reversed order.

4.2 DAG REGISTERS

Each DAG has four types of registers: *Index (I), Modify (M), Base (B),* and *Length (L)* registers.

An I register acts as a pointer to memory, and an M register contains the increment value for advancing the pointer. By modifying an I register with different M values, you can vary the increment as needed.

B registers and L registers are used only for circular data buffers. A B register holds the base address (i.e. the first address) of a circular buffer. The same-numbered L register contains the number of locations in (i.e. the length of) the circular buffer.

4 Data Addressing

Each DAG contains eight of each type of register:

DAG1 registers (32-bit)	*DAG2 registers (24-bit)*
B0-B7	B8-B15
I0-I7	I8-I15
M0-M7	M8-M15
L0-L7	L8-L15

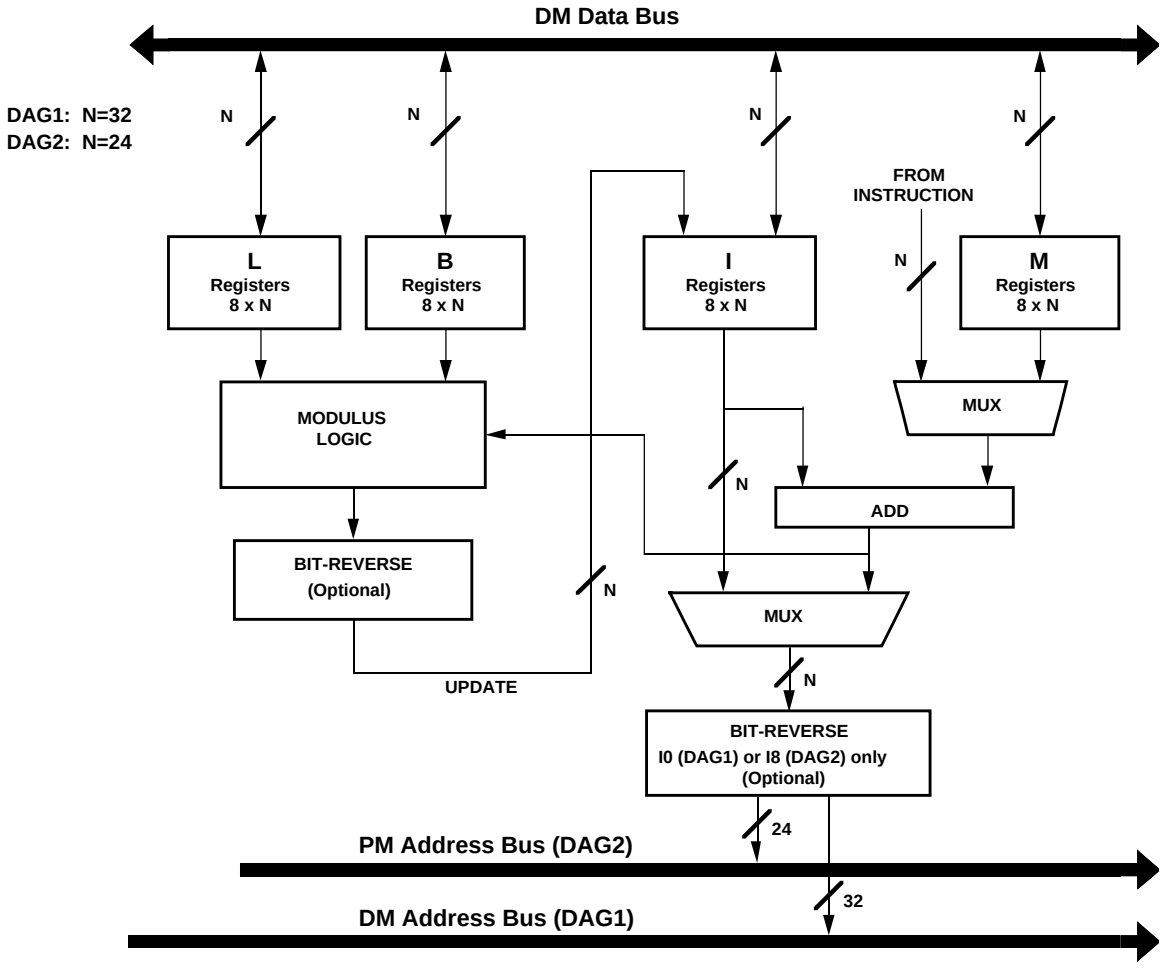


Figure 4.1 Data Address Generator Block Diagram

4.2.1 Alternate DAG Registers

Each DAG register has an alternate (secondary) register for context switching. For activating alternate registers, each DAG is organized into high and low halves, as shown in Figure 4.2. The high half of DAG1 contains the I, M, B and L registers numbered 4-7, and the low half, the registers numbered 0-3. Likewise, the high half of DAG2 consists of registers 12-15, and the low half consists of registers 8-11.

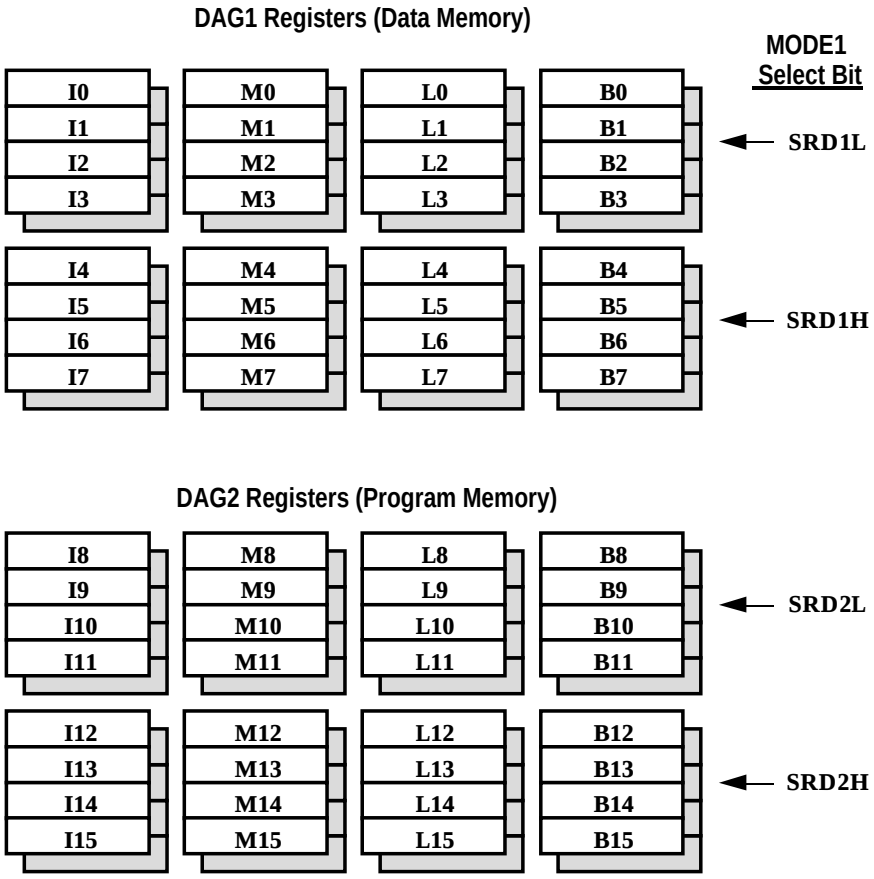


Figure 4.2 Alternate DAG Registers

Several control bits in the MODE1 register determine for each half whether primary or alternate registers are active (0=primary registers, 1=alternate registers):

MODE1

Bit	*Name*	*Definition*
3	SRD1H	DAG1 alternate register select (4-7)
4	SRD1L	DAG1 alternate register select (0-3)
5	SRD2H	DAG2 alternate register select (12-15)
6	SRD2L	DAG2 alternate register select (8-11)

This grouping of alternate registers lets you pass pointers between contexts in each DAG.

4.3 DAG OPERATION

DAG operations include:

- address output with pre-modify or post-modify,
- modulo addressing (for circular buffers), and
- bit-reversed addressing

Short word addresses (for 16-bit data) are right-shifted by one bit before being output onto the DM Address Bus. This allows internal memory to use the address directly. (See “16-Bit Short Words” in the *Memory* chapter of this manual for details on short word addresses.)

4.3.1 Address Output & Modification

The processor can add an offset (modifier), either an M register or an immediate value, to an I register and output the resulting address; this is called a *pre-modify without update* operation. Or it can output the I register value as it is, and then add an M register or immediate value to form a new I register value. This is a *post-modify* operation. These operations are compared in Figure 4.3. The pre-modify operation does not change the value of the I register. The width of an immediate modifier depends on the instruction; it can be as large as the width of the I register. The L register and modulo logic do not affect a pre-modified address—pre-modify addressing is always linear, not circular.

Pre-modify addressing operations must not change the memory space of the address; for example, pre-modification of an address in ADSP-2106x Internal Memory Space should not generate an address in External Memory Space. Refer to the *Memory* chapter for information on the ADSP-2106x memory map.

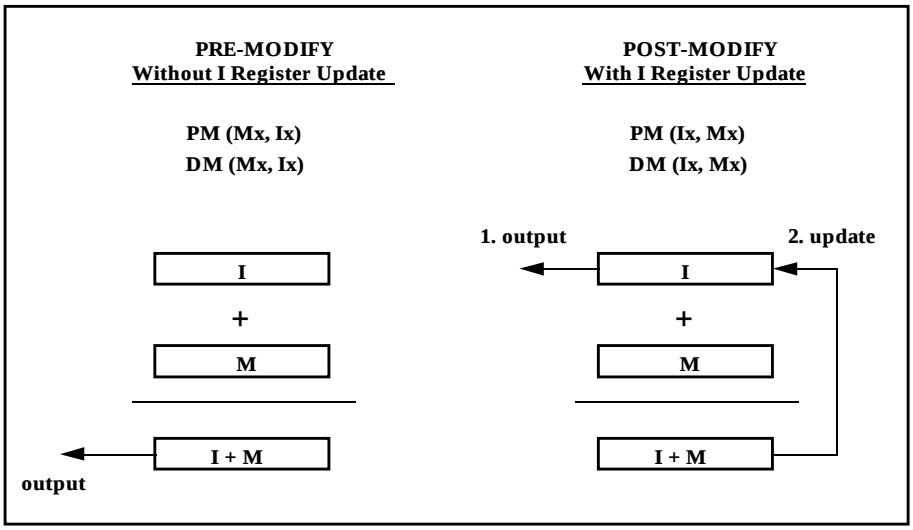


Figure 4.3 Pre-Modify & Post-Modify Operations

4.3.1.1 DAG Modify Instructions

In ADSP-2106x assembly language, pre-modify and post-modify operations are distinguished by the positions of the index and modifier (M register or immediate value) in the instruction. The I register before the modifier indicates a post-modify operation. If the modifier comes first, a pre-modify without update operation is indicated. The following instruction, for example, accesses the program memory location with an address equal to the value stored in I15, and the value I15 + M12 is written back to the I15 register:

```
R6 = PM(I15,M12);
```
Indirect addressing with post-modify

If the order of the I and M registers is switched, however,

```
R6 = PM(M12,I15);
```
Indirect addressing with pre-modify

the instruction accesses the location in program memory with an address equal to I15 + M12, but does not change the value of I15.

Any M register can modify any I register within the same DAG (DAG1 or DAG2). Thus,

```
DM(M0,I2) = TPERIOD;
```

is a legal instruction that accesses the data memory location M0 + I2; however,

```
DM(M0,I14) = TPERIOD;
```

is *not* a legal instruction because the I and M registers belong to different DAGs.

4.3.1.2 Immediate Modifiers

The magnitude of an immediate value that can modify an I register depends on the instruction type and whether the I register is in DAG1 or DAG2. DAG1 modify values can be up to 32 bits wide; DAG2 modify values can be up to 24 bits wide. Some instructions with parallel operations only allow modify values up to 6 bits wide. Here are two examples:

32-bit modifier:

```
R1=DM(0x40000000,I1);
```
DM address = I1 + 0x4000 0000

6-bit modifier:

```
F6=F1+F2,PM(I8,0x0B)=ASTAT;
```
PM address = I8, I8 = I8 + 0x0B

4.3.2 Circular Buffer Addressing

The DAGs provide for addressing of locations within a circular data buffer. A circular buffer is a set of memory locations that stores data. An index pointer steps through the buffer, being post-modified and updated by the addition of a specified value (positive or negative) for each step. If the modified address pointer falls outside the buffer, the length of the buffer is subtracted from or added to the value, as required to wrap the index pointer back to the start of the buffer (see Figure 4.4). There are no restrictions on the value of the base address for a circular buffer.

Circular buffer addressing must use M registers for post-modify of I registers, not pre-modify; for example:

```
F1=DM(I0,M0);
F1=DM(M0,I0);
```
Use post-modify addressing for circular buffers, not pre-modify.

Length = 11
Base address = 0
Modifier (step size) = 4

Sequence shows order in which locations are accessed in one pass.
Sequence repeats on subsequent passes.

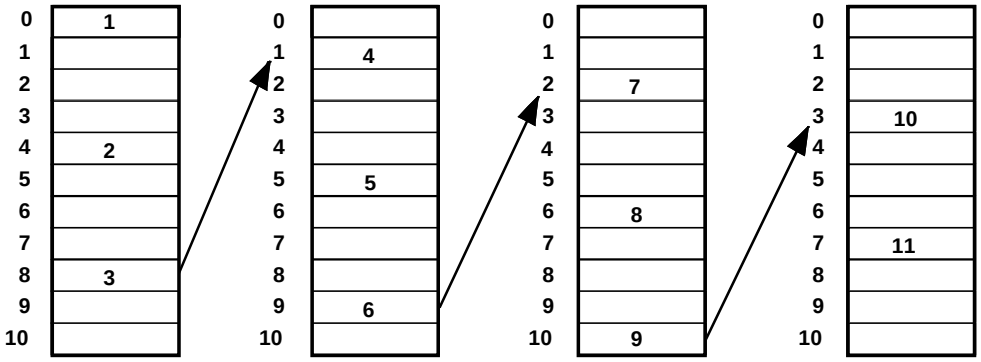


Figure 4.4 Circular Data Buffers

4.3.2.1 Circular Buffer Operation

You set up a circular buffer in assembly language by initializing an L register with a positive, nonzero value and loading the corresponding (same-numbered) B register with the base (starting) address of the buffer. The corresponding I register is automatically loaded with this same starting address.

On the first post-modify access using the I register, the DAG outputs the I register value on the address bus and then modifies it by adding the specified M register or immediate value to it. If the modified value is within the buffer range, it is written back to the I register. If the value is outside the buffer range, the L register value is subtracted (or, if the modify value is negative, added) first.

If M is positive,

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

If M is negative,

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

4.3.2.2 Circular Buffer Registers

All four types of DAG registers are involved in the operation of a circular buffer:

- The I register contains the value which is output on the address bus.

- The M register contains the post-modify amount (positive or negative) which is added to the I register at the end of each memory access. The M register can be any M register in the same DAG as the I register and does not have to have the same number. The modify value can also be an immediate number instead of an M register. The magnitude of the modify value, whether from an M register or immediate, must be less than the length (L register) of the circular buffer.

- The L register sets the size of the circular buffer and thus the address range that the I register is allowed to circulate through. L must be positive and cannot have a value greater than $2^{31} - 1$ (for L0-L7) or $2^{23} - 1$ (for L8-L15). If an L register's value is zero, its circular buffer operation is disabled.

- The B register, or the B register plus the L register, is the value that the modified I value is compared to after each access. When the B register is loaded, the corresponding I register is simultaneously loaded with the same value. When I is loaded, B is not changed. B and I can be read independently.

4.3.2.3 Circular Buffer Overflow Interrupts

There is one set of registers in each DAG that can generate an interrupt upon circular buffer overflow (i.e. address wraparound). In DAG1, the registers are B7, I7, L7, and in DAG2 they are B15, I15, L15. Circular buffer overflow interrupts can be used to implement a ping-pong (i.e. swap I/O buffer pointers) routine, for example.

Whenever a circular buffer addressing operation using these registers causes the address in the I register to be incremented (or decremented) past the end (or start) of the circular buffer, an interrupt is generated. Depending on which register set was used, the interrupt is either:

Interrupt	*DAG Registers To Use*	*Vector Address*	*Symbolic Name**
DAG1 circular buffer 7 overflow	B7, I7, L7	0x54	CB7I
DAG2 circular buffer 15 overflow	B15, I15, L15	0x58	CB15I

* These symbols are defined in the #include file `def21060.h`. See "Symbol Definitions File (def21060.h)" at the end of Appendix E, *Control/Status Registers.*

Specifically, an interrupt is generated during an instruction's address post-modify when:

(for M<0) $I + M < B$
(for M≥0) $I + M \geq B + L$

The interrupts can be masked by clearing the appropriate bit in IMASK.

There may be situations where you want to use I7 or I15 without circular buffering but with the circular buffer overflow interrupts unmasked. To disable the generation of these interrupts, set the B7/B15 and L7/L15 registers to values that assure the conditions that generate interrupts (as specified above) never occur. For example, when accessing the address range 0x1000–0x2000, your program could set B=0x0000 and L=0xFFFF. Note that setting the L register to zero will not achieve the desired results.

If you are using either of the circular buffer overflow interrupts, you should avoid using the corresponding I register(s) (I7, I15) in the rest of your program, or be careful to set the B and L registers as described above to prevent spurious interrupt branching.

The STKY status register includes two bits that also indicate the occurrence of a circular buffer overflow, bit 17 (DAG1 circular buffer 7 overflow) and bit 18 (DAG2 circular buffer 15 overflow). These bits are "sticky"—they remain set until explicitly cleared.

4.3.3 Bit-Reversal

Bit-reversal of memory addresses can be performed in two ways: by enabling the bit-reverse mode on DAG1 or DAG2 and using a specific I register (I0 or I8), or by using the explicit bit-reverse instruction (BITREV).

4.3.3.1 Bit-Reverse Mode

In bit-reverse mode, DAG1 bit-reverses 32-bit address values output from I0 and DAG2 bit-reverses 24-bit address values output from I8. These modes are enabled by the BR0 and BR8 bits in the MODE1 register. Only address values from I0 or I8 are bit-reversed. This mode affects both pre-modify and post-modify operations.

MODE1

Bit	*Name*	*Definition*
0	BR8	Bit-reverse mode for I8 (DAG2)
1	BR0	Bit-reverse mode for I0 (DAG1)

Bit-reversal occurs at the output of the DAG and does not affect the value in I0 or I8. In the case of a post-modify operation, the update value is not bit-reversed.

Example:

```
I0=0x80400000;
R1=DM(I0,3);          DM address=0x201,  I0=0x80400003
```

4.3.3.2 Bit-Reverse Instruction

The BITREV instruction modifies and bit-reverses addresses in any DAG index register (I0-I15) without actually accessing memory. This instruction is independent of the bit-reverse mode. The BITREV instruction adds a 32-bit immediate value to a DAG1 index register (or a 24-bit immediate value to a DAG2 index register), bit-reverses the result and writes the result back to the same index register.

Example:

```
BITREV(I1,4);          I1 = Bit-reverse of  (I1+4)
```

4.4 DAG REGISTER TRANSFERS

DAG registers are part of the universal register set and may be written to from memory, from another universal register, or from an immediate field in an instruction. DAG register contents may be written to memory or to a universal register.

Transfers between 32-bit DAG1 registers and the 40-bit DM Data Bus are aligned to bits 39-8 of the bus. When 24-bit DAG2 registers are read to the 40-bit DM Data Bus, M register values are sign-extended to 32 bits and I, L, and B register values are zero-filled to 32 bits. The results are aligned to bits 39-8 of the DM Data Bus. When DAG2 registers are written from the DM Data Bus, bits 31-8 are transferred and the rest are ignored. Figure 4.5 illustrates these transfers.

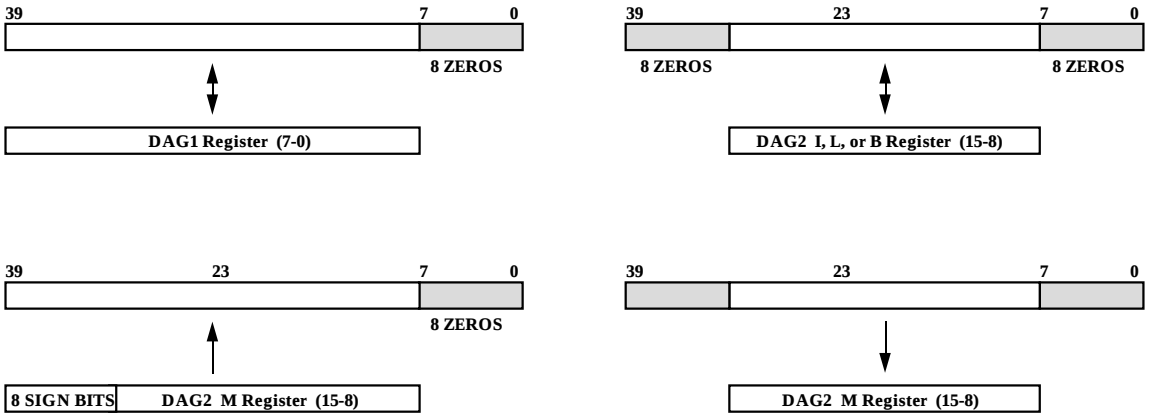


Figure 4.5 DAG Register Transfers

4.4.1 DAG Register Transfer Restrictions

For certain instruction sequences involving transfers to and from DAG registers, an extra (NOP) cycle is automatically inserted by the processor **(1)**. Certain other sequences cause incorrect results and are not allowed by the ADSP-21000 Family assembler **(2)**.

1.) When an instruction that loads a DAG register is followed by an instruction that uses any register in the same DAG for data addressing, modify instructions, or indirect jumps, the ADSP-2106x inserts an extra (NOP) cycle between the two instructions. This happens because the same bus is needed by both operations in the same cycle, therefore the second operation must be delayed.

Example:

```
L2=8;
DM(I0,M1)=R1;
```

Because L2 is in the same DAG as I0 (and M1), an extra cycle is inserted after the write to L2.

2.) The following types of instructions can execute on the processor, but cause incorrect results; these instructions are disallowed by the ADSP-21000 Family assembler:

- An instruction that stores a DAG register in memory using indirect addressing from the same DAG, with or without update of the index register. The instruction writes the wrong data to memory or updates the wrong index register.

  Examples:

  `DM(M2,I1)=I0;` *or* `DM(I1,M2)=I0;`

- An instruction that loads a DAG register from memory using indirect addressing from the same DAG, with update of the index register. The instruction will either load the DAG register or update the index register, but not both.

  Example:

  ```
  L2=DM(I1,M0);
  ```

Memory 5

5.1 OVERVIEW

ADSP-2106x processors contain a large dual-ported memory for on-chip program and data storage. On these processors, the two memory blocks are named Block 0 and Block 1. A comparison of on-chip memory (SRAM) available on ADSP-2106x processors is as follows:

On-chip SRAM	ADSP-21060	ADSP-21062	ADSP-21061
Total Size	4 MBit	2 MBit	1 MBit
Block size (2)	2 MBit	1 MBit	0.5 MBit
# of 48-bit words (per block)	40K	20K	8K
# of 32-bit words (per block)	64K	32K	16K
# of 16-bit words (per block)	128K	64K	32K

Addressing of up to 4 gigawords of additional, off-chip memory is also provided through the external port of ADSP-2106x processors.

32-bit memory words are used for single-precision IEEE floating-point data. 48-bit words contain either instructions or 40-bit extended-precision floating-point data. In addition, the ADSP-2106x supports a 16-bit short word format which can be used for integer or fractional data values.

The ADSP-2106x has three internal buses connected to its dual-ported memory, the PM bus, DM bus, and I/O bus. The PM bus and DM bus share one port of the memory and the I/O bus is connected to the other port. The ADSP-2106x's internal PM and DM buses are controlled by the processor core while the I/O bus is controlled by the ADSP-2106x's on-chip I/O processor. The I/O bus allows concurrent data transfers between either memory block and the ADSP-2106x's communication ports (link ports, serial ports, and external port).

5 Memory

With this dual-ported structure, accesses of internal memory by the processor core and I/O processor are independent and transparent to one another. Each block of memory can be accessed by both the core processor and the I/O processor in every cycle—no extra cycles are incurred when both the core and the I/O processor access the same block.

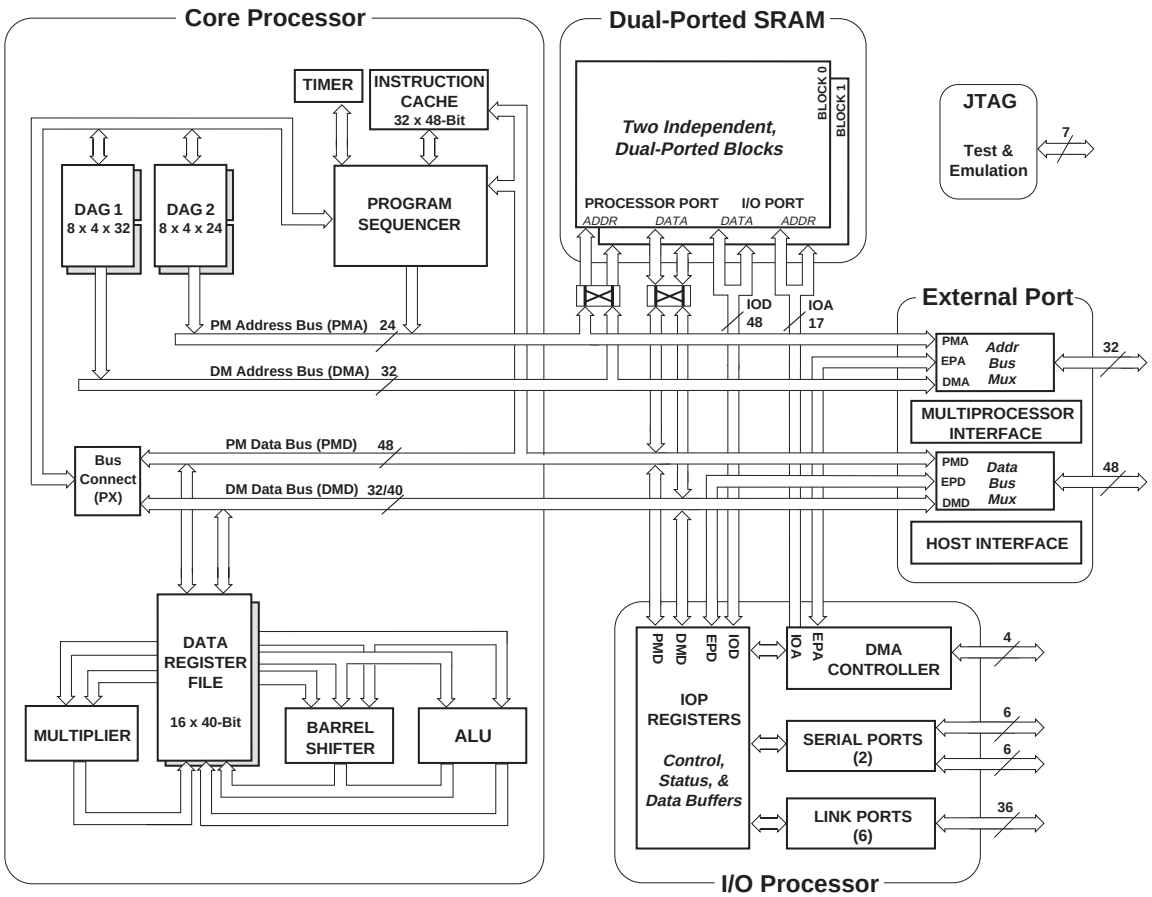


Figure 5.1 ADSP-2106x Block Diagram

Both the core processor and I/O processor have access to the external bus ($DATA_{47-0}$, $ADDR_{31-0}$), via the ADSP-2106x's external port. The external port provides access to off-chip memory and peripherals; it can also access the internal memory of other ADSP-2106xs connected in a multiprocessing system. This busing scheme allows the ADSP-2106x to have a single unified address space in which both code and data is stored.

External memory can be either 16, 32, or 48 bits wide; the ADSP-2106x's DMA controller automatically packs external data into the appropriate word width, either 48-bit instructions or 32-bit data.

Note that the ADSP-2106x's internal memory is divided into two *blocks*, called Block 0 and Block 1, while the external memory space is divided into four *banks*.

5.1.1 Dual Data Accesses

The ADSP-2100 and ADSP-21000 Family DSPs traditionally define memory as either program memory, for instructions, or as data memory, for data storage. The processors' modified Harvard architecture, however, allows data storage within program memory. The ADSP-2106x retains the ADSP-21000 Family's separate on-chip buses for program memory and data memory, but does not pre-define the two on-chip memory blocks as either PM or DM. This allows the memory to be freely configured to store different combinations of code and data.

The independent PM and DM buses allow the ADSP-2106x's processor core to simultaneously access instructions and data from both memory blocks. If the core tries to access two words from the same memory block (over the same bus) for a single instruction, however, an extra cycle is needed. Instructions are fetched over the PM bus or from the instruction cache. Data can be accessed over both the DM bus (using DAG1) and the PM bus (using DAG2). Figure 5.1 shows the memory bus connections on the ADSP-2106x.

The ADSP-2106x's two memory blocks can be configured to store different combinations of 48-bit instruction words and 32-bit data words. Maximum efficiency (i.e. single-cycle execution of dual-data-access instructions), though, is achieved when one block contains a mix of instructions and PM bus data while the other block contains DM bus data only.

This means that for an instruction requiring two data accesses, the PM bus (and DAG2) is used to access data from the mixed block, the DM bus (and DAG1) is used to access data from the data-only block, *and the instruction to be fetched must be available from the cache.* Another way to partition the data is to store one operand in external memory and the other in either block of internal memory.

In typical DSP applications such as digital filters and FFTs, two data operands must be accessed for some instructions. In a digital filter, for example, the filter coefficients can be stored in 32-bit words in the same memory block that contains the 48-bit instructions, while 32-bit data samples are stored in the other block. This provides single-cycle execution of dual-data-access instructions, with the filter coefficients being accessed by DAG2 over the PM bus and the instruction available from the cache.

In summary, to assure single-cycle, parallel accesses of two on-chip memory locations, the following conditions must be met:

- The two addresses must be located in different memory blocks (i.e. one in Block 0, one in Block 1).
- One address must be generated by DAG1 and the other by DAG2.
- The DAG1 address must not point to the same memory block that instructions are being fetched from.
- The instruction should be of the form:

```
compute, Rx=DM(I0-I7,M0-M7), Ry=PM(I8-I15,M8-M15);
```

(Note that reads and writes may be intermixed.)

Remember that a cache miss will occur if the fetched instruction is not valid during any DAG2 transfer.

5.1.2 Instruction Cache & PM Bus Data Accesses

Normally the ADSP-2106x fetches instructions over the 48-bit PM Data bus. When, however, the processor executes a dual-data-access instruction that requires data to be read or written over the PM bus, there is a conflict for use of the PM Data bus. The ADSP-2106x's on-chip instruction cache can resolve this conflict by providing the instruction (once it is stored in the cache, the first time it is executed).

By providing the instruction, the cache lets the core processor access data over the PM bus—the core processor fetches the instruction from the cache instead of from memory so that the processor can simultaneously transfer data over the PM bus. Only the instructions whose fetches conflict with PM bus data accesses are cached.

The instruction cache allows data to be accessed over the PM bus, without any extra cycles, whenever the instruction to be fetched is already cached (i.e. within a loop). An extra cycle will always occur in the event of a cache miss, even if the instruction and data are in different memory blocks but use the same bus.

5.1.3 On-Chip Memory Buses & Address Generation

The ADSP-2106x has three internal buses connected to its dual-ported memory, the PM bus, DM bus, and I/O bus. The PM bus and DM bus share one port of the memory and the I/O bus is connected to the other port.

The ADSP-2106x's program sequencer and data address generators (DAGs) supply memory addresses. The program sequencer supplies 24-bit PM bus addresses for instruction fetches. The DAGs supply addresses for data reads and writes. (See Figure 5.1.)

The two data address generators allow indirect addressing of data. DAG1 supplies 32-bit addresses over the DM bus. DAG2 supplies 24-bit addresses for PM bus data accesses. The two DAGs can generate simultaneous addresses—over the PM bus and DM bus—for dual operand read/writes, if the instruction to be fetched is available from the cache.

The 48-bit PM Data bus is used to transfer instructions (and data), and the 40-bit DM Data bus is used to transfer data. The PM Data bus is 48 bits wide to accommodate the 48-bit instruction width. When this bus is used to transfer 32-bit floating-point or 32-bit fixed-point data, the data is aligned to the upper 32 bits of the bus.

The 40-bit DM Data bus provides a path for the contents of any register in the processor to be transferred to any other register or to any external memory location in a single cycle. Data addresses come from one of two sources: an absolute value specified in the instruction (*direct addressing*), or the output of a data address generator (*indirect addressing*). 32-bit fixed-point and 32-bit single-precision floating-point data is also aligned to the upper 32 bits of the DM Data bus.

The PX bus connect registers permit data to be passed between the 48-bit PM Data bus and the 40-bit DM Data bus or between the 40-bit register file and the PM Data bus. These registers contain hardware to handle the 8-bit width difference.

The three memory buses—PM, DM, and I/O—are multiplexed at the processor's external port to create a single off-chip data bus ($DATA_{47-0}$) and address bus ($ADDR_{31-0}$).

5.1.4 Bus Exchange (PX Registers)

The PX register provides an internal bus exchange path for transferring data between the 48-bit PM Data Bus and the 40-bit DM Data Bus. The 48-bit PX register consists of PX1 and PX2. PX1 is 16 bits wide and PX2 is 32 bits wide. PX1 and PX2 can be used separately in instructions and can also be treated as the combined PX register. The alignment of PX1 and PX2 within PX is shown below in Figure 5.2.

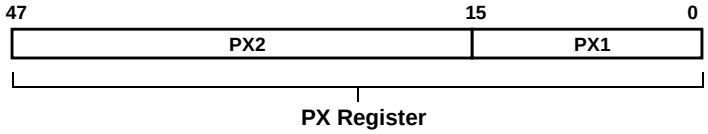


Figure 5.2 PX Register

Either PX1, PX2, or the combined PX register can be used in universal register-to-register transfers or in memory-to-register (and register-to-memory) transfers. These transfers may take place over the PM Data Bus or DM Data Bus. The PX register(s) can be read from or written to the PM Data Bus, the DM Data Bus, or the register file.

Data is aligned in PX register transfers as shown in Figure 5.3. When data is transferred between PX2 and the PM Data Bus, the upper 32 bits of the PM Data Bus are used. On transfers from PX2, the 16 LSBs of the PM Data Bus are filled with zeros. When data is transferred between PX1 and the PM Data Bus, the middle 16 bits of the PM Data Bus are used. On transfers from PX1, bits 15-0 and bits 47-32 are filled with zeros.

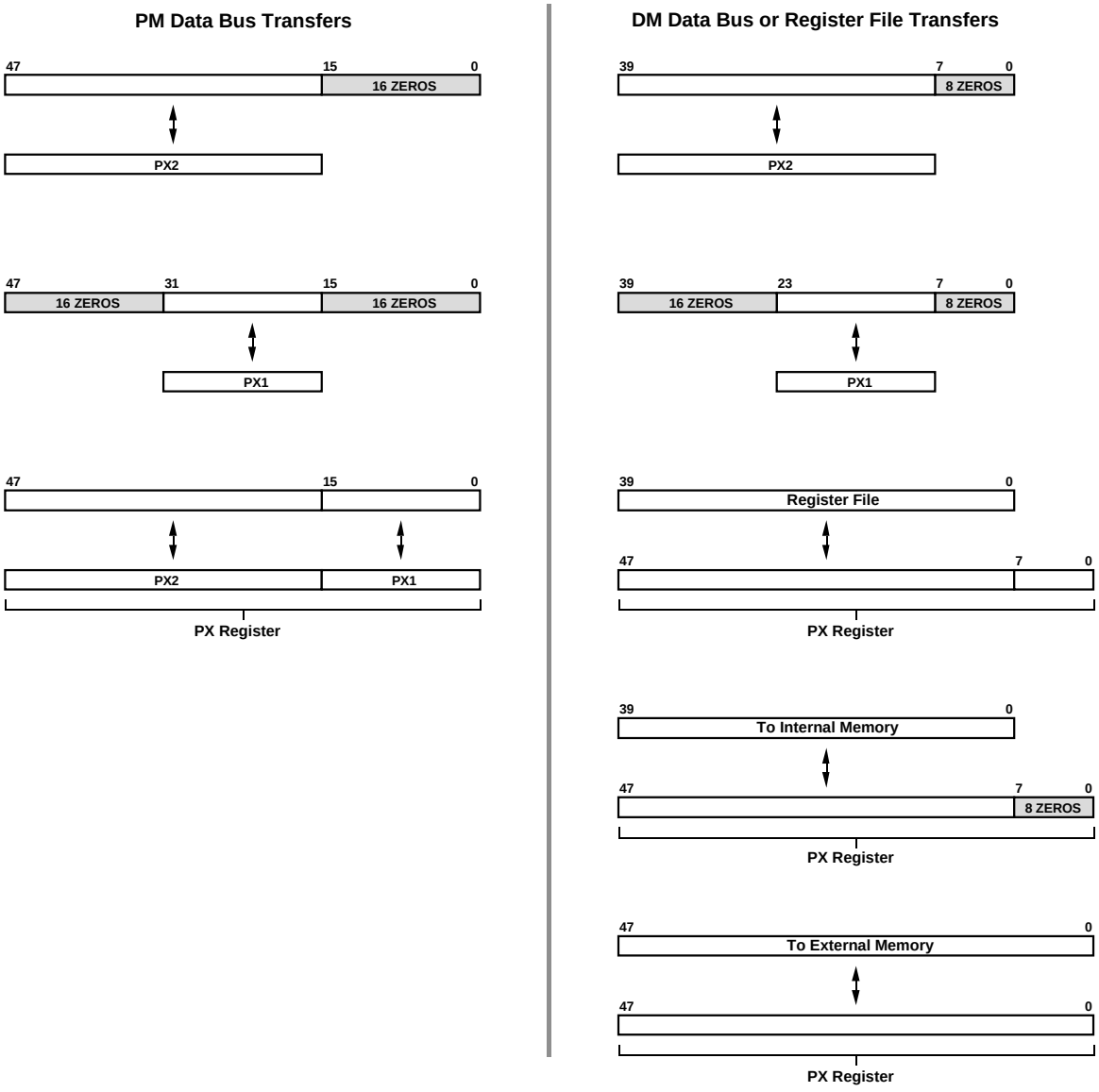


Figure 5.3 PX Register Transfers

When the combined PX register is used for PM Data Bus transfers, the entire 48 bits can be read from or written to program memory. PX2 contains the 32 MSBs of the 48-bit word while PX1 contains the 16 LSBs. (PM Bus data is left-justified in the 48-bit word.)

To write a 48-bit word to the memory location named *Port1* over the PM Data Bus, for example, the following instructions could be used:

```
R0=0x9A00;       /* load R0 with 16 LSBs */
R1=0x12345678;   /* load R1 with 32 MSBs */
PX1=R0;
PX2=R1;
PM(Port1)=PX;    /* write 16 LSBs on PM bus 15-0 */
                 /* and 32 MSBs on PM bus 47-16  */
```

When data is transferred between PX2 and the DM Data Bus or the register file, the upper 32 bits of the DM Data Bus (and register file) are used. On transfers from PX2, the eight LSBs are filled with zeros. (See Figure 5.3.) When data is transferred between PX1 and the DM Data Bus or the register file, bits 23-8 of the DM Data Bus (and register file) are used. On transfers from PX1, bits 7-0 and bits 39-24 are filled with zeros.

When the combined PX register is used for DM Data Bus transfers, the upper 40 bits of PX are read or written. For transfers to or from internal memory, the lower 8 bits are filled with zeroes. For transfers to or from external memory, the entire 48 bits are transferred.

5.1.5 Memory Block Accesses & Conflicts

Any of the ADSP-2106x's three internal buses, PM, DM, and I/O, may need to access one of the memory blocks at any given time. Each block of dual-ported memory can be accessed by both the ADSP-2106x's core processor (over either the PM or DM bus) and by the I/O processor (over the I/O bus) in every cycle—no extra cycles are incurred when both the core and I/O processor access the same block.

A conflict occurs, however, when two core accesses to a single block are attempted in the same cycle, for example over both the PM bus (by the program sequencer or DAG2) and DM bus (by DAG1). When this happens, an extra cycle is incurred—the DM bus access completes first and the PM bus access completes in the following (extra) cycle.

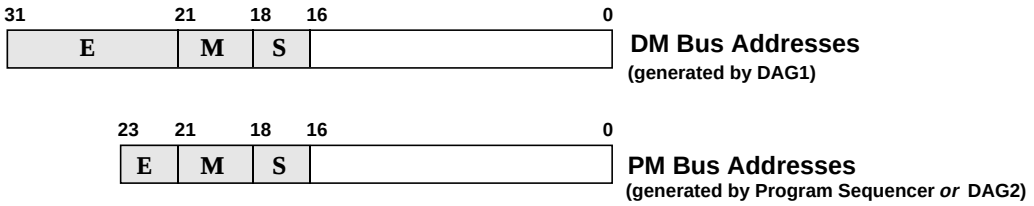
5.2 ADSP-2106x MEMORY MAP

The ADSP-2106x memory map, shown in Figure 5.5, is divided into three sections: internal memory space, multiprocessor memory space, and external memory space. Internal memory space consists of the ADSP-2106x's on-chip memory and resources. Multiprocessor memory space corresponds to the on-chip memory and resources of other ADSP-2106x's in a multiprocessor system. External memory space corresponds to off-chip memory and memory-mapped I/O devices.

The address boundaries of each memory space are:

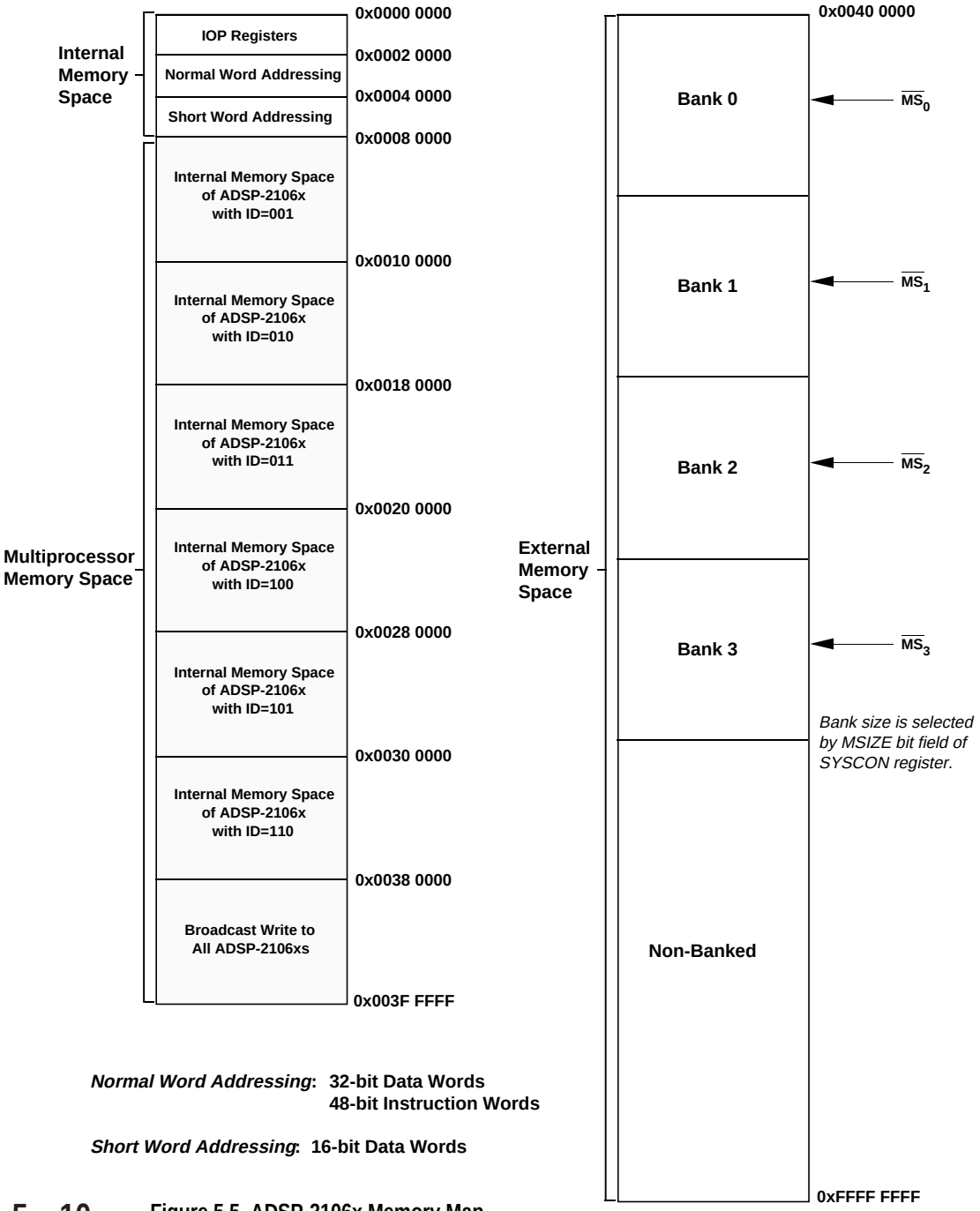
Internal memory	0x0000 0000	to	0x0007 FFFF
Multiprocessor memory	0x0008 0000	to	0x003F FFFF
External memory	0x0040 0000	to	0xFFFF FFFF

Addresses generated by the ADSP-2106x for DM bus and PM bus accesses are shown below in Figure 5.4. DM bus addresses are generated by DAG1, and PM bus addresses are generated either by the ADSP-2106x's program sequencer (for instructions) or by DAG2 (for data).



Note: Off-chip PM bus addresses are MSB-extended with zeros to create 32-bit external bus addresses ($ADDR_{31-0}$).

Figure 5.4 Memory Addresses (E = external, M = Multiprocessor, S = Internal)



Normal Word Addressing: **32-bit Data Words**
48-bit Instruction Words

Short Word Addressing: **16-bit Data Words**

Figure 5.5 ADSP-2106x Memory Map

The ADSP-2106x's I/O processor monitors the addresses of all memory accesses and routes them to the appropriate memory space. The E (external), M (multiprocessing), and S fields are decoded by the I/O processor as shown below. If the E bit field is all zeros, the M and S fields become active and are decoded.

Field	*Value*	*Meaning*
E	non-zero	–Address in external memory
	all zeros	–Address in the processor's own internal memory or in the internal memory of another ADSP-2106x *(M and S activated)*
M	000	–Address in the processor's own internal memory
	non-zero	– M = ID of another ADSP-2106x
	111	–Broadcast write to internal memory of all ADSP-2106xs
S	00	–Address of an IOP register
	01	–Address in Normal Word Addressing space
	1x	–Address in Short Word Addressing space *(x = MSB of short word address)*

5.2.1 ADSP-21060 Internal Memory Space

The internal memory space of the ADSP-21060 is shown in Figure 5.6. This memory has three address regions:

• I/O Processor (IOP) Registers	0x0000 0000	to	0x0000 00FF
• Normal Word Addresses	0x0002 0000	to	0x0003 FFFF
Interrupt Vector Table	*0x0002 0000*	*to*	*0x0002 007F*
• Short Word Addresses	0x0004 0000	to	0x0007 FFFF

The I/O Processor (IOP) Registers are 256 memory-mapped registers that control the system configuration of the ADSP-2106x as well as various I/O operations. The address space between the IOP registers and normal word addresses, locations 0x0000 0100 to 0x0001 FFFF, does not exist as usable memory and should not be written to.

Memory block 0 starts at the beginning of normal word space, at address 0x0002 0000. Block 1 starts at the middle of normal word space, at address 0x0003 0000.

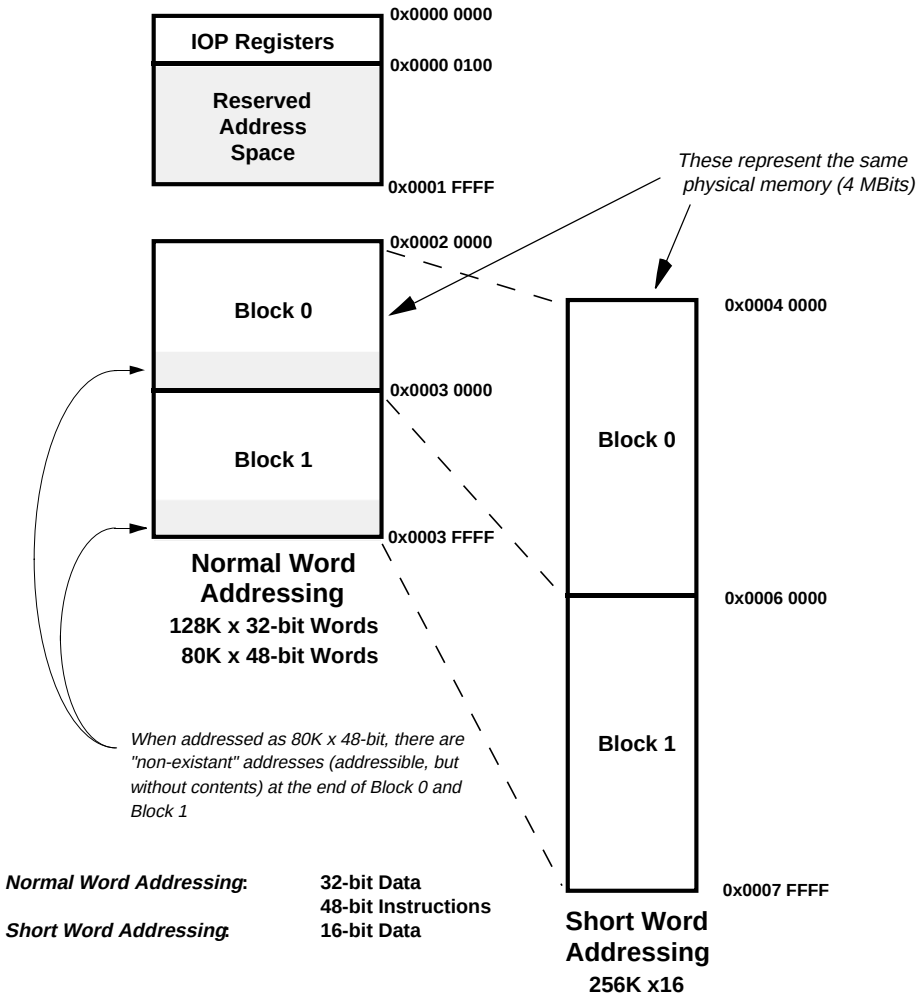


Figure 5.6 ADSP-21060 Internal Memory Space

0x0000 0000 – 0x0000 00FF	IOP Registers *(control/status registers)*
0x0000 0100 – 0x0001 FFFF	*Reserved addresses*
0x0002 0000 – 0x0002 FFFF	Block 0 – Normal Word Addressing *(32-bit, 48-bit words)*
0x0003 0000 – 0x0003 FFFF	Block 1 – Normal Word Addressing *(32-bit, 48-bit words)*
0x0004 0000 – 0x0005 FFFF	Block 0 – Short Word Addressing *(16-bit words)*
0x0006 0000 – 0x0007 FFFF	Block 1 – Short Word Addressing *(16-bit words)*

Table 5.1 ADSP-21060 Internal Memory Addresses

The normal word address space and short word address space actually access *the same physical memory*. For example, the normal word address 0x0002 0000 represents the same locations as short word addresses 0x0004 0000 and 0x0004 0001 (for a 32-bit data access in normal word space).

The ADSP-21060's 4 megabits of on-chip memory can be accessed with either normal word addressing, short word addressing, or combinations of both. The range of normal word addresses, from 0x0002 0000 through 0x0003 FFFF, is exactly 4 megabits when each word is 32 bits wide (128K x 32). (The normal word addressing range also accesses 4 megabits of 48-bit wide instruction words, 80K x 48, but with non-existant addresses at the end of Block 0 and Block 1—see "Internal Memory Organization & Word Size" for further details on physical mapping of 48-bit words and 32-bit words.) The range of short word addresses, from 0x0004 0000 through 0x0007 FFFF, is also exactly 4 megabits (256K x 16).

Using normal word addressing, each 2-Mbit block of memory contains 64K addressable locations (for 32-bit data words). Using short word addressing, each 2-Mbit block contains 128K addressable locations.

Normal word and short word addresses can be generated on all three on-chip buses: DM, PM, and I/O. Short word addresses only occur on the I/O bus when an external device is reading or writing to the ADSP-2106x's internal memory, and not for DMA operations.

Short word addressing increases the amount of 16-bit data that can be stored in internal memory, and also allows MSW (most significant word) and LSW (least significant word) addressing of 32-bit data words. Short word addressing of 16-bit data words is useful in array signal processing systems. The 16-bit short words are extended into 32-bit integers when they are read from memory, and may be either sign-extended or zero-filled (as determined by the SSE bit in the MODE1 register).

5 Memory

The ADSP-2106x's interrupt vector table is located at the start of normal word addressing, 0x0002 0000 – 0x0002 007F, when the processor is booted from an external source (EPROM, host port, or link port booting). If the processor is in "no boot" mode, the interrupt vector table is located in external memory, 0x0040 0000 to 0x0040 007F. If the IIVT bit of the SYSCON register is set, the interrupt table resides in internal memory regardless of booting mode.

5.2.2 ADSP-21062 Internal Memory Space

The ADSP-21062 is a memory-variant version of the ADSP-21060. The two processors include the following amounts of on-chip SRAM:

Processor	*Total Memory*	*Maximum Data Memory*	*Maximum Program Memory*
ADSP-21060	4 Mbits	128K x 32	80K x 48
ADSP-21062	2 Mbits	64K x 32	40K x 48

The on-chip memory of the ADSP-21062 is divided into two equal blocks, Block 0 and Block 1, in the same way as the ADSP-21060's. The ADSP-21062's multiprocessor memory space and external memory space are exactly the same as that of the ADSP-21060.

On the ADSP-21062, Block 0 starts at normal word address 0x0002 0000. Block 1 starts at normal word address 0x0002 8000. The memory map for the ADSP-21062's 2 Mbits of internal memory is shown in Figure 5.7a and in Table 5.2a below. The *Block 1 Alias* address ranges will access the actual Block 1, 0x0002 8000 – 0x0002 FFFF in normal word address space and 0x0005 0000 – 0x0005 FFFF in short word address space.

0x0000 0000 – 0x0000 00FF	IOP Registers (control / status registers)
0x0000 0100 – 0x0001 FFFF	*Reserved addresses*
0x0002 0000 – 0x0002 7FFF	Block 0 – Normal Word Addressing
0x0002 8000 – 0x0002 FFFF	Block 1 – Normal Word Addressing
0x0003 0000 – 0x0003 7FFF	Alias of Block 1 *(i.e. accesses Block 1)* – Normal Word Addressing
0x0003 8000 – 0x0003 FFFF	Alias of Block 1 *(i.e. accesses Block 1)* – Normal Word Addressing
0x0004 0000 – 0x0004 FFFF	Block 0 – Short Word Addressing
0x0005 0000 – 0x0005 FFFF	Block 1 – Short Word Addressing
0x0006 0000 – 0x0006 FFFF	Alias of Block 1 *(i.e. accesses Block 1)* – Short Word Addressing
0x0007 0000 – 0x0007 FFFF	Alias of Block 1 *(i.e. accesses Block 1)* – Short Word Addressing

Table 5.2a ADSP-21062 Internal Memory Addresses

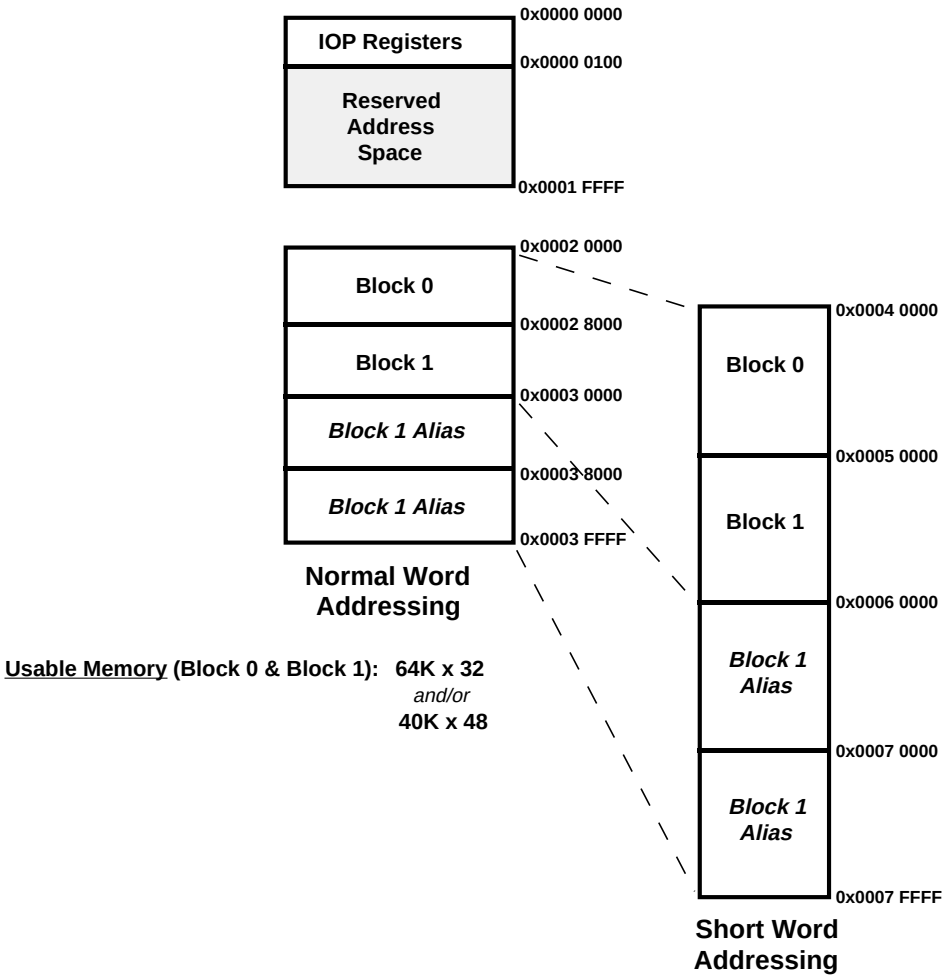


Figure 5.7a ADSP-21062 Internal Memory Space

5.2.3 ADSP-21061 Internal Memory Space

The ADSP-21061 is a memory-variant version of the ADSP-21060. The two processors include the following amounts of on-chip SRAM:

Processor	*Total Memory*	*Maximum Data Memory*	*Maximum Program Memory*
ADSP-21060	4 Mbits	128K x 32	80K x 48
ADSP-21062	1 Mbits	32K x 32	16K x 48

The on-chip memory of the ADSP-21061 is divided into two equal blocks, Block 0 and Block 1, in the same way as the ADSP-21060's. The ADSP-21061's multiprocessor memory space and external memory space are exactly the same as that of the ADSP-21060.

On the ADSP-21061, Block 0 starts at normal word address 0x0002 0000. Block 1 starts at normal word address 0x0002 4000. The memory map for the ADSP-21061's 1 Mbit of internal memory is shown in Figure 5.7b and in Table 5.2b below. The *Block 1 Alias* address ranges will access the actual Block 1, 0x0002 4000 – 0x0002 7FFF in normal word address space and 0x0004 8000 – 0x0004 FFFF in short word address space.

0x0000 0000 – 0x0000 00FF	IOP Registers (control / status registers)
0x0000 0100 – 0x0001 FFFF	*Reserved addresses*
0x0002 0000 – 0x0002 3FFF	Block 0 – Normal Word Addressing
0x0002 4000 – 0x0002 7FFF	Block 1 – Normal Word Addressing
0x0002 8000 – 0x0002 BFFF	Alias of Block 1 *(i.e. accesses Block 1)* – Normal Word Addressing
0x0002 C000 – 0x0002 FFFF	Alias of Block 1 *(i.e. accesses Block 1)* – Normal Word Addressing
0x0003 0000 – 0x0003 3FFF	Alias of Block 1 *(i.e. accesses Block 1)* – Normal Word Addressing
0x0003 4000 – 0x0003 7FFF	Alias of Block 1 *(i.e. accesses Block 1)* – Normal Word Addressing
0x0003 8000 – 0x0003 BFFF	Alias of Block 1 *(i.e. accesses Block 1)* – Normal Word Addressing
0x0003 C000 – 0x0003 FFFF	Alias of Block 1 *(i.e. accesses Block 1)* – Normal Word Addressing
0x0004 0000 – 0x0004 7FFF	Block 0 – Short Word Addressing
0x0004 8000 – 0x0004 FFFF	Block 1 – Short Word Addressing
0x0005 0000 – 0x0005 7FFF	Alias of Block 1 *(i.e. accesses Block 1)* – Short Word Addressing
0x0005 8000 – 0x0005 FFFF	Alias of Block 1 *(i.e. accesses Block 1)* – Short Word Addressing
0x0006 0000 – 0x0006 7FFF	Alias of Block 1 *(i.e. accesses Block 1)* – Short Word Addressing
0x0006 8000 – 0x0006 FFFF	Alias of Block 1 *(i.e. accesses Block 1)* – Short Word Addressing
0x0007 0000 – 0x0007 7FFF	Alias of Block 1 *(i.e. accesses Block 1)* – Short Word Addressing
0x0007 8000 – 0x0007 FFFF	Alias of Block 1 *(i.e. accesses Block 1)* – Short Word Addressing

Table 5.2b ADSP-21061 Internal Memory Addresses

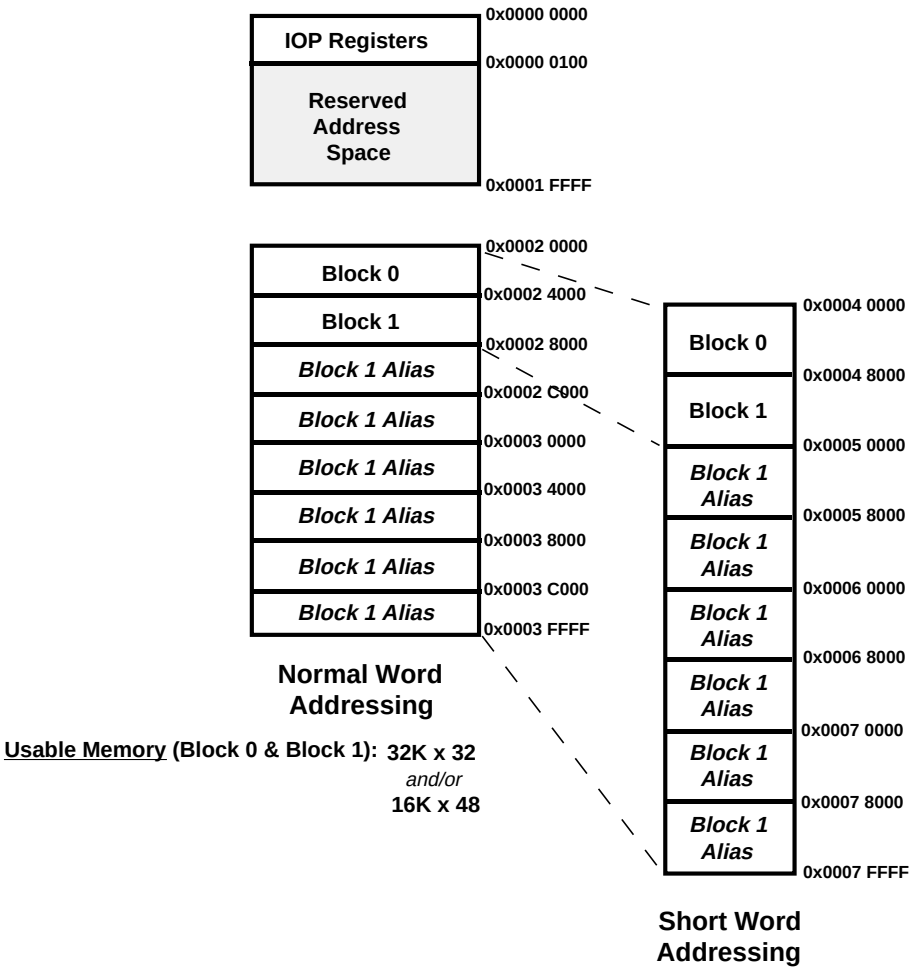


Figure 5.7b ADSP-21061 Internal Memory Space

5.2.4 Porting Code from ADSP-21060 to ADSP-21062 or ADSP-21061

To ease porting code between ADSP-2106x processor, a system for aliasing memory Block 1 eliminates the need to re-arrange (some) code placement. For example, memory Block 0 on the ADSP-21062 starts at the beginning of internal memory, normal word address 0x0002 0000. Block 1 on the ADSP-21062 starts at the end of Block 0, with contiguous addresses. The remaining addresses in internal memory are divided into blocks, which alias into Block 1. This aliasing allows any code or data stored in Block 1 on the ADSP-21060 to retain the same addresses on the ADSP-21062—these addresses will alias into the actual Block 1 of each processor.

A similar aliasing structure is built into the ADSP-21061. For more information on aliasing, see the memory maps for the ADSP-21061 and ADSP-21062 processors.

5.2.5 Multiprocessor Memory Space

Multiprocessor memory space maps to the internal memory of other ADSP-2106xs in a multiprocessor system. This allows each ADSP-2106x to access the internal memory and memory-mapped IOP registers of the other processors.

As shown in Figure 5.5, when the E field of an address is zero and the M field is non-zero, the address falls within multiprocessor memory space. The value of M specifies the processor ID_{2-0} of the external ADSP-2106x being accessed, and only that processor will respond to the read / write cycle. If M=111, however, a *broadcast write* is performed to all processors. All of the processors react to this address as if their individual ID_{2-0} was being used, enabling the write to their internal memory.

Instead of directly accessing its own internal memory, an ADSP-2106x can also access its memory through the multiprocessor memory space by using its own ID. In this case the processor simply reads or writes to its own internal memory and does not attempt an access on the external system bus. (Note that these self-accesses through multiprocessor memory space may only be accomplished with core-processor-generated addresses, not DMA-controller-generated addresses.)

If both the E and M fields of an address on the external bus are equal to zero, the address will be ignored unless the processor ID is also zero

($M=ID_{2-0}=000$). Addresses with $M=ID_{2-0}=000$ are only allowed in single-processor systems.

If the ADSP-2106x attempts to access an invalid address in multiprocessor memory space, data written will be ignored and reads will return invalid data.

For additional information about multiprocessor memory accesses, see "Direct Reads & Writes" and "Data Transfers Through The EPBx Buffers" in the *Multiprocessing* chapter of this manual.

5.2.6 External Memory Space

External memory can be accessed over the ADSP-2106x's DM bus, PM bus, and EP bus, all via the external port. The processor's DAG1, program sequencer (and DAG2), and IOP control these respective buses.

32-bit addresses are generated by DAG1 and the IOP over the DM address bus and I/O address bus, allowing addressing of the complete 4-gigaword memory map. The program sequencer and DAG2 generate 24-bit addresses over the PM address bus, limiting addressing to the low 12 megawords (0x0040 0000 to 0x00FF FFFF).

5.2.7 Memory Space Access Restrictions

The ADSP-2106x's three internal buses, PM, DM, and I/O, can be used to access the processor's memory map according to the following rules:

- The DM bus can access all memory spaces.
- The PM bus can access only Internal Memory Space and the lowest 12 megawords of External Memory Space.
- The I/O bus can access all memory spaces except for the memory-mapped IOP registers (in Internal Memory Space).

➠ Note that in silicon revision 1.0 and earlier pre-modify addressing operations must not change the memory space of the address; for example, pre-modification of an address in Internal Memory Space should not generate an address in External Memory Space. The one exception to this rule is: an indirect JUMP or CALL instruction with pre-modify addressing *can* jump from internal memory to external memory. Silicon revisions 2.x and later do not have this pre-modify limitation.

5 Memory

5.3 INTERNAL MEMORY ORGANIZATION & WORD SIZE

The ADSP-2106x's internal SRAM memory accommodates the following word types:

- 48-bit instructions
- 32-bit floating-point data
- 16-bit short word data

40-bit extended-precision floating-point data values are also accommodated, but are accessed in 48-bit words. The 40 bits are left-justified in the 48-bit word (bits 47-8).

When the ADSP-2106x processor core accesses its internal memory, the word width of the access is determined according to the following rules:

- Instruction fetches always read 48-bit words
- Read / writes using normal word addressing are either 32-bit words or 48-bit words, depending on how the block of memory is configured in the SYSCON register.
- Read / writes using short word addressing are always 16-bit words
- PM bus (DAG 2) read / writes of the PX register are always 48-bit words (unless they use short word addressing)
- DM bus (DAG 1) read / writes of the PX register are always 40-bit words (unless they use short word addressing)

An ADSP-2106x program should not attempt to access the same physical location in memory as a 32-bit word and as a 48-bit word. The internal SRAM employs a write-back scheme that will cause errors if this occurs.

5.3.1 32-Bit Words & 48-Bit Words

Each 2-Mbit block of ADSP-21060 memory is physically organized as 16 columns, each 16 bits wide, with a height of 8K. (On the ADSP-21062, each 1-Mbit block of memory is similarly organized but with each column having a height of 4K.) 48-bit instruction words require three columns of contiguous memory and 32-bit data words require two contiguous columns.

When an address is applied to memory for a read or write, the particular columns selected depends upon the word width of the access. For 48-bit words, the 16-bit columns are selected in groups of three. In a memory block consisting entirely of 48-bit instruction words,

16 columns ÷ 3 columns per group = 5 groups

there are 5 groups to select from and the 16th column is unused. Thus, an ADSP-21060 2-Mbit memory block that consists entirely of 48-bit words provides

8K × 5 groups = 40K words

of instruction storage. For 32-bit data words, the columns are selected in groups of two. In a memory block consisting entirely of 32-bit words,

16 columns ÷ 2 columns per group = 8 groups

there are 8 words to select from with no columns unused. Thus, an ADSP-21060 2-Mbit memory block that consists entirely of 32-bit words provides

8K × 8 groups = 64K words

of data storage.

Because the memory on the ADSP-21061 is arranged in eight 16-bit columns, a similar set of calculations for this processor yields the following:

4K × 2 groups = 8K words (of instruction storage)

4K × 4 groups = 16K words (of data storage)

Figure 5.8 shows the ordering of 16-bit words within 48-bit words and 32-bit words, and also shows initial addresses for each column of ADSP-21060 memory. Figure 5.9a shows the same information for the ADSP-21062, and Figure 5.9b shows this information for the ADSP-21061.

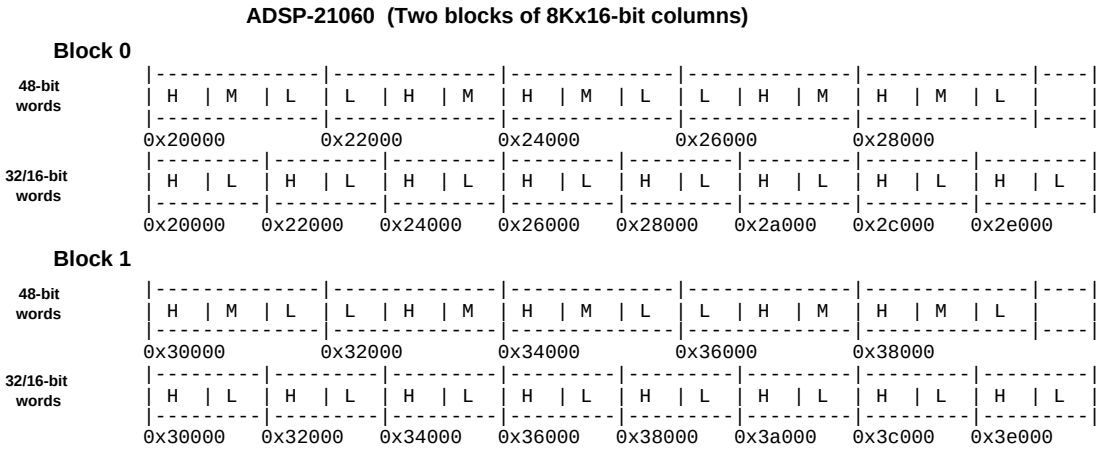


Figure 5.8 Memory Organization vs. Address (ADSP-21060)

Notes: All addresses denote the first location of each column.

"Non-existant" 48-bit addresses occur when a block is filled with 48-bit instructions. Because there is a set number of addresses per block (which does not vary with the size of the word at the address), you can end up with a range of 48-bit "non-existant" addresses (addressable, but having no contents) at the end of each block. This memory arrangement feature applies to all ADSP-2106x processors (shown in Figures 5.8, 5.9a, and 5.9b).

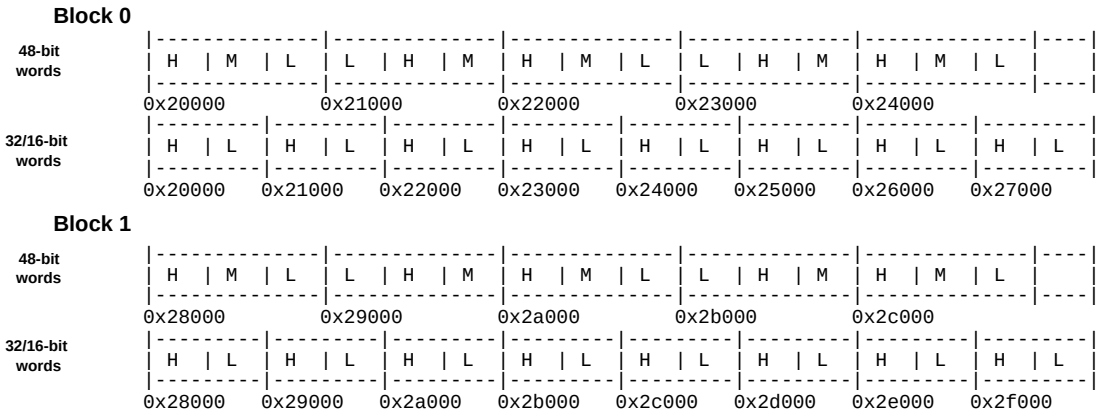


Figure 5.9a Memory Organization vs. Address (ADSP-21062)
Note: All addresses denote the first location of each column.

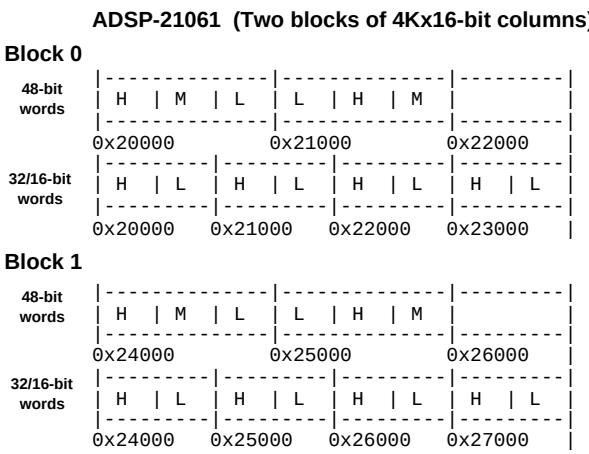


Figure 5.9b Memory Organization vs. Address (ADSP-21061)
Note: All addresses denote the first location of each column.

5.3.2 Mixing 32-Bit & 48-Bit Words In One Memory Block

32-bit data words and 48-bit instruction words can be stored in the same memory block, with the restriction that *all instructions must reside at addresses lower than the data.* No instruction may be stored at an address higher than the lowest address of any data word. This restriction is necessary to prevent addresses for 32-bit words and 48-bit words from overlapping.

The rules for combining 48-bit instruction words and 32-bit data words within the same block of memory are as follows:

- Instruction storage must start at the lowest address in the block.
- Data storage must start on an even column number
- All data must be located at addresses higher than all instructions
- Instructions require three contiguous 16-bit columns
- Data words require two contiguous 16-bit columns

5.3.3 Basic Examples Of Mixed 32-Bit & 48-Bit Words

Each block of memory is physically organized as 16 columns, each 16 bits wide, with a height of 8K on the ADSP-21060 and 4K on the ADSP-21062. Figure 5.10 illustrates four basic combinations of mixed 48-bit instructions and 32-bit data within a single block:

A. 3 columns for instructions, 1 unused column, and 12 columns for data. This provides 8K of instruction storage and 48K of data storage on the ADSP-21060 (4K of instruction storage and 24K of data storage on the ADSP-21062). One column is unused because the 32-bit data words must start on an even column number (in this case column 4).

{Columns one through eight on this example apply to the ADSP-21061.}

B. 6 columns for instructions and 10 columns for data. This provides 16K of instruction storage and 40K of data storage on the ADSP-21060 (8K of instruction storage and 20K of data storage on the ADSP-21062).

{Columns one through eight on this example apply to the ADSP-21061.}

C. 9 columns for instructions, 1 unused column, and 6 columns for data. This provides 24K of instruction storage and 24K of data storage on the ADSP-21060 (12K of instruction storage and 12K of data storage on the ADSP-21062). One column is unused because the 32-bit data words must start on an even column number (in this case column 10).

{Because there are only eight columns on the ADSP-21061, this example does not apply to the ADSP-21061.}

D. 12 columns for instructions and 4 columns for data. This provides 32K of instruction storage and 16K of data storage on the ADSP-21060 (16K of instruction storage and 8K of data storage on the ADSP-21062).

{Because there are only eight columns on the ADSP-21061, this example does not apply to the ADSP-21061.}

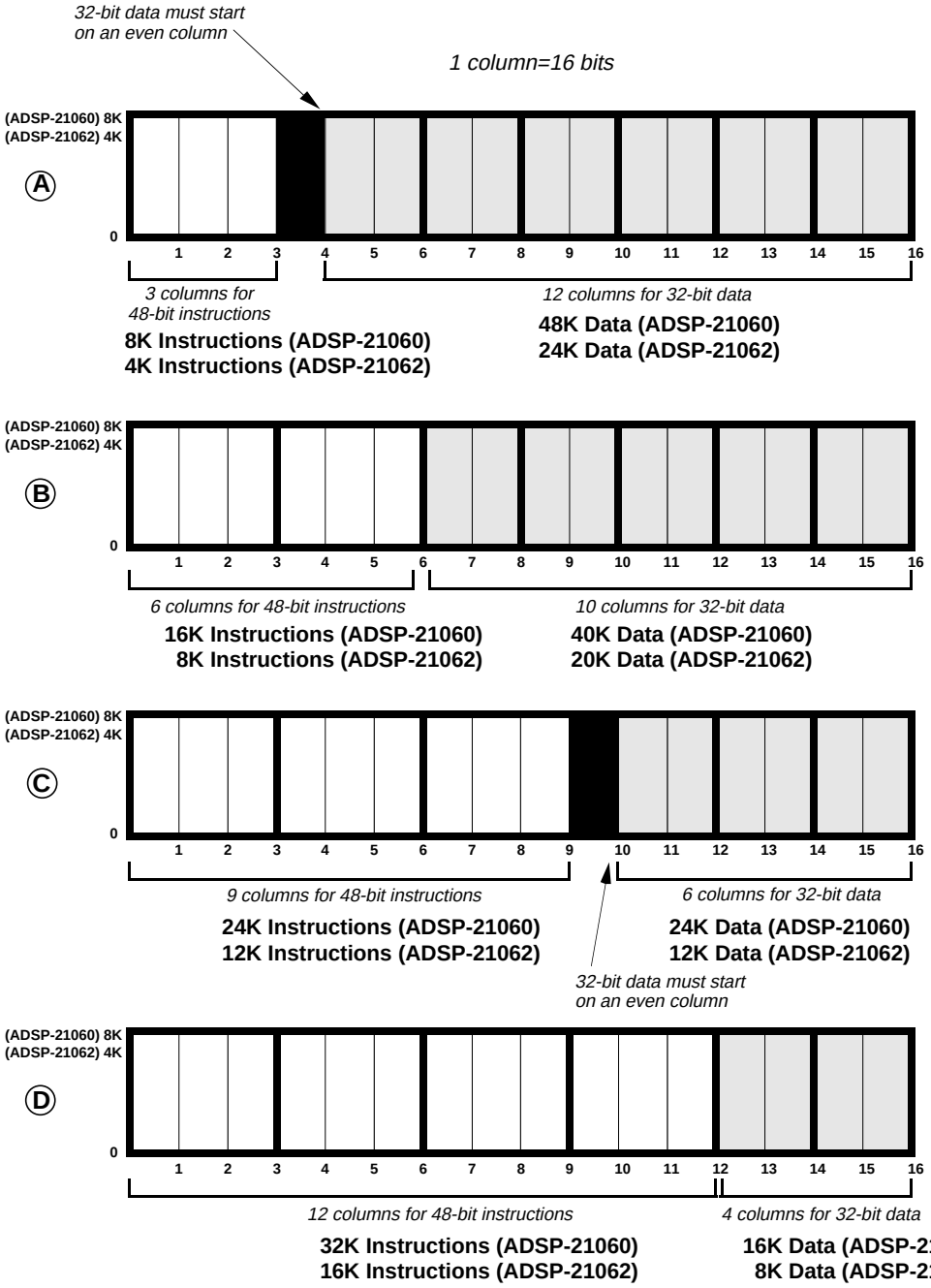


Figure 5.10 Basic Examples of Mixed Instructions & Data In A Memory Block

Table 5.3 shows the addressing in Block 0 (beginning address = 0x0002 0000) for each of the instruction and data combinations of Figure 5.10, on the ADSP-21060:

	48-Bit Instructions		*32-Bit Data*	
	start address	*end address*	***start address***	*end address*
A.	0x0002 0000	0x0002 1FFF	**0x0002 4000**	0x0002 FFFF
B.	0x0002 0000	0x0002 3FFF	**0x0002 6000**	0x0002 FFFF
C.	0x0002 0000	0x0002 5FFF	**0x0002 A000**	0x0002 FFFF
D.	0x0002 0000	0x0002 7FFF	**0x0002 C000**	0x0002 FFFF

Table 5.3 Address Ranges For Instructions & Data (ADSP-21060)

To determine the starting address of the 32-bit data, the following equations are used (for the ADSP-21060):

i	*Starting Address of 32-Bit Data*
0	B + 8K + m + 1
1	B + 16K + m + 1
2	B + 32K + m + 1
3	B + 40K + m + 1

B= beginning address of memory block
n= number of 48-bit instruction word locations
i= integer portion of [(n – 1) ÷ 8192]
m= (n – 1) mod 8192

Table 5.4 shows the addressing in Block 0 (beginning address = 0x0002 0000) for each of the instruction and data combinations of Figure 5.10, on the ADSP-21062:

	48-Bit Instructions		*32-Bit Data*	
	start address	*end address*	***start address***	*end address*
A.	0x0002 0000	0x0002 0FFF	**0x0002 2000**	0x0002 7FFF
B.	0x0002 0000	0x0002 1FFF	**0x0002 3000**	0x0002 7FFF
C.	0x0002 0000	0x0002 2FFF	**0x0002 5000**	0x0002 7FFF
D.	0x0002 0000	0x0002 3FFF	**0x0002 6000**	0x0002 7FFF

Table 5.4 Address Ranges For Instructions & Data (ADSP-21062)

To determine the starting address of the 32-bit data, the following equations are used (for the ADSP-21062 and ADSP-21061):

i	*Starting Address of 32-Bit Data*
0	B + 4K + m + 1
1	B + 8K + m + 1
2	B + 16K + m + 1
3	B + 20K + m + 1

B = beginning address of memory block
n = number of 48-bit instruction word locations
i = integer portion of [(n – 1) ÷ 4096]
m = (n – 1) mod 4096

5.3.4 16-Bit Short Words

Normal word addressing is used for accesses of 32-bit or 48-bit words. All instruction fetches and 32-bit data accesses are accomplished with normal word addresses. Short word addresses can be used, however, to access 16-bit data. Short word addressing increases the amount of 16-bit data that can be stored in internal memory, and also allows MSW (most significant word) and LSW (least significant word) addressing of 32-bit words. Bit 0 of the address selects between the MSW and LSW of the 32-bit word.

A single location in memory (i.e. the lower 16 bits of a 32-bit word) can be accessed in two ways: with a normal word address or a short word address. The short word address is a left shift of the corresponding normal word address. This allows easy conversion between short word address and normal word address for the same physical location. Figure 5.11 shows how the short word addresses are related to normal word addresses for 32-bit words. (Figures 5.9 and 5.10 show how these addresses are related to normal word addresses for 48-bit words.) Note that the 16-bit data words are transferred over lines 31-16 of the internal PM Data Bus and DM Data Bus as well as the external bus ($DATA_{47-0}$)

Arithmetically shifting a short word address to the right by one bit produces the corresponding normal word address. Arithmetically shifting a normal word address to the left produces the short word address of the LSW of the 32-bit normal word. To generate the short word address of the MSW, the left shift is performed and bit 0 is then set to 1.

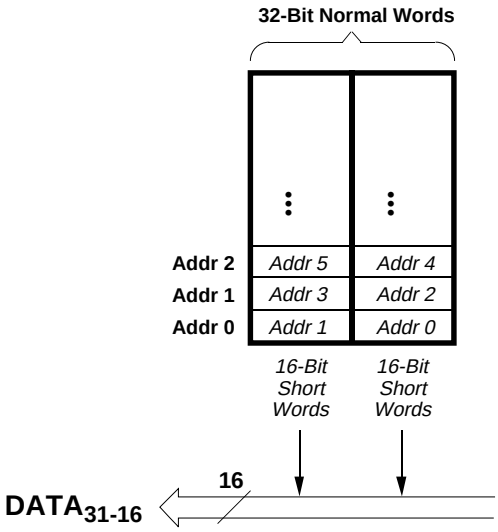


Figure 5.11 Short Word Addresses

16-bit short words read into ADSP-2106x registers are automatically extended into 32-bit integers. The upper 16 bits can be zero-filled or sign-extended, as determined by the value of the SSE bit in the MODE1 register. If SSE=0, the upper 16 bits are zero-filled. If SSE=1, the upper 16 bits are sign-extended (except when reading a short word into the PX register, which is always zero-filled).

5.3.5 Mixing 32-Bit & 48-Bit Words With Finer Granularity

If 48-bit instructions and 32-bit data words must be mixed with a finer granularity than the basic combinations described above, an in-depth understanding of the ADSP-2106x's internal memory is required. The following sections describe in detail the low-level organization and addressing of the internal memory blocks.

5.3.5.1 Low-Level Physical Mapping Of Memory Blocks

Each block of memory is organized as 16 columns. On the ADSP-21060, each column contains 8K 16-bit words; on the ADSP-21062, each column contains 4K 16-bit words. For reads or writes of 48-bit and 32-bit words, the 13 LSBs of the address select a row from each column. The MSBs of the address control which columns are selected. For reads or writes of 16-bit short words, the address is right-shifted one place before being applied to memory (see Figure 5.12). This allows bit 0 of the address to be used to select between the MSW and LSW of 32-bit data.

When a block is memory is accessed, how many and which columns are selected depends upon the word width of the access. For 48-bit words, the 16-bit columns are selected in groups of three and address bits 13-15 determine which group is selected. For 32-bit words, the columns are selected in groups of two and address bits 13-15 also select the group.

16-bit short word accesses are handled in a slightly different fashion, in order to provide easy access to the MSW and LSW of 32-bit data. In the ADSP-2106x's data address generators (DAGs), a single arithmetic right shift of the short word address gives the physical address of the 32-bit word being written to. If the bit shifted out is zero, the access is to the LSW, otherwise it is to the MSW. This is implemented by selecting columns in groups of two with address bits 13-15 and then selecting between the two columns in the group with the short word address bit shifted out.

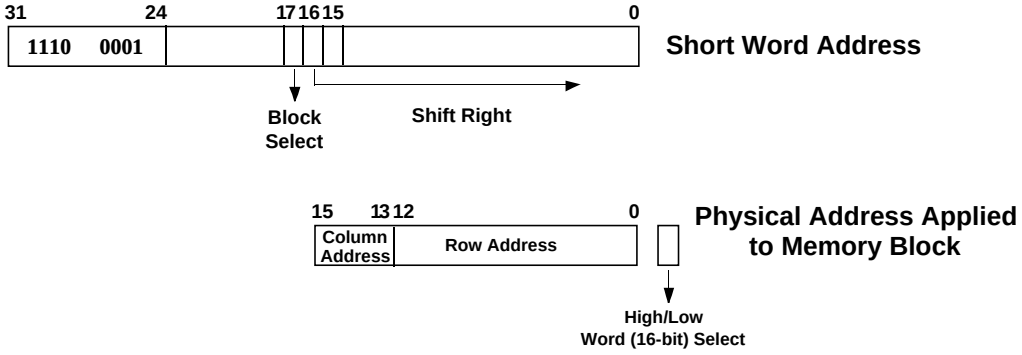


Figure 5.12 Preprocessing of 16-Bit Short Word Addresses

5.3.5.2 Placement Restrictions For Mixed 32-Bit & 48-Bit Words

32-bit and 48-bit words are grouped differently within a memory block and try to use the same address area. This may cause errors when mixing 48-bit instructions and 32-bit data within the same block. (Since 32-bit and 16-bit words use the same grouping structure and different addresses, they can be freely mixed within a memory block.) The overall guideline for placement of mixed word sizes is that all 48-bit instructions must reside at addresses lower than all 32-bit data. This restriction is necessary to prevent addresses for instructions and data from overlapping.

Figure 5.13 shows how the 48-bit words fill a memory block and exactly where 32-bit words can be placed, for the ADSP-21060. Figure 5.14 shows the equivalent information for the ADSP-21062. If the number of 48-bit word locations to be allocated is *n* and the beginning address of the block is *B*, the address where *contiguous* 32-bit data may begin can be determined by Table 5.5:

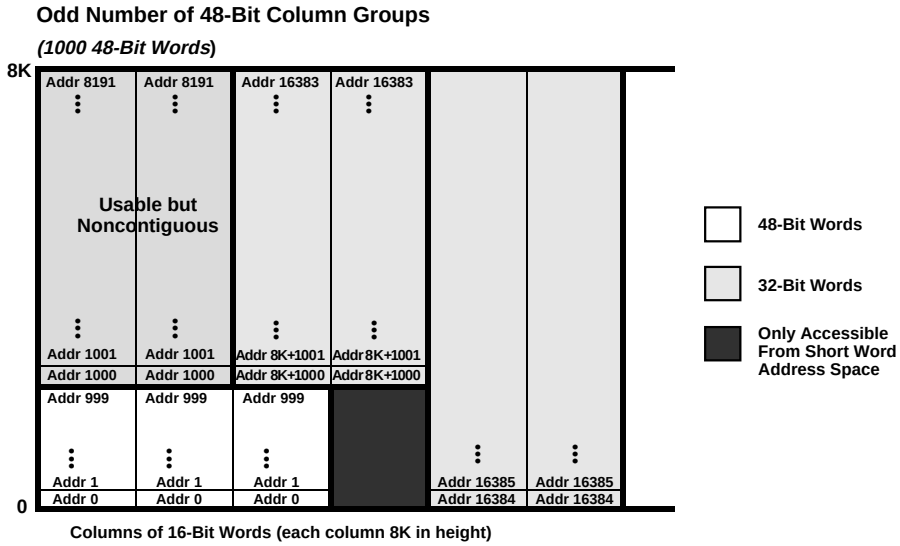
(n – 1) ÷ 8192	*Starting Address for Contiguous 32-Bit Data*	*Noncontiguous Address Range of Memory Block*
0	B + 8K + m + 1	(B + n) *to* (8K – 1)
1	B + 16K + m + 1	—
2	B + 32K + m + 1	(B + 24K + n) *to* (32K – 1)
3	B + 40K + m + 1	—
4	B + 56K + m + 1	(B + 48K + n) *to* (56K – 1)

m=(n – 1) mod 8192

Table 5.5 Starting Address for Contiguous 32-Bit Data (ADSP-21060)

Figure 5.13 also shows that when an odd number of 3-column groups are allocated for 48-bit words (i.e. one, three, or five 3-column groups), a usable but discontiguous block of 32-bit memory will exist. This is also specified in Table 5.5.

To fully use all of the memory block, 48-bit words should be allocated in 16K word increments (i.e. six columns). Even when all memory is used, there will exist a range of addresses between the 48-bit word region and the *contiguous* 32-bit word region that do not access any valid word. Any 48-bit write to this non-valid region will corrupt 32-bit data, and any 32-bit write will corrupt 48-bit data.



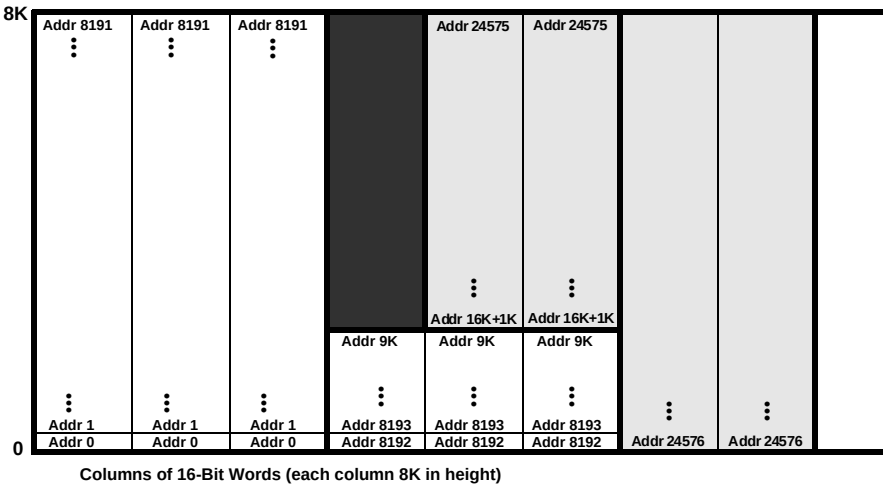


Figure 5.13 48-Bit Words & 32-Bit Words Mixed In A Memory Block (ADSP-21060)

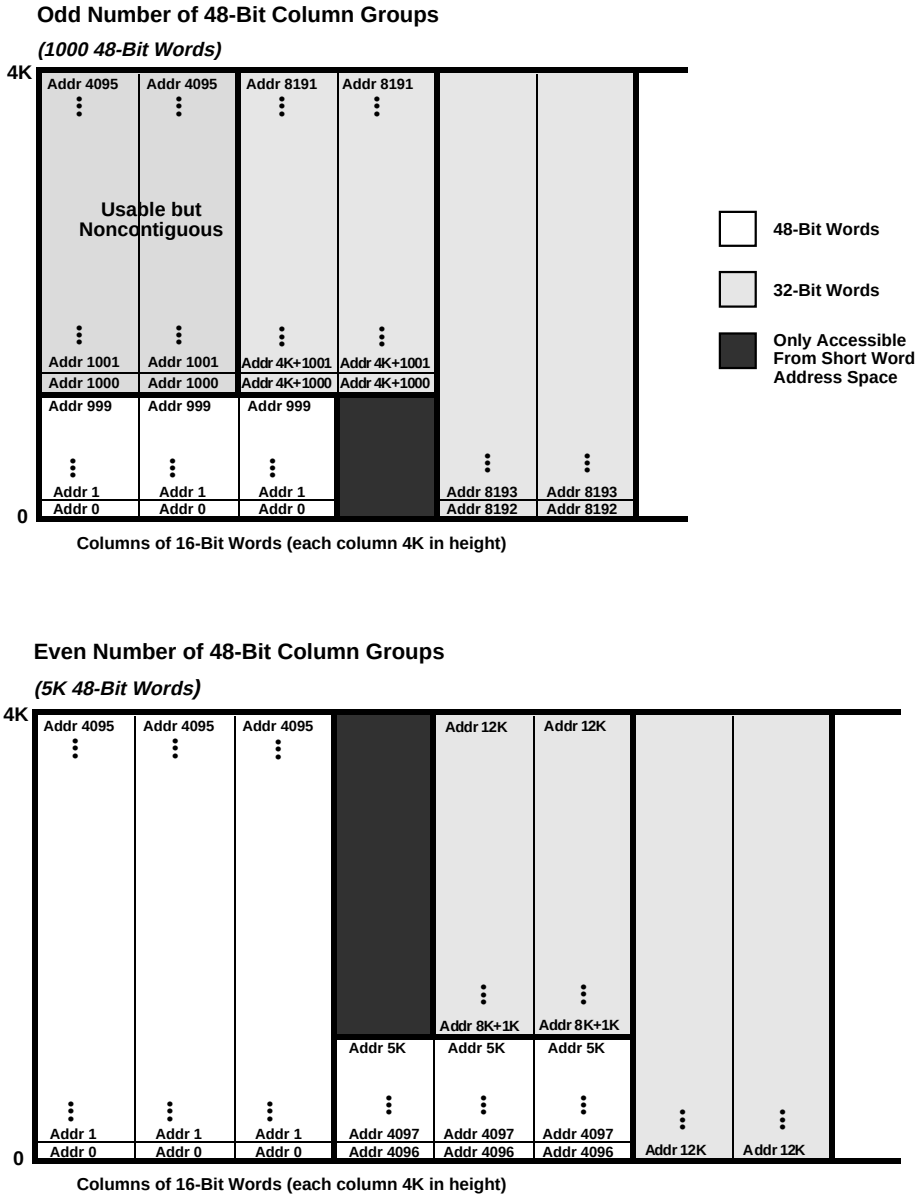


Figure 5.14 48-Bit Words & 32-Bit Words Mixed In A Memory Block (ADSP-21062 or ADSP-21061)

To determine, however, exactly which addresses are valid again requires an analysis of how the data is placed in memory. The simplest solution is to think of the 16-bit words as being mapped into 32-bit word space and allocate memory with the same method described above for 32-bit words.

Figure 5.14 shows (for the ADSP-21062 or the ADSP-21061) how the 48-bit words fill a memory block and exactly where 32-bit words can be placed. If the number of 48-bit word locations to be allocated is *n* and the beginning address of the block is *B*, the address where *contiguous* 32-bit data may begin can be determined by Table 5.6:

(n – 1) ÷ 4096	*Starting Address for Contiguous 32-Bit Data*	*Noncontiguous Address Range of Memory Block*
0	B + 4K + m + 1	(B + n) *to* (4K – 1)
1	B + 8K + m + 1	—
2	B + 16K + m + 1	(B + 12K + n) *to* (16K – 1)
3	B + 20K + m + 1	—
4	B + 28K + m + 1	(B + 24K + n) *to* (28K – 1)

m=(n – 1) mod 4096

Table 5.6 Starting Address for Contiguous 32-Bit Data (ADSP-21062 or ADSP-21061)

5.3.5.3 Shadow Write FIFO

Because the ADSP-2106x's internal memory must operate at high speeds, writes to the memory do not go directly into the memory array, but rather to a two-deep FIFO called the *shadow write FIFO*.

When an internal memory write cycle occurs, data in the FIFO from the previous write is loaded into memory and the new data goes into the FIFO. This operation is normally transparent, since any reads of the last two locations written are intercepted and routed to the FIFO. There is only one case in which you need to be aware of the shadow write FIFO: mixing 48-bit and 32-bit word accesses to the same locations in memory.

The shadow FIFO cannot differentiate between the mapping of 48-bit words and mapping of 32-bit words. (See Figures 5.8 and 5.9.) Thus if you write a 48-bit word to memory and then try to read the data with a 32-bit word access, the shadow FIFO will not intercept the read and incorrect data will be returned.

If 48-bit accesses and 32-bit accesses to the *same* locations absolutely must be mixed in this way, you must flush out the shadow FIFO with two dummy writes before attempting to read the data.

5.3.6 Configuring Memory For 32-Bit or 40-Bit Data

Each block of internal memory can be configured to store either single-precision 32-bit data or extended-precision 40-bit data. This configuration is selected by setting or clearing the IMDW0 and IMDW1 bits in the SYSCON register. If the IMDWx bit is equal to zero, 32-bit data is selected; when a data access occurs, a 32-bit access is performed. If the IMDWx bit is equal to one, 40-bit data is selected; when a data access occurs, a 48-bit access is performed.

If an ADSP-2106x program attempts to write 40-bit data (in a 48-bit word) to a memory block configured for 32-bit data, the lower 16 bits (of the 48-bit word) are truncated. If a 40-bit data read is attempted, the lower 8 bits will be zeros. The PX register is the only exception to these rules—all read / writes of the PX register are performed as 48-bit accesses. If any 40-bit data must be stored in a memory block configured for 32-bit words, the PX register should be used to access the 40-bit data in 48-bit words. For 48-bit writes of this kind, from the PX register to 32-bit memory, be sure that the physical memory space of the 48-bit destination does not corrupt any 32-bit data.

Changing the value of the IMDWx bits during system operation is possible, but be aware that any kind of memory access will be affected. This includes ADSP-2106x–to–ADSP-2106x direct read / writes, host processor–to–ADSP-2106x direct read / writes, DMA transfers, and interrupt data areas.

(Note that the word width of data accesses is not related to the value of the arithmetic precision mode bit, RND32. This allows the occasional use of 32-bit data in extended-precision 40-bit systems, without having to toggle the value of RND32 in your program.)

Because the ADSP-2106x's memory blocks must be configured for either 32-bit or 40-bit data, DMA transfers automatically read or write the proper word width. This simplifies setting up DMA channels for a system. DMA transfers between serial ports and memory are limited to a 32-bit word width (maximum).

(Note also that 32-bit words and 16-bit short words can be freely mixed in the same memory block, with no restrictions.)

5.4 EXTERNAL MEMORY INTERFACING

In addition to its on-chip SRAM, the ADSP-2106x provides addressing of up to 4 gigawords of off-chip memory through its external port. This external address space includes *multiprocessor memory space*, the on-chip memory of all other ADSP-2106xs connected in a multiprocessor system, as well as *external memory space*, the region for standard addressing of off-chip memory.

Table 5.7 defines the ADSP-2106x pins used for interfacing to external memory. Memory control signals allow direct connection to fast static RAM devices. Memory-mapped peripherals and slower memories are also supported, with a user-defined combination of programmable wait states and hardware acknowledge signals. The suspend bus tristate pin ($\overline{\text{SBTS}}$) and page boundary pin (PAGE) can be used with DRAM memory.

External memory can hold both instructions and data. The external data bus ($DATA_{47-0}$) must be 48 bits wide to transfer instructions and/or 40-bit extended-precision floating-point data, or 32 bits wide to transfer single-precision floating-point data. If external memory contains only data or packed instructions that will be transferred by DMA, the external data bus width can be either 16 or 32 bits. In this type of system, the ADSP-2106x's on-chip I/O processor handles unpacking operations on data coming into it and packing operations on data going out. Figure 5.a shows how different data word sizes are transferred over the external port.

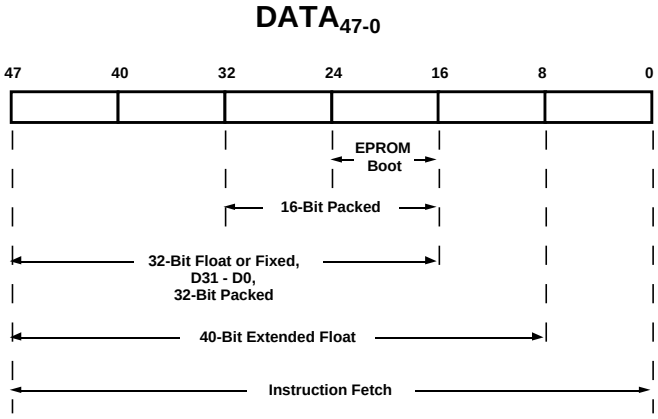


Figure 5.a External Port Data Alignment

5 Memory

The internal 32-bit DM Address bus and the I/O processor can access the entire 4-gigaword external memory space. The 24-bit PM Address bus, however, can only access 12 megawords of external memory because of its smaller width.

Pin	*Type*	*Function*
$ADDR_{31-0}$	I/O/T	**External Bus Address**. The ADSP-2106x outputs addresses for external memory and peripherals on these pins. In a multiprocessor system the bus master outputs addresses for read/writes of the internal memory or IOP registers of other ADSP-2106xs. The ADSP-2106x inputs addresses when a host processor or multiprocessing bus master is reading or writing its internal memory or IOP registers.
$DATA_{47-0}$	I/O/T	**External Bus Data**. The ADSP-2106x inputs and outputs data and instructions on these pins. 32-bit single-precision floating-point data and 32-bit fixed-point data is transferred over bits 47-16 of the bus. 40-bit extended-precision floating-point data is transferred over bits 47-8 of the bus. 16-bit short word data is transferred over bits 31-16 of the bus. Pull-up resistors on unused DATA pins are not necessary.
$\overline{MS}_{3-0}$	O/T	**Memory Select Lines**. These lines are asserted (low) as chip selects for the corresponding banks of external memory. Memory bank size must be defined in the ADSP-2106x's system control register (SYSCON). The $\overline{MS}_{3-0}$ lines are decoded memory address lines that change at the same time as the other address lines. When no external memory access is occurring the $\overline{MS}_{3-0}$ lines are inactive; they are active, however, when a conditional memory access instruction is executed, whether or not the condition is true. $\overline{MS}_0$ can be used with the PAGE signal to implement a bank of DRAM memory (Bank 0). In a multiprocessing system the $\overline{MS}_{3-0}$ lines are output by the bus master.
$\overline{RD}$	I/O/T	**Memory Read Strobe**. This pin is asserted (low) when the ADSP-2106x reads from external memory devices or from the internal memory of other ADSP-2106xs. External devices (including other ADSP-2106xs) must assert $\overline{RD}$ to read from the ADSP-2106x's internal memory. In a multiprocessing system $\overline{RD}$ is output by the bus master and is input by all other ADSP-2106xs.

Table 5.7 External Memory Interface Signals (cont. on next page)

Pin	*Type*	*Function*
$\overline{WR}$	I/O/T	**Memory Write Strobe**. This pin is asserted (low) when the ADSP-2106x writes to external memory devices or to the internal memory of other ADSP-2106xs. External devices must assert $\overline{WR}$ to write to the ADSP-2106x's internal memory. In a multiprocessing system $\overline{WR}$ is output by the bus master and is input by all other ADSP-2106xs.
PAGE	O/T	**DRAM Page Boundary**. The ADSP-2106x asserts this pin to signal that an external DRAM page boundary has been crossed. DRAM page size must be defined in the ADSP-2106x's memory control register (WAIT). DRAM can only be implemented in external memory Bank 0; the PAGE signal can only be activated for Bank 0 accesses. In a multiprocessing system PAGE is output by the bus master.
$\overline{SW}$	I/O/T	**Synchronous Write Select**. This signal is used to interface the ADSP-2106x to synchronous memory devices (including other ADSP-2106xs). The ADSP-2106x asserts $\overline{SW}$ (low) to provide an early indication of an impending write cycle, which can be aborted if $\overline{WR}$ is not later asserted (e.g. in a conditional write instruction). In a multiprocessing system, $\overline{SW}$ is output by the bus master and is input by all other ADSP-2106xs to determine if the multiprocessor memory access is a read or write. $\overline{SW}$ is asserted at the same time as the address output. A host processor using synchronous writes must assert this pin when writing to the ADSP-2106x(s).
ACK	I/O/S	**Memory Acknowledge**. External devices can deassert ACK (low) to add wait states to an external memory access. ACK is used by I/O devices, memory controllers, or other peripherals to hold off completion of an external memory access. The ADSP-2106x deasserts ACK as an output to add wait states to a synchronous access of its internal memory. In a multiprocessing system, a slave ADSP-2106x deasserts the bus master's ACK input to add wait state(s) to an access of its internal memory. The bus master has a keeper latch on its ACK pin that maintains the input at the level it was last driven to.

I=Input S=Synchronous (o/d)=Open Drain
O=Output A=Asynchronous (a/d)=Active Drive

T=Tristate (when $\overline{SBTS}$ or $\overline{HBR}$ is asserted, or when the ADSP-2106x is a bus slave)

Table 5.7 External Memory Interface Signals

5.4.1 External Memory Banks

External memory is divided into four banks of equal size, each associated with its own wait-state generator. This allows slower peripheral devices to be memory-mapped into a bank for which a specific number of wait states are specified. By mapping peripherals into different banks, you can accommodate I/O devices with different timing requirements.

Bank 0 starts at address 0x0040 0000 in external memory and is followed in order by Banks 1, 2, and 3. Whenever the ADSP-2106x generates an address located within one of the four banks, the corresponding memory select line, $\overline{MS}_{3\text{-}0}$, is asserted.

The $\overline{MS}_{3\text{-}0}$ outputs can be used as chip selects for memories or other external devices, eliminating the need for external decoding logic. $\overline{MS}_0$ provides a select line for a bank of DRAM memory, when used in combination with the PAGE signal (see "DRAM Page Boundary Detection").

The size of the memory banks can range from 8K words to 256 megawords, and must be a power of two. Selection of memory bank size is accomplished by setting the MSIZE bit field of the SYSCON register in the following way:

MSIZE = $\log_2$ (desired bank size) – 13

The $\overline{MS}_{3\text{-}0}$ lines are decoded memory address lines that change at the same time as the other address lines. When no external memory access is occurring the $\overline{MS}_{3\text{-}0}$ lines are inactive; they are active, however, when a conditional memory access instruction is executed, whether or not the condition is true. Systems using the $\overline{SW}$ signal that cannot abort such accesses should not use conditional memory write instructions, to ensure proper operation.

(Note that the ADSP-2106x's internal memory is divided into two *blocks*, called Block 0 and Block 1, while the external memory space is divided into four *banks*.)

5.4.2 Unbanked Memory

The region of memory above Banks 0-3 is called *unbanked* external memory space. No $\overline{MS}_x$ memory select line is asserted for accesses in this address space. Unbanked memory space accesses can also have wait states specified, in the UBWS and UBWM fields of the WAIT register.

5.4.3 Boot Memory Select ($\overline{\text{BMS}}$)

The $\overline{\text{BMS}}$ memory select line is asserted (low) only when the ADSP-2106x is configured for EPROM booting. This allows access of a separate external memory space for booting. Unbanked memory wait states and wait state mode are applied to $\overline{\text{BMS}}$-selected accesses.

The $\overline{\text{BMS}}$ output is only driven by the ADSP-2106x bus master. For details on EPROM booting, see "Booting" in the *System Design* chapter of this manual.

5.4.4 Wait States & Acknowledge

The ADSP-2106x's WAIT register is used to set up external memory wait states and response to the ACK signal. The WAIT register is one of the ADSP-2106x's IOP control registers.

To simplify the interface to slow external memories and peripherals, the ADSP-2106x provides a variety of methods for extending off-chip memory accesses:

- **External.** The ADSP-2106x samples its acknowledge input (ACK) during each clock cycle. If it latches a low value, it inserts a wait state by holding the address and strobes valid for an additional cycle. If the value of ACK is high, the ADSP-2106x completes the cycle.

- **Internal.** The ADSP-2106x ignores the ACK input. Control bits in the WAIT register specify the number of wait states for the access. You can specify a different number of wait states for each bank of external memory.

- **Both.** The ADSP-2106x samples its ACK input in each clock cycle. If it latches a low value, it inserts a wait state. If the value of ACK is high, it completes the cycle only if the number of wait states (specified in WAIT) have expired. In this mode, the WAIT-programmed wait states specify a minimum number of cycles per access, and an external device can use the ACK pin to extend the access as necessary. The ACK signal may be undefined (transitioning) until the internally programmed waitstates have completed; i.e. ACK is not sampled until the programmed waitstates have completed. No metastability problems will occur.

- **Either.** The ADSP-2106x completes the cycle as soon as it samples the ACK input as high or when the WAIT-programmed number of wait states have expired, whichever occurs first. In this mode, a system with two different types of peripherals could shorten the access for the faster peripheral using ACK but use the programmed wait states for the slower peripheral.

The method selected for each bank of memory is independent of the other banks. Thus, you can map devices of different speeds into different memory banks for the appropriate wait state control.

5.4.4.1 WAIT Register

The WAIT register is defined in Table 5.8 and shown in Figure 5.15. The bit values shown in Figure 5.15 are the default initialization; the WAIT register is initialized to 0x21AD 6B5A after a processor reset.

A *bus idle cycle* is an inactive bus cycle that is automatically generated to avoid bus driver conflicts. Such a conflict can occur when a device with a long output disable time continues to drive after $\overline{RD}$ is deasserted while another device begins driving in the following cycle.

To avoid this conflict, the ADSP-2106x will generate an inactive bus cycle on a transition from a read of a memory bank with bus idle cycle enabled to an access of any other bank or to a write in the same bank or to MMS (multiprocessor memory space). In other words, a bus idle cycle is always generated after a read, except in the case of consecutive reads of the same bank. A device with a slow disable time should enable bus idle cycle generation by using *# of wait states* code 001, 010, 011, or 111.

When a bus idle cycle is specified for unbanked memory, an idle cycle is inserted after every read cycle, not just after a bank change. This allows several external devices to be used in this region of memory. The ADSP-2106x cannot distinguish when there is a device change so it inserts an idle cycle after each read.

A *hold time cycle* is an inactive bus cycle automatically generated at the end of a read or write to allow a longer hold time for address and data. The address and data will remain unchanged and driven for one cycle after the read or write strobes are deasserted.

A single idle cycle on a page boundary crossing can be enabled by setting the PAGEIS bit of the WAIT register; the address is asserted in the same cycle that the PAGE pin is asserted, but read / write strobe assertion is delayed for one cycle. See “DRAM Page Boundary Detection” for further details.

Bit(s)	Name	Function
1-0	EB0WM	External Bank 0 wait state mode*
4-2	EB0WS	External Bank 0 number of wait states**
6-5	EB1WM	External Bank 1 wait state mode*
9-7	EB1WS	External Bank 1 number of wait states**
11-10	EB2WM	External Bank 2 wait state mode*
14-12	EB2WS	External Bank 2 number of wait states**
16-15	EB3WM	External Bank 3 wait state mode*
19-17	EB3WS	External Bank 3 number of wait states**
21-20	UBWM	Unbanked memory wait state mode*
24-22	UBWS	Unbanked memory number of wait states**
27-25	PAGSZ	Page size for DRAM (only in Bank 0) †
28	PAGEIS	Single idle cycle on DRAM page boundary crossing
29	MMSWS	Single wait state for Multiprocessor Memory Space access
30	HIDMA	Single idle cycle for DMA handshake ††
31	*reserved*	

Table 5.8 WAIT Register Bit Definitions

*** Wait state mode:**

EBxWM	Wait State Mode
00	External acknowledge only (ACK)
01	Internal wait states only
10	Both internal and external acknowledge required
11	Either internal or external acknowledge sufficient

**** Number of wait states:**

EBxWS	# of Wait States	Bus Idle Cycle?	Hold Time Cycle?
000	0	no	no
001	1	yes	no
010	2	yes	no
011	3	yes	no
100	4	no	yes
101	5	no	yes
110	6	no	yes
111	0	yes	no

Note that the bus idle cycle or hold time cycles will occur if programmed, regardless of the waitstate mode. For example, the ACK-only waitstate mode may have a hold time cycle programmed for it.

† DRAM page size:

PAGSZ	DRAM Page Size
000	256 words
001	512 words
010	1024 words (1K)
011	2048 words (2K)
100	4096 words (4K)
101	8192 words (8K)
110	16384 words (16K)
111	32768 words (32K)

(See “DRAM Page Boundary Detection” for more information on DRAM control.)

†† Setting the HIDMA bit to 1 causes an idle cycle to be inserted after every read (with $\overline{\text{DMAG}}$x asserted) from an external DMA latch.

This allows a device with a slow tristate time to get off the local bus before the next ADSP-2106x access begins. The idle cycle is inserted for every read from the DMA latch, not just for a changeover. See “DMA Hardware Interfacing” in the “External Port DMA” section of the *DMA* chapter for an example showing an external DMA latch.

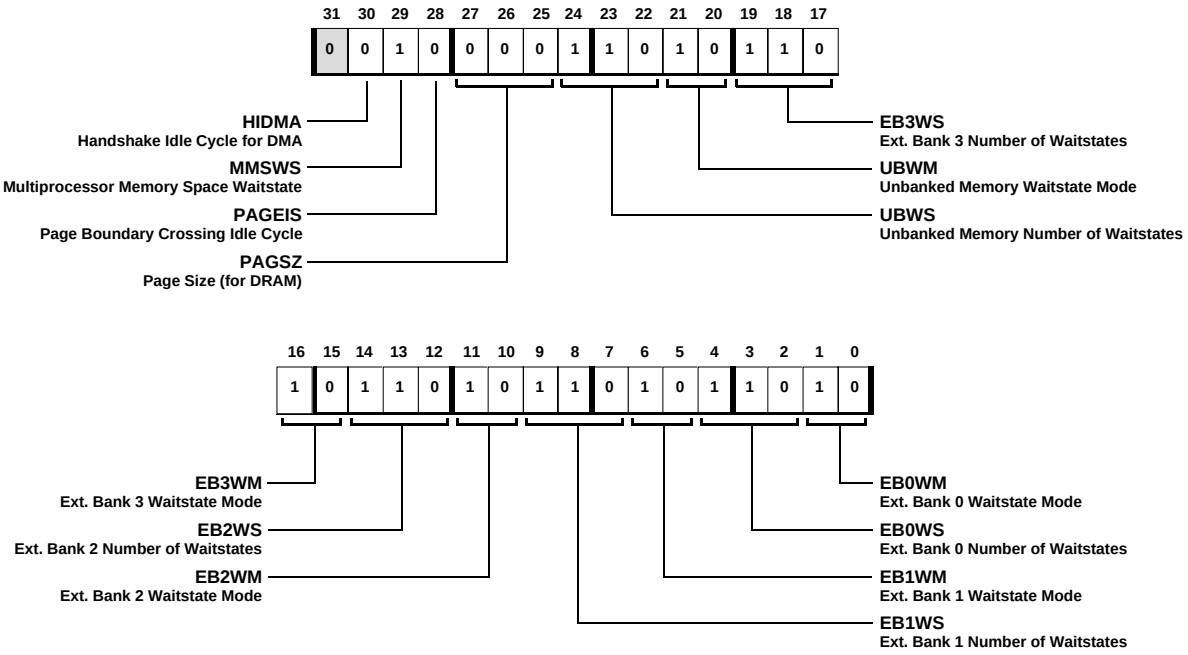


Figure 5.15 WAIT Register

Figure 5.16 (on the following page) shows the effects of the bus idle cycle, hold time cycle, and page idle cycle options.

The WAIT register is initialized to 0x21AD 6B5A after processor reset. This configures the ADSP-2106x for the following:

- no idle state on page boundary crossings
- 6 internal wait states
- dependence on both software-programmed waitstates and external acknowledge for all memory banks and for unbanked memory
- multiprocessor memory space wait state enabled (see the next section)

Unbanked memory wait states and wait state mode are applied to $\overline{\text{BMS}}$-selected accesses.

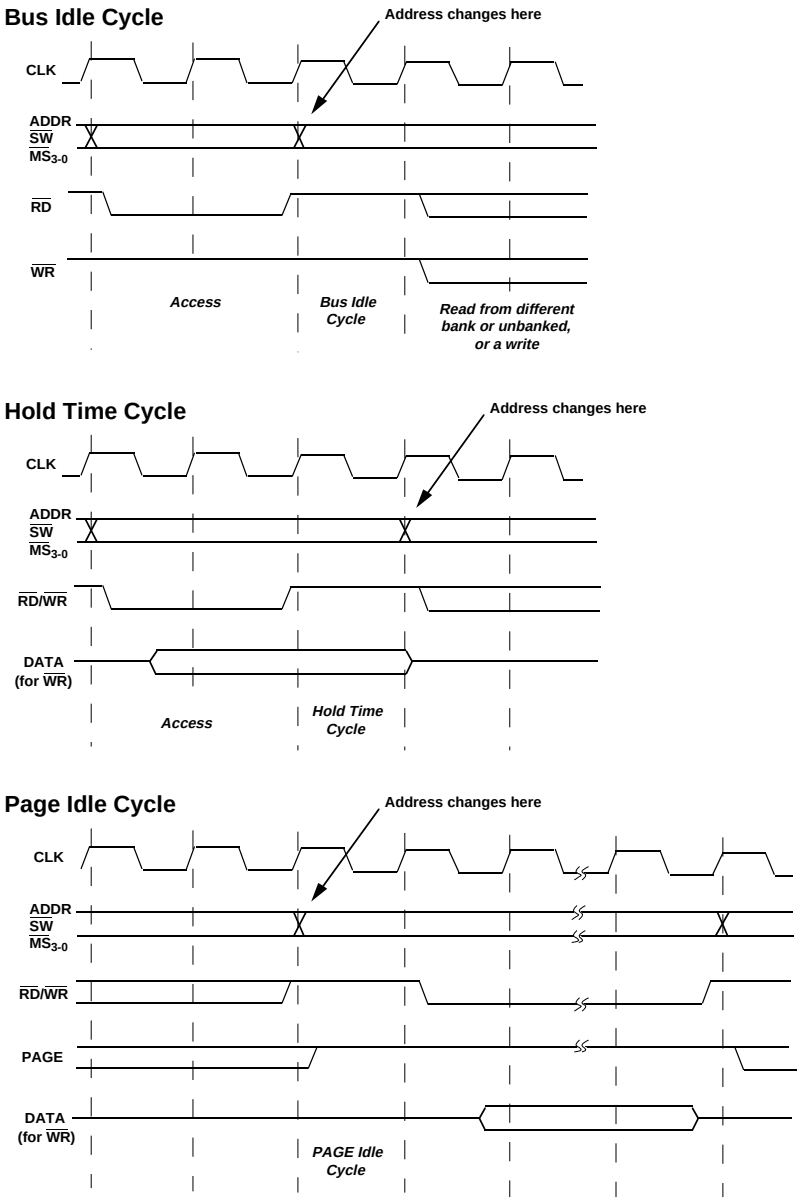


Figure 5.16 Bus Idle Cycle, Hold Time Cycle, Page Idle Cycle

5.4.4.2 Multiprocessor Memory Space Wait States & Acknowledge

Completion of reads and writes to multiprocessor memory space depends only on the ACK signal. This is facilitated by using the $\overline{SW}$ signal as an early indication of whether the access is a write or a read (see Figure 5.19 at the end of this chapter), as well as the use of the automatic wait state option for multiprocessor memory space—the MMSWS bit of the WAIT register.

Setting the MMSWS bit (bit 29) of the WAIT register causes the insertion of a single wait state into all multiprocessor memory space reads and writes. This option should be used whenever the external system bus is heavily loaded (i.e. such that the synchronous timing requirements for interprocessor communications cannot be met; refer to the *ADSP-2106x Data Sheet* for these specifications.)

The ADSP-2106x bus master inserts the wait state. The slave ADSP-2106x(s) respond with ACK (low) in the first cycle, even if they have MMSWS=1. If MMSWS=1 on the master ADSP-2106x, it will ignore ACK in the first cycle and respond to it in the second cycle. This setting allows longer set up times for the following slave SHARC's signals: ADDR, RD, WR, and DATA (written to the slave). Also, this setting allows a longer set up time for the master SHARC's ACK signal. Other hold and set up times are not influenced by MMSWS=1. This setting does not change hold time requirements for the slave SHARC's RD, WR, or DATA (written to the slave). Also, this setting does not change the master SHARC's set up or hold times for DATA (read from the slave).

All of the ADSP-2106xs in a multiprocessor system *must* have the same value for the MMSWS bit.

5.4.5 DRAM Page Boundary Detection

Applications with large amounts of data may want to use DRAM memory for bulk storage. To simplify interfacing to page-mode or static-column DRAMs, the ADSP-2106x detects page boundary crossings and outputs the PAGE signal to an external DRAM controller. Page boundaries are user-defined; they must be programmed in the WAIT register.

Automatic page boundary detection is provided by the ADSP-2106x's PAGE signal. DRAM memory must be implemented in bank 0 of external memory—the PAGE signal is only active within bank 0. The page size for page boundary detection is specified in the PAGSZ field (bits 27-25) of the WAIT register:

PAGSZ	*DRAM Page Size*
000	256 words
001	512 words
010	1024 words (1K)
011	2048 words (2K)
100	4096 words (4K)
101	8192 words (8K)
110	16384 words (16K)
111	32768 words (32K)

The ADSP-2106x asserts its PAGE pin whenever an external access crosses a page boundary and the address is within bank 0. The processor detects a boundary crossing by comparing each address output for bank 0 to the address of the last successful external access (which is stored in the IOP register ELAST). If a memory access is aborted, for example due to a conditional write, the PAGE pin is not asserted and the current page is not updated in ELAST. The PAGE pin will not be asserted nor the current page updated if the access is to multiprocessor memory space, or to any memory space other than bank 0 of external memory space.

The PAGE pin remains asserted as long as the access is active. It is not asserted if no access is performed.

The current page is automatically invalidated and the PAGE pin asserted upon the next external access if: 1) the ADSP-2106x loses mastership of the external bus to another ADSP-2106x or to a host processor, or 2) the processor is reset. ELAST should not be read in the cycle immediately after it is written, as it may be in the process of updating.

A single idle cycle on a page boundary crossing can be enabled in order to give the DRAM controller enough time to assert the $\overline{\text{SBTS}}$ bus tristate pin (see “Suspend Bus Tristate” below). This option is enabled by setting the PAGEIS bit (bit 28) of the WAIT register. The address is asserted in the same cycle that the PAGE pin is asserted, but read/write strobe assertion is delayed for one cycle. If wait states are enabled for the bank, they will begin after the idle cycle. The page change applies only to Bank 0, thus allowing interleaved reads and writes of other external memory or peripherals without always incurring a page change to the DRAM in Bank 0. This option should be disabled when the PAGE signal is not being used.

The host bus request pin ($\overline{HBR}$) is disabled when the PAGE pin is asserted. This prevents the possibility of the ADSP-2106x becoming a bus slave (by means of the deadlock resolution functionality) while the DRAM controller is servicing a page change. (See “Suspend Bus Tristate” below.)

Figure 5.17 shows an example of an ADSP-2106x system with DRAM. Different interfacing methods may be needed in some applications, however, especially if buffers are needed for the DRAM.

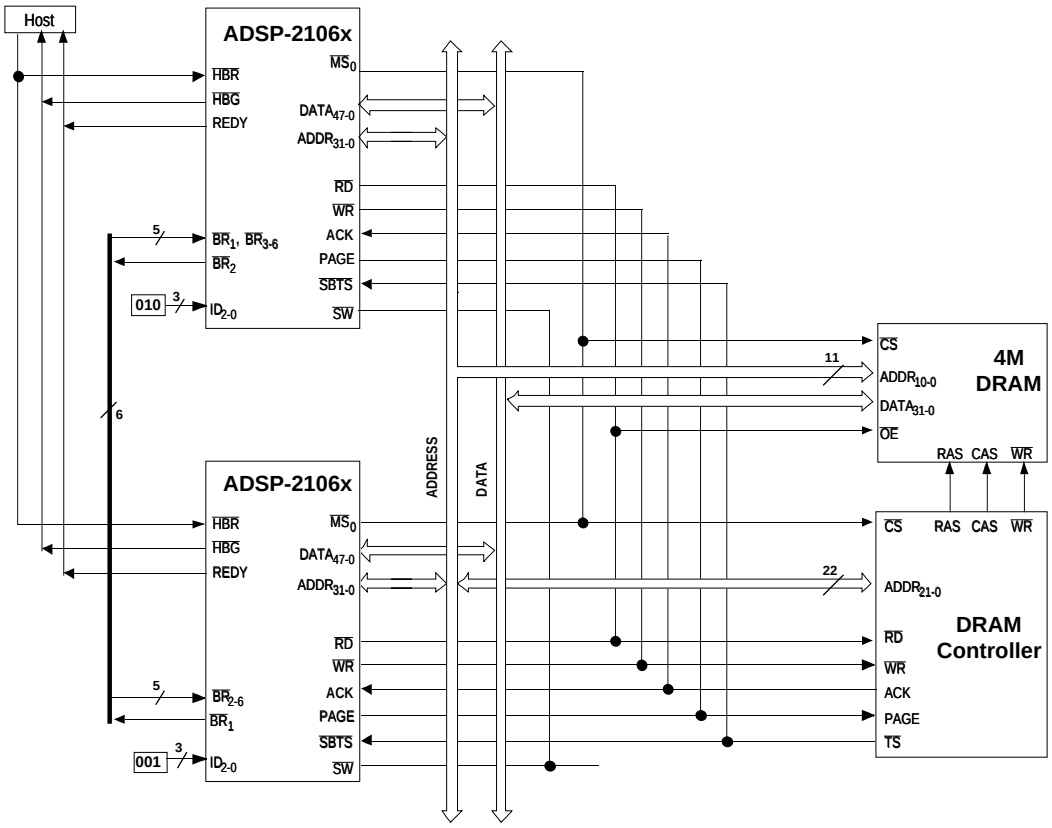


Figure 5.17 Example DRAM Interface

5.4.5.1 Suspend Bus Tristate ($\overline{SBTS}$)

External devices can assert the ADSP-2106x's $\overline{SBTS}$ input to place the external bus address, data, selects, and strobes in a high-impedance state for the following cycle. If the ADSP-2106x attempts to access external memory while $\overline{SBTS}$ is asserted, the processor will halt and the memory access will not be completed until $\overline{SBTS}$ is deasserted.

$\overline{SBTS}$ should only be used to recover from DRAM page faults or host processor/ADSP-2106x deadlock. (See "Deadlock Resolution" in the "System Bus Interfacing" section of the *Host Interface* chapter.) In the case of DRAM page faults, $\overline{SBTS}$ allows the external DRAM controller to take control of the external bus.

$\overline{SBTS}$ causes the following pins to be tristated:

$ADDR_{31-0}$	$\overline{RD}$	PAGE
$DATA_{47-0}$	$\overline{WR}$	$\overline{DMAG1}$
$\overline{MS}_{3-0}$	$\overline{SW}$	$\overline{DMAG2}$
$\overline{BMS}$	ADRCLK	

5.4.5.2 Normal $\overline{SBTS}$ Operation: $\overline{HBR}$ Not Asserted

Asserting $\overline{SBTS}$ places the external bus address, data, selects, and strobes in a high-impedance state for the following cycle. If an external access is underway when $\overline{SBTS}$ is asserted, the access will be held off (as if ACK were deasserted). If $\overline{SBTS}$ is asserted while there is no external access occurring, the external bus pins will tristate and the ADSP-2106x will continue running until it tries to perform an external access (at which time it will halt). In this case, the memory access will begin in the cycle after the deassertion of $\overline{SBTS}$.

When $\overline{SBTS}$ is deasserted, the $\overline{RD}$, $\overline{WR}$, and $\overline{DMAG}$x strobes will be reasserted (if they had been asserted prior to $\overline{SBTS}$) after the external address has become valid (i.e. at their normal timing within the cycle). The wait state counter will be reset. This applies even if the processor is held in reset ($\overline{RESET}$ asserted).

$\overline{SBTS}$ differs from $\overline{HBR}$ in that it takes effect in the next cycle, even if an external access is occurring (but not finished). $\overline{SBTS}$ should only be used when the external access is to a device such as a DRAM or cache memory, where the access must be held off in order to prepare for it. Use of $\overline{SBTS}$ at other times—such as during ADSP-2106x-to-ADSP-2106x accesses or when $\overline{DMAG}$x is asserted—will result in incorrect operation.

5.5 EXTERNAL MEMORY ACCESS TIMING

Memory access timing for external memory space and multiprocessor memory space is described below. For exact timing specifications, refer to the *ADSP-2106x Data Sheet*.

5.5.1 External Memory

The ADSP-2106x can interface asynchronously, without reference to CLKIN, to external memories and and memory-mapped peripherals. In a multiprocessing system, the ADSP-2106x must be the bus master in order to access external memory.

Figure 5.18 shows representative timing for an asynchronous read or write of external memory. Note that the clock signal is shown only to indicate that the access occurs within a single cycle.

5.5.1.1 External Memory Read – Bus Master

External memory reads occur with the following sequence of events (see Figure 5.18):

1. The ADSP-2106x drives the read address and asserts a memory select signal ($\overline{MS}_{3-0}$) to indicate the selected bank. The memory select signal is not deasserted between successive accesses of the same memory bank.

2. The ADSP-2106x asserts the read strobe (unless the access is aborted because of a conditional instruction).

3. The ADSP-2106x checks whether wait states are needed. If so, the memory select and read strobe remain active for additional cycle(s). Wait states are determined by the state of the external acknowledge signal (ACK), the internally programmed wait state count, or a combination of the two.

4. The ADSP-2106x latches in the data.

5. The ADSP-2106x deasserts the read strobe.

6. If initiating another memory access, the ADSP-2106x drives the address and memory select for the next cycle.

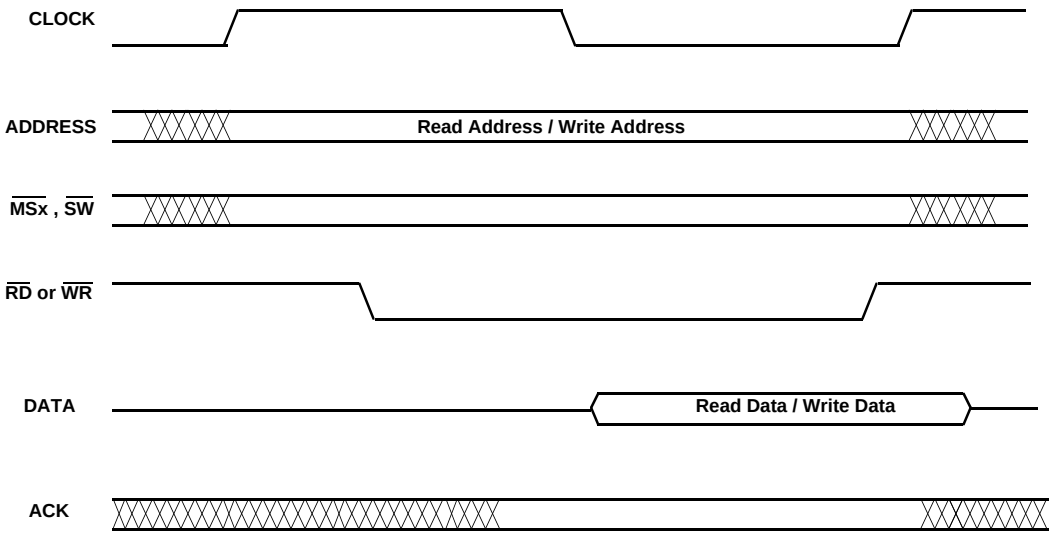


Figure 5.18 External Memory Access Timing

Note that if a memory read is part of a conditional instruction that is not executed because the condition is false, the ADSP-2106x still drives the address and memory select for the read, but does not assert the read strobe or read any data.

5.5.1.2 External Memory Write – Bus Master

External memory writes occur with the following sequence of events (refer again to Figure 5.18):

1. The ADSP-2106x drives the write address and asserts a memory select signal to indicate the selected bank. A memory select signal is not deasserted between successive accesses of the same memory bank.

2. The ADSP-2106x asserts the write strobe and drives the data (unless the memory access is aborted because of a conditional instruction).

3. The ADSP-2106x checks whether wait states are needed. If so, the memory select and write strobe remain active for additional cycle(s). Wait states are determined by the state of the external acknowledge signal, the internally programmed wait state count, or a combination of the two.

4. The ADSP-2106x deasserts the write strobe near the end of the cycle.

5. The ADSP-2106x tristates its data outputs.

6. If initiating another memory access, the ADSP-2106x drives the address and memory select for the next cycle.

Note that if a memory write is part of a conditional instruction that is not executed because the condition is false, the ADSP-2106x still drives the address and memory select for the write, but does not assert the write strobe or drive any data.

5.5.2 Multiprocessor Memory

Timing for multiprocessor memory accesses is shown in Figure 5.19. For complete information on multiprocessor memory accesses, see "Direct Reads & Writes" and "Data Transfers Through The EPBx Buffers" in the *Multiprocessing* chapter of this manual.

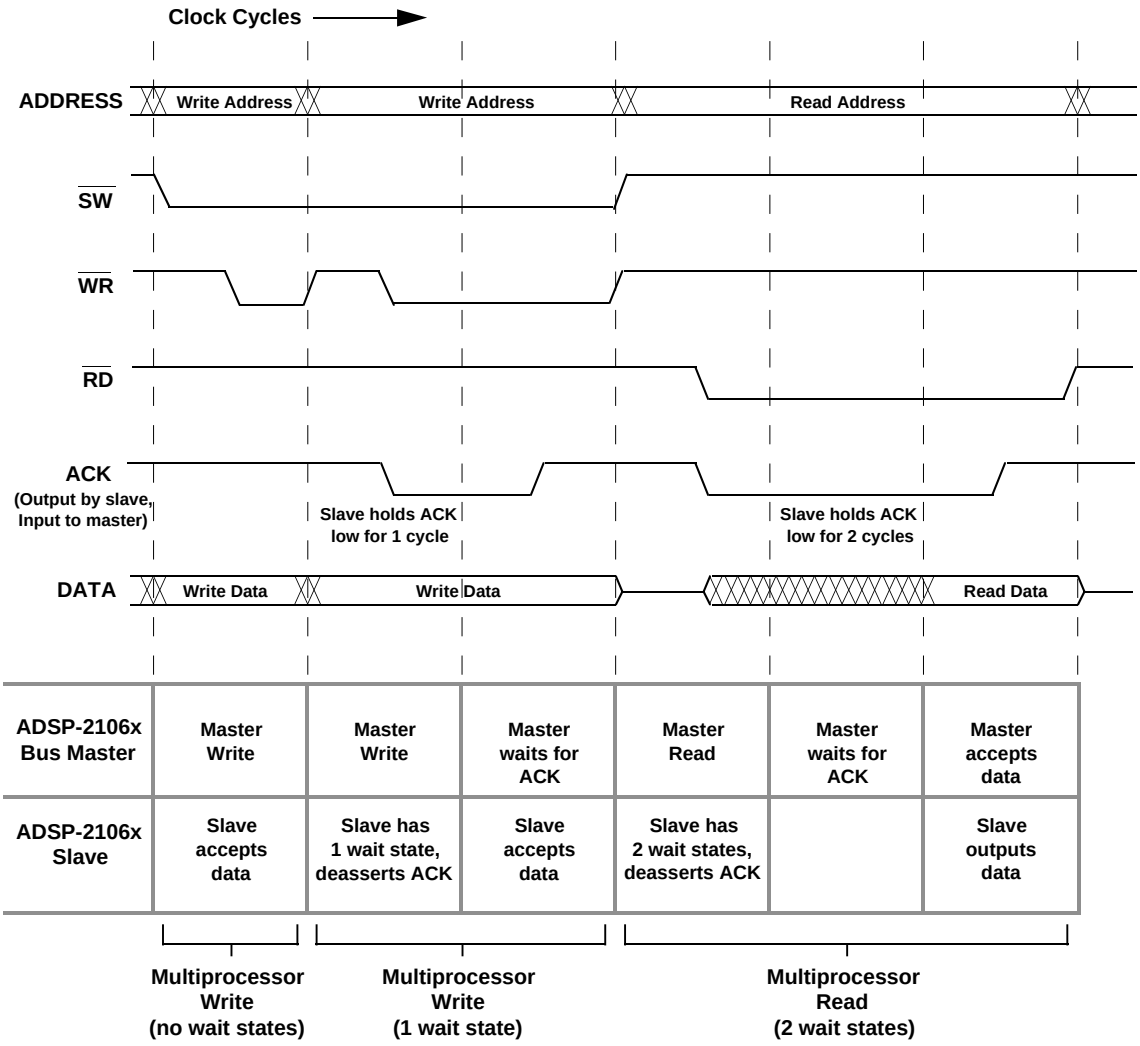


Figure 5.19 Multiprocessor Memory Access Timing

Note—Minimum access time is: 1 wait state (2 cycles) for IOP register reads
3 wait states (4 cycles) for memory reads

5 Memory

DMA 6

6.1 OVERVIEW

Direct Memory Access (DMA) provides a mechanism for transferring an entire block of data. The ADSP-2106x's on-chip DMA controller relieves the core processor of the burden of moving data between internal memory and an external data source or external memory. The fully integrated DMA controller allows the ADSP-2106x core processor, or an external device, to specify data transfer operations and return to normal processing while the DMA controller carries out the data transfers independently and invisibly to the core.

Figure 6.1 shows a block diagram of the ADSP-2106x's DMA controller, I/O processor, external port, and internal memory. Figure 6.2 shows a more detailed block diagram of the external port, the DMA controller, FIFO buffers, and DMA data paths and control.

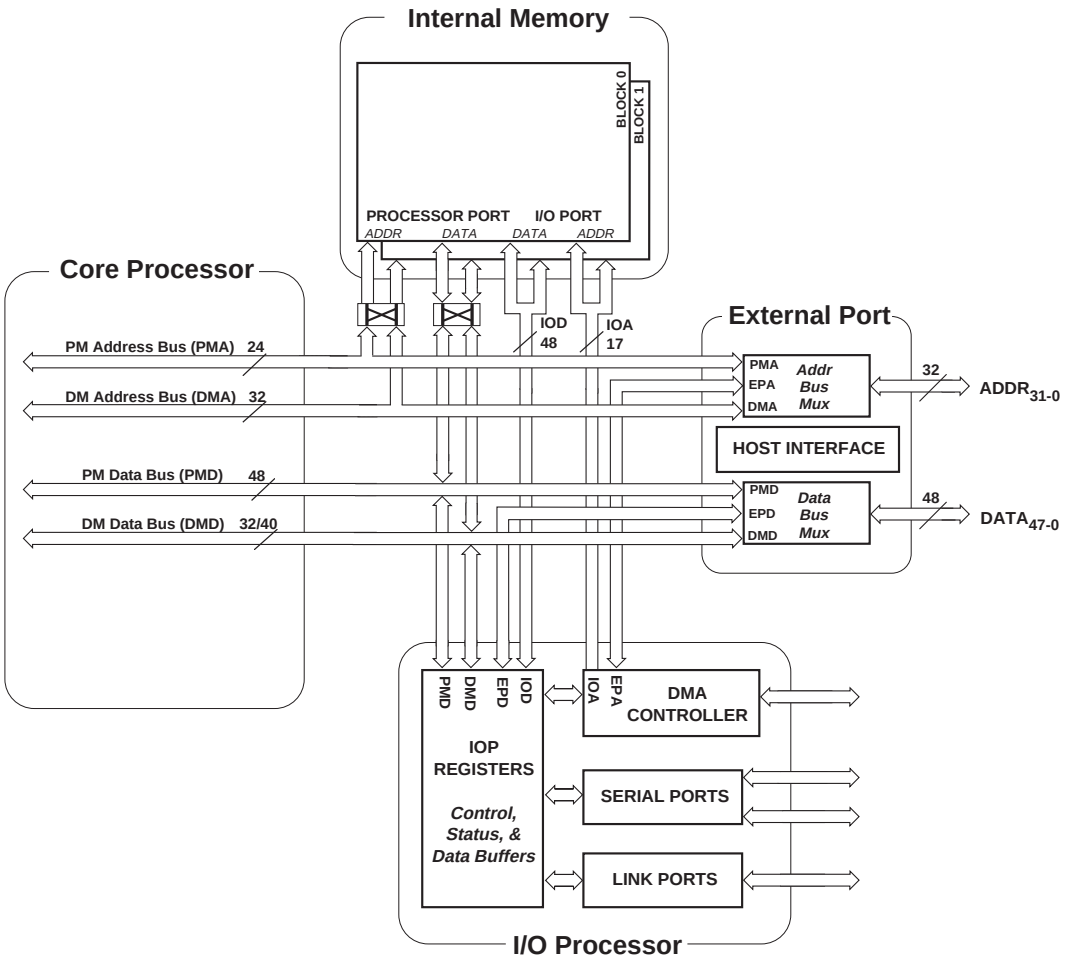
The DMA controller can perform several types of data transfers:

- internal memory ↔ external memory and memory-mapped peripherals
- internal memory ↔ internal memory of other ADSP-2106xs
- internal memory ↔ host processor
- internal memory ↔ serial port I/O
- internal memory ↔ link port I/O*
- external memory ↔ external peripherals

* Not applicable to the ADSP-21061.

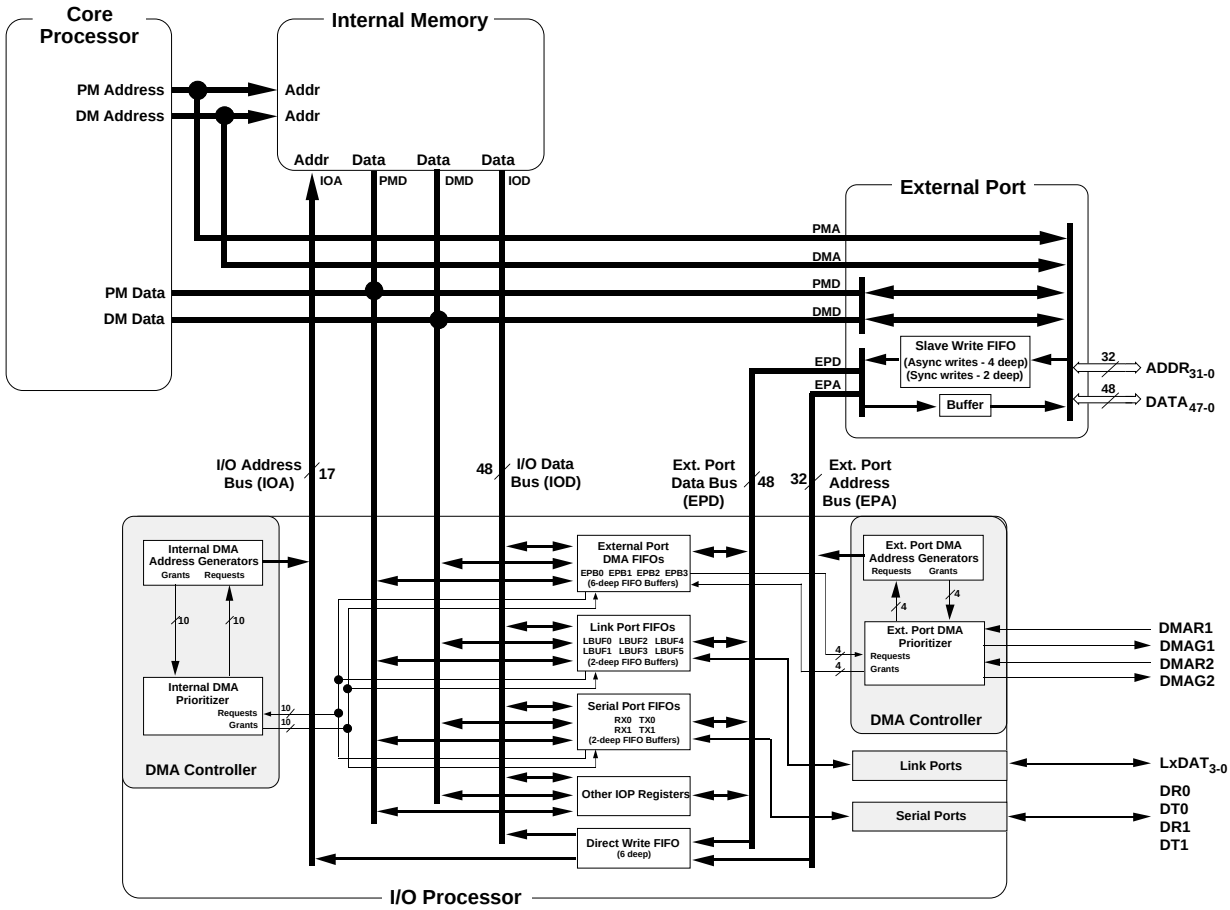
External bus word packing is used to facilitate compatibility between the ADSP-2106x's internal 32/48-bit structure and external 16- and 32-bit peripheral devices. Control of bus packing is accomplished in each of the four external port DMA Control Registers (DMAC6, DMAC7, DMAC8, and DMAC9).

6 DMA



* Note that link ports are not available on the ADSP-21061.

Figure 6.1 ADSP-2106x Block Diagram



* Note that link ports are not available on the ADSP-21061.

Figure 6.2 DMA Data Paths & Control

For external DMA requests, the ADSP-2106x includes the DMA request inputs $\overline{\text{DMAR1}}$ and $\overline{\text{DMAR2}}$, along with the DMA grant outputs $\overline{\text{DMAG1}}$ and $\overline{\text{DMAG2}}$, to support DMA transfers to and from external asynchronous peripheral devices. By pulling a $\overline{\text{DMAR}}$x line low and waiting for the appropriate $\overline{\text{DMAG}}$x signal to come back from the ADSP-2106x, a simple I/O device can transfer data to ADSP-2106x internal memory or to external memory.

6 DMA

The ten DMA channels of the ADSP-21060 and ADSP-21062 are numbered as shown in Table 6.1a, which also shows the corresponding data buffer used with each channel.

DMA Channel#	*Data Buffer*	*Description*
DMA Channel 0	RX0	Serial Port 0 Receive
DMA Channel 1	RX1 *(or LBUF0)*	Serial Port 1 Receive *(or Link Buffer 0)*
DMA Channel 2	TX0	Serial Port 0 Transmit
DMA Channel 3	TX1 *(or LBUF1)*	Serial Port 1 Transmit *(or Link Buffer 1)*
DMA Channel 4	LBUF2	Link Buffer 2
DMA Channel 5	LBUF3	Link Buffer 3
DMA Channel 6	EPB0 *(or LBUF4)*	Ext. Port FIFO Buffer 0 *(or Link Buffer 4)*
DMA Channel 7*	EPB1 *(or LBUF5)*	Ext. Port FIFO Buffer 1 *(or Link Buffer 5)*
DMA Channel 8*	EPB2	Ext. Port FIFO Buffer 2
DMA Channel 9	EPB3	Ext. Port FIFO Buffer 3

Table 6.1a ADSP-2106x DMA Channels & Data Buffers

* $\overline{\text{DMAR1}}$ and $\overline{\text{DMAG1}}$ are handshake controls for DMA Channel 7.
* $\overline{\text{DMAR2}}$ and $\overline{\text{DMAG2}}$ are handshake controls for DMA Channel 8.

The six DMA channels of the ADSP-21061 are numbered as shown in Table 6.1b, which also shows the corresponding data buffer used with each channel.

DMA Channel#	*Data Buffer*	*Description*
DMA Channel 0	RX0	Serial Port 0 Receive
DMA Channel 1	RX1	Serial Port 1 Receive
DMA Channel 2	TX0	Serial Port 0 Transmit
DMA Channel 3	TX1	Serial Port 1 Transmit
DMA Channel 6*	EPB0	Ext. Port FIFO Buffer 0
DMA Channel 7*	EPB1	Ext. Port FIFO Buffer 1

Table 6.1b ADSP-2106x DMA Channels & Data Buffers

* $\overline{\text{DMAR2}}$ and $\overline{\text{DMAG2}}$ are handshake controls for DMA Channel 6.
* $\overline{\text{DMAR1}}$ and $\overline{\text{DMAG1}}$ are handshake controls for DMA Channel 7.

The following terms are used throughout this chapter, and are defined below for reference:

external port FIFO buffers	EPB0, EPB1, EPB2, and EPB3—the IOP registers used for external port DMA transfers and single-word data transfers (from other ADSP-2106xs or from a host processor); these buffers are 6-deep FIFOs
DMACx control registers	the DMA control registers for the EPBx external port buffers: DMAC6, DMAC7, DMAC8, and DMAC9 (corresponding respectively to EPB0, EPB1, EPB2, and EPB3)
DMA parameter registers	the address (IIx), modifier (IMx), count (Cx), chain pointer (CPx), etc., registers used to set up a DMA transfer
transfer control block (TCB)	a set of DMA parameter register values stored in memory that are downloaded by the ADSP-2106x's DMA controller for chained DMA operations
TCB chain loading	the process in which the ADSP-2106x's DMA controller downloads a TCB from memory and autoinitializes the DMA parameter registers

6.1.1 DMA Controller Features

The ADSP-2106x's DMA controller is designed to perform two basic types of operations: external port block data transfers and I/O port data transfers. The I/O ports on the ADSP-21060 and ADSP-21062 are the link ports and serial ports. The I/O ports on the ADSP-21061 are the serial ports.

External port block data transfers move data between ADSP-2106x internal memory and external memory. The DMA controller must be programmed with the internal memory buffer size and address, the address increment, and the direction of transfer. Once setup programming is complete, DMA transfers begin automatically and continue until the entire buffer is transferred to or from internal memory.

I/O port DMA transfers handle data transmitted and received through the ADSP-2106x's serial ports and link ports. When performing I/O DMA, the same type of buffer is set up in internal memory, but instead of accessing the external memory, the DMA controller accesses the I/O port. The direction of data transfer is determined by the direction of the I/O port. When data is received at the port, it is automatically transferred to internal memory. Likewise, when the port needs to transmit a word, it is automatically fetched from internal memory.

An additional DMA capability allows the ADSP-2106x to support data transfers between an external device and external memory. This transfer does not interfere with internal ADSP-2106x operations that do not use the external port.

External devices can participate in DMA transfers in two ways. The external device can read or write to a DMA buffer on the ADSP-2106x, or it can assert a DMA Request input ($\overline{\text{DMAR}}$x) to request service.

In chained DMA operations, a DMA transfer can be programmed to autoinitialize another DMA operation upon completion.

6.1.2 Setting Up DMA Transfers

DMA operations can be programmed by the ADSP-2106x core processor, by an external host processor, or by the (external) ADSP-2106x bus master. The operation is programmed by writing to the memory-mapped DMA control registers and parameter registers. A DMA channel is set up by writing a set of memory buffer parameters to the DMA parameter registers. The II, IM, and C registers must be loaded with a starting address for the buffer, an address modifier, and a word count, respectively.

The external ports, link ports, and serial ports each have a DMA enable bit (DEN) in their main control register. Once a DMA channel is set up and enabled, data words received are automatically transferred to the buffer in internal memory. Likewise, when the ADSP-2106x is ready to transmit data, a word is automatically transferred from internal memory to the DMA buffer register. These transfers continue until the entire data buffer is received or transmitted.

DMA interrupts can be generated when an entire block of data has been transferred. This occurs when the DMA channel's count register (C) has decremented to zero (or EC register, in master mode only).

DMA interrupts are latched and masked in the IRPTL and IMASK registers, respectively; these registers are located in the ADSP-2106x processor core, not in the memory-mapped IOP register space.

➠ TO START A NEW DMA SEQUENCE AFTER THE CURRENT ONE IS FINISHED, YOUR PROGRAM MUST FIRST CLEAR THE DMA ENABLE BIT, WRITE NEW PARAMETERS TO THE II, IM, AND C REGISTERS, AND THEN SET THE DMA ENABLE BIT TO RE-ENABLE DMA.

(For chained DMA operations, however, this is not necessary; see "DMA Chaining.")

For further details, see the "DMA Controller Operation," "DMA Channel Parameter Registers," and "DMA Interrupts" sections of this chapter.

6.2 DMA CONTROL REGISTERS

The registers used to control and configure DMA operations are part of the memory-mapped IOP register set. These registers are accessed by writing to or (reading from) the appropriate address in memory.

Succeeding sections of this chapter describe the different operating modes of the DMA controller together with the associated control registers and bits. For complete information about the IOP registers, see the *Control/Status Registers* appendix of this manual.

The DMA control registers and data buffer registers are listed in Table 6.2. Note that the serial port and link port DMA control bits are located in the SPORT and link port control registers, not listed in Table 6.2—these control bits are described below under "Serial Port DMA Control" and "Link Port DMA Control."

Two-dimensional DMA mode is enabled by the L2DDMA bit in the LCOM control register and the D2DMA bit in the SRCTL0 and SRCTL1 registers. These bits should be cleared (to 0) for standard DMA operations. Note that references to two-dimensional DMA are not applicable to the ADSP-21061.

Register Name(s)	*Width*	*Description*
EPB0	48	External Port FIFO Buffer 0
EPB1	48	External Port FIFO Buffer 1
EPB2	48	External Port FIFO Buffer 2
EPB3	48	External Port FIFO Buffer 3
DMAC6	16	DMA Channel 6 Control Register (Ext. Port Buffer 0 or Link Buffer 4)[1, 2]
DMAC7	16	DMA Channel 7 Control Register (Ext. Port Buffer 1 or Link Buffer 5)[1, 2]
DMAC8	16	DMA Channel 8 Control Register (Ext. Port Buffer 2)[3]
DMAC9	16	DMA Channel 9 Control Register (Ext. Port Buffer 3)[3]
DMASTAT	32	DMA Channel Status Register
II0, IM0, C0, CP0 GP0, DB0, DA0	16-18	DMA Channel 0 Parameter Registers (SPORT0 Receive)[4]
II1, IM1, C1, CP1 GP1, DB1, DA1	16-18	DMA Channel 1 Parameter Registers (SPORT1 Receive or Link Buffer 0)[1, 2, 4, 5]
II2, IM2, C2, CP2 GP2, DB2, DA2	16-18	DMA Channel 2 Parameter Registers (SPORT0 Transmit)[4, 5]
II3, IM3, C3, CP3 GP3, DB3, DA3	16-18	DMA Channel 3 Parameter Registers (SPORT1 Transmit or Link Buffer 1)[1, 2, 4, 5]
II4, IM4, C4, CP4 GP4, DB4, DA4	16-18	DMA Channel 4 Parameter Registers (Link Buffer 2)[1, 5]
II5, IM5, C5, CP5 GP5, DB5, DA5	16-18	DMA Channel 5 Parameter Registers (Link Buffer 3)[1, 5]
II6, IM6, C6, CP6 GP6, EI6, EM6, EC6	16-32	DMA Channel 6 Parameter Registers (Ext. Port Buffer 0 or Link Buffer 4)[1, 2]
II7, IM7, C7, CP7 GP7, EI7, EM7, EC7	16-32	DMA Channel 7 Parameter Registers (Ext. Port Buffer 1 or Link Buffer 5)[1, 2]
II8, IM8, C8, CP8 GP8, EI8, EM8, EC8	16-32	DMA Channel 8 Parameter Registers (Ext. Port Buffer 2)[3]
II9, IM9, C9, CP9 GP9, EI9, EM9, EC9	16-32	DMA Channel 9 Parameter Registers (Ext. Port Buffer 3)[3]

Table 6.2 DMA Control, Buffer, & Parameter Registers

1. DMA control, buffer, and parameter registers associated with the link ports are not applicable to the ADSP-21061.
2. There are no shared DMA channels on the ADPS-21061.
3. DMA control, buffer, and parameter registers associated wiht DMA channels 8 and 9 are not applicalbe to the ADSP-21061.
4. The IM0, IM1, IM2, and IM3 registers contain the fixed value of 1 on the ADSP-21061.
5. The DBx and DAx registers are not available on the ADSP-21061 because there is no 2-D DMA on the ADSP-21061.

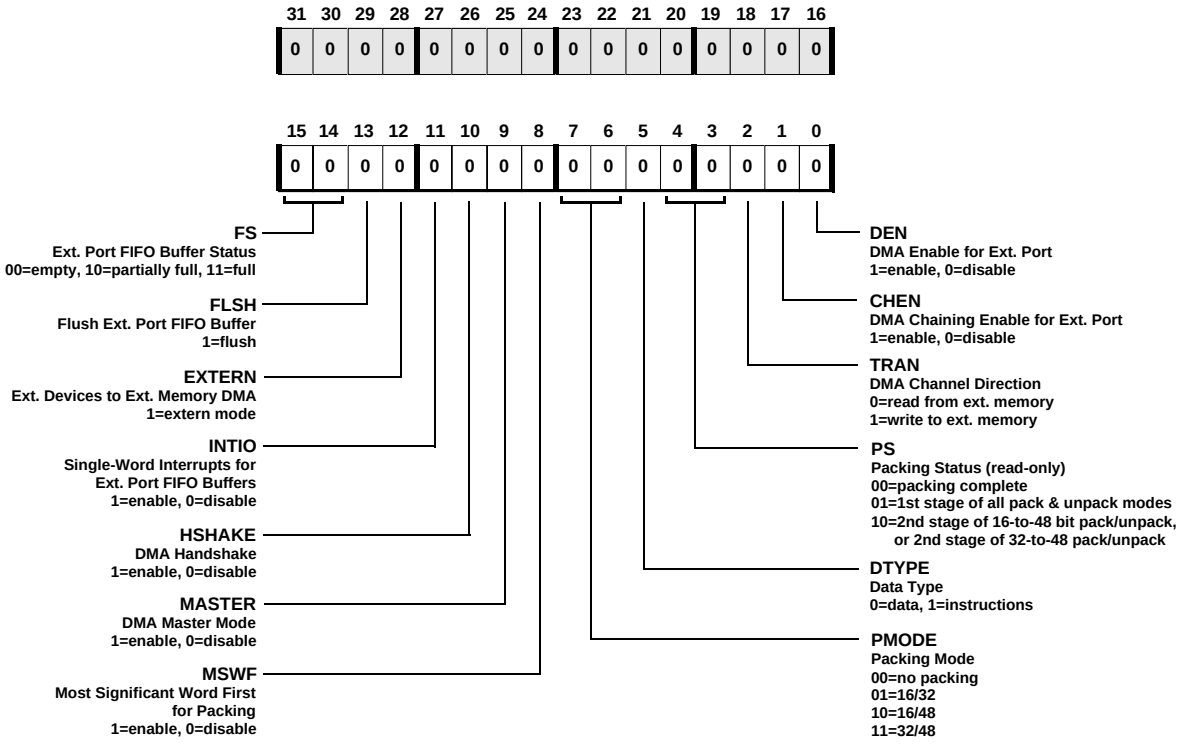


Figure 6.3 DMACx Registers

6.2.1 External Port DMA Control Registers

Each external port DMA channel has its own control register. The registers are named DMAC6, DMAC7, DMAC8, and DMAC9, corresponding to channels 6-9. Note that for the ADSP-21061 only DMA channels 6 and 7 of the external port are applicable. Table 6.3 shows the contents of the DMACx registers. All bits are active high unless stated otherwise.

The control bits in the DMACx registers take effect during the second cycle after the write to the register is completed. The exception to this rule is the FLSH bit, which takes effect in the third cycle after the write.

To start a new DMA sequence after the current one is finished, your program must first clear the DEN enable bit, write new parameters to the II, IM, and C registers, and then set the DEN bit to re-enable DMA. (For

chained DMA operations, however, this is not necessary.)

Bit(s)	*Name*	*Definition*
0	DEN	DMA Enable for External Port
1	CHEN	DMA Chaining Enable for External Port
2	TRAN	Transmit / Receive (1=transmit, 0=receive)
3-4	PS	Pack Status (read-only)
5	DTYPE	Data Type (0=data, 1=instructions)
6-7	PMODE	Packing Mode (00=none, 01=16 / 32, 10=16 / 48, 11=32 / 48)
8	MSWF	Most Significant Word First during packing
9	MASTER	Master Mode Enable
10	HSHAKE	Handshake Mode Enable ($\overline{\text{DMAR}}$x, $\overline{\text{DMAG}}$x)
11	INTIO	Single-Word Interrupt Enable for external port buffers
12	EXTERN	External Handshake Mode Enable
13	FLSH	Flush DMA Buffers & Status
14-15	FS	External port buffer status (00=empty, 11=full, 10=partially full)
16-31	*reserved*	

Table 6.3 External Port DMA Control Registers (DMACx)

The control and status bits in the DMACx registers are further described below:

DEN — Enables DMA for the external port buffers. (Note that the DMA channels shared between the external port and link ports, channels 6 and 7, may also become enabled by the link buffers; see the "Selection Of Shared DMA Channels" section of this chapter. Also note that for the ADSP-21061 there are no shared DMA channels.)

CHEN — Enables chained DMA transfers. When CHEN=1 and DEN=0, the DMA channel is placed in *chain insertion* mode in which a new DMA chain can be inserted into the current chain without affecting the current DMA transfer. This mode of operation is identical to CHEN=1 and DEN=1 except that automatic chaining is disabled when the current DMA transfer ends. The complete list of modes selected by the CHEN and DEN bits are as follows:

CHEN	*DEN*	*Mode of Operation*
0	0	Chaining disabled, DMA disabled
0	1	Chaining disabled, DMA enabled
1	0	*Chain Insertion* mode (chaining enabled, DMA enabled, auto-chaining disabled)
1	1	Chaining enabled, DMA enabled, auto-chaining enabled

TRAN Transmit (1) or Receive (0). (1=read from ADSP-2106x, 0=write to ADSP-2106x.) This bit specifies the data transfer direction as internal-to-external when set to 1. (When EXTERN=1, setting TRAN=1 specifies a read from external memory and TRAN=0 specifies a write to external memory.)

PS PS is a two-bit status field that indicates whether the packing buffer is on its first, second, or last pack:

PS	*Status*
00	pack complete
01	1st stage of all pack and unpack modes
10	2nd stage of 16-to-48 bit pack or unpack modes, or 2nd stage of 32-to-48 bit pack or unpack modes
11	*reserved*

DTYPE Specifies the type of data being transferred; this information is used by internal memory to determine the word width. DTYPE=1 overrides the IMDW bits and forces a 48-bit (3-column) memory transfer. DTYPE=0 defers to the data word setting of the IMDW bits in the SYSCON register. The data word may be 32-bit or 40-bit, as determined by the setting of the IMDW bits in the SYSCON register.

PMODE PMODE is a two-bit value specifying the EPBx buffer packing mode. For host processor accesses of the EPBx buffers, the HPM bits of the SYSCON register must be set to correspond to the external bus width specified by PMODE.

PMODE	*Packing Mode*
00	No packing/unpacking
01	16-bit external bus to/from 32-bit internal packing
10	16-bit external bus to/from 48-bit internal packing
11	32-bit external bus to/from 48-bit internal packing

MSWF Specifies the order in which words are packed, for 16-to-32 bit packing and 16-to-48 bit packing. MSWF is ignored for 32-to-48 bit packing. When MSWF=1, packing is done MSW first (most significant 16-bit word first). When MSWF=0, packing is done LSW first.

INTIO Used when DEN=0, to allow the external port DMA interrupts to occur for individual words received and transmitted. Generating DMA interrupts in this fashion is useful for implementing interrupt-driven single-word transfers under control of the ADSP-2106x core processor. Setting INTIO=1 causes the interrupts to occur when an EPBx input buffer is “not empty” (for TRAN=0) or when an output buffer is “not full” (for TRAN=1).

FLSH — Reinitializes the state of the DMA channel, clearing the FS and PS status bits to zero. The external port FIFO buffer and DMA request counter are flushed and any internal DMA states are reset. Any partially packed data words are also flushed. The entire flushing operation has a two-cycle latency. FLSH is a self-clearing control bit which is not latched and will always read as a 0.

The FLSH bit should only be used to clear the DMA channel when the channel is inactive. *Use of the FLSH bit while the channel is active may cause unexpected results.* The DMASTAT register can be read to determine if the channel is active. (For a particular channel, the *channel active* status bit in DMASTAT will be set if DMA is enabled and the current DMA sequence has not completed.)

The FLSH bit should only be set to 1 at the same time the DEN enable bit is cleared, or when the DEN bit is already equal to 0. Do not set FLSH to 1 in the same write that sets DEN to 1.

FS — FS is a two-bit status field that indicates whether data is present in the EPBx FIFO buffer. When data is being transferred out from the ADSP-2106x, these status bits indicate whether there is room in the buffer for more data. When data is being transferred into the ADSP-2106x, these status bits indicate whether new (unread) data is available in the buffer.

FS	*Status*
00	empty
01	*undefined*
10	partially full
11	full

MASTER — Master Mode DMA Enable. The MASTER, HSHAKE, and EXTERN bits are used in combination, as described below.

HSHAKE — DMA Handshake Enable. The MASTER, HSHAKE, and EXTERN bits are used in combination, as described below.

EXTERN — Specifies an external memory to external device DMA transfer. HSHAKE must equal 1 and MASTER equal 0 in this mode.

The MASTER, HSHAKE, and EXTERN bits configure the DMA mode in the following manner:

M	*H*	*E*	*DMA Mode of Operation*
0	0	0	**Slave Mode.** The DMA request is generated whenever the receive buffer is not empty or the transmit buffer is not full.[1]
0	0	1	*Reserved*
0	1	0	**Handshake Mode.** (For the ADSP-21060 and ADSP-21062, applies to EPB1, EPB2 buffers, channels 7, 8 only. For the ADSP-21061, applies to EPB0, EPB1 buffers, channels 6, 7 only.) The DMA request is generated when the $\overline{\text{DMAR}}$x line is asserted. The transfer occurs when $\overline{\text{DMAG}}$x is asserted.[1]
0	1	1	**External Handshake Mode.** (For the ADSP-21060 and ADSP-21062, applies to EPB1, EPB2 buffers, channels 7, 8 only. For the ADSP-21061, applies to EPB0, EPB1 buffers, channels 6, 7 only.) Identical to Handshake Mode, but with data transferred between external memory and an external device.
1	0	0	**Master Mode.** The DMA controller will attempt a transfer whenever the receive buffer is not empty or the transmit buffer is not full and the DMA counter is non-zero.[1] $\overline{\text{DMAR}}$1 should be kept high (inactive) if channel 7 is in master mode, and $\overline{\text{DMAR}}$2 should be kept high if channel 8 is in master mode on the ADSP-21060 or ADSP-21062. $\overline{\text{DMAR}}$2 should be kept high if channel 6 is in master mode on the ADSP-21061.
1	0	1	*Reserved*
1	1	0	**Paced Master Mode.** (For the ADSP-21060 and ADSP-21062, applies to EPB1, EPB2 buffers, channels 7, 8 only. For the ADSP-21061, applies to EPB0, EPB1 buffers, channels 6, 7 only.) In this mode the transfers are paced by the $\overline{\text{DMAR}}$x signal—the DMA request is generated when $\overline{\text{DMAR}}$x is asserted. $\overline{\text{DMAR}}$x requests operate in the same way as in handshake mode. The bus transfer occurs when $\overline{\text{RD}}$ or $\overline{\text{WR}}$ is asserted. The address is driven as in normal master mode. No external gates are required to OR the $\overline{\text{RD}}$-$\overline{\text{DMAG}}$x and $\overline{\text{WR}}$-$\overline{\text{DMAG}}$x pairs, thus allowing the buffer access to be zero-waitstate with no idle states. Waitstates and acknowledge (ACK) apply to Paced Master Mode transfers; see Section 5.4.4, “Wait States & Acknowledge” in Chapter 5, *Memory*.
1	1	1	*Reserved*

1. If data is to be read from the ADSP-2106x (i.e. TRAN=1), the EPBx buffer will be filled as soon as the DEN enable bit is set to 1.

6.2.2 Serial Port DMA Control

The ADSP-2106x's two serial ports, SPORT0 and SPORT1, can use DMA transfers to handle transmit and receive data. DMA channels 0-3 are assigned to the serial ports, with channels 1 and 3 for SPORT1 being shared with link buffers 0 and 1 on the ADSP-21060 and ADSP-21062. See Table 6.4 below. The direction of SPORT DMA transfers is hardwired—receive channels send data to internal memory, while transmit channels take data from internal memory.

DMA Channel #	*Data Buffer*	*Description*
DMA Channel 0	RX0	Serial Port 0 Receive
DMA Channel 1	RX1 *(or LBUF0)*	Serial Port 1 Receive *(or Link Buffer 0)*[1]
DMA Channel 2	TX0	Serial Port 0 Transmit
DMA Channel 3	TX1 *(or LBUF1)*	Serial Port 1 Transmit *(or Link Buffer 1)*[1]

1. There are no shared DMA channels on the ADSP-21061.

Table 6.4 Serial Port DMA Channels

32-bit words are transferred internally between the RX/TX buffers and memory. If 16-bit serial words are being received or transmitted, they can be transferred two at a time by using the SPORTs' packing capability. See "Data Packing & Unpacking" in the "Data Word Formats" section of the *Serial Ports* chapter for details.

Serial port DMA transfers must be set up in the DMA parameter registers for channels 0-3. Table 6.2 lists these registers. The serial port DMA enable bits are located in the SPORT transmit and receive control registers, STCTL0, STCTL1, SRCTL0, and SRCTL1. These registers are fully described in the *Serial Ports* chapter. Table 6.5 below shows the control bits relating to serial port DMA. These bits are active high: 0=disabled, 1=enabled.

Bit	*Function*
SDEN	SPORT DMA enable
SCHEN	SPORT DMA chaining enable
D2DMA	2-D DMA enable *(for receive only, in SRCTLx register)* *(Two -dimensional DMA is not available on the ADSP-21061.)*

Table 6.5 STCTLx/SRCTLx Control Bits For Serial Port DMA

The D2DMA bit places the DMA controller in two-dimensional SPORT DMA mode on the ADSP-21060 and ADSP-21062. Two-dimensional SPORT DMA mode is not applicable to the ADPS-21061. This bit should be cleared (to 0) for standard operation.

Each serial port has a transmit DMA interrupt and a receive DMA interrupt. When serial port DMA is not enabled, a TX interrupt occurs when the TX buffer is not full and a RX interrupt occurs when the RX buffer is not empty.

Interrupt Name	*Interrupt*	
SPR0I	SPORT0 Receive DMA Channel	**HIGHEST PRIORITY**
SPR1I	SPORT1 Receive DMA Channel	
SPT0I	SPORT0 Transmit DMA Channel	
SPT1I	SPORT1 Transmit DMA Channel	**LOWEST PRIORITY**

Table 6.6 SPORT DMA Interrupts

6.2.3 Link Port DMA Control

The six link ports on ADSP-21060 and ADSP-21062 DSPs can also use DMA transfers to handle transmit and receive data. DMA channels 4 and 5 are dedicated to link buffers 2 and 3, respectively. The other link buffers share DMA channels with the serial ports and external port. [Note that the discussion in this section applies only to ADSP-21060 and ADSP-21062 DSPs; the topics here do not apply to the ADSP-21061 DSP because this DSP does not have link ports.]

DMA Channel #	*Data Buffer*	*Description*
DMA Channel 1	RX1 *(or LBUF0)*	Serial Port 1 Receive *(or Link Buffer 0)*
DMA Channel 3	TX1 *(or LBUF1)*	Serial Port 1 Transmit *(or Link Buffer 1)*
DMA Channel 4	LBUF2	Link Buffer 2
DMA Channel 5	LBUF3	Link Buffer 3
DMA Channel 6	EPB0 *(or LBUF4)*	External Port Buffer 0 *(or Link Buffer 4)*
DMA Channel 7	EPB1 *(or LBUF5)*	External Port Buffer 1 *(or Link Buffer 5)*

Table 6.7 Link Port DMA Channels

Link port DMA operations are set up in the DMA parameter registers for each channel. Table 6.2 lists these registers. Either 32- or 48-bit word widths can be used in link port DMA transfers.

The link buffer DMA enable and control bits are located in the LCTL register. Table 6.8 shows these control bits, which are active high (i.e. 0=disabled, 1=enabled). The LCOM register contains the L2DDMA bit; this bit places the DMA controller in two-dimensional DMA mode for the link ports. This bit should be cleared (to 0) for standard operation.

Bit(s)	*Name*	*Definition*
0-3	*	Link Buffer 0 controls
4-7	*	Link Buffer 1 controls
8-11	*	Link Buffer 2 controls
12-15	*	Link Buffer 3 controls
16-19	*	Link Buffer 4 controls
20-23	*	Link Buffer 5 controls
24	LEXT0	Extended word size**
25	LEXT1	Extended word size**
26	LEXT2	Extended word size**
27	LEXT3	Extended word size**
28	LEXT4	Extended word size**
29	LEXT5	Extended word size**
30-31	*reserved*	

Table 6.8 LCTL Control Bits For Link Port DMA

* Each four-bit group includes the following control bits for each link buffer (x=0,1,2,3,4,5):

Bit#	*Name*	*Definition*
0+4x	LxEN	LBUFx enable
1+4x	LxDEN	LBUFx DMA enable
2+4x	LxCHEN	LBUFx chaining enable
3+4x	LxTRAN	LBUFx direction: 1=transmit, 0=receive

** Extended word size: 1=48-bit link port transfers, 0=32-bit link port transfers

Each link buffer has a DMA interrupt, listed in Table 6.9 below. When link port DMA is not enabled, an interrupt is generated whenever a receive buffer is not empty or a transmit buffer is not full.

Interrupt Name	*Interrupt*
SPR1I	DMA Channel 1 – SPORT1 Rx (or Link Buffer 0)
SPT1I	DMA Channel 3 – SPORT1 Tx (or Link Buffer 1)
LP2I	DMA Channel 4 – Link Buffer 2
LP3I	DMA Channel 5 – Link Buffer 3
EP0I	DMA Channel 6 – Ext. Port Buffer 0 (or Link Buffer 4)
EP1I	DMA Channel 7 – Ext. Port Buffer 1 (or Link Buffer 5)

Table 6.9 Link Buffer DMA Interrupts

6.2.4 Port Selection For Shared DMA Channels

DMA Channel 1 and Channel 3 are shared by Serial Port 1 and Link Buffers 0 and 1. Similarly, DMA Channel 6 and Channel 7 are shared by External Port Buffers 0 and 1 and Link Buffers 4 and 5. [Note that the discussion in this section applies only to ADSP-21060 and ADSP-21062 DSPs; the topics here do not apply to the ADSP-21061 DSP because this DSP does not have shared DMA channels.]

DMA Channel #	*Data Buffer*	*Description*
DMA Channel 1	RX1 *(or LBUF0)*	SPORT1 Receive *(or Link Buffer 0)*
DMA Channel 3	TX1 *(or LBUF1)*	SPORT1 Transmit *(or Link Buffer 1)*
DMA Channel 6	EPB0 *(or LBUF4)*	External Port Buffer 0 *(or Link Buffer 4)*
DMA Channel 7	EPB1 *(or LBUF5)*	External Port Buffer 1 *(or Link Buffer 5)*

Channel 1 is assigned to either the SPORT1 Receive buffer or Link Buffer 0 according the following rules:

- If the SPORT1 Receive DMA enable bit is set (SDEN=1), then Channel 1 is assigned to it.
- If the Link Buffer 0 DMA enable bit is set (L0DEN=1), then Channel 1 is assigned to it.
- If both enables are set, SPORT1 Receive is selected.
- If neither enable is set, then the interrupts from the two buffers are ORed together.

Channel 3 is assigned to either SPORT1 Transmit or Link Buffer 1 in the same way.

Channel 6 is assigned to either External Port Buffer 0 or Link Buffer 4 according the following rules:

- If the External Port DMA enable bit is set in the DMAC6 control register (DEN=1), then Channel 6 is assigned to EPB0.
- If the Link Buffer 4 DMA enable bit is set (L4DEN=1), then Channel 6 is assigned to it.
- If both enables are set, EPB0 is selected.
- If neither enable is set, then the interrupts from the two buffers are ORed together.

Channel 7 is assigned to either External Port Buffer 1 or Link Buffer 5 in the same way.

6.2.5 DMA Channel Status Register (DMASTAT)

The ADSP-2106x's DMA controller maintains a 32-bit read-only status register called DMASTAT, described in Table 6.10. Bits 0-9 of DMASTAT indicate which DMA channels are active, with bit 0 corresponding to channel 0, and so on. Bits 10-19 indicate DMA chaining status for each channel. [Note that bits 4, 5, 8, 9, 14, 15, 18, and 19 are not valid for the ADSP-21061.]

Bit#	*Definition*
0	DMA Channel 0 Status[1]
1	DMA Channel 1 Status[1]
2	DMA Channel 2 Status[1]
3	DMA Channel 3 Status[1]
4	DMA Channel 4 Status[1, 3]
5	DMA Channel 5 Status[1, 3]
6	DMA Channel 6 Status[1]
7	DMA Channel 7 Status[1]
8	DMA Channel 8 Status[1, 3]
9	DMA Channel 9 Status[1, 3]
10	DMA Channel 0 Chaining Status[2]
11	DMA Channel 1 Chaining Status[2]
12	DMA Channel 2 Chaining Status[2]
13	DMA Channel 3 Chaining Status[2]
14	DMA Channel 4 Chaining Status[2, 3]
15	DMA Channel 5 Chaining Status[2, 3]
16	DMA Channel 6 Chaining Status[2]
17	DMA Channel 7 Chaining Status[2]
18	DMA Channel 8 Chaining Status[2, 3]
19	DMA Channel 9 Chaining Status[2, 3]
20-31	*reserved*

Table 6.10 DMASTAT Register

1. **Channel Status: 1 (active)**=transferring data or waiting to transfer the current block, and not transferring TCB. **0 (inactive)**=DMA disabled, transfer complete, or transferring TCB.
2. **Channel Chaining Status: 1=**transferring TCB or waiting to transfer TCB. **0**=chaining disabled, or not transferring TCB.
3. **Does not apply to the ADSP-21061**.

Note 1: Status does not change on the master ADSP-2106x during external port DMA until the external portion is completed (i.e., the EPBx buffers are emptied).

Note 2: If in chain insertion mode (DEN=0, CHEN=1), then *channel chaining status* will never go to 1. Therefore, test *channel status* to see if it is ready so that your program can rewrite the chain pointer (CPx register).

For a particular channel, the *channel active* status bit will be set if DMA is enabled and the current DMA sequence has not completed. The *chaining* status bit will be set if the channel is currently performing chaining operations or if chaining is pending. There will be a single cycle of latency between internal status changes and the update of the DMASTAT register.

As an alternative to interrupt-driven DMA, polling DMASTAT can be used to determine when a single DMA sequence has completed:

1. Read DMASTAT.
2. If both status bits for the channel are inactive, the DMA sequence has completed.

If chaining is enabled, however, polling should not be used since the next DMA sequence may be under way by the time the polled status is returned.

6.3 DMA CONTROLLER OPERATION

The following sections discuss the operation of the ADSP-2106x's DMA controller and describe how DMA transfers occur.

In the ADSP-2106x, the DMA controller operations are centered on the internal I/O bus. The serial ports, link ports, and external port are connected to the internal memory via the I/O Data bus (IOD), and the DMA controller generates internal memory addresses on the I/O Address bus (IOA).

The DMA controller maintains 10 DMA channels that are used by the external port, the link ports, and the serial ports. A DMA channel consists of a set of parameter registers which specify a data buffer in internal memory, plus the hardware required by an I/O port to request DMA service.

To transfer data, the DMA controller accepts internal requests from I/O ports and sends back an internal grant when they are serviced. The DMA controller contains priority logic to determine which channel can drive the bus in any given cycle. The DMA transfer never conflicts with the core for internal memory accesses because the internal memory has separate ports for core and I/O accesses.

Each external port DMA channel has a control/status register which is used to set the operating mode of the channel and to return status information. All of the DMA control and parameter registers are accessible to external devices. This allows a host, or another ADSP-2106x, to set up a DMA channel and initiate transfers without local ADSP-2106x involvement. The local ADSP-2106x can set up a DMA

channel on itself by writing to its DMA control and parameter registers.

The external port and link port DMA channels can be configured to transmit or receive data from internal memory. The serial port DMA channels, however, are unidirectional, either transmit or receive only.

6.3.1 DMA Channel Parameter Registers

The DMA channels operate in a similar fashion as the ADSP-2106x's Data Address Generators (DAGs). Each channel has of a set of parameter registers including an index register (IIx) and modify register (IMx) which are used to set up a data buffer in internal memory. The index register must be initialized with a starting address for the data buffer. The address in the index register is output onto the ADSP-2106x's IOA (I/O Address) bus and applied to internal memory during each DMA cycle. (A DMA cycle is defined as a clock cycle in which a DMA transfer is taking place.)

All addresses in the 17-bit index registers are offset by 0x0002 0000, the first internal RAM location, before they are used by the DMA controller. Since the index registers are only 17 bits wide, DMA transfers cannot be made to short word address space. (16-bit short word data can be transferred within 32-bit words, however, using the packing capability of the external port and serial port DMA channels.)

After each data word is transferred to or from internal memory, the DMA controller adds the modify value to the index register to generate the address for the next DMA transfer; the modify value is added to the index value and written back into the index register. The modify value in the IM register is a signed integer, to allow both incrementing and decrementing. Note that if the index register is modified past its maximum 17-bit value (i.e. out of the address range of internal memory), it will wraparound to zero (offset by 0x0002 0000). The modify value for DMA channels 0-3 is fixed to 1 on the ADSP-21061; this DSP does not have the IM0-3 registers.

Each DMA channel has a count register (Cx) which must be initialized with a word count to be transferred. The count register is decremented after each DMA transfer on that channel; when the count reaches zero, the interrupt for that channel can be generated.

Caution: If the count register is initialized with zero, DMA transfers on

that channel are *not* disabled. Rather, 2^{16} transfers will be performed. This occurs because the first transfer is started *before* the count value is tested. The correct way to disable a DMA channel is to clear its DMA enable bit in the corresponding control register.

> *To start a new DMA sequence after the current one is finished, your program must first clear the DEN enable bit, write new parameters to the II, IM, and C registers, and then set the DEN bit to re-enable DMA.*

(For chained DMA operations, however, this is not necessary.)

Each DMA channel also has a chain pointer register (CPx) and a general-purpose register (GPx). The CP register is used in chained DMA operations (as described below in "DMA Chaining"), and the GP register can be used for any purpose.

The external port DMA channels each contain three additional parameter registers, the external index register (EIx), external modify register (EMx), and external count register (ECx). (These registers are not included in the serial port and link port DMA channels.) The EI, EM, and EC registers are used to generate 32-bit addresses driven out of the external port, for master mode DMA transfers between internal memory and external memory or devices. Master mode is configured by the MASTER bit of each DMACx control register. The EC register must be loaded with the number of external bus *transfers* to occur (in master mode only). (**Note:** This differs from the number of *words* transferred by the DMA controller if word packing is used.) EIX may not index internal memory. If DMA data is broadcast, write space data is not written to the internal memory of the broadcaster.

Instead of the EI, EM, and EC registers, the serial port and link port DMA channels have the DA and DB registers. These registers are used for two-dimensional array addressing in mesh multiprocessing applications, but may also be used as general-purpose registers in standard, one-dimensional DMA operations.

Figure 6.4 shows a block diagram of the DMA controller's address generator. Table 6.11 defines the DMA parameter registers, and Table 6.12 lists the parameter registers for each DMA channel. The parameter registers are uninitialized following a processor reset.

Parameter Register	*#of Bits*	*Function*
IIx	17	Internal Index (starting address for data buffer – 0x0002 0000)
IMx	16	Internal Modifier (address increment)[1]
Cx	16	Internal Count (number of words to transfer)
CPx	18*	Chain Pointer (address of next set of buffer parameters)[2]
GPx	17	General-Purpose (or 2D DMA)[3]
EIx	32	External Index (Ext. Port DMA channels only)
EMx	32	External Modifier (Ext. Port DMA channels only)
ECx	32	External Count (Ext. Port DMA channels only)
DBx	16	General-Purpose or 2D DMA (Link / SPORT channels only)[3]
DAx	16	General-Purpose or 2D DMA (Link / SPORT channels only)[3]

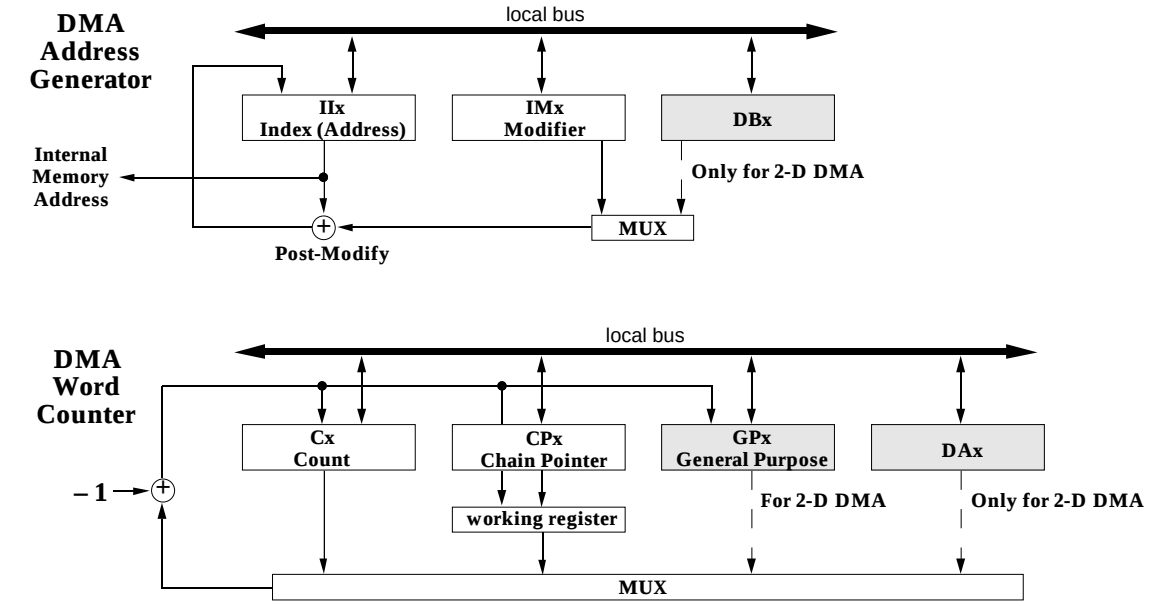
Table 6.11 DMA Parameter Registers

1. The modify value of DMA channels 0-3 is fixed to '1' on the ADSP-21061.
2. Lower 17 bits (bits 16-0) contain memory address of the next set of parameters for chained DMA operations. Most significant bit (bit 17) is the PCI bit (Program-Controlled Interrupts), which determines whether the DMA interrupts occur at the completion of each DMA sequence.
3. Two-dimensional DMA is not available on the ADSP-21061; this DSP does not have the DBx or DAx registers.

Register Names	*Description*
II0, IM0, C0, CP0 GP0, DB0, DA0	DMA Channel 0 Parameter Registers (SPORT0 Receive)[1]
II1, IM1, C1, CP1 GP1, DB1, DA1	DMA Channel 1 Parameter Registers (SPORT1 Receive *or* Link Buffer 0)[1]
II2, IM2, C2, CP2 GP2, DB2, DA2	DMA Channel 2 Parameter Registers (SPORT0 Transmit)[1]
II3, IM3, C3, CP3 GP3, DB3, DA3	DMA Channel 3 Parameter Registers (SPORT1 Transmit *or* Link Buffer 1)[1]
II4, IM4, C4, CP4 GP4, DB4, DA4	DMA Channel 4 Parameter Registers (Link Buffer 2)[2]
II5, IM5, C5, CP5 GP5, DB5, DA5	DMA Channel 5 Parameter Registers (Link Buffer 3)[2]
II6, IM6, C6, CP6 GP6, EI6, EM6, EC6	DMA Channel 6 Parameter Registers (Ext. Port Buffer 0 *or* Link Buffer 4)
II7, IM7, C7, CP7 GP7, EI7, EM7, EC7	DMA Channel 7 Parameter Registers (Ext. Port Buffer 1 *or* Link Buffer 5)
II8, IM8, C8, CP8 GP8, EI8, EM8, EC8	DMA Channel 8 Parameter Registers (Ext. Port Buffer 2)[2]
II9, IM9, C9, CP9 GP9, EI9, EM9, EC9	DMA Channel 9 Parameter Registers (Ext. Port Buffer 3)[2]

Table 6.12 Parameter Registers For Each DMA Channel

1. The values in the IM0-3 registers are fixed to '1' on the ADSP-21061; this DSP does not have DAx and DBx registers.
2. These sets of registers are not available on the ADSP-21061.



Only for External Port DMA Channels

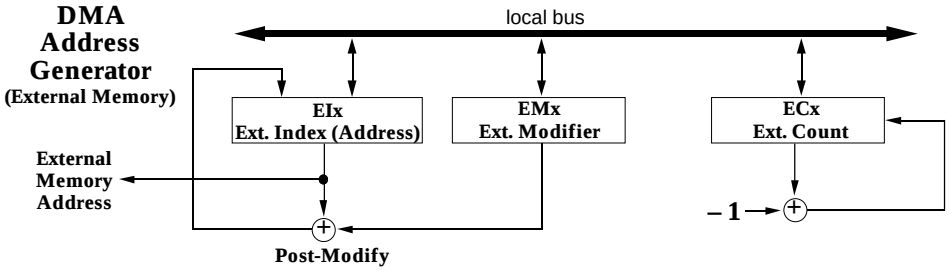


Figure 6.4 DMA Address Generation

6.3.2 Internal Request & Grant

The ADSP-2106x's I/O ports communicate with the DMA controller by means of an internal DMA request/grant handshake. Each I/O port (link ports, serial ports, and external port) has one or more DMA channels, with each channel having a single request and a single grant.

When a particular I/O port needs to write data to internal memory, it asserts its request. This request is prioritized with all other valid DMA requests. See Figure 6.2.

When a channel becomes the highest priority requester, its internal grant is asserted by the DMA controller. In the next clock cycle, the DMA transfer is started. When an I/O port wishes to read data from internal memory, the sequence is the same.

If a DMA channel is disabled, no grants will be given for that channel, regardless of whether it has data to transfer.

6.3.3 DMA Channel Prioritization

Since more than one DMA channel may have a request active in a particular cycle, a prioritization scheme is used to select the channel to service. Prioritization is needed to determine which channel can use the IOD (I/O Data) bus to access memory. The ADSP-2106x always uses a fixed prioritization (except for the external port DMA channels, as described below). Table 6.13 lists in descending order of priority the possible I/O bus accesses including DMA channels .

– *Core Accesses to DA Group Registers* **HIGHEST PRIORITY**
Channel 0 – Serial Port 0 Receive
Channel 1 – Serial Port 1 Receive (or Link Buffer 0)
Channel 2 – Serial Port 0 Transmit
Channel 3 – Serial Port 1 Transmit (or Link Buffer 1)
– *TCB Chain Loading Requests*[1]
– *External Accesses of Internal Memory (Direct Reads, Direct Writes)*[2]
Channel 4 – Link Buffer 2[3]
Channel 5 – Link Buffer 3[3]
Channel 6 – Ext. Port Buffer 0 (or Link Buffer 4)[4]
Channel 7 – Ext. Port Buffer 1 (or Link Buffer 5)[4]
Channel 8 – Ext. Port Buffer 2[3, 4]
Channel 9 – Ext. Port Buffer 3[3, 4] **LOWEST PRIORITY**

Table 6.13 Internal Memory I/O Bus Access Priority

1. TCB chain loading uses the I/O bus and therefore requires prioritization. (See "DMA Chaining" below.)
2. Direct reads and writes use the I/O bus and therefore require prioritization. (See "Direct Writes" and "Direct Reads" in the *Host Interface* chapter.)
3. These DMA channels are not available on the ADSP-21061
4. Rotating priority can be selected for External Port Buffers.

The DMA controller determines the highest priority requesting channel during every cycle, between each individual data transfer. Master/slave bus request prioritization, however, occurs only when the ADSP-2106x master gives up control of the external bus—this occurs only after an entire DMA block transfer has completed.

Note that external direct accesses of internal memory and TCB chain loading are prioritized along with the DMA channels. This is necessary to prevent I/O bus contention, because these accesses are also performed over the internal I/O bus. TCB chain loading is given a higher priority than external port accesses to allow serial port DMA transfers (which cannot be held off) to be chained, even when the external port is attempting an access in every cycle. (TCB chain loading is explained in the section "DMA Chaining" below.)

6.3.3.1 Rotating Priority For Ext. Port Channels

The DMA controller can be programmed to use a *rotating priority* scheme for the four external port channels by setting the DCPR bit in the SYSCON register:

Bit	*Function*
DCPR	Enables rotating priority for external port DMA channels 6-9 on the ADSP-21060 and ADSP-21062. Enables rotating priority for external port DMA channels 6-7 on the ADSP-21061. (0=disabled, 1=enabled)

When rotating priority is enabled, high priority shifts to a new channel after each single-word transfer. The order of channel priority then rotates. Thus, a single-word transfer is serviced, then priority rotates to the next higher-numbered channel, and so on until all four are serviced. Figure 6.5 illustrates this process, according to the following example (applies to the ADSP-21060 and ADSP-21062):

1) After reset, the default priority ordering from high to low is 6, 7, 8, 9.
2) A single transfer is performed on channel 7.
3) Assuming that rotating priority is enabled (DCPR=1), the priority ordering then changes to 8, 9, 6, 7.

For the ADSP-21061, rotating priority works in the same manner described above, except there are only two DMA channels (6-7).

The external port channel priorities do not change relative to the serial port and link port channel priorities. At reset, the DCPR bit is cleared and rotating priority is disabled.

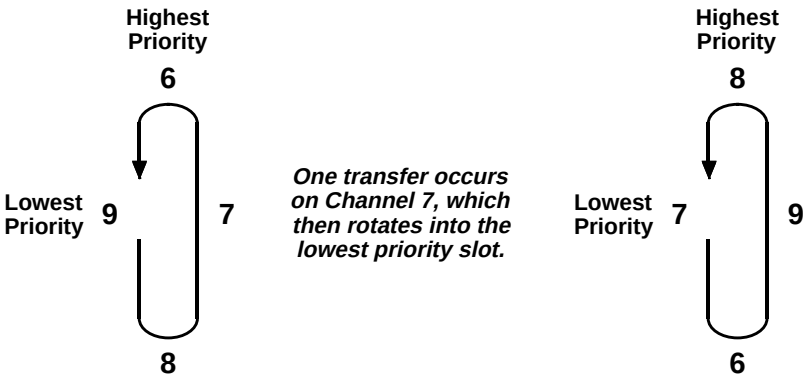


Figure 6.5 Rotating Priority Example (ADSP-21060 & ADSP-21062)

Note: Even though the external port channel DMA priority can rotate, the *interrupt* priorities of all DMA channels are fixed.

When using fixed priority for the external port DMA channels, the highest priority of the four is assigned to Channel 6 and the lowest priority is assigned to Channel 9 (as shown in Table 6.13) for the ADSP-21060 and ADSP-21062. The lowest priority channel on the ADSP-21061 is channel 7. This order of priority can be redefined by assigning one of the other channels the highest priority. To change the fixed priority sequence of the external port DMA channels, use the following procedure:

1) Disable all external port DMA channels except the one which is to have lowest priority.
2) Select rotating priority.
3) Cause at least one transfer to occur on the enabled channel.
4) Disable rotating priority and re-enable all of the external port DMA

channels.

The channel immediately after the selected channel will now have the highest fixed priority, for example (ADSP-21060 & ADSP-21062):

	HIGHEST			LOWEST
Priority at Reset:	**DMA6**	**DMA7**	**DMA8**	**DMA9**
Follow steps 1-4 above to make DMA7 the lowest priority.				
New Priority Ordering:	**DMA8**	**DMA9**	**DMA6**	**DMA7**

6.3.4 DMA Chaining

DMA chaining allows the ADSP-2106x's DMA controller to autoinitialize itself between multiple DMA transfers. Using chaining, you can set up multiple DMA operations in which each operation can have different attributes.

In chained DMA operations, the ADSP-2106x automatically sets up another DMA transfer when the entire contents of the current buffer have been transmitted or received. The chain pointer register (CP) is used to point to the next set of DMA parameters stored in internal memory. This new set of parameters is called a *transfer control block (TCB)*. The ADSP-2106x's DMA controller automatically reads the TCB from internal memory and loads the values into the channel parameter registers to set up the next DMA sequence; this process is called *TCB chain loading*.

A *DMA sequence* is defined as the sum of the DMA transfers for a single channel, from the parameter registers initialization to when the count register decrements to zero.

Each DMA channel has a chaining enable bit (CHEN) in the corresponding control register. This bit must be set to 1 to enable chaining. Writing all zeros to the address field of the chain pointer register (CP) also disables chaining.

When chaining is enabled, DMA transfers are initiated by writing a memory address to the CP register. This is also an easy way to start a single DMA sequence, with no subsequent chained DMAs. The CP register can be loaded at any time during the DMA sequence—this allows a DMA channel to have chaining disabled (CP register address field = 0x0000) until some event occurs that loads the CP register with a non-zero value. DMA chaining operations may only occur within the same channel; cross-channel chaining is not supported.

The CP register is 18 bits wide, of which the lower 17 bits are the memory address field. The memory address field is offset by 0x0002 0000 before it is used by the DMA controller. The most significant bit (bit 17) of the CP register is a control bit called PCI (Program-Controlled Interrupts). The PCI bit selects whether or not an interrupt occurs at the completion of the current DMA sequence (in addition to the interrupt's mask bit in IMASK). When PCI=1, the corresponding DMA channel interrupt is enabled and will occur when the count register reaches zero. When PCI=0, the channel's interrupt is disabled. Note that the PCI bit only affects DMA channels for which chaining is enabled (i.e. CHEN bit set to 1). For non-chained DMA operations, the IMASK register must be used to disable the interrupt. Interrupt requests enabled by the PCI bit can still be masked out (i.e. disabled) in the IMASK register. Figure 6.6 illustrates the PCI bit within the CPx register.

Caution: Because the PCI bit is not part of the memory address in the CP register, care should be taken when writing and reading addresses to and from the register. To prevent errors, it is a good practice to mask out the PCI bit (bit 17) when copying the address in CP to another address register.

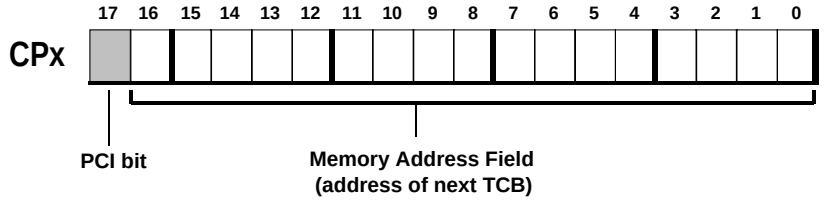


Figure 6.6 Chain Pointer Register & PCI Bit

The general-purpose register (GP) can be useful during chained DMA sequences. It is loaded from memory with the other parameter registers, and can be used to point to the last DMA sequence that was completed. This allows a program to determine where the last full (or empty) data buffer is located. Since it is a general-purpose register with no dedicated functionality, it can be used for any purpose.

6.3.4.1 Transfer Control Blocks & Chain Loading

During TCB chain loading, the DMA channel parameter registers are loaded with values retrieved from internal memory. The CP register contains the chain pointer—the highest address of the TCB. The TCB is stored in consecutive locations.

Table 6.14 below shows the TCB-to-register loading sequence for the external port, link port, and serial port DMA channels (i.e. the order in which the DMA controller reads each word of the TCB and loads it into the corresponding register.) Figure 6.7 shows how you must set up the TCB in memory (for an external port DMA chain), referenced to the address pointer contained in the CP register of the previous DMA operation of the chain.

Address	*External Port & Link Buffers 4,5*	*Serial Ports & Link Buffers 0,1,2,3*
CPx + 0x0002 0000	IIx	IIx
CPx – 1 + 0x0002 0000	IMx	IMx
CPx – 2 + 0x0002 0000	Cx	Cx *(and DAx for 2D DMA)*
CPx – 3 + 0x0002 0000	CPx	CPx
CPx – 4 + 0x0002 0000	GPx	GPx
CPx – 5 + 0x0002 0000	EIx	*DBx (loaded during 2D DMA only)*
CPx – 6 + 0x0002 0000	EMx	*LPATH1 (mesh multiproc. links only)*
CPx – 7 + 0x0002 0000	ECx	*LPATH2 (mesh multiproc. links only)*
CPx – 8 + 0x0002 0000	–	*LPATH3 (mesh multiproc. links only)*

Table 6.14 TCB Chain Loading Sequence

Notes:

1. An “x” denotes the DMA channel number.
2. The DAx and DBx registers are not loaded during chaining in normal, one-dimensional DMA. In 2D DMA operations, only DBx is loaded. The DAx register is automatically loaded with the same value as the Cx register. (2D DMA operations are not applicable to the ADSP-21061.)
3. The link transmit chain also downloads the LPATH1, LPATH2, and LPATH3 registers when the LMSP bit in the LCOM control register is set, enabling mesh multiprocessing. (These registers are not available of the ADSP-21061.)
4. Link Buffers 4 and 5 use the same chaining registers as the external port. All 8 registers are always loaded when chaining on DMA channels 6-9, but EIx, EMx, and ECx are not used when Link Buffers 4 and 5 are enabled. (Link buffers 4-5 are not available on the ADSP-21061.)

A working register is loaded from the CP register before the chain loading sequence begins, and is decremented after each register is loaded. The working register allows the CP register to be updated with the new CP value without interfering with the current register loading.

When the chain loading is complete, the working register is loaded with the new CP value. This allows chained DMA sequences to be set up in a continuous loop. (Note: The contents of the working register are not accessible.)

TCB chain loading is requested like all other DMA operations. A TCB loading request is latched and held in the DMA controller until it becomes the highest priority request. The IOP individually prioritizes and transfers the TCB register like normal DMA. If multiple chaining requests are present, the TCB registers for the highest priority DMA channel are transferred first. A channel which is in the process of chain loading cannot be interrupted by a higher priority channel. Refer to Table 6.13 for the DMA channel request priorities.

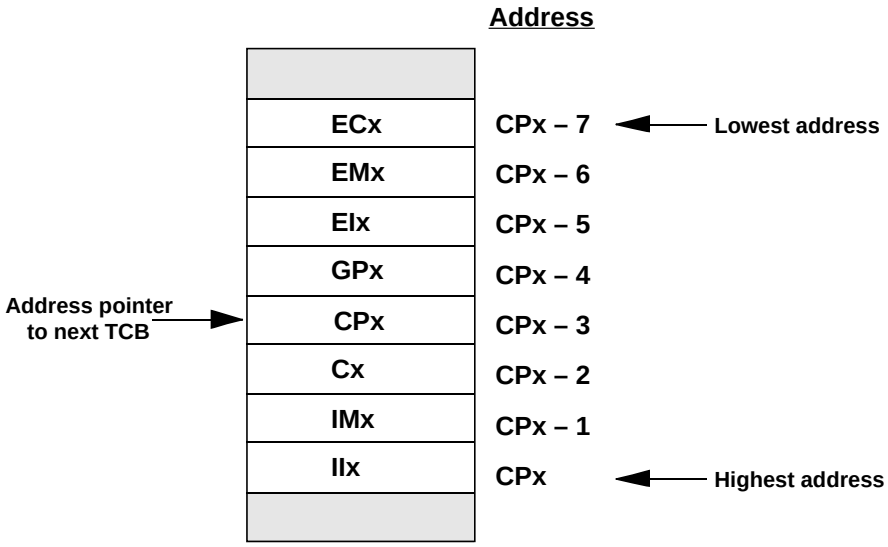


Figure 6.7 TCB Setup In Memory (For External Port DMA Channel)

6.3.4.2 Setting Up & Starting The Chain

To setup and initiate a chain of DMA operations, your program should follow this sequence:

1. Set up all TCBs in internal memory.
2. Write to the appropriate DMA control register, setting the DEN enable bit to 1 and the CHEN chaining enable bit to 1.
3. Write the *last* address (i.e. the address of the IIx register value) of the *first* TCB to the CPx register—this will start the chain.

The DMA controller will autoinitialize itself with the first TCB and then start the first transfer. When this transfer is completed, the next one will begin if the current chain pointer address is non-zero. This address will be used as the pointer to the next TCB.

Remember that the address field of the CPx registers is only 17 bits wide. If a symbolic address is written directly to CPx, bit 17 may conflict with the PCI bit. Be sure to clear the upper bits of the address, then AND in the PCI bit separately if needed.

6.3.4.3 Chain Insertion

A high priority DMA operation or chain can be inserted into an active DMA chain. When CHEN=1 and DEN=0, the DMA channel is placed in *chain insertion* mode in which a new DMA chain can be inserted into the current chain without affecting the current DMA transfer. The new chain is inserted by the ADSP-2106x core writing a TCB into the channel parameter registers. This mode of operation is identical to that selected by CHEN=1 and DEN=1; except that when the current DMA transfer ends, automatic chaining is disabled and an interrupt request occurs. This interrupt request is independent of the PCI bit state.

The following sequence should be used to insert a DMA subchain while another chain is active:

1. Enter chain insertion mode by setting CHEN=1 and DEN=0 in the appropriate DMA control register.
2. The DMA interrupt will indicate when the current DMA sequence has completed.
3. Write the CPx register value into the CP position of the last TCB in the new chain.
4. Set DEN=1 and CHEN=1.
5. Write the start address of the first TCB of the new chain into the CP register.

Chain insertion should not be set up as an initial mode of operation; it is intended for use only while another DMA operation is in progress.

6.3.5 DMA Interrupts

When the count register (C) of an active DMA channel decrements to zero, an interrupt is generated. For the external port DMA channels, both the C and EC (external count) registers must equal zero before the interrupt is generated (EC register only in MASTER mode). The count register(s) must be decremented to zero as a result of actual DMA transfers in order for a DMA interrupt to be generated—writing zero to a count register will not generate the interrupt.

Each DMA channel has its own interrupt; the DMA interrupts are latched in the IRPTL and are enabled in the IMASK register. Table 6.15 shows the IRPTL and IMASK bits of the ten DMA channel interrupts, in order of priority. (Note: Although the external port channel access priority can rotate, the *interrupt* priorities of all DMA channels are fixed.)

IRPTL/ IMASK Bit #	*Vector Address*[1]	*Interrupt Name*	*DMA Channel Interrupt*	
10	0x28	SPR0I	DMA Channel 0 – SPORT0 Receive	**HIGHEST PRIORITY**
11	0x2C	SPR1I	DMA Channel 1 – SPORT1 Receive (or Link Buffer 0)	
12	0x30	SPT0I	DMA Channel 2 – SPORT0 Transmit	
13	0x34	SPT1I	DMA Channel 3 – SPORT1 Transmit (or Link Buffer 1)	
14	0x38	LP2I	DMA Channel 4 – Link Buffer 2[2]	
15	0x3C	LP3I	DMA Channel 5 – Link Buffer 3[2]	
16	0x40	EP0I	DMA Channel 6 – Ext. Port Buffer 0 (or Link Buffer 4)	
17	0x44	EP1I	DMA Channel 7 – Ext. Port Buffer 1 (or Link Buffer 5)	
18	0x48	EP2I	DMA Channel 8 – Ext. Port Buffer 2[2]	
19	0x4C	EP3I	DMA Channel 9 – Ext. Port Buffer 3[2]	**LOWEST PRIORITY**

Table 6.15 DMA Interrupt Vectors & Priority

1. Offset from base address: 0x0002 0000 for interrupt vector table in internal memory, 0x0040 0000 for interrupt vector table in external memory

2. These locations are reserved, not used, on the ADSP-21061.

In addition to IMASK, DMA interrupts for each channel can be enabled or disabled by the PCI bit of the CP register, when DMA chaining is enabled. When PCI=1, DMA interrupt requests occur when the count register reaches zero. When PCI=0, no DMA interrupts are generated. The PCI bit is valid only when DMA chaining is enabled. If

chaining is disabled, the IMASK register must be used to disable interrupts. Interrupt requests enabled by PCI can still be masked out by the IMASK register.

DMA interrupts can also be generated by ADSP-2106x's I/O ports without using DMA. In this case, a DMA interrupt is generated whenever data becomes available at the receive buffer, or whenever the transmit buffer does not have new data to transmit. Generating DMA interrupts in this fashion is useful for implementing interrupt-driven I/O under control of the ADSP-2106x core processor. Multiple interrupts can occur if several I/O ports transmit or receive data in the same cycle. To perform single-word, non-DMA, interrupt-driven transfers on the external port, the INTIO bit must be set in a DMACx control register.

The following list describes the various conditions for which an interrupt will be generated by a DMA channel or its corresponding I/O port:

Condition	*Interrupt Mask*
Chaining disabled, current DMA sequence completes	IMASK
Chaining enabled, current DMA sequence completes	IMASK & PCI
Chain insertion mode, current DMA sequence completes	IMASK
DMA disabled and a buffer is accessed by the I/O port*	IMASK

* INTIO bit must be set in DMACx control register for external port.

If the interrupt mask is a 1 (i.e. unmasked), the interrupt is enabled and will be acknowledged.

The IMASK register is not directly accessible to external devices, via the external port, because it is one of the universal registers in the ADSP-2106x processor core (and is not memory-mapped like the IOP registers). IMASK may be read or written via the external port, however, by using an interrupt vector to a routine set up to handle this task. The VIRPT vector interrupt register may be used for this purpose.

As an alternative to interrupts, polling DMASTAT can be used to determine when a single DMA sequence has completed:

1. Read DMASTAT.
2. If both status bits for the channel are inactive, the DMA sequence has completed.

If chaining is enabled, however, polling DMASTAT should not be used since the next DMA sequence may be under way by the time the polled status is returned.

6.3.6 Starting & Stopping DMA Sequences

DMA sequences are started in different ways depending on whether DMA chaining is enabled. When chaining is not enabled, only the DMA enable bit (DEN) allows DMA transfers to occur.

A DMA sequence starts when one of the following occurs:

- Chaining is disabled and the DMA enable bit (DEN) transitions from low to high.
- Chaining is enabled, DMA is enabled (DEN=1), and the CP register address field is written with a non-zero value. (In this case, TCB chain loading of the channel parameter registers occurs first.)
- Chaining is enabled, the CP register address field is non-zero, and the current DMA sequence finishes. (Again, TCB chain loading occurs.)

A DMA sequence ends when one of the following occurs:

- The count register decrements to zero (both C and EC for external port channels).
- Chaining is disabled and the channel's DEN bit transitions from high to low. If the DEN bit goes low and chaining is enabled, the channel enters chain insertion mode and the DMA sequence continues. (See "Chain Insertion" for details.)

Note that whenever the DEN bit goes high again, the DMA sequence continues from where it left off (for non-chained operations only).

To start a new DMA sequence after the current one is finished, your program must first clear the DEN enable bit, write new parameters to the II, IM, and C registers, and then set the DEN bit to re-enable DMA. (For chained DMA operations, however, this is not necessary; see "DMA Chaining.")

Warning: If a DMA operation completes and the count register is rewritten before the DMA enable bit is cleared, the DMA transfer will

restart at the new count.

6.4 EXTERNAL PORT DMA

Channels 6, 7, 8, and 9 are the external port DMA channels, which are available on the ADSP-21060 and ADSP-21062. On the ADSP-21061, only channels 6-7 are available. These DMA channels allow efficient data transfers between the ADSP-2106x's internal memory and external memory or devices.

6.4.1 External Port FIFO Buffers (EPBx)

DMA Channels 6, 7, 8, and 9 are associated with the external port FIFO data buffers, EPB0, EPB1, EPB2, and EPB3. Each buffer acts as a six-location FIFO. It has two ports, a read port and a write port. Each port can be connected to either the EPD (External Port Data) bus or to a local bus which in turn can connect to the IOD (I/O Data) bus, PM Data bus, or DM Data bus. (See Figure 6.2.) This structure allows data to be written to the FIFO on one port while it is being read from the other port—allowing DMA transfers at the full processor clock frequency.

The external port FIFO buffers can also be used for non-DMA, single-word data transfers, as described in the *Host Interface* chapter of this manual.

Caution: The ADSP-2106x core should not attempt to read or write an EPBx buffer when a DMA operation using that buffer is in progress; this will corrupt the DMA data.

Each external port buffer can be flushed (i.e. cleared) by writing a 1 to the FLSH bit in the corresponding DMACx control register. This should only be done when DMA is disabled for the channel. The FLSH bit is not latched internally and will always be read as a 0. Status can change in the following cycle. An external port buffer should not be enabled and flushed in the same cycle.

6.4.1.1 External Port DMA Data Packing

Each external port buffer contains data packing logic to allow 16-bit or 32-bit external bus words to be packed into 32-bit or 48-bit internal words. The packing logic is also fully reversible, depending on the setting of the TRAN bit in the DMACx control register, so that 32-bit or 48-bit internal data can be unpacked into 16-bit or 32-bit external word

widths. The packing mode is selected by the PMODE bits in the DMACx control register for each external port buffer.

PMODE	*Packing Mode*
00	No packing / unpacking
01	16-bit external bus to / from 32-bit internal packing
10	16-bit external bus to / from 48-bit internal packing
11	32-bit external bus to / from 48-bit internal packing

The external port buffer can pack data most significant word (MSW) first or least significant word (LSW) first. Setting the MSWF bit to 1 in the DMACx control register selects MSW-first. When MSWF is set, data is also unpacked MSW-first. The MSWF bit has no effect when PMODE=11 or PMODE=00.

The packing sequence for downloading ADSP-2106x instructions from a 32-bit bus (PMODE=11) takes 3 cycles for every 2 words, as shown below. (Note that for host processor transfers to or from the EPBx buffers, the HPM bits of the SYSCON register must be set to correspond to the external bus width specified by PMODE.) 32-bit data is transferred on data bus lines 47-16. If an odd number of instruction words are transferred, the packing buffer must be flushed by a dummy access to remove the unused word.

32-Bit to 48-Bit Word Packing (External Bus ↔ ADSP-2106x):

	Data Bus Pins 47-32	*Data Bus Pins 31-16*
1st DMA transfer	Word1 bits 47-32	Word1 bits 31-16
2nd DMA transfer	Word2 bits 15-0	Word1 bits 15-0
3rd DMA transfer	Word2 bits 47-32	Word2 bits 31-16

The MSWF bit of the DMACx control register is ignored for 32-to-48-bit packing.

The packing sequence for downloading ADSP-2106x instructions from a 16-bit bus is shown below. The MSWF bit determines whether the most significant 16-bit word or least significant 16-bit word is packed first.

16-Bit to 48-Bit Word Packing w/MSWF=1 (External Bus ↔ ADSP-2106x):

	Data Bus Pins 31-16
1st DMA transfer	Word1 bits 47-32
2nd DMA transfer	Word1 bits 31-16
3rd DMA transfer	Word1 bits 15-0

40-bit extended precision data may be transferred using the 48-bit packing mode. Refer to the *Memory* chapter of this manual for a description of memory allocation for different word widths.

6.4.1.2 Packing Status

Each external port DMA control register also contains a two-bit PS field which contains the number of short words currently packed in the EPBx buffer. During unpacking, the PS status behaves the same as for packing. All of the packing functions are available for any type of DMA transfer.

6.4.2 Internal & External Address Generation

DMA transfers between ADSP-2106x internal memory and external memory require the DMA controller to generate addresses for both. The external port DMA channels contain EI (External Index) and EM (External Modifier) registers to perform external address generation. The EI register provides the external port address for the current DMA cycle, and is updated with the modifier value in EM for the next external memory access.

In order to support the wide range of data packing operations provided for external DMA transfers, the EI and EM registers are able to generate addresses at a different rate than the internal address registers (II and IM). For this reason the internal and external address generators are decoupled from each other, and the EC (External Count) register is used as the external DMA word counter.

If, for example, a 16-bit DMA device is reading data from ADSP-2106x internal memory, then two external 16-bit transfers will occur for each 32-bit internal memory word and the EC (external) word count should be twice the value of the C (internal) word count.

6.4.3 External Port DMA Modes

The MASTER, HSHAKE, and EXTERN bits of each DMACx control register are used to select the DMA mode of operation. Each external port DMA channel can be set up to operate in one of five DMA modes. The master mode initiates transfers while the other modes act as “slaves” where an external device must initiate each transfer.

The MASTER, HSHAKE, and EXTERN bits configure the DMA mode

in the following manner:

M	*H*	*E*	*DMA Mode of Operation*[1]
0	0	0	**Slave Mode.** The DMA request is generated whenever the receive buffer is not empty or the transmit buffer is not full.[2]
0	0	1	*Reserved*
0	1	0	**Handshake Mode.** (For the ADSP-21060 and ADSP-21062, applies to EPB1, EPB2 buffers, channels 7, 8 only. For the ADSP-21061, applies to EPB0, EPB1 buffers, channels 6, 7 only.) The DMA request is generated when the $\overline{\text{DMAR}}$x line is asserted. The transfer occurs when $\overline{\text{DMAG}}$x is asserted.[1]
0	1	1	**External Handshake Mode.** (For the ADSP-21060 and ADSP-21062, applies to EPB1, EPB2 buffers, channels 7, 8 only. For the ADSP-21061, applies to EPB0, EPB1 buffers, channels 6, 7 only.) Identical to Handshake Mode, but with data transferred between external memory and an external device.
1	0	0	**Master Mode.** The DMA controller will attempt a transfer whenever the receive buffer is not empty or the transmit buffer is not full and the DMA counter is non-zero.[1] $\overline{\text{DMAR}}$1 should be kept high (inactive) if channel 7 is in master mode, and $\overline{\text{DMAR}}$2 should be kept high if channel 8 is in master mode on the ADSP-21060 or ADSP-21062. $\overline{\text{DMAR}}$2 should be kept high if channel 6 is in master mode on the ADSP-21061.
1	0	1	*Reserved*
1	1	0	**Paced Master Mode.** (For the ADSP-21060 and ADSP-21062, applies to EPB1, EPB2 buffers, channels 7, 8 only. For the ADSP-21061, applies to EPB0, EPB1 buffers, channels 6, 7 only.) In this mode the transfers are paced by the $\overline{\text{DMAR}}$x signal—the DMA request is generated when $\overline{\text{DMAR}}$x is asserted. $\overline{\text{DMAR}}$x requests operate in the same way as in handshake mode. The bus transfer occurs when $\overline{\text{RD}}$ or $\overline{\text{WR}}$ is asserted. The address is driven as in normal master mode. No external gates are required to OR the $\overline{\text{RD}}$-$\overline{\text{DMAG}}$x and $\overline{\text{WR}}$-$\overline{\text{DMAG}}$x pairs, thus allowing the buffer access to be zero-waitstate with no idle states. Waitstates and acknowledge (ACK) apply to Paced Master Mode transfers; see Section 5.4.4, “Wait States & Acknowledge” in Chapter 5, *Memory*.
1	1	1	*Reserved*

1. When an external port DMA channel is configured for output (i.e., TRAN=1), the EPBx buffer will start to fill as soon as that DMA channel is enabled. The EPBx buffer will start to fill up even if no DMAR assertions or slave mode DMA buffer reads have been made yet.
2. If data is to be read from the ADSP-2106x (i.e. TRAN=1), the EPBx buffer will be filled as soon as the DEN enable bit is set to 1.

6.4.3.1 Master Mode

When the DMACx bits are set such that MASTER=1, HANDSHAKE=0, and EXTERN=0, then the corresponding DMA channel operates in master mode. This means that the ADSP-2106x's DMA controller will generate internal DMA requests for that channel until the DMA sequence is completed. Master mode can be specified independently for each external port DMA channel.

Examples of DMA master mode operations include transfers between internal memory and external memory and transfers from internal memory to external devices. In both cases, the data is set up in memory so that the ADSP-2106x can run the complete sequence without interaction with other devices.

Note: The serial port and link port DMA channels do not have the MASTER control bit and do not operate in master mode.

6.4.3.2 Paced Master Mode

In paced master mode, the $\overline{\text{DMAR}}$x requests operate in the same way as in handshake mode but $\overline{\text{DMAG}}$x is not active. The ADSP-2106x responds to the requests only with the $\overline{\text{RD}}$ or $\overline{\text{WR}}$ strobe; this method allows the same buffer to be shared for both DMA and core processor I/O without external gating. Paced Master Mode accesses can be extended by the ACK pin, by waitstates programmed in the WAIT register, and by holding the DMARx pin low.

6.4.3.3 Slave Mode

When the DMACx bits MASTER, HANDSHAKE, and EXTERN are cleared, then the corresponding DMA channel is configured as a slave. This means that the particular DMA channel cannot independently initiate external memory transfers no matter what the programmed direction of data transfer. To initiate a DMA transfer to or from an ADSP-2106x configured in slave mode, an external device must read or write to the appropriate EPBx buffer.

If the DMA channel is in slave mode and the direction of data transfer is internal to external, the channel will automatically perform enough transfers from internal memory to keep the EPBx buffer full. (Remember that each EPBx buffer is a six-location FIFO.) On the other hand, if the direction of data transfer is external to internal, then the DMA channel will not initiate any internal DMA transfers until the

EPBx buffer has valid data.

The EI, EM, and EC registers are not used in slave mode DMA.

External to Internal
To explore the operation of slave mode DMA, consider the case where an external device wishes to transfer a block of data into the ADSP-2106x's internal memory. First the external device would write to the DMA channel parameter registers, II, IM, and C, and to the DMACx control register to initialize the channel. Then the device would begin writing data to the EPBx buffer.

When the EPBx buffer contains a valid data word (requiring one or more external memory cycles, depending on the packing mode selected), it signals the ADSP-2106x's DMA controller to request an internal DMA cycle. When granted, the internal DMA transfer occurs and the EPBx buffer FIFO is emptied. If the internal DMA transfer was held off for some reason, the external device could still write to the EPBx buffer again because of its six-deep FIFO. When the EPBx FIFO eventually becomes full, the external device will be held off with the ACK signal (for synchronous accesses) or with the REDY signal (for asynchronous, host-driven accesses).

This state continues until the internal DMA transfer is completed and space freed up in the EPBx buffer. For the buffer to operate in this fashion, the BHD (Buffer Hang Disable) bit must be cleared (to 0) in the SYSCON register.

Internal to External
Now consider the case where the transfer direction is from internal memory to the external port. Immediately after the DMA channel is enabled, it will request internal DMA transfers to fill up the EPBx FIFO buffer. Once the buffer is filled, the request will be deasserted. When the external device reads the buffer (one or more times depending on the packing mode), it becomes "partially empty" and the internal DMA request is asserted again. If the internal DMA transfers cannot fill the EPBx FIFO buffer at the same rate as the external device empties it (e.g. due to internal bus conflicts), the external device will be held off with the ACK signal (for synchronous accesses) or with the REDY signal (for asynchronous, host-driven accesses) until valid data can be transferred to the EPBx buffer. Again, for the buffer to operate in this fashion, the BHD (Buffer Hang Disable) bit must be cleared (to

0) in the SYSCON register.

Note that ACK (or REDY) is only deasserted during a write when the EPBx FIFO buffer is full. ACK (or REDY) remains asserted at the end of a completed block transfer if the EPBx buffer is not full. When reading, the buffer will be empty at the end of the block transfer and ACK (or REDY) will be deasserted if an additional read is attempted.

System-Level Considerations
Slave mode DMA is useful in systems with a host processor because it allows the host to access any ADSP-2106x internal memory location while limiting the address space the host must recognize—only the address space of the ADSP-2106x's IOP registers. Slave mode DMA is also useful for ADSP-2106x to ADSP-2106x DMA transfers.

Slave mode DMA has one drawback when interfacing to a slow host—the fact that the external bus is held up during the transfer (whether initiated by the ADSP-2106x or the host) and no other transactions can proceed. To overcome this, the handshake DMA mode may be used. In handshake mode, the host does not have to master the bus in order to make a DMA request, nor does the ADSP-2106x (in master mode) have to wait on the bus for the transfer to complete. Instead, the host asserts the $\overline{\text{DMAR}}$x pin. When the ADSP-2106x is ready to make the transfer, it can complete it in one bus cycle. The following section provides further details.

6.4.3.4 Handshake Mode

On the ADSP-21060 or ADSP-21062, DMA channels 7 and 8, for external port buffers EPB1 and EPB2, each have a set of external handshake controls. $\overline{\text{DMAR1}}$ and $\overline{\text{DMAG1}}$ are the request and grant signals for EPB1 and channel 7, and $\overline{\text{DMAR2}}$ and $\overline{\text{DMAG2}}$ are the request and grant signals for EPB2 and channel 8.

On the ADSP-21061, DMA channels 7 and 6, for external port buffers EPB1 and EPB0, each have a set of external handshake controls. $\overline{\text{DMAR1}}$ and $\overline{\text{DMAG1}}$ are the request and grant signals for EPB1 and channel 7, and $\overline{\text{DMAR2}}$ and $\overline{\text{DMAG2}}$ are the request and grant signals for EPB0 and channel 6.

These signals serve as a hardware handshake to facilitate DMA transfers between the ADSP-2106x and an external peripheral device that does not have bus mastership capability.

➠ If external port DMA channel is enabled but the handshake signals will not be used, the corresponding $\overline{\text{DMAR}}$x signal should be kept high.

Handshake mode DMA is enabled when the HSHAKE bit is set to 1 in the corresponding DMACx control register (DMAC7 or DMAC8 on an ADSP-21060 or ADSP-21062; DMAC7 or DMAC6 on an ADSP-21061). If the MASTER bit is 0, the ADSP-2106x handshakes by returning $\overline{\text{DMAG}}$x. If the MASTER mode bit is 1, the DMA operates in paced master mode.

DMA handshaking operates asynchronously at up to the full clock speed of the ADSP-2106x. The data source / destination can be selected to be either ADSP-2106x internal memory or external memory. It is important to load the EC external count register whenever external DMA transfers are being made.

The $\overline{\text{MS}}_{3\text{-}0}$ memory select lines are deasserted during DMA transfers between an external device and an ADSP-2106x because there is no external memory space being accessed. The $\overline{\text{MS}}_{3\text{-}0}$ lines are, however, asserted by the ADSP-2106x in external handshake mode because it is providing the address and strobes for transfers between an external DMA device and external memory.

Refer to Figure 6.8, *DMA Handshake Timing with Asynchronous Requests*. The DMA handshake uses the rising and falling edges of $\overline{\text{DMAR}}$x. The ADSP-2106x interprets a falling edge to mean “begin a DMA access” and interprets the rising edge to mean “complete the DMA access.”

To request an access of the EPBx buffer, the external device pulls $\overline{\text{DMAR}}$x low. The falling edge is detected by the ADSP-2106x and synchronized to the processor’s system clock. To be recognized in a particular cycle, the $\overline{\text{DMAR}}$x low transition must meet the setup time specified in the data sheet; otherwise it may take effect in the following cycle. When the ADSP-2106x recognizes the request, it begins to arbitrate for the external bus, if it is not already the bus master or if the buffer is not blocked (see discussion of *blocked condition* below). When the ADSP-2106x becomes the bus master, it drives $\overline{\text{DMAG}}$x low. The ADSP-2106x will keep $\overline{\text{DMAG}}$x asserted until it sees $\overline{\text{DMAR}}$x deasserted. This allows the external device to hold the ADSP-2106x until it is ready to proceed. Provided there are no pipelined requests, $\overline{\text{DMAG}}$x will deassert in the cycle after $\overline{\text{DMAR}}$x is deasserted. If the external device does not wish to extend the grant cycle, it can deassert $\overline{\text{DMAR}}$x immediately after asserting it, provided it meets the minimum pulse width timing requirements specified in the data sheet. In this case, $\overline{\text{DMAG}}$x will be a short pulse and the external bus will only be used for one cycle.

6 DMA

The DMA controller has a three-cycle pipeline, similar to the fetch–decode–execute pipeline of the core processor's program sequencer. The DMA request and arbitration occur in the fetch cycle. The DMA address generation and bus arbitration occur in the decode cycle and the data transfer occurs in the execute cycle. Use of the rising and falling edges of $\overline{\text{DMAR}}$x allows better utilization of the pipeline and, if desired, allows data transfers at up to the full clock rate of the ADSP-2106x.

The external device does not have to wait for the $\overline{\text{DMAG}}$x grant signal before making another request. The requests are stored in a working counter maintained internally by the ADSP-2106x. The counter holds a maximum of seven requests, so the external device can make up to seven requests before the first one has been serviced. *Note that more than seven requests without a grant will cause unpredictable results.* $\overline{\text{DMAG}}$x will be asserted in response to $\overline{\text{DMAR}}$x only for the number of transfers specified in the counter. If more requests than this are made, $\overline{\text{DMAG}}$x will remain deasserted. The flush bit (FLSH) in the DMACx control register should be used to clear any extra requests.

The external device must make sure that when the $\overline{\text{DMAG}}$x grant signal arrives, the data for each write request is immediately available (or that it can accept each word for a read). This can be accomplished by placing the data in an external FIFO. When DMAing data at the full clock speed of the ADSP-2106x, a two- or three-deep data pipeline may be needed to handle the latency between request and grant. Thus, the external device might issue three requests rapidly and condition the fourth request on whether a grant has been given in the meantime. Given this caveat, handshake DMA can occur at up to the full clock rate of the ADSP-2106x for both reads and writes. The stored requests are cleared when a 1 is written to the flush bit (FLSH) in the DMACx control register.

Since the external device can control the completion of a request, it does not need to have data available before making a request. If, however, the data is not available within two cycles and $\overline{\text{DMAR}}$x is kept low for this time, the ADSP-2106x and the external bus may be held inactive. The external bus is occupied for only one cycle for each DMA transfer if the request is deasserted before the grant has been asserted. Otherwise the external bus is held as long as $\overline{\text{DMAR}}$x is asserted.

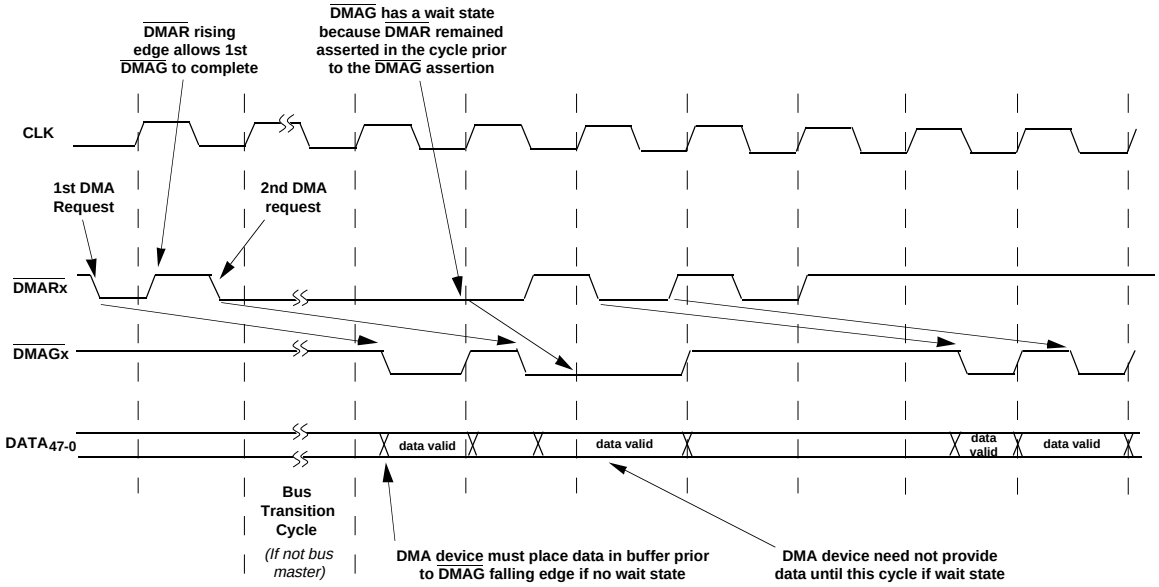


Figure 6.8 DMA Handshake Timing With Asynchronous Requests

Notes:

– DMA requests (DMARx) can be asynchronous. The DMARx falling edge initiates a DMA request on the ADSP-2106x. When writing, data must be provided by the device before DMAGx has been deasserted. If data is not available, the device may hold DMARx asserted (low) until the data is available. When this happens, the ADSP-2106x will attempt to service the request but will be delayed until the DMARx rising edge.

– There is a minimum delay of three cycles before DMAGx is asserted and the transfer from the external DMA device to the ADSP-2106x (or to external memory) occurs. However, the ADSP-2106x may not be able to issue a DMAGx grant for several cycles after a DMA request if a higher priority DMA operation is requesting service or if the bus is currently being used by another ADSP-2106x. Thus the external DMA device must not assume that the grant will arrive within two cycles unless higher priority DMA operations are disabled and the external bus is available for the transfer.

– DMA requests are pipelined in the ADSP-2106x. The ADSP-2106x keeps track of up to seven requests if it cannot service them immediately. It then services them on a prioritized basis. The request tracking allows DMA transfers at up to the full clock rate of the ADSP-2106x. The external DMA device is responsible for keeping track of requests, monitoring grants, and pipelining the data when operating at full speed.

The ADSP-2106x will not begin external bus arbitration in response to $\overline{\text{DMAR}}$x if the EPBx buffer is full during a write or empty during a read. This is a *blocked condition*. Bus arbitration will begin when the EPBx buffer is serviced by the DMA controller and the full or empty state changes (i.e. becomes unblocked).

If an external port DMA channel is disabled, its corresponding $\overline{\text{DMAR}}$x and $\overline{\text{DMAG}}$x pins are disabled and $\overline{\text{DMAR}}$x assertions are ignored for a maximum of 2 cycles after the instruction that enables DMA (setting DEN=1) in handshake mode. See Figure 6.9. $\overline{\text{DMAG}}$x will be held high by the ADSP-2106x.

The $\overline{\text{DMAR}}$x input must be kept high (not low or changing) during the instruction that enables DMA in handshake mode (see Figure 6.9.)

Several ADSP-2106xs in a multiprocessing cluster may share a $\overline{\text{DMAG}}$x signal. $\overline{\text{DMAG}}$x is only driven by the bus master and is tristated otherwise, or when $\overline{\text{HBG}}$ is asserted. This eliminates the need for external gating if more than one ADSP-2106x or the host needs to drive the DMA buffer. A pullup resistor may be needed on this line if the host is not connected to the pin and does not drive it when it acquires the bus. $\overline{\text{DMAG}}$x has the same timing and transitions as the $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes and responds to the $\overline{\text{SBTS}}$ and $\overline{\text{HBR}}$ signals in the same way as $\overline{\text{RD}}$ and $\overline{\text{WR}}$.

6.4.3.5 External Handshake Mode

External devices can also use the $\overline{\text{DMAR}}$x and $\overline{\text{DMAG}}$x handshake signals to control DMA transfers between an external device and external memory (instead of ADSP-2106x internal memory). In this mode, the ADSP-2106x operates as an independent DMA controller. This mode is configured by setting the EXTERN bit in the DMAC7 or DMAC8 control register; the corresponding HSHAKE bit must equal 1 and MASTER bit equal 0. External handshake mode transfers are similar to standard DMA transfers, but with some differences.

External handshake mode transfers require the ADSP-2106x's DMA controller to generate external memory access cycles. $\overline{\text{DMAR}}$x and $\overline{\text{DMAG}}$x retain the same functionality in this mode but instead of simply generating $\overline{\text{DMAG}}$x, the ADSP-2106x also outputs addresses, $\overline{\text{MS}}_{3\text{-}0}$ memory selects, and $\overline{\text{RD}}$ / $\overline{\text{WR}}$ strobes, and responds to ACK. ($\overline{\text{DMAG}}$x will be held low until the ACK line is released or any waitstates complete.) The external memory access behaves exactly as if

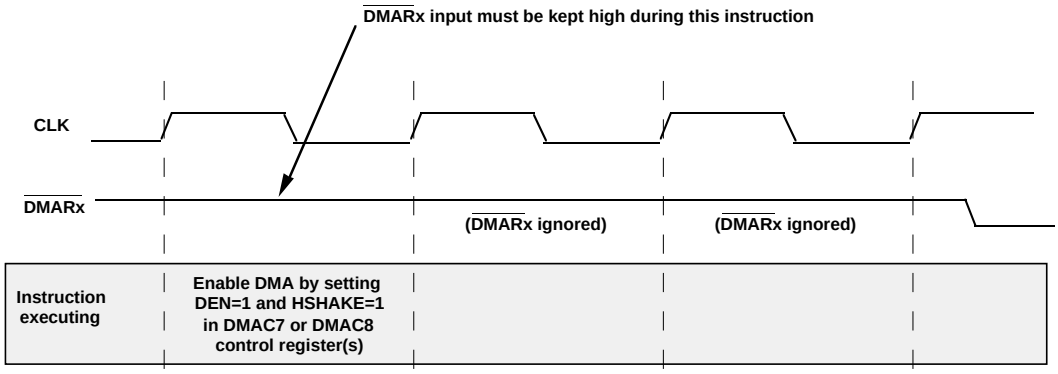


Figure 6.9 DMARx Delay After Enabling Handshake DMA

the ADSP-2106x core had requested it. The ADSP-2106x's EPBx buffers do not latch or drive any data, however, and no internal memory DMA transfers are performed. The EI, EM, and EC parameter registers (of the DMA channel) must be preloaded to generate the external memory addresses and word count.

Since internal DMA transfers do not occur in this mode, the PCI bit of the CP register cannot be used to disable the DMA interrupt—the IMASK register must be used. The DMA interrupt is always enabled and generated, unless it is masked out in IMASK. Also, because data does not pass through the ADSP-2106x in external handshake mode it cannot be packed or unpacked into different word widths.

6.4.4 System Configurations For ADSP-2106x Interprocessor DMA

Figure 6.10 shows the different ways you can set up external port DMA transfers between two ADSP-2106xs. The advantages and disadvantages of each configuration should be taken into account when designing a multiprocessor system.

6.4.5 DMA Hardware Interfacing

Figure 6.11 shows a typical DMA interface between two multiprocessing ADSP-2106xs and an external device. The ADSP-2106xs are configured for handshake mode operation. The external latches acts as a mailbox between the external device and the ADSP-2106xs. The latch allows a DMA transfer to take only one ADSP-2106x bus cycle, even with a slow external device. The latch is directly controlled by the DMARx and the DMAGx signals.

If the external device is writing data to the latch, the $\overline{\text{DMAG}}$x signal is used as the output enable signal for the latch. If the external device is reading from the latch, $\overline{\text{DMAG}}$x is used to clock the data on its rising edge. Figure 6.12 shows the timing relationships between $\overline{\text{DMAR}}$x, $\overline{\text{DMAG}}$x, and the data transfer. Refer to the *ADSP-2106x Data Sheet* for exact specifications.

6.5 DMA THROUGHPUT

This section discusses overall DMA throughput when several DMA channels are trying to access internal or external memory at the same time.

Internal Memory DMA

The DMA channels arbitrate for access to the ADSP-2106x's internal memory. The DMA controller determines, on a cycle-by-cycle basis, which channel is allowed access to the internal I/O bus and consequently which channel will read or write to internal memory. (The priority of the DMA channels is shown in Table 6.13 in the "DMA Channel Prioritization" section of this chapter.)

Each DMA transfer takes one clock cycle even when different DMA channels are being allowed access on sequential cycles; i.e. there is no overall throughput loss in switching between channels. Thus, four link port DMA channels, each transferring one byte per cycle, would have the same I/O transfer rate as one external port DMA channel transferring data to internal memory on every cycle. Any combination of link ports, serial ports, and external port transfers has the same maximum transfer rate.

External Memory DMA

When the DMA transfer is between ADSP-2106x internal memory and external memory, the external memory may have one or more wait states. External memory wait states, however, do not reduce the overall internal DMA transfer rate if other channels have data available to transfer. In other words, the ADSP-2106x's internal I/O data bus will not be held up by an incomplete external transfer.

ADSP-2106x Configuration *(Data Source)*	**ADSP-2106x Configuration** *(Data Destination)*	**Throughput**** *(cycles/transfer)*	**Advantages, Disadvantages**
Bus Master DMA Master Mode (MASTER=1) TRAN=1 EIx = address of EPBx buffer in destination EMx=0	Bus Slave DMA Slave Mode (MASTER=0) TRAN=0	**1**	*Advantage*: Destination automatically generates interrupt upon completion. *Disadvantage*: DMA must be programmed on both source and destination.
Bus Master DMA Master Mode (MASTER=1) TRAN=1 EIx = MMS address in destination * EMx=1	Bus Slave Direct Write	**1**	*Advantage*: No programming required for destination. *Disadvantage*: No interrupt generated upon completion—source must issue vector interrupt to destination.
Bus Slave DMA Slave Mode (MASTER=0) TRAN=1	Bus Master DMA Master Mode (MASTER=1) TRAN=0 EIx = address of EPBx buffer in source EMx=0	**2**	*Advantage*: Source automatically generates interrupt upon completion. *Disadvantages*: Slower throughput. DMA must be programmed on both source and destination.
Bus Slave Direct Read	Bus Master DMA Master Mode (MASTER=1) TRAN=0 EIx = MMS address in source * EMx=1	**4**	*Advantage*: No programming required for source. *Disadvantages*: Slowest throughput. No interrupt generated upon completion—destination must issue vector interrupt to source.

Figure 6.10 System Configurations For ADSP-2106x-To-ADSP-2106x DMA

* MMS=Multiprocessor Memory Space
** Assumes that no MMS wait state is configured in WAIT register. If the single MMS wait state is selected, add 1 to each throughput value.

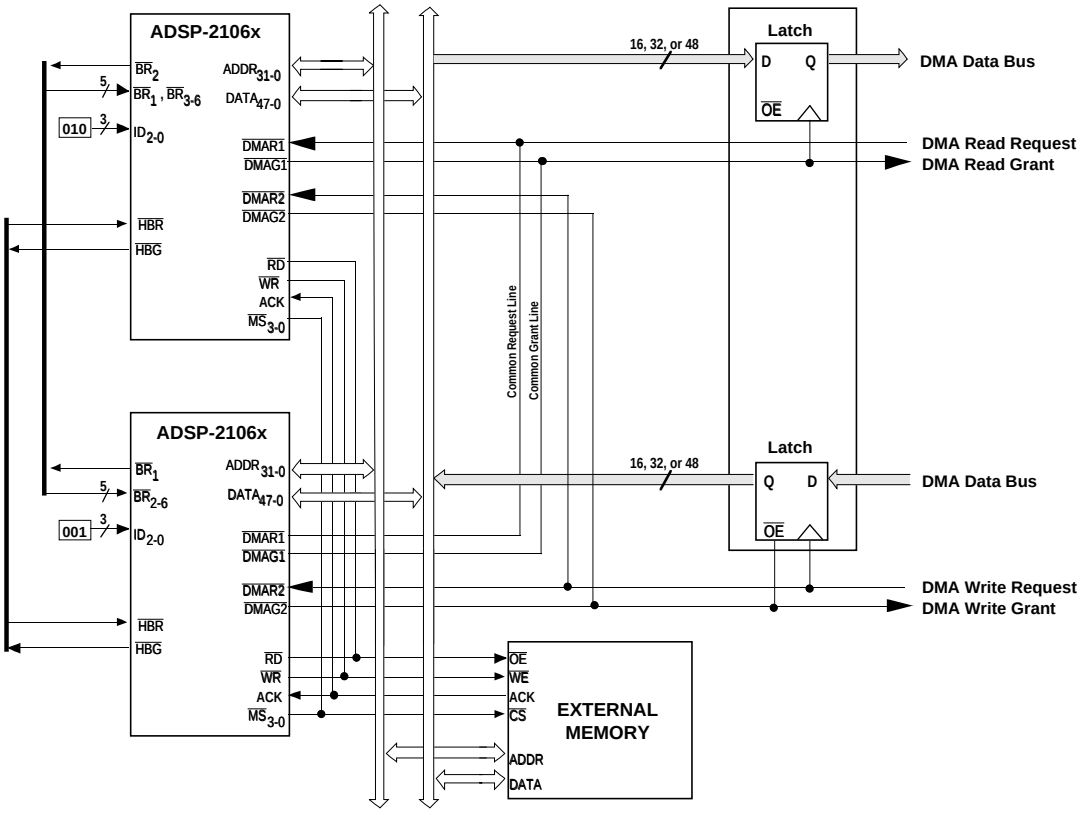


Figure 6.11 Example DMA Hardware Interface

Notes:

– Because $\overline{\text{DMAR}}$x and $\overline{\text{DMAG}}$x are tied together, only one of the ADSP-2106xs may have DMA enabled at a time.
– $\overline{\text{DMAG}}$x is only driven by the ADSP-2106x bus master.
– The DMA Write Grant signal can be the combination of $\overline{\text{RD}}$ and $\overline{\text{MS}}$x instead of $\overline{\text{DMAG2}}$ if paced master mode is used.
– The DMA Read Grant signal can be the combination of $\overline{\text{WR}}$ and $\overline{\text{MS}}$x instead of $\overline{\text{DMAG1}}$ if paced master mode is used.
– DMA transfers may be to either ADSP-2106x or to external memory (in external handshake mode).

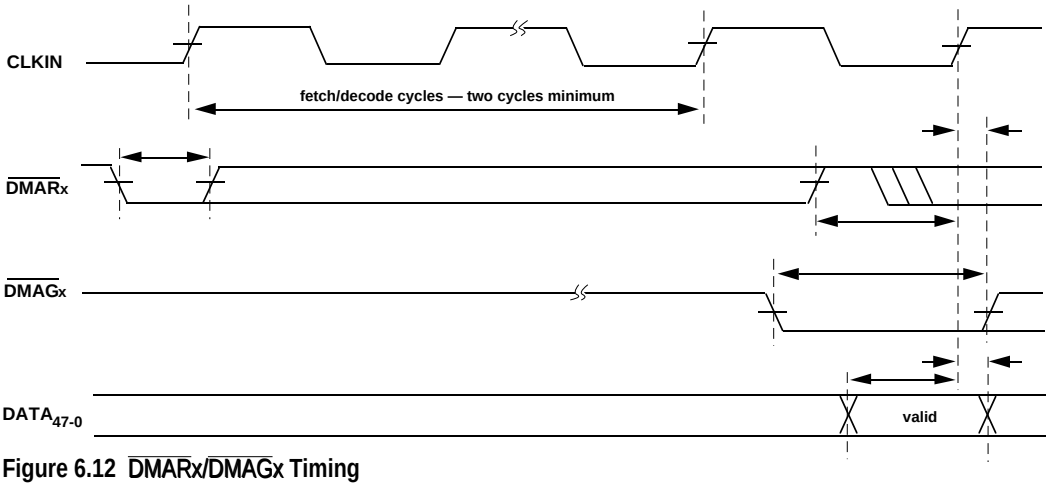


Figure 6.12 $\overline{\text{DMAR}}$x/$\overline{\text{DMAG}}$x Timing

Notes:
– $\overline{\text{DMAR}}$x setup times relate to the use of the signal in that cycle by the ADSP-2106x. DMA requests may be asserted asynchronously to CLKIN.
– $\overline{\text{DMAG}}$x drives $\text{DATA}_{47\text{-}0}$ if ADSP-2106x is receiving. $\overline{\text{DMAG}}$x latches $\text{DATA}_{47\text{-}0}$ if ADSP-2106x is transmitting.

When data is to be transferred from internal to external memory, the internal memory data is first placed in the external port's EPBx buffer by the DMA controller; the external memory access is then begun independently. (Likewise for external-to-internal DMA, the internal DMA request will not be made until the external memory data is in the EPBx buffer.) In both cases, the external DMA address generator—the EI and EM parameter registers—maintains the external address until the data transfer is completed. The internal and external address generators of a DMA channel are decoupled and operate independently.

When EXTERN mode DMA transfers occur between an external device and external memory, no internal resources of the ADSP-2106x are utilized and internal DMA throughput is not affected.

6.6 TWO-DIMENSIONAL DMA

This section describes the changes in functionality that occur when the ADSP-21060 or ADSP-21062 is placed in two-dimensional DMA mode. (Note that two-dimensional DMA mode does not apply to the ADSP-21061.) 2-D DMA mode is enabled by the L2DDMA bit in the LCOM control register and the D2DMA bit in the SRCTL0 and SRCTL1 registers. If a particular mode of operation is not explicitly mentioned below, then it is unchanged in 2-D mode.

6.6.1 2-D DMA Channel Organization

In 2-D mode, two-dimensional DMA array addressing can be performed for the link buffers and serial ports. DMA channels 0-5 support 2-D DMA. Link buffers 4 and 5 (DMA channels 6 and 7) do not support 2-D DMA. Table 6.16 shows the 2-D registers and their mapping into the DMA channel registers. For the purpose of discussion here, the 2-D array is addressed in *row-major* order.

2-D Function	*DMA Channel Register*
Index (address)	IIx
X Increment	IMx
X Count	Cx
Next Pointer	CPx
Y Increment	DBx
Y Count	GPx
X Initial Count	DAx *(not part of chain; loaded by Cx)*

Table 6.16 2-D Register Mapping

In Table 6.16 the DMA channel number, x, is distinguished from the standard numbering of serial ports and link ports as follows:

Transmit Link Buffer – Uses DMA Channel 5
Receive Link Buffer – Uses DMA Channel 4
Transmit SPORT – Uses DMA Channel 3 or 1
Receive SPORT – Uses DMA Channel 2 or 0

The Index register (II) is loaded with the first address in the data array and maintains the current address by subtracting the X increment after each transfer. The X Increment register (IM) contains the offset added to the current address to point to the next element in the X dimension (next column). The X Initial Count register (DA) contains the number of data

elements in the X dimension. This is used to reload the X count register when it decrements to zero. The X Count register (C) contains the number of data elements left in the current row. This initially has the same value as X initial count. It is decremented after each transfer.

The Y Increment register (DB) contains the offset added to the current address to point to the next element in the Y dimension (first location in next row). When the X count register reaches zero, this register is added to the current address on the following cycle and the Y count register is decremented. The value of DB should be the row distance minus the column distance since both the X and Y increments are done on a row change. Note that two DMA cycles are required for a row change.

The Y Count register (GP) initially contains the number of data elements in the Y dimension (number of rows). It is decremented each time the X count register reaches zero. When Y Count reaches zero, the DMA block transfer is done. The Next Pointer register (CP) points to the start of a buffer in internal memory containing the next set of DMA parameters.

The DMA controller and the link ports communicate via the same internal DMA request / grant handshake as is used by the other I / O ports. The receive link buffer uses channel 4 while the transmit link buffer uses channel 5. For more information, refer to the *Link Ports* chapter of this manual. The DMA controller and the serial port also communicate via the same internal DMA request / grant handshake as is used by the other I / O ports. For more information, refer to the *Serial Ports* chapter.

6.6.2 2-D DMA Operation

A two-dimensional DMA transfer occurs in the following manner:

First cycle:

- The current address stored in the II register is output and a DMA memory cycle is started.
- In the same cycle, the X Increment value stored in the IM register is added to the current address in the II register.
- The X Count in the C register is decremented.
- If the decremented X Count is zero, do the second cycle.

Second cycle:
- The X Count is restored into the C register from the DA register.
- The Y Increment value in the DB register is added to the current address in II.
- The Y Count in GP is decremented.
- If the Y Count is zero, the DMA sequence is ended and the channel becomes inactive until the Next Pointer is written again.

A key point about the 2-D DMA sequence (or any DMA sequence) is that the first DMA transfer begins before the address is modified. This means that DMA cannot be disabled by setting either the X Count or the Y Count to zero. To do one-dimensional DMA transfers in 2-D mode, the Y Count must be initialized to one.

When the X Count becomes zero but the Y Count is non-zero, the X Count must be reloaded with the original value. The C register functions as the working count register. The DA register holds the original count value. C is loaded from DA to restore the count. The DA register is written automatically whenever the C register is written.

Multiprocessing 7

7.1 OVERVIEW

The ADSP-2106x includes functionality and features that allow the design of multiprocessing DSP systems. These features include distributed on-chip arbitration for bus mastership and multiprocessor accesses of the internal memory and IOP registers of other ADSP-2106xs. The ADSP-2106x also has the ability to lock the bus in order to perform indivisible *read-modify-write* sequences for semaphores.

In a multiprocessor system with several ADSP-2106xs sharing the external bus, any of the processors can become the bus master. The bus master has control of the bus, which consists of the $DATA_{47-0}$, $ADDR_{31-0}$, and associated control lines. Figure 7.1 illustrates a basic multiprocessing system.

Table 7.1 shows which pins are connected between the SHARC processors.

$DATA_{47-0}$	$ADDR_{31-0}$	MS_{3-0}
RD	WR	ACK
PAGE	SBTS	SW
ADRCLK	BMS	BR_{6-1}
RESET	HBR†	HBG†
REDY†	CPA‡	CLKIN

† If Host Interface is used.

‡ If Core Priority Access function is used.

Table 7.1 Pin Connections For Cluster Multiprocessor System

The internal memory and IOP registers of the system's ADSP-2106xs is called *multiprocessor memory space*. Multiprocessor memory space is mapped into the unified address space of each ADSP-2106x.

Once an ADSP-2106x becomes the bus master, it can directly read and write the internal memory of any other (slave) ADSP-2106x. It can also read and write to any of the slave's IOP registers, including their external port FIFO data buffers. The master ADSP-2106x may write to a slave's IOP registers to set up DMA transfers, for example, or to send a vector interrupt.

7 Multiprocessing

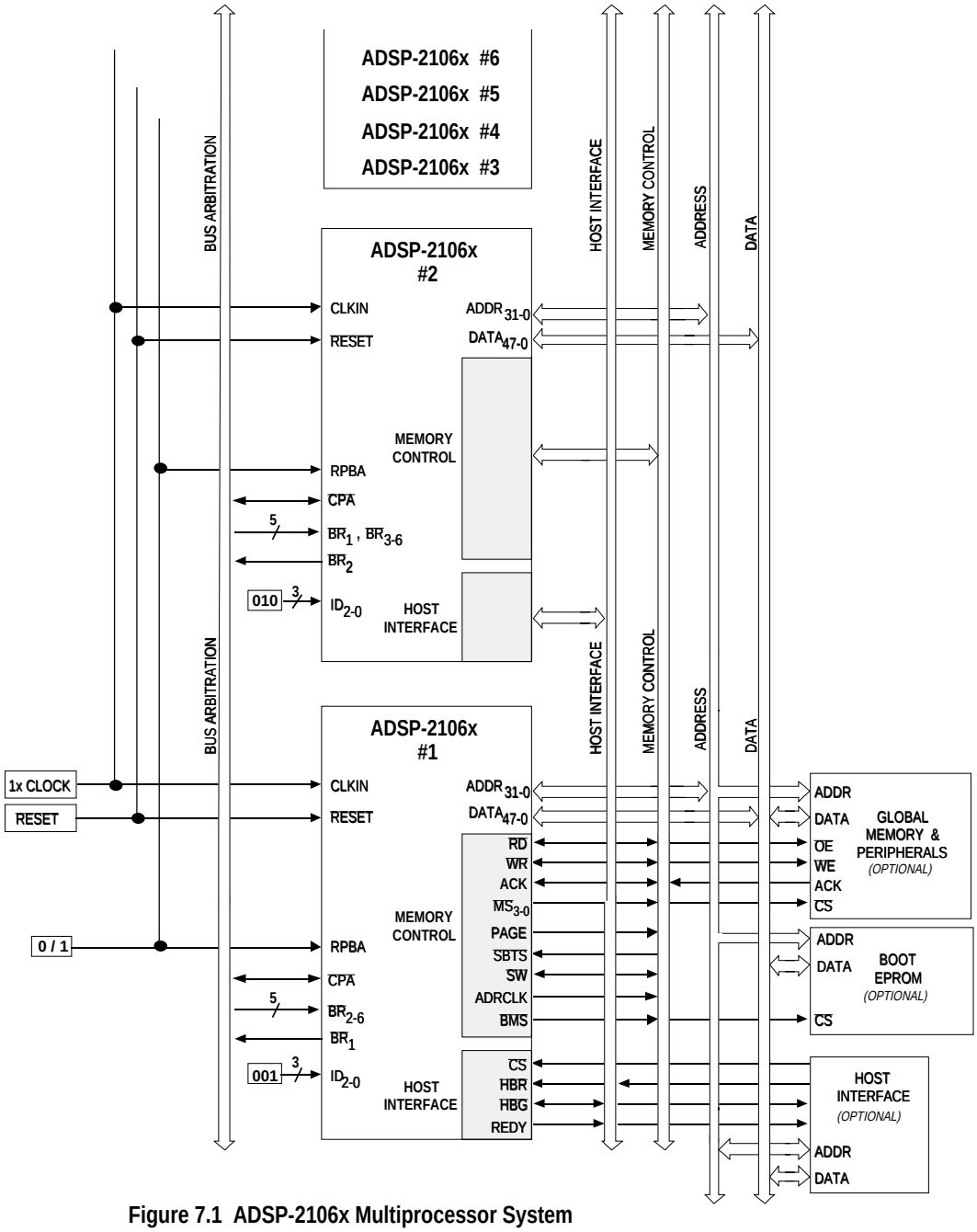


Figure 7.1 ADSP-2106x Multiprocessor System

Multiprocessing 7

The following terms are used throughout this chapter, and are defined below for reference:

external bus	$DATA_{47-0}$, $ADDR_{31-0}$, $\overline{RD}$, $\overline{WR}$, $\overline{MS}_{3-0}$, $\overline{BMS}$, ADRCLK, PAGE, $\overline{SW}$, ACK, and $\overline{SBTS}$ signals
multiprocessor system	a system with multiple ADSP-2106xs, with or without a host processor; the ADSP-2106xs are connected by the external bus and / or link ports
multiprocessor memory space	portion of the ADSP-2106x's memory map that includes the internal memory and IOP registers of each ADSP-2106x in a multiprocessing system; this address space is mapped into the unified address space of the ADSP-2106x
IOP register	one of the control, status, or data buffer registers of the ADSP-2106x's on-chip I / O processor
bus slave *or* slave mode	an ADSP-2106x can be a bus slave to another ADSP-2106x or to a host processor
direct reads & writes	a direct access of the ADSP-2106x's internal memory or IOP registers by another ADSP-2106x or by a host processor
single-word data transfers	reads and writes to the EPBx external port buffers, performed externally by the ADSP-2106x bus master or internally by the ADSP-2106x slave's core; these occur when DMA is disabled in the DMACx control register
bus transition cycle (BTC)	a cycle in which control of the external bus is passed from one ADSP-2106x to another
external port FIFO buffers	EPB0, EPB1, EPB2, and EPB3, the IOP registers used for external port DMA transfers and single-word data transfers (from other ADSP-2106xs or from a host processor); the EPBx buffers are 6-deep FIFOs
DMACx control registers	the DMA control registers for the EPBx external port buffers: DMAC6, DMAC7, DMAC8, and DMAC9, corresponding respectively to EPB0, EPB1, EPB2, and EPB3 (see the *DMA* chapter or *Control/Status Registers* appendix of this manual for a complete description of the DMACx control registers)

7 Multiprocessing

7.2 MULTIPROCESSING SYSTEM ARCHITECTURES

Multiprocessor systems typically use one of two schemes to communicate between processor nodes. One scheme uses dedicated point-to-point communication channels. In the other, nodes communicate through a single shared global memory via a parallel bus.

The ADSP-2106x SHARC supports the implementation of point-to-point communication through its six link ports. It also supports an enhanced version of shared parallel bus communication called *cluster multiprocessing*. Cluster multiprocessing features of the ADSP-2106x are described in this chapter, while point-to-point connections are described in the *Link Ports* chapter of this manual.

Multiprocessing systems must overcome two problems: interprocessor communication overhead and data bandwidth bottlenecks. The ADSP-2106x SHARC architecture addresses these concerns in several ways, as illustrated in the following discussion of three basic multiprocessing topologies.

7.2.1 Data Flow Multiprocessing

Data flow multiprocessing is best suited for applications requiring high computational bandwidth but only limited flexibility. Programmers partition their algorithm sequentially across multiple processors and pass data linearly down an “assembly line” of processors, as shown in Figure 7.2.

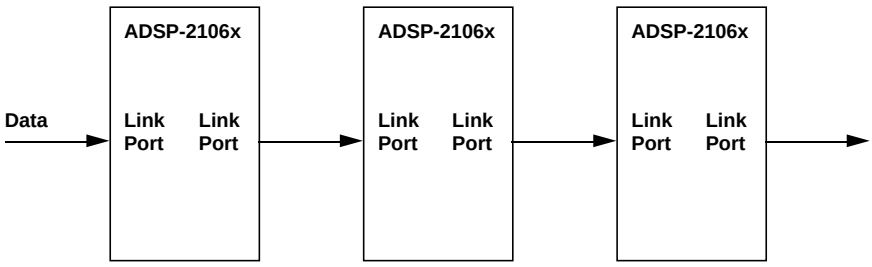


Figure 7.2 Data Flow Multiprocessing

The ADSP-2106x SHARC is ideally suited for data flow multiprocessing applications because it eliminates the need for interprocessor data FIFOs and external memory. The internal memory of the SHARC is usually large enough to contain both code and data for most applications using this topology. All a data flow system requires are a number of SHARC processors and point-to-point signals connecting them. This yields a substantial savings in complexity, board space, and system cost.

7.2.2 Cluster Multiprocessing

Cluster multiprocessing is best suited for applications where a fair amount of flexibility is required. This is especially true when a system must be able to support a variety of different tasks, some of which may be running concurrently. The cluster multiprocessing configuration is shown in Figure 7.3. SHARC processors also have an on-chip host interface that allows a cluster to be easily interfaced to a host processor or even to another cluster.

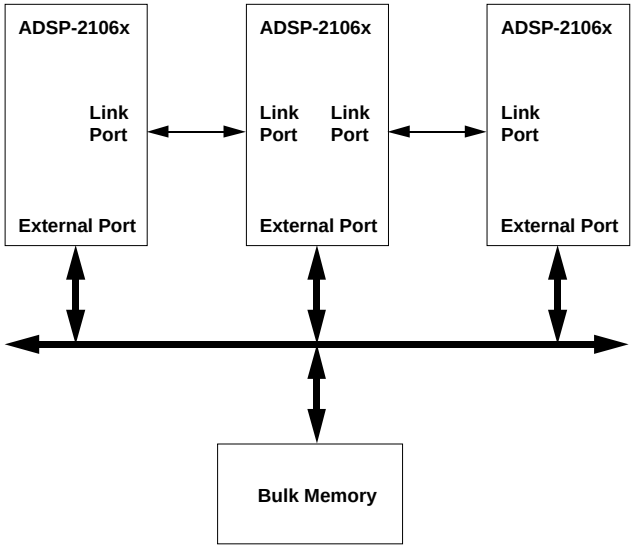


Figure 7.3 Cluster Multiprocessing

7 Multiprocessing

Cluster multiprocessing systems include multiple SHARC processors connected by a parallel bus that allows interprocessor access of on-chip memory as well as access to shared global memory. In a typical cluster of SHARCs, up to six processors and a host can arbitrate for the bus. The on-chip bus arbitration logic allows these processors to share the common bus. The SHARC's other on-chip features help eliminate the need for any extra hardware in the cluster multiprocessor configuration. External memory, both local and global, can frequently be eliminated in this type of system.

Both fixed and rotating priority schemes are supported as well as bus locking, timed release, and core processor access preemption of background DMA transfers. The on-chip arbitration logic allows transitions in bus mastership to take up to only one cycle of overhead. Bus requests are generated implicitly whenever a processor accesses an external address. Because each processor monitors all bus requests and applies the same priority logic to the requests, each can independently determine who will be the next bus master. With complete bus sharing features built into the processor, designers are spared the time and risk of developing their own shared-bus logic and timing.

Once a SHARC gains mastership of the bus, it can access not only external memory but also the internal memory and IOP registers of all other processors. A processor can directly transfer data to another processor or set up a DMA channel to transfer the data. Each of the processors are mapped into a common memory map—to identify the address space of each processor within the unified memory map of the system cluster, each processor has a unique ID. The SHARC's IOP registers, internal memory, and external memory are all part of the unified address space. This shared on-chip memory eliminates the need to use external memory for message passing between processors and simplifies software communications. Processors can write directly into each other's memory, saving an extra transfer step. Local memory may also no longer be needed due to the SHARC's large amount of on-chip SRAM. For larger applications, however, blocks of data and code can be stored in shared bulk memory and transparently swapped in and out of a processor's internal memory.

Communication between processors is also facilitated by the ability of a processor to broadcast a write to all processors simultaneously. This can be used to implement reflective semaphores, where a processor polls its own internal copy of the semaphore and only uses the external bus for a broadcast write to all other processors when it wants to change it. This reduces communications traffic on the external bus.

The cluster configuration allows the SHARCs to have a very fast node-to-node data transfer rate. It also allows for a simple, efficient, software communication model. For example, all of the required setup operations for a DMA transfer can be accomplished by a single SHARC on one side of the transfer. The other processor is not interrupted until the DMA transfer is complete.

The SHARC's internal memory is designed to facilitate the I/O needs of multiprocessor systems. The on-chip dual-ported RAM allows full-speed interprocessor transfers concurrent with dual accesses by the processor's computational core. No cycles are stolen from the core, and the processor's full 40 MIPS, 120 MFLOPS performance is maintained.

7.2.2.1 Link Port Data Transfers In A Cluster

A bottleneck exists within the cluster because only two processors can communicate over the shared bus during each cycle—other processors are held off until the bus is released. Since the SHARC can also perform point-to-point link port transfers within a cluster, this bottleneck is easily eliminated. Data links between processors can be dynamically set up and initiated over the common bus. All six link ports can operate simultaneously on each processor.

A disadvantage of the link ports is that individual transfers occur at only 40 Mbyte/sec (for a 40 MHz system clock), a lower rate than that of the shared parallel bus. Since the link ports' 4-bit data path is smaller than the processor's native word size, the transfer of each word requires multiple clock cycles. Link ports may also require more software overhead and complexity because they must be set up on both sides of the transfers before they can occur.

7.2.3 SIMD Multiprocessing

For certain classes of applications such as radar imaging, a SIMD array may be the most efficient topology to coordinate a large number of processors in a single system. The SIMD array of Figure 7.3 consists of multiple SHARCs connected in a 2-D or 3-D mesh. The data link ports provide nearest neighbor communications as well as through-routing of data. A single master SHARC provides the instruction stream that the array executes. Data flow in and out the array can be managed through multiple serial port streams.

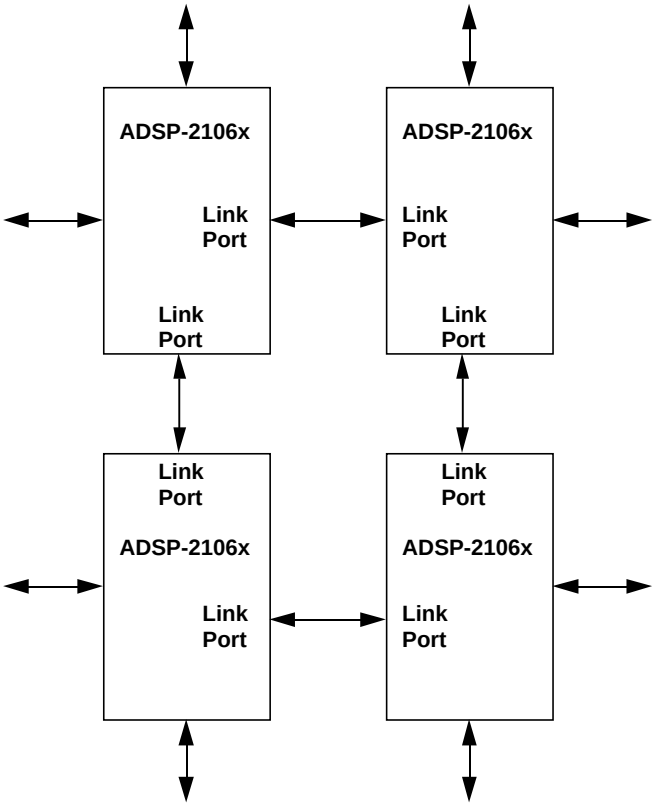


Figure 7.4 Two-Dimensional SIMD Mesh Multiprocessing

7.3 MULTIPROCESSOR BUS ARBITRATION

Multiple ADSP-2106xs can share the external bus with no additional arbitration circuitry. Arbitration logic is included on-chip to allow the connection of up to six ADSP-2106xs and a host processor.

Bus arbitration is accomplished with the use of the $\overline{BR1}$–$\overline{BR6}$, $\overline{HBR}$, and $\overline{HBG}$ signals. $\overline{BR1}$–$\overline{BR6}$ arbitrate between multiple ADSP-2106xs, and $\overline{HBR}$–$\overline{HBG}$ pass control of the bus from the ADSP-2106x bus master to the host (and back). The priority scheme for bus arbitration is determined by the setting of the RPBA pin. Table 7.2 defines the ADSP-2106x pins used in multiprocessing systems.

Signal	*Type*	*Definition*
$\overline{BR}_{6-1}$	I/O/S	**Multiprocessing Bus Requests**. Used by multiprocessing ADSP-2106xs to arbitrate for bus mastership. An ADSP-2106x only drives its own $\overline{BR}$x line (corresponding to the value of its ID_{2-0} inputs) and monitors all others. In a multiprocessor system with less than six ADSP-2106xs, the unused $\overline{BR}$x pins should be tied high; the processor's own $\overline{BR}$x line must not be tied high or low because it is an output.
ID_{2-0}	I	**Multiprocessing ID**. Determines which multiprocessing bus request ($\overline{BR1}$ – $\overline{BR6}$) is used by ADSP-2106x. ID=001 corresponds to $\overline{BR1}$, ID=010 corresponds to $\overline{BR2}$, etc. ID=000 in single-processor systems. These lines are a system configuration selection which should be hardwired or only changed at reset.
RPBA	I/S	**Rotating Priority Bus Arbitration Select**. When RPBA is high, rotating priority for multiprocessor bus arbitration is selected. When RPBA is low, fixed priority is selected. This signal is a system configuration selection which must be set to the same value on every ADSP-2106x. If the value of RPBA is changed during system operation, it must be changed in the same CLKIN cycle on every ADSP-2106x.
$\overline{CPA}$ (o/d)	I/O	**Core Priority Access**. Asserting its $\overline{CPA}$ pin allows the core processor of an ADSP-2106x bus slave to interrupt background DMA transfers and gain access to the external bus. $\overline{CPA}$ is an open drain output that is connected to all ADSP-2106xs in the system. The $\overline{CPA}$ pin has an internal 5 Kohm pullup resistor. If core access priority is not required in a system, the $\overline{CPA}$ pin should be left unconnected.

Table 7.2 ADSP-2106x Multiprocessor Signals

I=Input S=Synchronous (o/d)=Open Drain
O=Output A=Asynchronous (a/d)=Active Drive

7 Multiprocessing

The ID_{2-0} pins provide a unique identity for each ADSP-2106x in a multiprocessing system. The first ADSP-2106x should be assigned ID=001, the second should be assigned ID=010, and so on. One of the ADSP-2106xs must be assigned ID=001 in order for the bus synchronization scheme to function properly. This processor also holds the external bus control lines stable during reset.

When the ID_{2-0} inputs of an ADSP-2106x are equal to 001, 010, 011, 100, 101, or 110, it configures itself for a multiprocessor system and maps its internal memory and IOP registers into the multiprocessor memory space. ID=000 configures the ADSP-2106x for a single-processor system. ID=111 is reserved and should not be used.

An ADSP-2106x in a multiprocessor system can determine which processor is the current bus master, by reading the CRBM(2:0) bits of the SYSTAT register. These bits give the value of the ID_{2-0} inputs of the current bus master.

Conditional instructions can be written that depend upon whether the ADSP-2106x is the current bus master in a multiprocessor system. The assembly language mnemonic for this condition code is BM, and its complement is NBM (not bus master). The BM condition indicates whether the ADSP-2106x is the current bus master. For a complete list of condition codes, see “Conditional Instruction Execution” in the *Program Sequencer* chapter of this manual. To enable the use of the bus master condition, bits 17 and 18 of the MODE1 register must both be zeros; otherwise the condition is always evaluated as false.

7.3.1 Bus Arbitration Protocol

The $\overline{BR1}$–$\overline{BR6}$ pins are connected between each ADSP-2106x in a multiprocessing system, with the number of $\overline{BR}$x lines used equal to the number of ADSP-2106xs in the system. Each processor drives the $\overline{BR}$x pin corresponding to its ID_{2-0} inputs and monitors all others. If less than six ADSP-2106xs are used in the system, the unused $\overline{BR}$x pins should be tied high.

When one of the slave ADSP-2106xs needs to become bus master, it automatically initiates the bus arbitration process by asserting its $\overline{BR}$x line at the beginning of the cycle. Later in the same cycle it samples the value of the other $\overline{BR}$x lines.

The cycle in which mastership of the bus is passed from one ADSP-2106x to another is called a *bus transition cycle*. A bus transition cycle occurs when the current bus master's $\overline{BR}$x pin is deasserted and one of the slave's $\overline{BR}$x pins is asserted. The bus master can therefore retain bus mastership by keeping its $\overline{BR}$x pin asserted. Also, the bus master does not always lose bus mastership when it deasserts its $\overline{BR}$x line—another $\overline{BR}$x line must be asserted by one of the slaves at the same time. In this case, when no other $\overline{BR}$x is asserted, the master will not lose any bus cycles.

By observing all of the $\overline{BR}$x lines, each ADSP-2106x can detect when a bus transition cycle occurs and which processor has become the new bus master. A bus transition cycle is the only time that bus mastership is transferred.

Once it is determined that a bus transition cycle will occur, the priority of each $\overline{BR}$x line asserted within that cycle is evaluated (on every ADSP-2106x). (Refer to the following section for a description of bus arbitration priority.) The ADSP-2106x with the highest priority request becomes the bus master on the following cycle, and all of the ADSP-2106xs update their internal record of who the current bus master is. This information can be read from the current bus master field, CRBM, of the SYSTAT register.

Figure 7.5 shows typical timing for bus arbitration.

The actual transfer of bus mastership is accomplished by the current bus master tristating the external bus—$DATA_{47-0}$, $ADDR_{31-0}$, ADRCLK, $\overline{RD}$, $\overline{WR}$, $\overline{MS}_{3-0}$, PAGE, $\overline{HBG}$, $\overline{DMAG1}$, and $\overline{DMAG2}$—at the end of the bus transition cycle and the new bus master driving these signals at the beginning of the next cycle. $\overline{MS}_{3-0}$ is driven high (inactive) before tristating occurs. See Figure 7.6.

Execution of external accesses will be delayed during transfers of bus mastership. When one of the slave ADSP-2106xs needs to perform an external read or write, for example, it automatically initiates the bus arbitration process by asserting its $\overline{BR}$x line; the read or write is delayed until the processor receives bus mastership. If the read or write was generated by the ADSP-2106x's processor core (not the DMA controller), program execution stops until the instruction is completed.

7 Multiprocessing

The following steps summarize the actions a slave takes to acquire bus mastership and perform an external read or write over the bus (see Figure 7.6):

1. The slave determines that it is executing an instruction which requires an off-chip access. It asserts its $\overline{BR}$x line at the beginning of the cycle. Extra cycles are generated by the core processor (or DMA controller) until the slave acquires bus mastership.

2. To acquire bus mastership, the slave waits for a bus transition cycle in which the current bus master deasserts its $\overline{BR}$x line. If the slave has the highest priority request in the bus transition cycle, it becomes the bus master in the next cycle. If not, it continues waiting.

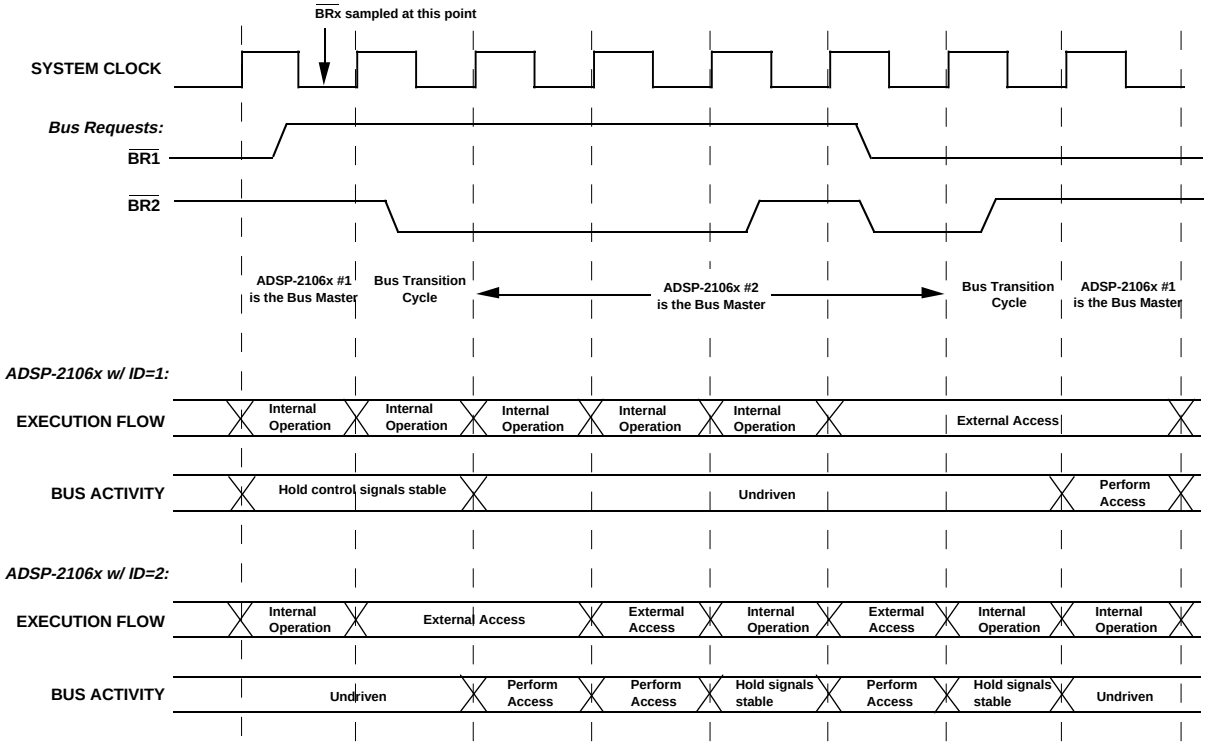


Figure 7.5 Bus Arbitration Timing

3. At the end of the bus transition cycle the current bus master releases the bus and the new bus master starts driving.

Whenever the bus master stops using the bus its $\overline{BR}$x line is deasserted, allowing other ADSP-2106xs to arbitrate for mastership if they need it. If no other ADSP-2106xs are asserting their $\overline{BR}$x line when the master deasserts his, the master retains control of the bus and continues to drive the memory control signals until: 1) it needs to use the bus again, or 2) another ADSP-2106x asserts its $\overline{BR}$x line.

Note: An ADSP-2106x will try to become bus master whenever it executes a conditional external access, even if the access is aborted.

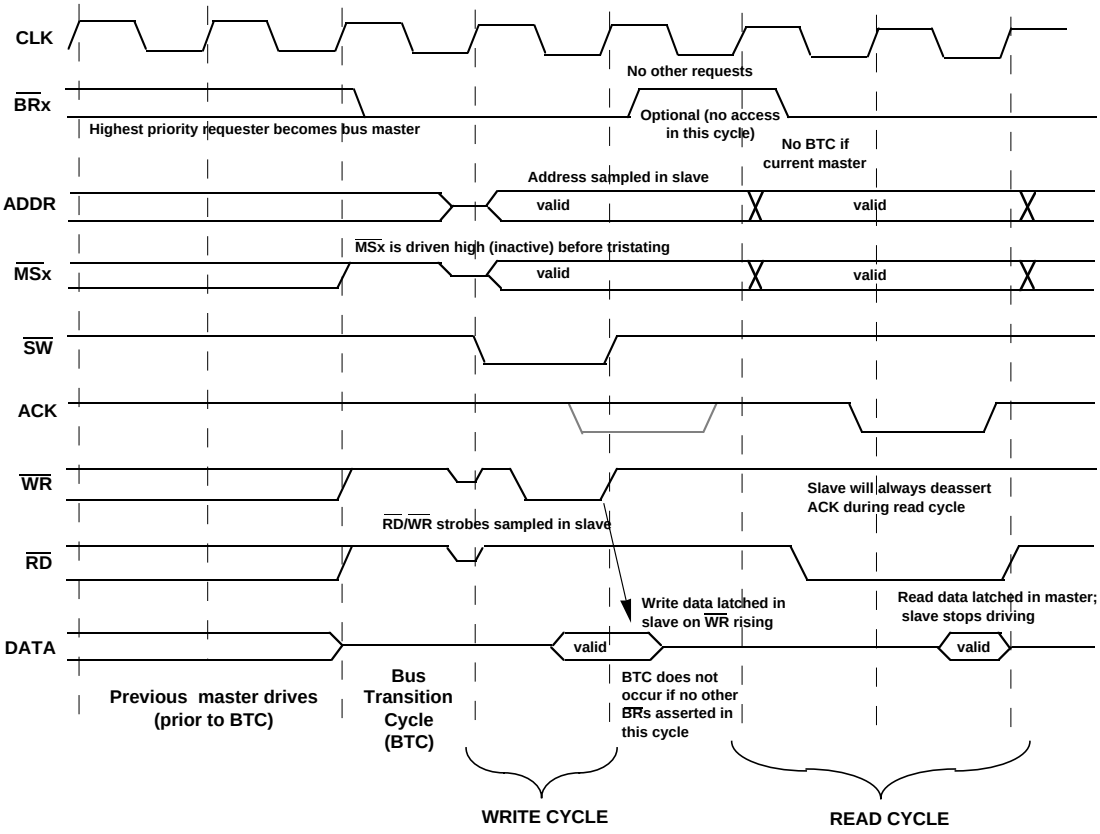


Figure 7.6 Bus Request & Read/Write Timing

While a slave waits to be a master for a DMA transfer, it asserts $\overline{BR}$x. If that slave's core accesses the DA group regsiters, the $\overline{BR}$x will be deasserted during that access..

7.3.2 Bus Arbitration Priority (RPBA)

Two different priority schemes are available to resolve competing bus requests, fixed and rotating. The RPBA pin selects which scheme is used: when RPBA is high, rotating priority bus arbitration is selected and when RPBA is low, fixed priority is selected.

The RPBA pin must be set to the same value on each ADSP-2106x in a multiprocessing system. If the value of RPBA is changed during system operation, it must be changed synchronously to CLKIN and must meet a setup time (specified in the data sheet) to allow all ADSP-2106xs to recognize the change in the same cycle. The priority scheme will change in that (same) cycle.

In the fixed priority scheme, the ADSP-2106x with the lowest ID number among the competing bus requests becomes the bus master. If, for example, the processor with ID=010 and the processor with ID=100 request the bus simultaneously, the processor with ID=010 becomes bus master in the following cycle. Each ADSP-2106x knows the ID of the other processor(s) requesting the bus because their ID corresponds to the $\overline{BR}$x line being used.

The rotating priority scheme gives roughly equal priority to each ADSP-2106x. When rotating priority is selected, the priority of each processor is reassigned after every transfer of bus mastership. Highest priority is rotated from processor to processor as if they were arranged in a circle—the ADSP-2106x located next to (one place down from) the current bus master is the one that receives highest priority. Table 7.3 shows an example of how rotating priority changes on a cycle-by-cycle basis.

	Hardwired Processor IDs:						
Cycle#	*ID1*	*ID2*	*ID3*	*ID4*	*ID5*	*ID6*	
1	M	1	2BR	3	4	5	***Initial priority assignments***
2	4	5BR	M-BR	1	2	3	
3	4	5BR	M	1	2	3	
4	5BR	M	1	2	3	4BR	
5	1BR	2	3	4	5	M	***Final priority assignments***

1-5 = assigned priority
M = bus mastership (in that cycle)
BR = requesting bus mastership with $\overline{BR}$x

Table 7.3 Rotating Priority Arbitration Example

7.3.3 Bus Mastership Timeout

In either bus arbitration priority scheme, it may be desirable to limit how long a bus master can own the bus. This is accomplished by forcing the bus master to deassert its $\overline{BR}$x line after a specified number of cycles, giving the other processors a chance to acquire bus mastership.

To setup a bus master timeout, your program must load the BMAX register with the maximum number of cycles (minus 2) for which the ADSP-2106x can retain bus mastership:

BMAX = (maximum # of bus mastership cycles) – 2

The minumum value that BMAX can be set to is 2, which lets the processor retain bus mastership for 4 cycles. Setting BMAX=1 is not allowed. To disable the bus master timeout function, set BMAX=0.

Each time an ADSP-2106x acquires bus mastership, its BCNT register is loaded with the value in BMAX. BCNT is then decremented in every cycle that the master performs a read or write over the bus *and* any other (slave) ADSP-2106xs are requesting the bus. Any time the bus master deasserts its $\overline{BR}$x line, BCNT is reloaded from BMAX.

When BCNT decrements to zero, the bus master first completes its off-chip read / write and then deasserts its own $\overline{BR}$x (any new off-chip accesses are delayed)—this allows transfer of bus mastership. If none of the slave processors has its $\overline{BR}$x request asserted when the master's BCNT reaches zero, the master's $\overline{BR}$x is *not* deasserted and BCNT is reloaded from BMAX. If the ACK signal is holding off an access when BCNT reaches zero, bus mastership will not be relinquished until the access can complete.

If BCNT reaches zero while bus lock is active, the bus master will not deasserts its $\overline{BR}$x line until bus lock is removed. (Bus lock is enabled by the BUSLK bit in the MODE2 register; see "Bus Lock & Semaphores" later in this chapter.)

If $\overline{HBR}$ is being serviced, BCNT stops decrementing and continues only after $\overline{HBR}$ is deasserted.

7.3.4 Core Priority Access

The Core Priority Access signal, $\overline{\text{CPA}}$, allows external bus accesses by the core processor of a slave ADSP-2106x to take priority over ongoing DMA transfers. Normally when external port DMA transfers are in progress, the core processors of the slave ADSP-2106xs cannot use the external bus until the DMA transfer is finished. By asserting its $\overline{\text{CPA}}$ pin, the core processor of a slave ADSP-2106x can acquire the bus without waiting for the DMA operation to complete.

If the $\overline{\text{CPA}}$ signal is not used in a multiprocessor system, the ADSP-2106x bus master will not give up the bus to another ADSP-2106x until either: 1) a cycle in which it does not perform an external bus access, or 2) a bus timeout. If a slave ADSP-2106x needs to send a high priority message or perform an important data transfer, it normally must wait until any DMA operation completes. Using the $\overline{\text{CPA}}$ signal allows the slave to perform its higher priority bus access with less delay.

A slave ADSP-2106x core with a pending access to the bus will assert the $\overline{\text{CPA}}$ pin at the same time as its bus request pin ($\overline{\text{BR}}$x). $\overline{\text{CPA}}$ is an open-drain output which is connected to all ADSP-2106xs in the system. Each ADSP-2106x has a 5 Kohm pull-up resistor on this pin, allowing it to be shared among all ADSP-2106xs in the system. Any ADSP-2106x may assert $\overline{\text{CPA}}$ low, and the internal resistors (or an additional external resistor for faster pull-up) will pull it high when it is released. Multiple ADSP-2106xs can be asserting this line at the same time.

When $\overline{\text{CPA}}$ is asserted, the current ADSP-2106x bus master will deassert its $\overline{\text{BR}}$x and give up the bus, provided its core does not have an external access pending. In addition, any ADSP-2106x cores that do not have an external access pending will remove their $\overline{\text{BR}}$x pins in the next cycle. Note that the current bus master never asserts $\overline{\text{CPA}}$ because it already has control of the bus.

In the cycle after $\overline{\text{CPA}}$ has been asserted, only the ADSP-2106x cores with a pending external access have their bus requests asserted. Bus arbitration now proceeds as usual, with the highest priority device becoming the master (when the previous bus master releases its $\overline{\text{BR}}$x line). The ADSP-2106x that becomes bus master releases $\overline{\text{CPA}}$ immediately on becoming master. If there are no other ADSP-2106x cores that need to perform an external access, the $\overline{\text{CPA}}$ signal will be pulled high by the pull-up resistors and arbitration will proceed normally. ADSP-2106xs that have deasserted their $\overline{\text{BR}}$x in response to $\overline{\text{CPA}}$ will reassert it in the cycle after $\overline{\text{CPA}}$ is sampled as high.

If there are lower priority ADSP-2106xs that still require access to the bus, they will continue to assert their $\overline{\text{CPA}}$. In this case, when the bus master core has completed its bus access (or accesses), it will release its $\overline{\text{BR}}$x even if it has DMA accesses pending. When this happens, the bus is acquired by the ADSP-2106x with the highest priority $\overline{\text{BR}}$x.

The overall sequence of events that takes place when an ADSP-2106x uses its $\overline{\text{CPA}}$ signal is as follows (Figure 7.7 shows timing for this sequence):

1. The core processor of an ADSP-2106x bus slave asserts its $\overline{\text{CPA}}$ pin (with the same timing as $\overline{\text{BR}}$x) when it has a pending external bus access.

2. When the common $\overline{\text{CPA}}$ line is asserted, the ADSP-2106x cores with no pending external accesses will deassert their $\overline{\text{BR}}$x in the next cycle. If the current ADSP-2106x bus master core does not have a pending access, it will proceed to give up the bus (i.e. deassert its $\overline{\text{BR}}$x) after completing its current access.

3. In the cycle after $\overline{\text{CPA}}$ is asserted, arbitration occurs normally among the ADSP-2106xs that have their $\overline{\text{BR}}$x asserted. The highest priority device becomes bus master when the previous bus master releases its $\overline{\text{BR}}$x.

4. The new bus master releases $\overline{\text{CPA}}$ after acquiring the bus.

All ADSP-2106xs arbitrate as usual while $\overline{\text{CPA}}$ is asserted, but only assert their $\overline{\text{BR}}$x if their core processor needs to make an access over the external bus.

7 Multiprocessing

When $\overline{\text{CPA}}$ is released, all ADSP-2106xs resume normal $\overline{\text{BR}}$x assertion one cycle after $\overline{\text{CPA}}$ is sampled as high.

After releasing its $\overline{\text{CPA}}$, the bus master will ignore the $\overline{\text{CPA}}$ pin for two cycles. This reduces the possibility of the bus master unnecessarily losing bus mastership while the $\overline{\text{CPA}}$ signal is pulled high by the common pullup resistors.

Because $\overline{\text{CPA}}$ is pulled up by a resistor and may have a time constant greater than one cycle, it may not be recognized as high by all ADSP-2106xs in the same cycle. In some very rare cases this may result in a lower priority ADSP-2106x temporarily gaining control of the bus, but the correct prioritization will be implemented eventually.

If core access priority is not required in a system, the $\overline{\text{CPA}}$ pin should be left unconnected and the ADSP-2106xs will arbitrate normally.

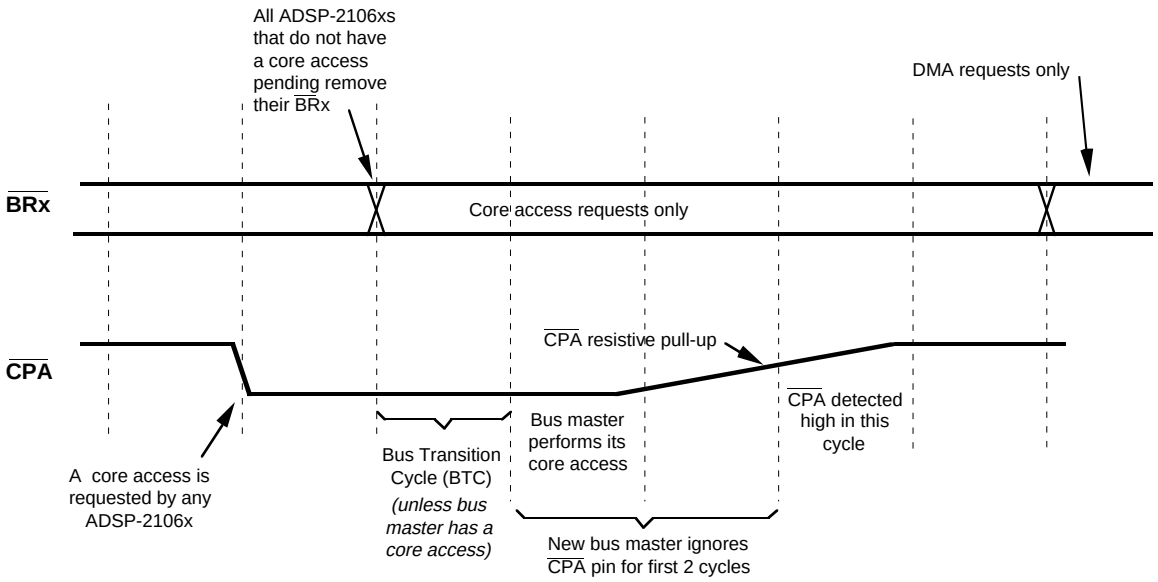


Figure 7.7 Core Priority Access Timing

7.3.5 Bus Synchronization After Reset

When a multiprocessing system is reset by the $\overline{\text{RESET}}$ pin, the bus arbitration logic on each processor must synchronize to insure that only one ADSP-2106x will drive the external bus. One ADSP-2106x must become the bus master, and all other processors must recognize which one it is before actively arbitrating for the bus. The bus synchronization scheme also allows the system to safely bring individual ADSP-2106xs into and out of reset.

A soft reset (SRST) does *not* resynchronize ADSP-21062 silicon revision 1.x parts or ADSP-21060/62 silicon revision 2.x (or later) parts.

Note that a soft reset (SRST) does resynchronize ADSP-21062 silicon revision 0.x parts and ADSP-21060 silicon revision 1.x parts.

One of the ADSP-2106xs in the system must be assigned ID=001 in order for the bus synchronization scheme to function properly. This processor also holds the external bus control lines stable during reset. Bus arbitration synchronization is disabled if the ADSP-2106x is in a single-processor system (ID=000).

To synchronize their bus arbitration logic and define the bus master after a system reset, the multiple ADSP-2106xs obey the following rules:

- All ADSP-2106xs except the one with ID=001 will deassert their $\overline{\text{BR}}$x line during reset. They will keep their $\overline{\text{BR}}$x deasserted for at least two cycles after reset and until their bus arbitration logic is synchronized.

- After reset, an ADSP-2106x will consider itself synchronized when it sees a cycle in which only one $\overline{\text{BR}}$x line is asserted. The ADSP-2106x will identify the bus master by recognizing which $\overline{\text{BR}}$x is asserted, and will update its internal record of who the current master is (in the current bus master field, CRBM, of the SYSTAT register).

- The ADSP-2106x with ID=001 will assert its $\overline{\text{BR}}$x ($\overline{\text{BR1}}$) during reset and for at least two cycles after reset. If no other $\overline{\text{BR}}$x lines are asserted during these cycles, the ADSP-2106x with ID=001 will drive the memory control signals to prevent them from glitching. (Although it is asserting its $\overline{\text{BR}}$x and driving the memory control signals during these cycles, this processor does not perform reads or writes over the bus.)

  If the ADSP-2106x with ID=001 is synchronized by the end of the two cycles following reset, it becomes the bus master. If it is not synchronized at this time, it will deassert its $\overline{\text{BR}}$x ($\overline{\text{BR1}}$) and wait until it is.

When an ADSP-2106x has synchronized itself, it sets the BSYN bit in the SYSTAT register.

If one ADSP-2106x comes out of reset after the others have synchronized and started program execution, that processor may not be able to synchronize immediately (e.g. if it sees more than one $\overline{BR}$x line asserted). If the unsynchronized processor tries to execute an instruction with an off-chip read or write, it cannot assert its $\overline{BR}$x line to request the bus and execution is delayed until it can synchronize and correctly arbitrate for the bus.

Synchronization cannot occur while $\overline{HBG}$ is asserted, because bus arbitration is suspended while the bus is controlled by a host. If $\overline{HBR}$ is asserted immediately after reset and no bus arbitration has taken place, the ADSP-2106x with ID=001 is considered to be the last bus master.

As mentioned above, the ADSP-2106x with ID=001 maintains correct logic levels on the $\overline{RD}$, $\overline{WR}$, $\overline{MS}_{3-0}$, PAGE, and $\overline{HBG}$ signals during reset.

Because the “001” processor can be accidently reset by an erroneous write to the soft reset bit (SRST) of the SYSCON register, it behaves in the following manner during reset:

- While it is in reset, the ADSP-2106x with ID=001 attempts to gain control of the bus by asserting $\overline{BR1}$.
- While it is in reset, the ADSP-2106x with ID=001 will drive the $\overline{RD}$, $\overline{WR}$, $\overline{MS}_{3-0}$, $\overline{DMAG1}$, $\overline{DMAG2}$, PAGE, and $\overline{HBG}$ signals only if it determines that it has control of the bus. For the processor to decide it has control of the bus, two conditions must be true: 1) $\overline{BR1}$ was asserted and no other $\overline{BR}$x lines were asserted in the previous cycle, and 2) $\overline{HBG}$ was deasserted in the previous cycle.

The ADSP-2106x with ID=001 will continue to drive the $\overline{RD}$, $\overline{WR}$, $\overline{MS}_{3-0}$, $\overline{DMAG1}$, $\overline{DMAG2}$, PAGE, and $\overline{HBG}$ signals for two cycles after reset, as long as neither $\overline{HBG}$ nor any other $\overline{BR}$x lines are asserted. At the end of the second cycle it assumes bus mastership (if it is synchronized), and normal bus arbitration begins in the following cycle. If it is not synchronized, it deasserts $\overline{BR1}$, stops driving the memory control signals, and does not arbitrate for the bus until it becomes synchronized.

Although the bus synchronization scheme allows individual processors to be reset, the ADSP-2106x with ID=001 may fail to drive the memory control signals if it is in reset while any other processors are asserting their $\overline{BR}$x line.

If the ADSP-2106x with ID=001 has asserted $\overline{HBG}$ while it is in reset, it will be synchronized when $\overline{RESET}$ is deasserted. This allows the host to start using the bus while the ADSP-2106xs are still in reset.

If a host processor attempts to reset the ADSP-2106x bus master (which is driving the $\overline{HBG}$ output), the host will immediately lose control of the bus.

During reset, the ACK line is pulled high internally by the ADSP-2106x bus master (with a 2 kΩ equivalent resistor).

7.4 SLAVE DIRECT READS & WRITES

The ADSP-2106x bus master can directly access the internal memory and IOP registers of a slave ADSP-2106x by simply reading or writing to the appropriate address in multiprocessor memory space—this is called a *direct read* or *direct write*. Each ADSP-2106x bus slave monitors addresses driven on the external bus and responds to any that fall within its region of multiprocessor memory space.

> *These accesses are invisible to the slave ADSP-2106x's core processor because they are performed through the external port and via the on-chip I/O bus—not the DM bus or PM bus. (See Figure 8.1 in the Host Interface chapter.)*

This is an important distinction, because it allows the slave's core processor to continue program execution uninterrupted.

The ADSP-2106x bus master can directly read and write the slave's IOP registers to send a vector interrupt, for example, or to set up a DMA transfer.

To read or write 48-bit instruction words, the IWT (Instruction Word Transfer) bit of the SYSCON register must be set to 1. To read or write 32-bit data words, the IWT bit must be cleared to 0. When this bit is set, it overrides the IMDWx (Internal Memory Data Width) bit of each memory block.

For heavily loaded buses, or when external data buffers are used, a single wait state can be added to all multiprocessor memory accesses. This option is selected by the MMSWS bit of the WAIT register.

7.4.1 Direct Writes

When a direct write to a slave ADSP-2106x occurs, the address and data are latched on-chip by the I/O processor. The I/O processor buffers the address and data in a special set of FIFO buffers. If additional direct writes are attempted when the FIFO buffer is full, the slave ADSP-2106x deasserts its ACK line until the buffer is no longer full. Up to six direct writes can be performed before another is delayed. (The direct write buffer itself may be held off for up to four cycles if all of the serial port DMA channels are active or for up to nine cycles per chain if DMA chaining is occurring.)

7.4.1.1 Direct Write Latency

When data is written to an ADSP-2106x bus slave, the data and address are latched at the I/O pins in a four-level FIFO buffer; this buffer is called the *slave write FIFO* (see again Figure 8.1 in the *Host Interface* chapter). In the following cycle, the slave write FIFO attempts to complete the write internally. This allows the master ADSP-2106x to perform writes at the full clock rate. The slave write FIFO cannot be explicitly read by the slave ADSP-2106x's core processor, nor can its status be determined.

Writes to the IOP registers will usually occur in the following one or two cycles, or when any current DMA transfer is completed. The write will take more than two cycles only if a direct write in the previous cycle was held off by a full buffer.

If the buffer is full when a write is attempted, the slave ADSP-2106x will deassert its ACK line until the buffer is not full. The buffer will usually flush out within one cycle, thus creating a write latency, unless higher priority on-chip DMA transfers are occurring.

Slave reads will be held off when there is data in the write FIFO—this prevents false data reads and out-of-sequence operations.

The DWPD (Direct Write Pending) bit of the SYSTAT register indicates when a direct write to internal memory is pending in the I/O processor's direct write FIFO or data is pending in the slave write FIFO (at the external port I/O pins). Direct writes and IOP register accesses may be completed in different sequences. If, for example, the ADSP-2106x master performs a direct memory write and then writes to an IOP register on a slave, the IOP register write may complete before the direct write.

7.4.2 Direct Reads

When a direct read of a slave ADSP-2106x occurs, the address is latched on-chip by the I/O processor and ACK is deasserted. When the corresponding location in memory is read internally, the ADSP-2106x drives the data off-chip and asserts its ACK line. Direct reads cannot be pipelined like direct writes—they only occur one at a time.

Note that while direct writes have a maximum pipelined throughput of one per cycle, direct reads have a maximum throughput of one per every two cycles (for synchronous IOP register reads) or one per every four cycles (for synchronous internal memory reads). See Table 11.5, "Data Delays & Throughputs", in Chapter 11. Because of this low bandwidth, direct reads are not the most efficient method of transferring data out of a slave ADSP-2106x—setting up a master mode DMA channel on the slave to perform writes is more efficient, although it requires additional programming. The advantage of direct reads is that no programming of the DMA controller is required.

7.4.3 Broadcast Writes

Broadcast writes allow simultaneous transmission of data to all of the ADSP-2106xs in a multiprocessing system. The master ADSP-2106x can perform broadcast writes to the same memory location or IOP register on all of the slaves. During broadcast writes, the master also writes to itself unless the broadcast is a DMA write. Broadcast writes can be used to implement *reflective semaphores* in a multiprocessing system (see "Bus Lock & Semaphores" later in this chapter). Broadcast writes can also be used to simultaneously download code or data to multiple processors.

The highest region of multiprocessor memory space, addresses 0x0038 0000 to 0x003F FFFF, is used for broadcast writes. When a write address falls within this region, each ADSP-2106x slave responds by accepting the access; the master ADSP-2106x also accepts its own broadcast write. A read cycle generated in the broadcast write region reads the corresponding location in that processor's internal memory and does not assert the processor's $\overline{BR}_x$.

Figure 7.8 shows the timing for a typical broadcast write for MMSWS=0. In this example, the first broadcast write completes without a wait state. In the second broadcast write, one or more of the slaves have 3 wait states and are deasserting ACK for 3 cycles. Note that ACK is sampled by the master on odd cycles (*wrt* $\overline{\text{WR}}$ asserted). If the multiprocessor memory space wait state is enabled, the master does not sample or pre-charge ACK for the first two cycles.

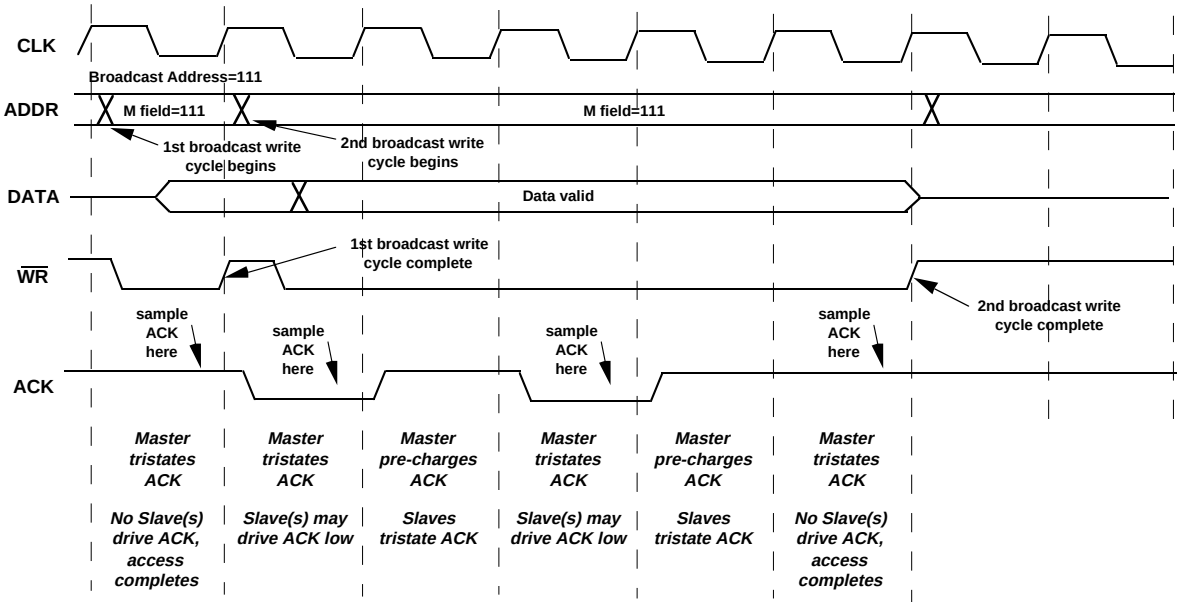


Figure 7.8 Broadcast Write Timing Example

Because the master ADSP-2106x must wait for a broadcast write to complete on *all* of the slaves, the acknowledge signal is handled differently to prevent drive conflicts on the ACK line. A wired-OR acknowledge signal is implemented to respond to broadcast writes. This signal operates as follows:

1. In the first cycle of the broadcast write (and in all succeeding odd cycles), a slave ADSP-2106x will pull ACK low if it is not ready to accept the data. If it is ready, it will not drive the ACK line.

   If the master ADSP-2106x sees that ACK is high, indicating that all slaves are accepting the broadcast write, it completes the write.

2. During all succeeding even cycles in which the broadcast write is not finished, the slave ADSP-2106xs will not drive ACK. Instead, the master ADSP-2106x drives (i.e. pre-charges) ACK high and must continue the write. *(Go to Step 1.)*

In most cases the ACK signal will be high and the ADSP-2106x slaves will be ready to accept data at the start of the broadcast write—the write completes in one cycle. If the ACK signal is low, however, or one of the slaves is not ready to accept the data, the broadcast write will take a minimum of three cycles.

When the wait state for multiprocessor memory space is enabled (with the MMSWS bit of the WAIT register), none of the ADSP-2106xs will drive ACK in the first cycle, the master pre-charges ACK in the second cycle, and the slaves may drive ACK in the third cycle. In this case the broadcast write will again take a minimum of three cycles to complete.

(**Note:** The ADSP-2106x bus master enables a keeper latch on the ACK line to prevent the signal from drifting. This eliminates any power consumption caused by the signal drifting to the switching point and improves the robustness of broadcast writes. Multiprocessor systems that use broadcast writes should keep the ACK signal line as free of noise as possible.)

7.4.4 Shadow Write FIFO

Because the ADSP-2106x's internal memory must operate at high speeds, writes to the memory do not go directly into the memory array, but rather to a two-deep FIFO called the *shadow write FIFO*.

When an internal memory write cycle occurs, data in the FIFO from the previous write is loaded into memory and the new data goes into the FIFO. This operation is normally transparent, since any reads of the last two locations written are intercepted and routed to the FIFO. There is only one case in which you need to be aware of the shadow write FIFO: mixing 48-bit and 32-bit word accesses to the same locations in memory.

The shadow FIFO cannot differentiate between the mapping of 48-bit words and mapping of 32-bit words. (See Figures 5.8 and 5.9 in the *Memory* chapter.) Thus if you write a 48-bit word to memory and then try to read the data with a 32-bit word access, the shadow FIFO will not intercept the read and incorrect data will be returned.

If 48-bit accesses and 32-bit accesses to the *same* locations absolutely must be mixed in this way, you must flush out the shadow FIFO with two dummy writes before attempting to read the data.

7.5 DATA TRANSFERS THROUGH THE EPBx BUFFERS

In addition to direct reads and writes, the ADSP-2106x bus master can transfer data to and from the slave ADSP-2106xs through the external port FIFO buffers, EPB0, EPB1, EPB2, and EPB3. Each of these buffers, which are part of the IOP register set, is a six-location FIFO. Both single-word transfers and DMA transfers can be performed through the EPBx buffers. DMA transfers are handled internally by the ADSP-2106x's DMA controller, but single-word transfers must be handled by the ADSP-2106x core.

Each EPBx buffer has a read port and a write port, both of which can connect internally to either the EPD (External Port Data) bus or to a local bus which in turn can connect to the IOD (I/O Data) bus, PM Data bus, or DM Data bus. This is shown in Figure 8.1 in the *Host Interface* chapter. Note that direct reads and writes bypass the EPBx buffers and go directly to internal memory.

7.5.1 Single-Word Transfers

When the ADSP-2106x master writes a single data word to a slave's EPBx buffers, the slave core's program must read the data. Conversely, when the slave's core writes a single piece of data to one of its EPBx buffers, the master must perform an external bus read cycle to obtain it. Because the EPBx buffers are six-deep FIFOs (in both directions), the master and the slave's core are allowed extra time to read the data—efficient, continuous, single-word transfers can thus be performed in real-time, with low latency and without using DMA.

If the ADSP-2106x master attempts a read from an empty EPBx buffer on a slave, the access will be held off with the ACK signal until the buffer receives data from the slave's core. If the slave's core attempts to write to a full EPBx buffer, the access is also delayed and the core will hang until the buffer is externally read by the master. To prevent this from happening, the BHD (Buffer Hang Disable) bit should be set to 1 in the SYSCON register. The *full or empty* status of a particular EPBx buffer can be determined by reading the appropriate DMACx control/status register.

Similarly, if the ADSP-2106x master attempts a write to a full EPBx buffer on a slave, the access will be held off with ACK until the buffer is read by the slave's core. If the slave's core attempts to read from an empty buffer, the access is also held off and the core will hang until the buffer is externally written from the bus master. The BHD bit can also be used to prevent a hang condition in this case.

Each EPBx buffer can be flushed (i.e. cleared) by writing a 1 to the FLSH bit in the corresponding DMACx control register. This bit is not latched internally and will always be read as a 0. Status can change in the following cycle. An EPBx buffer should not be enabled and flushed in the same cycle.

Note: To perform single-word, non-DMA transfers through the EPBx buffers, the DMA enable bit (DEN) must be cleared in the appropriate DMACx control register.

7.5.1.1 Interrupts For Single-Word Transfers

The interrupts for the four external port DMA channels can be used to control single-word data transfers between the ADSP-2106x bus master and a slave. To do this, the DMACx control register must have the following bit settings: DEN=0 and INTIO=1. This disables DMA (DEN=0) and enables interrupt-driven I/O (INTIO=1). See the *DMA* chapter or *Control/Status Registers* appendix of this manual for a complete description of the DMACx control registers.

In this case the interrupt is generated whenever data becomes available in the read port of the EPBx buffer, or whenever the write port does not have new data to transmit. The EPBx buffer can then be read or written, either internally by the ADSP-2106x slave's core or externally by the master. Generating interrupts in this fashion is useful for implementing interrupt-driven I/O controlled by the ADSP-2106x core processor.

This interrupt may be masked out (i.e. disabled) in the IMASK register. If the interrupt is later enabled in IMASK, the corresponding IRPTL latch bit must be cleared to clear any interrupt request that may have occurred.

7.5.2 DMA Transfers

The ADSP-2106x bus master can also set up DMA transfers to and from a slave ADSP-2106x. The master can write to the slave's DMA control and parameter registers to set up an external port DMA operation. This is the most efficient way to transfer blocks of data between two ADSP-2106xs.

- **DMA Transfers to Internal Memory.** The ADSP-2106x master can set up external port DMA channels to transfer data to and from a slave's internal memory.

- **DMA Transfers to External Memory.** The ADSP-2106x master can set up an external port DMA channel to transfer data directly to external memory using the DMA request and grant lines ($\overline{\text{DMAR}}$x, $\overline{\text{DMAG}}$x).

Refer to the *DMA* chapter of this manual for details on setting up DMA operations. Figure 6.9 in the "System Configurations For ADSP-2106x Interprocessor DMA" section of the *DMA* chapter shows the different ways you can set up external port DMA transfers between two ADSP-2106xs, as well as the advantages and disadvantages of each.

7.5.2.1 DMA Transfers To Internal Memory

The ADSP-2106x master can set up external port DMA channels to transfer blocks of data to and from a slave's internal memory. To set up the DMA transfer, the master must initialize the slave's control and parameter registers for that channel. Once the DMA channel is set up, the master may simply read from (or write to) the corresponding EPBx buffer on the slave, or it may set up its own DMA controller to perform the transfers. If the slave's buffer is empty (or full), the access is extended until data is available (or stored). This method allows fast and efficient data transfers.

Packing and unpacking of DMA data words is selected by the PMODE bits in the external port DMA control registers (DMAC6, DMAC7, DMAC8, and DMAC9). Either 16-to-32, 16-to-48, or 32-to-48 bit packing / unpacking can be selected.

The ADSP-2106x master may also use the $\overline{\text{DMAR}}$x / $\overline{\text{DMAG}}$x handshake signals to control a DMA transfer, but not when a host processor has gained control of the bus.

7.5.2.2 DMA Transfers To External Memory

The ADSP-2106x's DMA controller can also be used to transfer data directly to external memory. The *external handshake* mode for external port DMA channel 7 or 8 will provide the $\overline{\text{DMAR}}$x / $\overline{\text{DMAG}}$x handshaking for this type of transfer. Again, this is not possible when a host processor has gained control of the bus.

7.6 BUS LOCK & SEMAPHORES

Semaphores can be used in multiprocessor systems to allow the processors to share resources such as memory or I/O. A semaphore is a flag that can be read and written by any of the processors sharing the resource. The value of the semaphore tells the processor when it can access the resource. Semaphores are also useful for synchronizing the tasks being performed by different processors in a multiprocessing system.

With the use of its bus lock feature, the ADSP-2106x has the ability to read and modify a semaphore in a single indivisible operation—a key requirement of multiprocessing systems.

Because both external memory and each ADSP-2106x's internal memory (and IOP registers) are accessible by every other ADSP-2106x, semaphores can be located almost anywhere. Read-modify-write operations on semaphores can be performed if all of the ADSP-2106xs obey two simple rules:

1. An ADSP-2106x must not write to a semaphore unless it is the bus master. This is especially important if the semaphore is located in the ADSP-2106x's own internal memory or IOP registers.

2. When attempting a read-modify-write operation on a semaphore, the ADSP-2106x must have bus mastership for the duration of the operation.

Both of these rules are adhered to when an ADSP-2106x uses its bus lock feature, which "locks in" its mastership of the bus and prevents the other processors from simultaneously accessing the semaphore.

Bus lock is requested by setting the BUSLK bit in the MODE2 register. When this happens, the ADSP-2106x initiates the bus arbitration process in the usual fashion, by asserting its $\overline{BR}$x line. When it becomes bus master, it locks the bus (i.e. retains bus mastership) by keeping its $\overline{BR}$x line asserted even when it is not performing an external read or write. Host bus request ($\overline{HBR}$) is also ignored during a bus lock. When the BUSLK bit is cleared, the ADSP-2106x gives up the bus by deasserting its $\overline{BR}$x line.

While the BUSLK bit is set, the ADSP-2106x can determine if it has acquired bus mastership by executing a conditional instruction with the *BM* or *NOT BM* condition codes, for example:

```
IF NOT BM JUMP(PC,0);      /* wait for bus mastership */
```

If it has become the bus master, the ADSP-2106x can proceed with the external read or write. If not, it can clear its BUSLK bit and try again later.

A *read-modify-write* operation is accomplished with the following steps:

1. Request bus lock by setting the BUSLK bit in MODE2.
2. Wait for bus mastership to be acquired.
3. Wait until Direct Write Pending (DWPD) is zero.
4. Read the semaphore, test it, and then write to it.

Locking the bus prevents other processors from writing to the semaphore while the read-modify-write is occurring. (**Note:** If the semaphore is reflective, located in the ADSP-2106x's internal memory or an IOP register, the processor must write to it only when it has bus lock.) After bus mastership is aquired, the Direct Write Pending status in SYSTAT must be checked to ensure that a semaphore write by another processor is not pending.

Bus lock can be used in combination with broadcast writes to implement *reflective semaphores* in a multiprocessing system. The reflective semaphore should be located at the same address in internal memory (or IOP register) of each ADSP-2106x. To check the semaphore, each ADSP-2106x simply reads from its own internal memory. To modify the semaphore, an ADSP-2106x requests bus lock and then performs a broadcast write to the semaphore address on every ADSP-2106x, including itself. Before modifying the semaphore, though, the ADSP-2106x must re-check it to verify that another processor has not changed it. With reflective semaphores, the external bus is used only for updating the semaphore, not for reading it. This greatly reduces bus traffic.

7.6.1 Example: Sharing A DMA Channel With Reflective Semaphores

A single DMA channel can be shared by more than one ADSP-2106x by using the channel's control register as a reflective semaphore. The DMA channel control register is a memory-mapped IOP register on each ADSP-2106x. If the control register is equal to zero, the channel is disabled and is not being used by any processor. If the control register is non-zero, the DMA channel is in use.

Before an ADSP-2106x can use the DMA channel, it must read the semaphore to determine if the channel is in use. If not, the ADSP-2106x can request bus lock and then execute a read-modify-write operation to set the semaphore on each of the processors sharing the DMA channel. Before performing the read-modify-write, though, the ADSP-2106x should recheck the semaphore to assure that the DMA channel is still free. Once this is done, the ADSP-2106x should clear the BUSLK bit to unlock the bus and can proceed with the DMA transfer(s). When the transfer(s) are completed, this ADSP-2106x must clear the semaphore to tell the other processors that the channel is available for use.

The following code performs the read-modify-write operation described above:

```
#define semaphore 0x0038001C      /* Broadcast write to     */
                                  /* DMAC6 control register */
                                  /* on all ADSP-2106xs.    */
  ...

   BIT SET MODE2 BUSLK;       /* Request bus lock           */
   IF NOT BM JUMP(PC,0);      /* Wait for bus mastership    */
   USTAT1=DM(SYSTAT);         /* Check Direct Write Pending */
   BIT TST USTAT1 0X1000;     /* Status to ensure no
                                 writes happen              */
   IF NE JUMP (PC,-2);        /* After bus lock             */
   R0=DM(semaphore);          /* Read semaphore             */
   R0=PASS RO                 /* Set condition codes        */

   IF NE JUMP(PC,3);          /* Test semaphore - don't write
                                 if resource is unavailable.*/
   R0=R0+1;                   /* Modify semaphore           */
   DM(semaphore)=R0;          /* Write semaphore            */
   BIT CLR MODE2 BUSLK;       /* Release bus lock           */
```

Notes:

1.) The `IF NOT BM JUMP(PC,0)` instruction through the `IF NE JUMP (PC, -2)` instruction is only necessary for internal semaphores.
2.) The `R0=DM(semaphore)` instruction will not be executed until bus mastership is acquired and locked.
3.) The `DM(semaphore)=R0` instruction is a broadcast write.

7.7 INTERPROCESSOR MESSAGES & VECTOR INTERRUPTS

The ADSP-2106x bus master can communicate with slave ADSP-2106xs by writing messages to their IOP registers. The MSGR0-MSGR7 registers are general-purpose registers which can be used for convenient message passing between ADSP-2106xs. They are also useful for semaphores and resource sharing between multiple ADSP-2106xs. The MSGRx and VIRPT registers can be used for message passing in the following ways:

- **Message Passing.** The master ADSP-2106x can communicate with a slave ADSP-2106x by writing and/or reading any of the 8 message registers, MSGR0–MSGR7, on the slave.

- **Vector Interrupts.** The master ADSP-2106x can issue a vector interrupt to a slave by writing the address of an interrupt service routine to the slave's VIRPT register. This causes an immediate high-priority interrupt on the slave which, when serviced, will cause it to branch to the specified service routine.

The MSGRx and VIRPT registers also support the host processor interface. Since these registers may be shared resources within a single ADSP-2106x, conflicts may occur—your system software must prevent this. For further discussion of IOP register access conflicts, refer to the *Control/Status Registers* appendix of this manual.

7.7.1 Message Passing (MSGRx)

There are three methods by which the ADSP-2106x bus master can communicate with a slave through the MSGRx message registers: 1) *vector-interrupt-driven*, 2) *register handshake*, and 3) *register write-back*.

For the *vector-interrupt-driven* method, the master fills predetermined MSGRx registers on the slave with data and triggers a vector interrupt by writing the address of the service routine to the slave's VIRPT register. The slave's service routine should read the data from the MSGRx registers and then write "0" into VIRPT to tell the master it is done. The service routine could also use one of the slave's $FLAG_{3-0}$ pins to tell the master it has finished.

For the *register handshake* method, four of the MSGRx registers should be designated as follows: a receive register (R), a receive handshake register (RH), a transmit register (T), and a transmit handshake register (TH). To pass data to the slave ADSP-2106x, the master would write data into T and then write a "1" into TH. When the slave sees a "1" in TH, it reads the data from T and then writes back a "0" into TH. When

the master sees a “0” in TH, it knows that the transfer is complete. A similar sequence of events occurs when the slave passes data to the master through R and RH.

The *register write-back* method is similar to register handshaking, but uses only the T and R data registers. The master writes data to T. When the slave sees a non-zero value in T, it retrieves it and writes back a “0” to T. A similar sequence occurs when the master is receiving data. This simpler method works well as long as the data to be passed does not include “0.”

7.7.2 Vector Interrupts (VIRPT)

Vector interrupts are used for interprocessor commands between two ADSP-2106xs or between a host and the ADSP-2106x. When the external processor writes an address to the ADSP-2106x’s VIRPT register, a vector interrupt is caused.

When the vector interrupt is serviced, the ADSP-2106x automatically pushes the status stack and begins executing the service routine located at the address specified in VIRPT. The lower 24 bits of VIRPT contain the address; the upper 8 bits may be optionally used as data to be read by the interrupt service routine. At reset, VIRPT is initialized to its standard address in the ADSP-2106x’s interrupt vector table.

The minimum latency for vector interrupts is six cycles, five of which are NOPs. When the RTI (return from interrupt) instruction is reached in the service routine, the ADSP-2106x automatically pops the status stack.

The VIPD bit in the SYSTAT register reflects the status of the VIRPT register. If VIRPT is written while a previous vector interrupt is pending, the new vector address replaces the pending one. If VIRPT is written while a previous vector interrupt is being serviced, the new vector address is ignored and no new interrupt is triggered. If the ADSP-2106x writes to its own VIRPT register it is ignored.

To use the slave ADSP-2106x’s vector interrupt feature, the master ADSP-2106x should perform the following sequence of actions:

1. Poll the slave’s VIRPT register until it reads a certain token value (i.e. zero).
2. Write the vector interrupt service routine address to VIRPT.
3. When the service routine is finished, the slave ADSP-2106x should write the token back into VIRPT to indicate that it is finished and that another vector interrupt can be initiated.

The DWPD (Direct Write Pending) bit of the SYSTAT register indicates when a direct write to internal memory is pending. Pending direct writes may occur in different sequences. If, for example, the master ADSP-2106x performs a direct write to a slave and then writes to an IOP register on the slave, the IOP register write may complete before the direct write. Because of this, direct writes performed just before vector interrupt writes (to VIRPT) may be delayed until after the branch to the interrupt vector:

1. The master ADSP-2106x performs a direct write to the internal memory of a slave.
2. The master ADSP-2106x writes to the VIRPT register of the slave to initiate a vector interrupt. This causes the direct write to be delayed.
3. The slave ADSP-2106x jumps to the vector interrupt service routine.
4. The direct write is completed after the interrupt service routine is underway.

To prevent this from happening, the master ADSP-2106x should check that all direct writes have completed before writing to the slave's VIRPT register. This can be done by polling the slave's DWPD bit (in SYSTAT) after performing a direct write, waiting for it to become cleared, and then proceeding with the write to VIRPT.

7.8 SYSTAT REGISTER STATUS BITS

The SYSTAT register provides status information, primarily for multiprocessor systems. Table 7.4 shows the status bits in this register.

Bit(s) *Name*	*Definition*
HSTM	Host Mastership
BSYN	Bus Synchronization
CRBM	Current Bus Master (ID_{2-0} of ADSP-2106x bus master)
IDC	ID Code (ID_{2-0} of this ADSP-2106x)
DWPD	Direct Write Pending
VIPD	Vector Interrupt Pending
HPS	Host Packing Status

Table 7.4 SYSTAT Status Bits

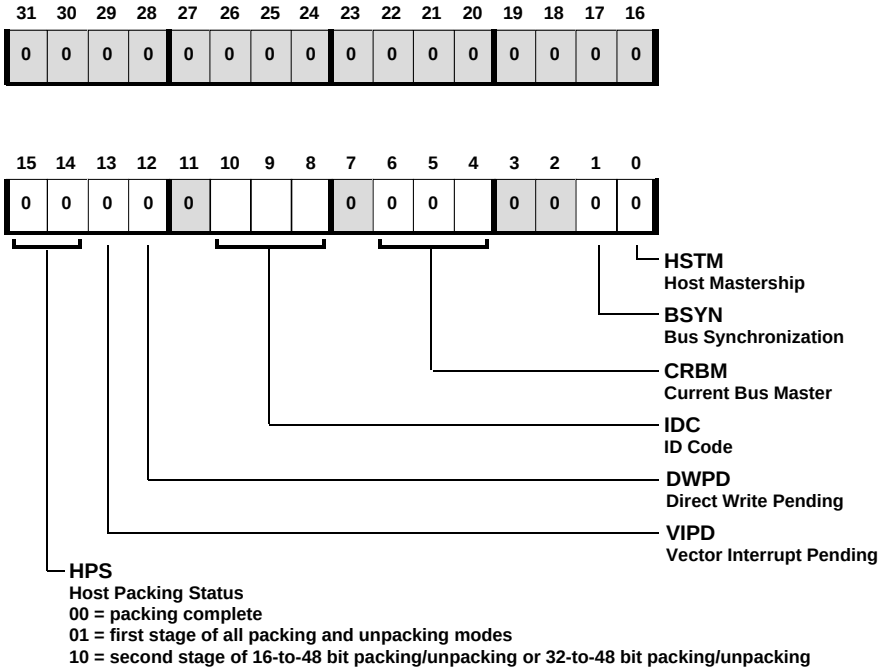


Figure 7.9 SYSTAT Register

HSTM **Host Mastership.** Indicates whether the host processor is has been granted control of the bus.

1=Host is bus master
0=Host is not bus master

BSYN **Bus Synchronization.** Indicates when the ADSP-2106x's bus arbitration logic is synchronized after reset. (See "Bus Synchronization After Reset.")

1=Bus arbitration logic is synchronized
0=Bus arbitration logic is not synchronized

CRBM **Current Bus Master.** Indicates the ID code of the ADSP-2106x that is the current bus master. If CRBM is equal to the ID of this ADSP-2106x then it is the current bus master. CRBM is only valid for $ID_{2-0} > 0$ (greater than zero). When ID_{2-0}=000, CRBM is always 1.

IDC **ID Code.** Indicates the ID_{2-0} inputs of this ADSP-2106x.

DWPD **Direct Write Pending.** Indicates when a direct write to the ADSP-2106x's internal memory is pending. The DWPD bit is cleared when the direct write has been completed. (Direct writes may be delayed for several cycles is DMA chaining is underway or if higher priority DMA requests occur. Maximum delay is 12 cycles.)

1=Direct write pending
0=No direct write pending

VIPD **Vector Interrupt Pending.** Indicates that a pending vector interrupt has not yet been serviced. The VIPD bit is set when the VIRPT register is written to and is cleared upon return from the interrupt service routine. The master ADSP-2106x (or host processor) that issued the vector interrupt should monitor this bit to determine when the service routine has been completed, and when a new vector interrupt may be issued.

1=Vector interrupt pending
0=No vector interrupt pending

HPS **Host Packing Status.** Indicates when host word packing is completed or, if not, what stage of the process is taking place.

00=Packing complete
01=1st stage of all packing and unpacking modes.
10=2nd stage of 16-to-48 bit packing/unpacking or 32-to-48 bit packing/unpacking

Host Interface 8

8.1 OVERVIEW

The ADSP-2106x's host interface allows easy connection to standard microprocessor buses, both 16-bit and 32-bit, with little additional hardware required. The ADSP-2106x accommodates either synchronous or asynchronous data transfers, allowing the host to use a different clock frequency. Asynchronous transfers at speeds up to the full clock rate of the processor are supported. The host accesses the ADSP-2106x through its external port, via the external bus ($DATA_{47-0}$ and $ADDR_{31-0}$). The host interface is memory-mapped into the unified address space of the ADSP-2106x. Figure 8.1 shows a block diagram of the external port, I/O processor, and FIFO data buffers, illustrating the on-chip data paths for host-driven transfers. The four external port DMA channels are available for use by the host—DMA transfers of code and data can be performed with low software overhead.

The host processor requests and controls the ADSP-2106x's external bus with the host bus request ($\overline{HBR}$), host bus grant ($\overline{HBG}$), and ready (REDY) signals. Once it has gained control of the bus, the host can can directly read and write the internal memory of the ADSP-2106x. It can also read and write to any of the ADSP-2106x's IOP registers, including the EPBx FIFO buffers. The host uses certain IOP registers to control and configure the ADSP-2106x, SYSCON and SYSTAT for example, and to set up DMA transfers. DMA transfers are controlled by the ADSP-2106x's on-chip DMA controller once they have been set up by the host (or by the ADSP-2106x core). In a multiprocessor system, the host can access the internal memory and IOP registers of every ADSP-2106x. Vector interrupts provide efficient execution of host commands.

8 Host Interface

Any host microprocessor with a standard memory interface can easily connect to the ADSP-2106x bus through buffers. By providing an address, a data bus, and memory control signals—i.e. read, write and chip select—a host may access any device on the ADSP-2106x bus as if it were a memory. The host data bus width may be either 16 or 32 bits, and the host-driven address may be either 8 or 32 bits. Any one of ADSP-2106xs on the bus can be addressed, either with their chip select ($\overline{CS}$) signal or with a memory-mapped address. All of the internal registers and resources of the ADSP-2106x's I/O processor, such as the DMA control registers, are available to the host. A host bus acknowledge signal, REDY, is provided to indicate the completion of each transfer.

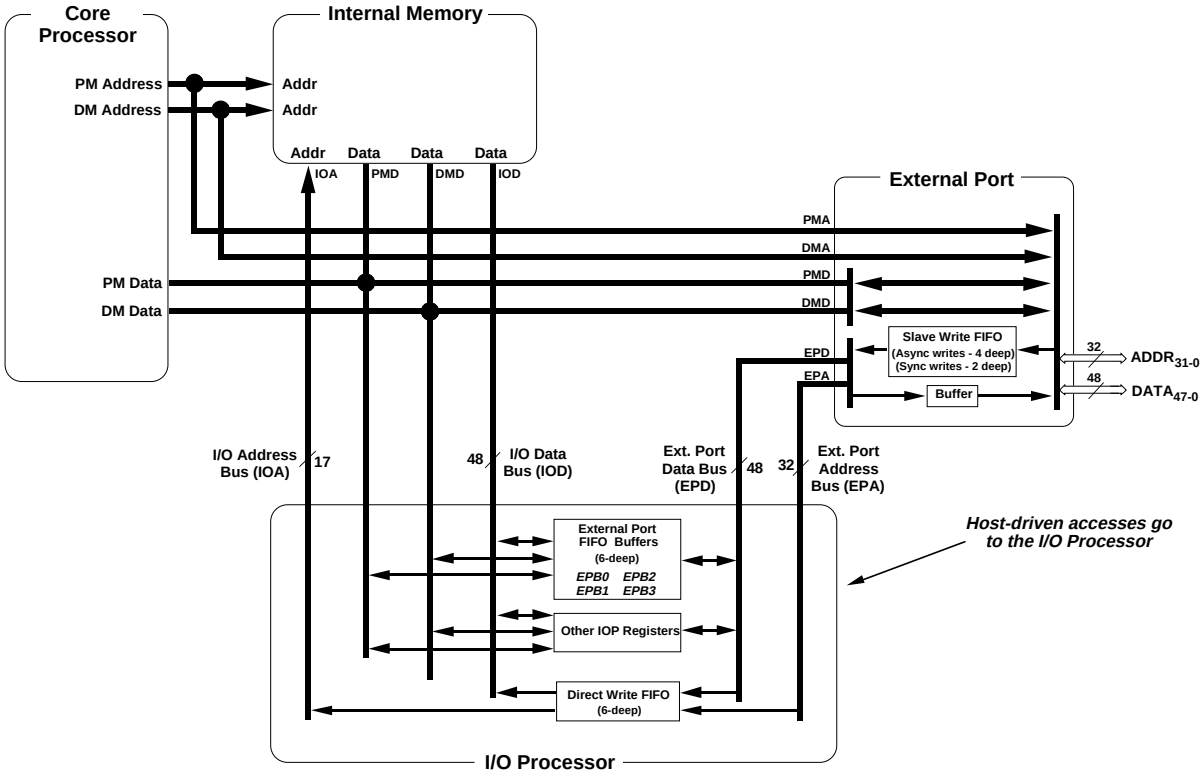


Figure 8.1 External Port & Host Interface

Host Interface 8

Table 8.1 defines the ADSP-2106x pins used in host processor interfacing.

Signal	*Type*	*Definition*
$\overline{\text{HBR}}$	I / A	**Host Bus Request**. Must be asserted by a host processor to request control of the ADSP-2106x's external bus. When $\overline{\text{HBR}}$ is asserted in a multiprocessing system, the ADSP-2106x that is bus master will relinquish the bus and assert $\overline{\text{HBG}}$. To relinquish the bus, the ADSP-2106x places the address, data, select, and strobe lines in a high-impedance state. $\overline{\text{HBR}}$ has priority over all ADSP-2106x bus requests ($\overline{\text{BR}}_{1\text{-}6}$) in a multiprocessing system.
$\overline{\text{HBG}}$	I / O	**Host Bus Grant**. Acknowledges an $\overline{\text{HBR}}$ bus request, indicating that the host processor may take control of the external bus. $\overline{\text{HBG}}$ is asserted (held low) by the ADSP-2106x until $\overline{\text{HBR}}$ is released. In a multiprocessing system, $\overline{\text{HBG}}$ is output by the ADSP-2106x bus master and is monitored by all others.
$\overline{\text{CS}}$	I / A	**Chip Select**. Asserted by host processor to select the ADSP 2106x.
REDY (o / d)	O	**Host Bus Acknowledge**. The ADSP-2106x deasserts REDY (low) to add wait states to an asynchronous access of its internal memory or IOP registers by a host. Open-drain output (o / d) by default; can be programmed in ADREDY bit of SYSCON register to be active drive (a / d). REDY will only be output if the $\overline{\text{CS}}$ and $\overline{\text{HBR}}$ inputs are asserted.
$\overline{\text{SBTS}}$	I / S	**Suspend Bus Tristate**. External devices can assert $\overline{\text{SBTS}}$ (low) to place the external bus address, data, selects, and strobes in a high-impedance state for the following cycle. If the ADSP-2106x attempts to access external memory while $\overline{\text{SBTS}}$ is asserted, the processor will halt and the memory access will not be completed until $\overline{\text{SBTS}}$ is deasserted. $\overline{\text{SBTS}}$ should only be used to recover from PAGE faults or host processor / ADSP-2106x deadlock.

I=Input S=Synchronous (o / d)=Open Drain
O=Output A=Asynchronous (a / d)=Active Drive

Table 8.1 Host Interface Signals

8 Host Interface

The following terms are used throughout this chapter, and are defined below for reference:

external bus	$DATA_{47-0}$, $ADDR_{31-0}$, $\overline{RD}$, $\overline{WR}$, $\overline{MS}_{3-0}$, $\overline{BMS}$, ADRCLK, PAGE, $\overline{SW}$, ACK, and $\overline{SBTS}$ signals
multiprocessor system	a system with multiple ADSP-2106xs, with or without a host processor; the ADSP-2106xs are connected by the external bus and/or link ports
multiprocessor memory space	portion of the ADSP-2106x's memory map that includes the internal memory and IOP registers of each ADSP-2106x in a multiprocessing system; this address space is mapped into the unified address space of the ADSP-2106x
IOP register	one of the control, status, or data buffer registers of the ADSP-2106x's on-chip I/O processor
bus slave *or* slave mode	an ADSP-2106x can be a bus slave to another ADSP-2106x or to a host processor; the ADSP-2106x becomes a "host bus slave" when the $\overline{HBG}$ signal is returned
bus transition cycle (BTC)	a cycle in which control of the external bus is passed from one ADSP-2106x to another (in a multiprocessor system)
host transition cycle (HTC)	a cycle in which control of the external bus is passed from the ADSP-2106x to the host processor—during this cycle the ADSP-2106x stops driving the $\overline{RD}$, $\overline{WR}$, $ADDR_{31-0}$, $\overline{MS}_{3-0}$, ADRCLK, PAGE, $\overline{SW}$, and $\overline{DMAG}$x signals, which must then be driven by the host
asynchronous transfers	asynchronous host accesses of the ADSP-2106x; after acquiring control of the ADSP-2106x's external bus, the host must assert the $\overline{CS}$ pin of the ADSP-2106x it wants to access
synchronous transfers	synchronous host accesses of the ADSP-2106x; $\overline{CS}$ is not asserted and the host must act like another ADSP-2106x in a multiprocessor system, by generating an address in multiprocessor memory space, asserting $\overline{SW}$ and $\overline{WR}$ or $\overline{RD}$, and driving out or latching in the data

direct reads & writes	a direct access of the ADSP-2106x's internal memory or IOP registers by another ADSP-2106x or by a host processor
external port FIFO buffers	EPB0, EPB1, EPB2, and EPB3—the IOP registers used for external port DMA transfers and single-word data transfers (from other ADSP-2106xs or from a host processor); these buffers are 6-deep FIFOs
single-word data transfers	reads and writes to the EPBx external port buffers, performed externally by the host or internally by the ADSP-2106x core; these occur when DMA is disabled in the DMACx control register
DMACx control registers	the DMA control registers for the EPBx external port buffers: DMAC6, DMAC7, DMAC8, and DMAC9, corresponding respectively to EPB0, EPB1, EPB2, and EPB3 (see the *DMA* chapter or *Control/Status Registers* appendix of this manual for a complete description of the DMACx control registers)

8.2 HOST PROCESSOR CONTROL OF THE ADSP-2106X

The $\overline{\text{HBR}}$ and $\overline{\text{HBG}}$ signals allow a host processor to gain control of the ADSP-2106x and its external bus. Once the host is granted control of the ADSP-2106x bus, it may transfer data either synchronously or asynchronously. Asynchronous transfers are most commonly used. The host bus may be 16, 32, or 48 bits wide for synchronous transfers, but only 16 or 32 bits wide for asynchronous transfers.

Data written to and read from the ADSP-2106x can be packed or unpacked into different word widths. When the width of the host bus is 16 bits, data can be packed into 32-bit words or 48-bit words. When the host bus is 32 bits wide, data can be packed into 48-bit words. The host packing mode control bits (HPM) in the SYSCON register are used to configure data packing and unpacking.

8.2.1 Acquiring The Bus

For a host processor to gain access to the ADSP-2106x, it must first assert $\overline{\text{HBR}}$, the host bus request signal. $\overline{\text{HBR}}$ has priority over all $\overline{\text{BR}}$x multiprocessor bus requests, and when asserted will cause the current ADSP-2106x master to give up the bus to the host as soon as it has finished the current bus cycle.

The current ADSP-2106x bus master signals that it is transferring ownership of the bus by asserting $\overline{\text{HBG}}$ (low) as soon as the current bus operation has completed. The cycle in which control of the bus is transferred is called a *host transition cycle* (HTC).

Figure 8.2 shows the timing for bus acquisition by the host. $\overline{\text{HBG}}$ is asserted during the ADSP-2106x's bus transition cycle (BTC) and remains asserted until $\overline{\text{HBR}}$ is deasserted by the host. (The bus transition cycle (BTC) shown in Figure 8.2 is the same as that of a SHARC-to-SHARC multiprocessor BTC, as described in Chapter 7, *Multiprocessing*. $\overline{\text{HBG}}$ freezes ADSP-2106x multiprocessor bus arbitration during the time that the host owns the bus. ($\overline{\text{HBG}}$ should also be used to enable the host's signal buffers, as shown in Figure 8.8 at the end of this chapter.) While $\overline{\text{HBG}}$ is asserted, the ADSP-2106xs will continue to assert and deassert their $\overline{\text{BR}}$x lines as in normal operation, but no BTCs will occur. The current ADSP-2106x bus master will keep its $\overline{\text{BR}}$x asserted throughout the entire time the host controls the bus.

Once the host has gained control of the bus, it can choose to perform either synchronous or asynchronous transfers with the ADSP-2106x(s). To initiate asynchronous transfers, the host asserts (low) the $\overline{\text{CS}}$ pin of the ADSP-2106x that it intends to access and performs the read or write. ($\overline{\text{CS}}$ is ignored when $\overline{\text{HBG}}$ is not asserted.) To initiate synchronous transfers, the host keeps all ADSP-2106x $\overline{\text{CS}}$ pins deasserted (high) and reads or writes to the ADSP-2106xs' *multiprocessor memory space* (just as one ADSP-2106x reads or writes to another ADSP-2106x).

The host is responsible for driving the following signals during the HTC in which it gains control of the bus: $\text{ADDR}_{31\text{-}0}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{SW}}$, and PAGE. See Figure 8.3. These signals must also continue to be driven for the entire time the host has the bus. In addition, the $\overline{\text{MS}}_{3\text{-}0}$, ADRCLK, $\overline{\text{DMAG1}}$, and $\overline{\text{DMAG2}}$ lines must be driven or weakly pulled up or down—the ADSP-2106x bus master tristates these lines to allow the host the possibility of using them.

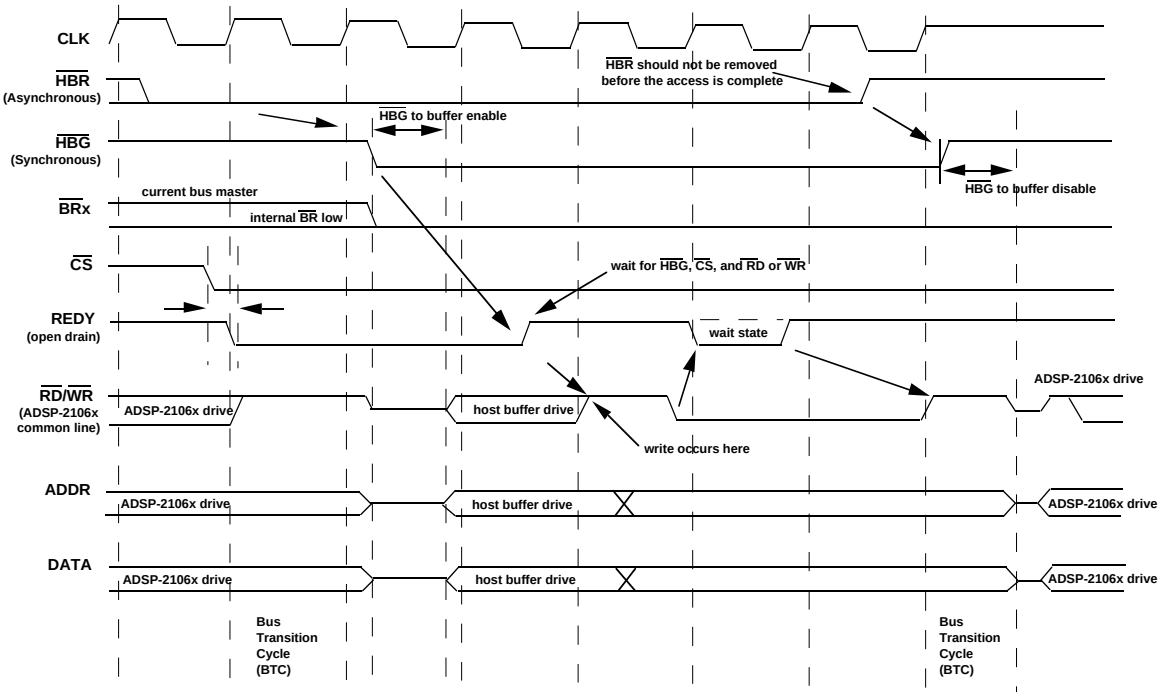


Figure 8.2 Example Timing For Bus Acquisition

During read-modify-write operations, the host should ensure that $\overline{HBR}$ is not deasserted to avoid temporary loss of bus mastership. $\overline{HBR}$ must remain asserted until after the host completes the last data transfer.

The following restrictions apply to bus acquisition by the host:

- If $\overline{HBR}$ is asserted while the ADSP-2106x is in reset, the ADSP-2106x will not respond with $\overline{HBG}$ until after reset and multiprocessor synchronization is completed.
- $\overline{HBR}$ should not be deasserted during a host access.
- If $\overline{SBTS}$ is asserted after $\overline{HBR}$, the ADSP-2106x may enter slave mode and suspend any unfinished access to the external bus. (See the discussion of “Deadlock Resolution” in the “System Bus Interface” section of this chapter for further details.)
- If the host is to execute both synchronous and asynchronous accesses during a single bus grant, it must allow at least one cycle to pass after the last access before switching $\overline{CS}$.

- Synchronous accesses may not be used in systems with only one ADSP-2106x (with ID_{2-0}=000).

Once the host has finished its task, it can relinquish control of the bus by deasserting $\overline{HBR}$. The ADSP-2106x bus master responds by deasserting $\overline{HBG}$. In the next cycle, the ADSP-2106x bus master again assumes control of the bus and normal multiprocessor arbitration resumes. The host should not deassert $\overline{HBR}$ until after it has completed its last data transfer with the ADSP-2106x.

8.2.2 Asynchronous Transfers

To initiate asynchronous transfers after acquiring control of the ADSP-2106x's external bus, the host must assert the $\overline{CS}$ pin of the ADSP-2106x it wants to access. This informs the ADSP-2106x that it will be transferring data asynchronously with the host. The host must then drive the address of the memory location or IOP register that it wants to access. To simplify the hardware requirements for external interface logic, only the address bits shown in Table 8.2 need to be driven.

*Address Bits **	*Comments*
$ADDR_{7-0}$	Must be driven in all cases.
$ADDR_{16-8}$	Must be driven only if the S field indicates an internal memory access.
$ADDR_{18-17}$	S field**—Must be driven "00" for IOP register accesses, "01" for internal memory accesses, or "1m" for short word accesses.
and either	
$ADDR_{21-19}$	M field**—Must be driven "000" to prevent other non-selected ADSP-2106xs on the bus from thinking that a synchronous *multiprocessor memory space* access is occurring.
or	
$ADDR_{31-22}$	E field**—one of the lines 31-22 driven as "1".

Table 8.2 Address Bits To Be Driven During Asynchronous Host Accesses

* Setup and hold times for these address lines are specified in the *ADSP-2106x Data Sheet*.

** For a complete description of these address fields, see "ADSP-2106x Memory Map" in the *Memory* chapter of this manual.

Table 8.2 covers all cases, including multiprocessor systems, but fewer address bits may need to be driven depending on the system. In a single-ADSP-2106x system, the host need not drive the M address field.

Host direct reads and writes can be performed with normal words (32-bit or 48-bit) or short words (16-bit). Short words are accessed if the S field of the address is "1m", where "m" is the most significant bit of the short word address. Normal words are accessed if the S field is "01". 48-bit words are accessed if the IWT bit in the SYSCON register is set to 1, selecting instruction word transfers.

When using asynchronous transfers and direct access to internal memory is not required, the address supplied to the ADSP-2106x need not be the full 32 bits. Only the lower 8 bits, $ADDR_{7-0}$, need be supplied. The $ADDR_{21-17}$ bits must be zeros and the $ADDR_{31-22}$ bits are ignored, or $ADDR_{21-19}$ need not be driven if one of the $ADDR_{31-22}$ bits is a 1. See Table 8.2.

Asynchronous Write Clock Rates
To allow full speed asynchronous writes, data is latched at the I/O pins in a four-level FIFO buffer; this buffer is called the *slave write FIFO* (see Figure 8.1). This buffering allows previously written words to be resynchronized while a new word is being written, and allows asynchronous writes to occur at up to the full clock rate of the ADSP-2106x.

Broadcast Writes
A host may broadcast write to several ADSP-2106xs by asserting each of their $\overline{CS}$ pins. Each ADSP-2106x will accept the write as if it were the only device being addressed. Because the REDY line is wire-ORed (if configured as an open-drain output), it will only appear asserted when all selected ADSP-2106xs are ready. (This is true only if REDY is not actively pulled up.) ACK is not active when $\overline{CS}$ is asserted.

Uniprocessor Host Interface
To eliminate the need for a host to drive the M field address lines ($ADDR_{21-19}$) in systems with only one ADSP-2106x (ID_{2-0}=000), this ADSP-2106x will not recognize synchronous accesses. The host must drive these address lines however, if the ADSP-2106x's ID_{2-0} is anything other than 000. (Note that this removes the requirement that $\overline{CS}$ be asserted before $\overline{RD}$.) To account for buffer delays when sampling the REDY signal, be careful to ensure that it is properly resynchronized by the host.

8.2.2.1 Asynchronous Transfer Timing

When a ADSP-2106x's $\overline{CS}$ chip select is asserted (low), the selected ADSP-2106x immediately deasserts the READY signal, with a delay of approximately 10 ns. Refer to the *ADSP-2106x Data Sheet* for timing exact specifications. (The READY deassertion is activated from $\overline{CS}$ and not from $\overline{RD}$ or $\overline{WR}$ because the host interface buffers for $\overline{RD}$ and $\overline{WR}$ may not yet be enabled if $\overline{HBG}$ has not been asserted. $\overline{CS}$ can be asserted before or after $\overline{HBR}$ is asserted, but READY will not be reasserted until after $\overline{HBG}$ has been asserted and a $\overline{RD}$ or $\overline{WR}$ strobe has been applied. This is true only if a $\overline{RD}$ or $\overline{WR}$ strobe is active when $\overline{HBG}$ is asserted, otherwise it is determined by the t_{TRDYHG} switching characteristic specified in the "Multiprocessor Bus Request & Host Bus Request" timing section of the *ADSP-2106x Data Sheet*.)

READY is asserted prior to the $\overline{RD}$ or $\overline{WR}$ being asserted and becomes deasserted only if the ADSP-2106x is not ready for the read or write to complete—the only exception is when $\overline{CS}$ is first asserted. The READY pin is an open-drain output to facilitate interfacing to common buses. It can be changed to an active-drive output by setting the ADREDY bit in the SYSCON register.

Figure 8.3 shows the timing of a *host write cycle*, discussed below, including details of data packing and unpacking. This timing assumes the use of the example host interface hardware shown in Figure 8.8 at the end of this chapter.

1. The host asserts the address. $\overline{HBR}$ and $\overline{CS}$ are decoded from the host bus interface address comparator and need not be supplied directly by the host. The selected ADSP-2106x deasserts REDY immediately.
2. The host asserts $\overline{WR}$ and drives data (according to the timing requirements specified in the data sheet).
3. The selected ADSP-2106x asserts REDY when it is ready to accept the data. This occurs after the current bus master has completed its current transfer and has asserted $\overline{HBG}$. $\overline{HBG}$ enables the host interface buffers to drive onto the ADSP-2106x bus.
4. The host deasserts $\overline{WR}$ when REDY is high and stops driving data.
5. The selected ADSP-2106x latches data on the rising edge of $\overline{WR}$.

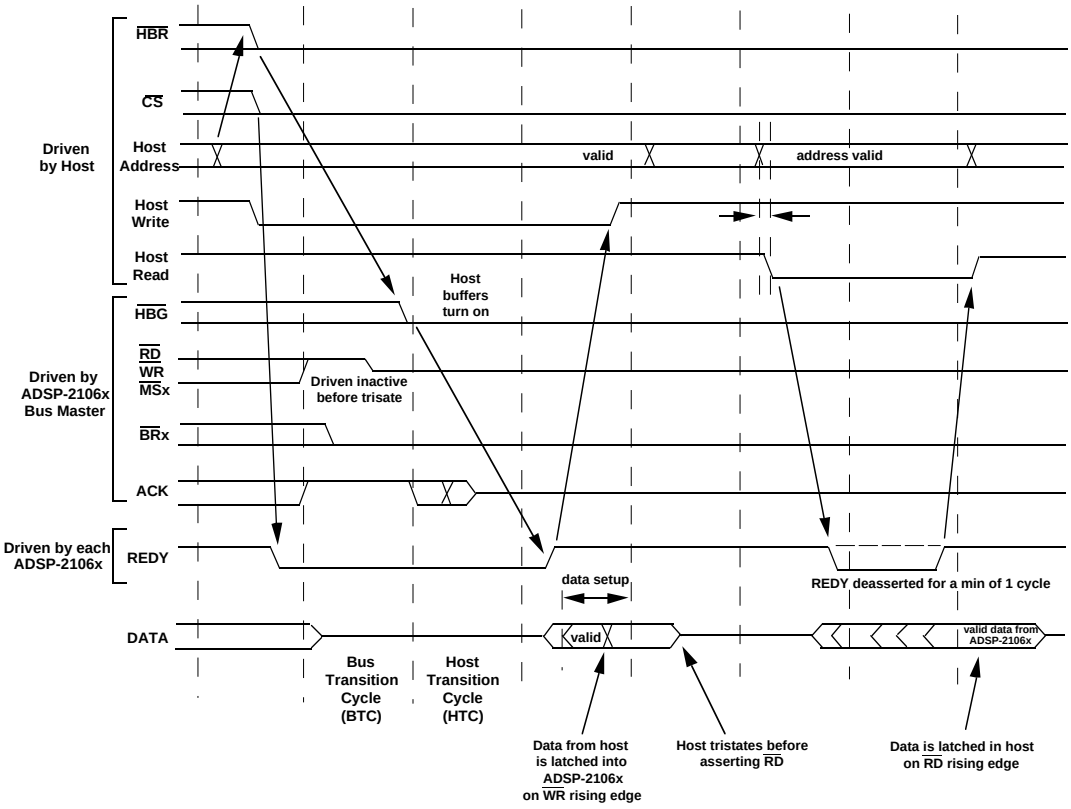


Figure 8.3 Example Timing For Host Read & Write Cycles

Note: In this example host interface, $\overline{HBR}$ and $\overline{CS}$ are derived from an address comparator circuit (see Figure 8.8).

After the first word, the write sequence is:

6. The host asserts $\overline{WR}$ and drives data (according to the timing requirements specified in the data sheet).
7. The ADSP-2106x deasserts REDY if it is not ready to accept data.
8. The host deasserts $\overline{WR}$ when REDY is high and stops driving data.
9. The selected ADSP-2106x latches data on the rising edge of $\overline{WR}$.

More than one ADSP-2106x may have its $\overline{CS}$ pin asserted at any one time during a write, but not during a read because of bus conflicts.

Figure 8.3 also shows the timing of a *host read cycle*, again assuming the use of the bus interface hardware of Figure 8.8, with the following sequence:

1. The host asserts the address. $\overline{HBR}$ and the appropriate $\overline{CS}$ line are again decoded by the host bus interface address comparator. The selected ADSP-2106x deasserts REDY immediately and asserts $\overline{HBG}$.
2. The host asserts $\overline{RD}$.
3. The selected ADSP-2106x drives data onto the bus and asserts REDY when the data is available.
4. The host latches the data and deasserts $\overline{RD}$.

After the first word, the read sequence is:

5. The host asserts $\overline{RD}$.
6. The selected ADSP-2106x deasserts REDY then asserts REDY, driving data when it becomes available.
7. The host deasserts $\overline{RD}$ when REDY is high and latches the data.

8.2.3 Synchronous Transfers

To perform synchronous data transfers, the $\overline{CS}$ input is not asserted and the host must act like another ADSP-2106x in a multiprocessor system. To do this, it must generate an address in the multiprocessor memory space of an ADSP-2106x, assert $\overline{SW}$ and $\overline{WR}$ or $\overline{RD}$, and drive out or latch in the data.

Synchronous accesses may not be used in uniprocessor systems (ID_{2-0}=000). To perform synchronous accesses in a multiprocessor system, the host must drive the M address field ($ADDR_{21-19}$) with a value of 0-7 to select one of the ADSP-2106xs (by its ID_{2-0}) *or* one of the E field address lines must be driven high to select an address in external memory. (Address fields are described in the “ADSP-2106x Memory Map” section of the *Memory* chapter.)

For synchronous host transfers, the ADSP-2106x uses its ACK signal instead of REDY to add waitstates to an access—the host must wait for the ADSP-2106x to assert ACK. Synchronous accesses will not be recognized during the Host Transition Cycle (see Figure 8.3). This prevents any spurious access from occurring while the external host buffers are starting to drive the address, data, and strobes. Note that ACK may glitch during the HTC and should not be relied on until the following cycle.

When performing synchronous transfers, the host should use the same number of wait states as are configured for *multiprocessor memory space wait states*; otherwise the system may hang. This is configured by the MMSWS bit of the WAIT register.

When an ADSP-2106x is responding to a synchronous read access, it will only drive valid data for one cycle, even if the access is prolonged by the host. Specifically, after the host synchronously asserts $\overline{RD}$, the ADSP-2106x will drive valid data only in the first cycle it asserts ACK and will tristate the data bus during the following cycles, even if the host continues to assert $\overline{RD}$.

8.2.4 Host Interface Deadlock Resolution With $\overline{SBTS}$

If $\overline{SBTS}$ and $\overline{HBR}$ are both asserted, the ADSP-2106x will enter slave mode. ACK, $\overline{HBG}$, REDY, and the data bus may all be active in slave mode. If the ADSP-2106x was performing an external access (which did not complete) in the same cycle that $\overline{SBTS}$ and $\overline{HBR}$ were asserted, the access will be suspended until $\overline{SBTS}$ and $\overline{HBR}$ are both deasserted again.

This functionality, i.e. using $\overline{SBTS}$ and $\overline{HBR}$ together, can be used for host / ADSP-2106x deadlock resolution. If $\overline{SBTS}$ and $\overline{HBR}$ are asserted while an external DMA access is occurring, $\overline{HBG}$ will not be asserted until the access is completed. If $\overline{SBTS}$ and $\overline{HBR}$ are asserted while bus lock is set, the ADSP-2106x will tristate its bus signals but will not go into slave mode.

See “Deadlock Resolution” in the “System Bus Interfacing” section of this chapter for further details.

8.3 SLAVE DIRECT READS & WRITES

The host can directly access the internal memory and IOP registers of an ADSP-2106x by simply reading or writing to the appropriate address in multiprocessor memory space—this is called a *direct read* or *direct write*. Each ADSP-2106x bus slave monitors addresses driven on the external bus and responds to any that fall within its region of multiprocessor memory space.

> *These accesses are invisible to the slave ADSP-2106x’s core processor because they are performed through the external port and via the on-chip I/O bus—not the DM bus or PM bus. (See Figure 8.1.)*

This is an important distinction, because it allows the core processor to continue program execution uninterrupted.

8 Host Interface

The host can directly read and write the IOP registers to control and configure the ADSP-2106x, for example in SYSCON and SYSTAT, and to set up DMA transfers.

The IWT (Instruction Word Transfer) bit controls internal memory width for instruction transfers. IWT=1 overrides the IMDW bits and forces a 48-bit (3-column) memory transfer. IWT=0 defers to the data word setting of the IMDW bits in the SYSCON register.

Synchronous and asynchronous direct read/writes are performed in the same way by the host, and the following sections apply to both synchronous and asynchronous direct accesses. The only difference is which signal, REDY or ACK, the ADSP-2106x uses to add wait states to these accesses. For asynchronous direct reads and writes, the REDY signal is used. For synchronous direct reads and writes, ACK is used.

Synchronous and asynchronous broadcast writes, however, are slightly different, as described below.

8.3.1 Direct Writes

When a direct write to a slave ADSP-2106x occurs, the address and data are latched on-chip by the I/O processor. The I/O processor buffers the address and data in a special set of FIFO buffers. If additional direct writes are attempted when the FIFO buffer is full, the ADSP-2106x deasserts ACK (or REDY) until the buffer is no longer full. Up to six direct writes can be performed before another is delayed. (The direct write buffer itself may be held off for up to four cycles if all of the serial port DMA channels are active or for up to nine cycles per chain if DMA chaining is occurring.)

8.3.1.1 Direct Write Latency

When data is written to an ADSP-2106x bus slave, the data and address are latched at the I/O pins in a four-level FIFO buffer; this buffer is called the *slave write FIFO* (see Figure 8.1). In the following cycle, the slave write FIFO attempts to complete the write internally. This allows the host (or ADSP-2106x bus master) to perform writes at the full clock rate. The slave write FIFO cannot be explicitly read by the slave ADSP-2106x's core processor, nor can its status be determined.

Writes to the IOP registers will usually occur in the following one or two cycles, or when any current DMA transfer is completed. The write will take more than two cycles only if a direct write in the previous cycle was held off by a full buffer.

If the buffer is full when a write is attempted, the ADSP-2106x will deassert ACK (or REDY) until the buffer is not full. The buffer will usually empty out within one cycle, thus creating a write latency, unless higher priority on-chip DMA transfers are occurring.

Slave reads will be held off when there is data in the write FIFO—this prevents false data reads and out-of-sequence operations.

The DWPD (Direct Write Pending) bit of the SYSTAT register indicates when a direct write to internal memory is pending in the I/O processor's direct write FIFO or data is pending in the slave write FIFO (at the external port I/O pins). Direct writes and IOP register accesses may be completed in different sequences. If, for example, the host performs a direct memory write and then writes to an IOP register on the ADSP-2106x, the IOP register write may complete before the direct write.

8.3.2 Direct Reads

When a direct read of an ADSP-2106x occurs, the address is latched on-chip by the I/O processor and ACK (or REDY) is deasserted. When the corresponding location in memory is read internally, the ADSP-2106x drives the data off-chip and asserts ACK (or REDY). Direct reads cannot be pipelined like direct writes—they only occur one at a time.

Note that while direct writes have a maximum pipelined throughput of one per cycle, direct reads have a maximum throughput of one per every two cycles (for synchronous IOP register reads) or one per every four cycles (for synchronous internal memory reads). See Table 11.5, "Data Delays & Throughputs", in Chapter 11. Because of this low bandwidth, direct reads are not the most efficient method of transferring data out of a slave ADSP-2106x—setting up a master mode DMA channel on the slave to perform writes is more efficient, although it requires additional programming. The advantage of direct reads is that no programming of the DMA controller is required.

8.3.3 Broadcast Writes

Broadcast writes allow simultaneous transmission of data to all of the ADSP-2106xs in a multiprocessing system. The host processor (or master ADSP-2106x) can perform broadcast writes to the same memory location or IOP register on all of the slaves. Broadcast writes can be used to implement reflective semaphores in a multiprocessing

system; see “Bus Lock & Semaphores” in the *Multiprocessing* chapter of this manual. Broadcast writes can also be used to simultaneously download code or data to multiple processors.

Asynchronous broadcast writes and synchronous broadcast writes are performed differently by the host. *For asynchronous* broadcast writes, the host must assert $\overline{CS}$ on each ADSP-2106x that it wants to write to. The host must also drive the M address field as zero: $ADDR_{21-19}$=000. The ADSP-2106xs use REDY to add wait states to the asynchronous broadcast write, if necessary, and the host should wire-OR the REDY lines together. REDY must be configured in SYSCON as an open-drain output, not active drive.

To perform *synchronous* broadcast writes, the host must generate an address in the highest region of multiprocessor memory space, addresses 0x0038 0000 to 0x003F FFFF. When a write address falls within this region, each ADSP-2106x responds by accepting the access. The ADSP-2106xs use ACK to add wait states to the synchronous broadcast write, if necessary, and the host should wire-OR the ACK lines together. Timing for synchronous broadcast writes is shown in the *Multiprocessing* chapter.

Because the host must wait for a synchronous broadcast write to complete on *all* of the ADSP-2106xs, the acknowledge signal is handled differently to prevent drive conflicts on the ACK line. A wired-OR acknowledge signal is used to respond to these accesses. This signal operates as follows:

1. In the first cycle of the synchronous broadcast write (and in all succeeding odd cycles), a slave ADSP-2106x will pull ACK low if it is not ready to accept the data. If it is ready, it will not drive the ACK line.

   If the host sees that ACK is high, indicating that all ADSP-2106xs are accepting the broadcast write, it completes the write.

2. During all succeeding even cycles in which the broadcast write is not finished, the slave ADSP-2106xs will not drive ACK. Instead, the master ADSP-2106x drives (i.e. pre-charges) ACK high and the host must continue the write. *(Go to Step 1.)*

In most cases the ACK signal will be high and the ADSP-2106x slaves will be ready to accept data at the start of the synchronous broadcast write—the write completes in one cycle. If the ACK signal is low, however, or one of the slaves is not ready to accept the data, the write will take a minimum of three cycles.

When the wait state for multiprocessor memory space is enabled (with the MMSWS bit of the WAIT register), the master ADSP-2106x will not sample or drive ACK during the first two cycles of a synchronous broadcast write. In this case the write will again take a minimum of three cycles to complete.

(**Note:** The ADSP-2106x bus master enables a keeper latch on the ACK line to prevent the signal from drifting. This eliminates any power consumption caused by the signal drifting to the switching point and improves the robustness of synchronous broadcast writes. Multiprocessor systems that use synchronous broadcast writes should keep the ACK line as free of noise as possible.)

8.3.4 Shadow Write FIFO

Because the ADSP-2106x's internal memory must operate at high speeds, writes to the memory do not go directly into the memory array, but rather to a two-deep FIFO called the *shadow write FIFO*.

When an internal memory write cycle occurs, data in the FIFO from the previous write is loaded into memory and the new data goes into the FIFO. This operation is normally transparent, since any reads of the last two locations written are intercepted and routed to the FIFO. There is only one case in which you need to be aware of the shadow write FIFO: mixing 48-bit and 32-bit word accesses to the same locations in memory.

The shadow FIFO cannot differentiate between the mapping of 48-bit words and mapping of 32-bit words. (See Figures 5.8 and 5.9 in the *Memory* chapter.) Thus if you write a 48-bit word to memory and then try to read the data with a 32-bit word access, the shadow FIFO will not intercept the read and incorrect data will be returned.

If 48-bit accesses and 32-bit accesses to the *same* locations absolutely must be mixed in this way, you must flush out the shadow FIFO with two dummy writes before attempting to read the data.

8.4 DATA TRANSFERS THROUGH THE EPBx BUFFERS

In addition to direct reads and writes, the host processor can transfer data to and from the ADSP-2106x through the external port FIFO buffers, EPB0, EPB1, EPB2, and EPB3. Each of these buffers, which are part of the IOP register set, is a six-location FIFO. Both single-word transfers and DMA transfers can be performed through the EPBx buffers. DMA transfers are handled internally by the ADSP-2106x's DMA controller, but single-word transfers must be handled by the ADSP-2106x core.

Each EPBx buffer has a read port and a write port, both of which can connect internally to either the EPD (External Port Data) bus or to a local bus which in turn can connect to the IOD (I/O Data) bus, PM Data bus, or DM Data bus. This is shown in Figure 8.1. Note that direct reads and writes bypass the EPBx buffers and go directly to internal memory.

8.4.1 Single-Word Transfers

When the host writes a single data word to the EPBx buffers, the ADSP-2106x core's program must read the data. Conversely, when the ADSP-2106x core writes a single piece of data to one of the EPBx buffers, the host must perform an external read cycle to obtain it. Because the EPBx buffers are six-deep FIFOs (in both directions), the host and ADSP-2106x core are allowed extra time to read the data—efficient, continuous, single-word transfers can thus be performed in real-time, with low latency and without using DMA.

If the host attempts a read from an empty EPBx buffer, the access will be held off with the ACK signal (for synchronous accesses) or with the REDY signal (for asynchronous accesses) until the buffer receives data from the ADSP-2106x core. If the ADSP-2106x core attempts to write to a full EPBx buffer, the access is also delayed and the core will hang until the buffer is read by the host. To prevent this from happening, the BHD (Buffer Hang Disable) bit should be set to 1 in the SYSCON register. The *full or empty* status of a particular EPBx buffer can be determined by reading the appropriate DMACx control/status register.

Similarly, if the host attempts a write to a full EPBx buffer, the access will be held off with ACK (or REDY) until the buffer is read by the ADSP-2106x core. If the core attempts to read from an empty buffer, the access is also held off and the core will hang until the buffer is written from the external host. The BHD bit can also be used to prevent a hang condition in this case.

Each EPBx buffer can be flushed (i.e. cleared) by writing a 1 to the FLSH bit in the corresponding DMACx control register. This bit is not latched internally and will always be read as a 0. Status can change in the following cycle. An EPBx buffer should not be enabled and flushed in the same cycle.

If packing and unpacking of individual data words is desired, the packing mode must be selected in the PMODE bits of the EPBx buffer control registers (DMAC6, DMAC7, DMAC8, and DMAC9). Either 16-to-32, 16-to-48, or 32-to-48 bit packing/unpacking can be selected. The external host bus width indicated by the host packing mode bits (HPM) in SYSCON must correspond to the external word width selected by the PMODE bits.

If any of the three packing modes are used for single-word transfers, the TRAN bit must also be appropriately set in the DMACx control register. Set TRAN=1 for host reads from the EPBx buffer, or set TRAN=0 for host writes to the EPBx buffer.

Note: To perform single-word, non-DMA transfers through the EPBx buffers, the DMA enable bit (DEN) must be cleared in the appropriate DMACx control register.

8.4.1.1 Interrupts For Single-Word Transfers

The interrupts for the four external port DMA channels can be used to control single-word data transfers between the host and the ADSP-2106x core. To do this, the DMACx control register must have the following bit settings: DEN=0 and INTIO=1. This disables DMA (DEN=0) and enables interrupt-driven I/O (INTIO=1). See the *DMA* chapter or *Control/Status Registers* appendix of this manual for a complete description of the DMACx control registers.

In this case the interrupt is generated whenever data becomes available in the read port of the EPBx buffer, or whenever the write port does not have new data to transmit. The EPBx buffer can then be read or written by either the ADSP-2106x core or by an external device such as the host. Generating interrupts in this fashion is useful for implementing interrupt-driven I/O controlled by the ADSP-2106x core processor.

This interrupt may be masked out (i.e. disabled) in the IMASK register. If the interrupt is later enabled in IMASK, the corresponding IRPTL latch bit must be cleared to clear any interrupt request that may have occurred.

8.4.2 DMA Transfers

The host processor can also set up DMA transfers to and from the ADSP-2106x. Once the host has gained control of the ADSP-2106x, it can access the on-chip DMA control and parameter registers to set up an external port DMA operation. This is the most efficient way to transfer blocks of data.

- **DMA Transfers to Internal Memory.** The host can set up external port DMA channels to transfer data to and from ADSP-2106x internal memory.

- **DMA Transfers to External Memory.** The host can set up an external port DMA channel to transfer data directly to external memory using the DMA request and grant lines ($\overline{\text{DMAR}}$x, $\overline{\text{DMAG}}$x).

Refer to the *DMA* chapter of this manual for details on setting up DMA operations.

8.4.2.1 DMA Transfers To Internal Memory

The host can set up external port DMA channels to transfer blocks of data to and from ADSP-2106x internal memory. To set up the DMA transfer, the host must initialize the ADSP-2106x's control and parameter registers for that channel. Once the DMA channel is set up, the host may simply read from (or write to) the corresponding EPBx buffer. If the buffer is empty (or full), the access is extended until data is available (or stored). This method allows fast and efficient data transfers.

If packing and unpacking of DMA data is desired, the packing mode must be selected in the PMODE bits of the external port DMA control registers (DMAC6, DMAC7, DMAC8, and DMAC9). Either 16-to-32, 16-to-48, or 32-to-48 bit packing / unpacking can be selected. The external host bus width indicated by the host packing mode bits (HPM) in SYSCON must correspond to the external word width selected by the PMODE bits.

The host may also use the $\overline{\text{DMAR}}$x / $\overline{\text{DMAG}}$x handshake signals for a DMA transfer, but not when $\overline{\text{HBR}}$ has been used to gain control of the bus.

8.4.2.2 DMA Transfers To External Memory

The ADSP-2106x's DMA controller can also be used to transfer data directly from the host to external memory. The *external handshake* mode for external port DMA channel 7 or 8 will provide the $\overline{\text{DMAR}}$x / $\overline{\text{DMAG}}$x handshaking for this type of transfer. Again, $\overline{\text{HBR}}$ should not be used to gain control of the bus. This type of transfer cannot be used if data packing is required, since the data does not pass through the ADSP-2106x.

8.5 DATA PACKING

The host interface has data packing logic to allow 16-bit or 32-bit external host bus words to be packed into 32-bit or 48-bit internal words. The packing logic is also fully reversible, so that 32-bit and 48-bit internal data can be unpacked into 16-bit and 32-bit external word widths. The SYSCON register is used to select the packing mode for synchronous and asynchronous transfers performed by the host.

8.5.1 Packing Control Bits In SYSCON

The SYSCON register bits for host packing control and memory width are shown in Table 8.3. After reset the SYSCON register is initialized to 0x0000 0010, causing the ADSP-2106x to assume a 16-bit bus for the host processor. Two 16-bit words must be written to SYSCON to change this selection (in the HPM bits), even if the host bus is 32 bits wide.

Bit(s) *Name*	*Definition*
IWT	Instruction Word Transfer (1=48-bit instruction, 0=32-bit data)
HPM*	Host Packing Mode (00=none, 01=16-to-32, 10=16-to-48, 11=32-to-48)
HMSWF	Host Packing Order – MSW First (1=MSW first, 0=LSW first)
HPFLSH	Host Packing Status Flush
IMDW0	Internal Memory Block 0 Data Width (0=32-bit data, 1=40-bit data)
IMDW1	Internal Memory Block 1 Data Width (0=32-bit data, 1=40-bit data)

Table 8.3 SYSCON Control Bits For Host Interface Packing

* If the host access is a read or write of the external port data buffers (EPB0, EPB1, EPB2, or EPB3), the external host bus width selected by HPM must correspond to the external word width selected in the PMODE bits of the DMACx control register (DMAC6, DMAC7, DMAC8, and DMAC9).

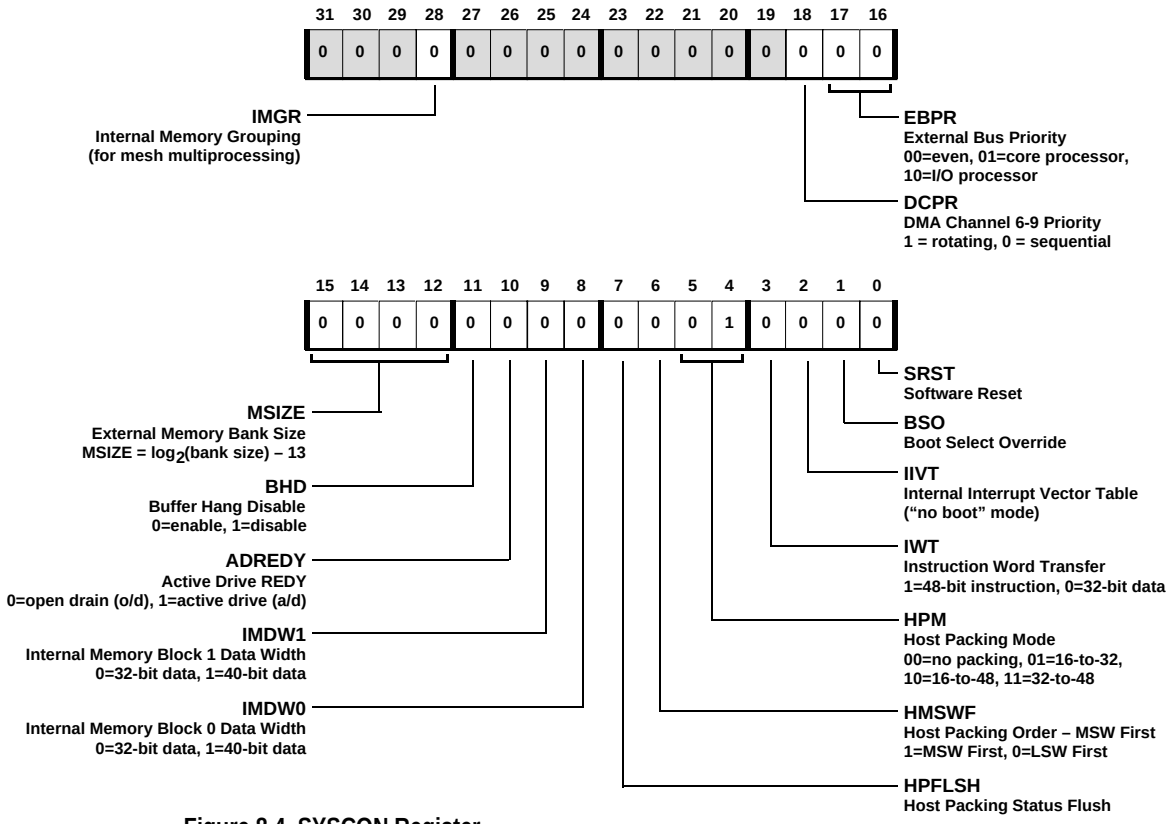


Figure 8.4 SYSCON Register

IWT **Instruction Word Transfer.** Specifies the word width for direct reads and direct writes of the ADSP-2106x's internal memory (by other ADSP-2106xs or by the host). IWT=1 overrides the IMDW bits (see below) and forces a 48-bit (3-column) memory transfer. IWT=0 defers to the data word setting of the IMDW bits in the SYSCON register. IWT should be set whenever the ADSP-2106x bus master or host processor is reading or writing instructions from (this) ADSP-2106x.

1 = 48-bit words for direct read/writes
0 = 32-bit words for direct read/writes

HPM **Host Packing Mode.** Specifies the internal word width and external host bus width for host processor accesses of the ADSP-2106x's internal memory or IOP registers. If the host access is a read or write of any IOP register other than the external port FIFO buffers (EPB0-EPB3) or link port buffers (LBUF0-LBUF5), the word width will always be 32 bits no matter what the host bus width is. If the host access is a read or write of the link port buffers, the word width is determined *only* by HPM, and not by the LEXT bit in LCTL.

00 = No packing. Maximum bus width is 32 bits for asynchronous transfers. The lower 16 bits of the 48-bit data bus will be written and read as zeros, even when reading 48-bit words. For synchronous transfers, the host bus should be 32 bits wide for data transfers or 48 bits wide for instruction word transfers. (Note: To read and write 48-bit words from internal memory, the IWT bit must be set to 1 or the IMDW bit for the block of memory being accessed must be set to 1.)

Default at Reset → **01 = 16-bit host bus, 32-bit words.** The host bus will be 16 bits wide; any memory access will be 32-bit words. (Note: If the memory access is to a block of ADSP-2106x internal memory for which the IMDW bit is set to 1, the access will read or write the upper 32 bits of the 48-bit word.)

10 = 16-bit host bus, 48-bit words. The host bus will be 16 bits wide; any memory access will be 48-bit words.

11 = 32-bit host bus, 48-bit words. The host bus will be 32 bits wide; any memory access will be 48-bit words.

HMSWF **Host Packing Order.** Specifies the order in which host-accessed words are packed, for 16-to-32 bit packing and 16-to-48 bit packing. HMSWF is ignored for 32-to-48 bit packing. When HMSWF=1, packing is done MSW first (most significant 16-bit word first). When HMSWF=0, packing is is done LSW first.

1=MSW first
0=LSW first

HPFLSH **Host Packing Status Flush.** Resets the host packing status. Host accesses must not occur while the HPFLSH bit is being written by the ADSP-2106x processor core. There is a two cycle latency before the reset takes effect, after which the host may resume normal operations. (Note: HPFLSH is always read as a zero.)

1=Flush packing status

IMDWx **Internal Memory Block Data Width.** Selects the data word width for each block of internal memory. For 32-bit data words, set IMDWx to 0. For 40-bit data (transferred within 48-bit words), set IMDWx to 1. IMDW0 (bit 8 of SYSCON) selects the data word width for memory block 0 and IMDW1 (bit 9) selects the data word width for memory block 1. (**Note:** 48-bit instructions can be stored in a memory block regardless of the setting of the IMDW bit. See "Configuring Memory For 32-Bit or 40-Bit Data" in the *Memory* chapter of this manual for more information.)

0=32-bit data
1=40-bit data

In addition to the HPM bits, the packing mode is also affected by the setting of the PMODE bits in the DMACx control register of each external port buffer (DMAC6, DMAC7, DMAC8, and DMAC9, which correspond to the EPB0, EPB1, EPB2, and EPB3 buffers):

PMODE	*Packing Mode*
00	No packing / unpacking
01	16-bit external bus to / from 32-bit internal packing
10	16-bit external bus to / from 48-bit internal packing
11	32-bit external bus to / from 48-bit internal packing

HPM and PMODE must select the same external bus width for host data transfers to and from the ADSP-2106x!!

For example, if HPM=11 for a 32-bit host bus, then PMODE must also be set to 11. If HPM equals 01 or 10 for a 16-bit bus, then PMODE can be either 01 or 10.

If any of the three packing modes are used for non-DMA, single-word transfers to or from an EPBx buffer, the TRAN bit must also be appropriately set in the DMACx control register. Set TRAN=1 for host reads from the EPBx buffer, or set TRAN=0 for host writes to the EPBx buffer.

To change the host packing mode, the following sequence must occur:

1. Write to the SYSCON register, changing the value of HPM.
2. Read SYSCON to ensure that the write was completed.
3. Repeat the write to SYSCON (to flush the read, since it may have occurred in the old packing mode).
4. Wait 4 cycles.

During packed transfers with a slow host, the host may relinquish the bus before the current word has been fully packed—the bus may be released after the first part of the word is written and then, some time later, $\overline{\text{HBR}}$ reasserted and the second part of the word written. This could allow, for example, another ADSP-2106x to write to this one without affecting the host packing operation.

8.5.2 Data Bus Lines Used For Different Packing Modes

Table 8.4 shows which data bus lines are used for different host bus widths and packing modes. If the host bus width is 32 bits and no packing is selected for an asynchronous access, the ADSP-2106x will ignore the lower 16 bits of the 48-bit external data bus when inputting data—*note that this only true for asynchronous accesses, not for synchronous accesses.* When outputting data onto a 32-bit host bus with 48-to-32 bit unpacking, the ADSP-2106x drives the lower 16 bits as zeroes; this is true for both asynchronous and synchronous accesses. When outputting data with no packing, the ADSP-2106x drives the lower 16 bits with whatever data is in the corresponding memory bits; this applies for both asynchronous and synchronous accesses.

If the host bus width is 16 bits and 16-to-32 or 16-to-48 bit packing is selected, the ADSP-2106x will ignore the upper and lower 16 bits of the 48-bit external data bus when inputting data (for both asynchronous and synchronous accesses). When outputting data, these bits will be driven as zeroes.

*Host Packing Mode**	*HPM Bits In SYSCON*	*Effect*
No packing	00	**Data In:** ADSP-2106x ignores lower 16 bits of external bus ($DATA_{15-0}$) for a 32-bit host (*asynchronous accesses only*). **Data Out:** ADSP-2106x outputs whatever is in memory onto the external bus ($DATA_{47-0}$).
16-to-32	01	**Data In:** ADSP-2106x ignores upper and lower 16 bits of external bus. **Data Out:** ADSP-2106x outputs zeroes on upper and lower 16 bits of external bus.
16-to-48	10	Same as 16-to-32 packing.
32-to-48	11	**Data In:** ADSP-2106x ignores lower 16 bits of external bus ($DATA_{15-0}$). **Data Out:** ADSP-2106x outputs zeroes on lower 16 bits of external bus.

Table 8.4 Data Bus Lines Used For Different Host Packing Modes

* 16-to-32 packing: 16-bit host bus, 32-bit memory words on ADSP-2106x.
16-to-48 packing: 16-bit host bus, 48-bit memory words on ADSP-2106x.
32-to-48 packing: 32-bit host bus, 48-bit memory words on ADSP-2106x.

Figure 8.a shows how different data word sizes are transferred over the external port.

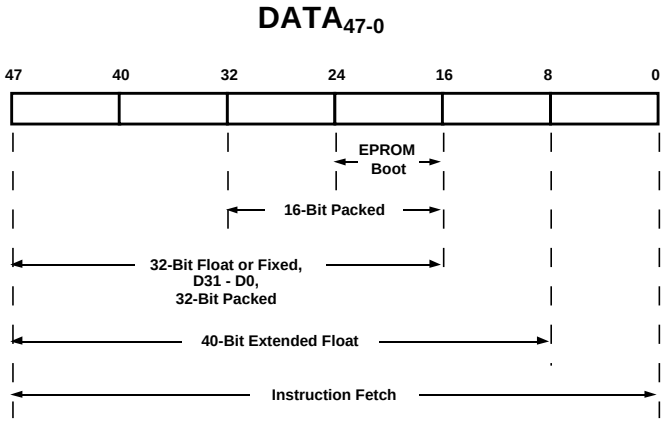


Figure 8.a External Port Data Alignment

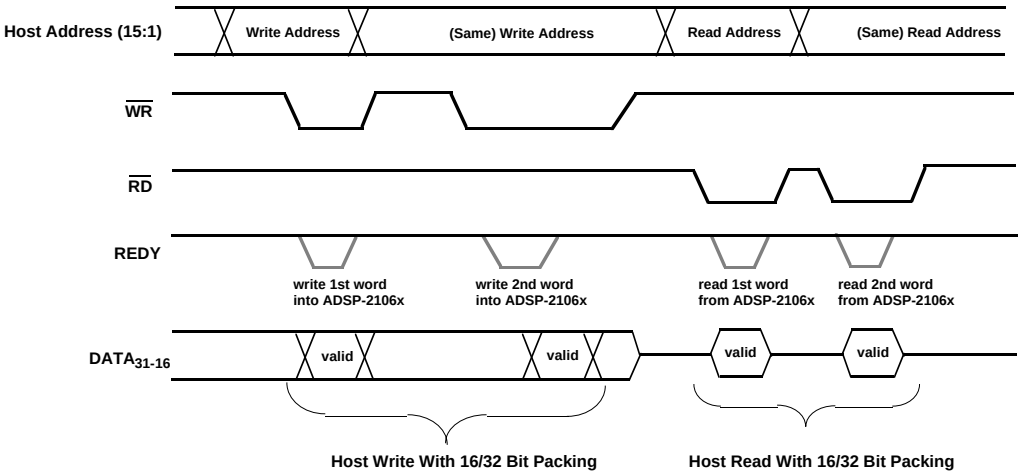
8.5.3 32-Bit Data Packing

For a 16-bit host bus, the incoming data is latched on $DATA_{31-16}$. Similarly, outgoing data is driven on $DATA_{31-16}$ with the other lines equal to zeroes. The sequence of events for 32-bit packing/unpacking is different for writes and reads, as described below. For a 16-bit host bus, the endian format of the transfers is controlled by the HMSWF bit in the SYSCON register. If HMSWF=0, the least significant 16-bit word will be packed first. If HMSWF=1, the most significant 16-bit word will be packed first.

When a host reads a 32-bit word with 16-bit unpacking, using the typical bus interface hardware shown in Figure 8.8 (at the end of this chapter), the following sequence of events occurs (as illustrated in Figure 8.5):

- The host initiates a read cycle by driving an address, asserting $\overline{CS}$ if the access is asynchronous, and asserting $\overline{RD}$ (low).
- The selected ADSP-2106x deasserts REDY, latches the address, and performs an internal direct read to get the data.
- When the ADSP-2106x has the data, it asserts REDY and drives the 1st 16-bit word.
- The host latches the data and deasserts $\overline{RD}$ (high).

16/32 Bit Packing



32/48 Bit Packing

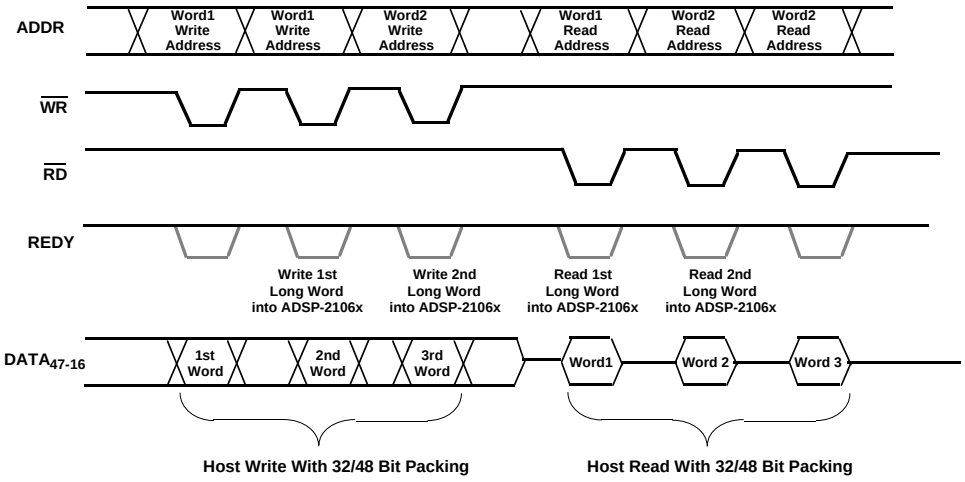


Figure 8.5 Example Timing For Host Interface Data Packing

- The host initiates another read access, driving the address of the data to be accessed and then asserting $\overline{RD}$.
- The ADSP-2106x transmits the 2nd 16-bit word.

When a host writes a 32-bit word with 16-bit packing, again using the typical bus interface hardware shown in Figure 8.8, the following sequence of events occurs (also illustrated in Figure 8.5):

- The host initiates a write cycle by driving the write address, asserting $\overline{CS}$ if the access is asynchronous, and asserting $\overline{WR}$ (low).
- The ADSP-2106x asserts REDY when it is ready to accept data.
- The host drives the address and the 1st 16-bit word, and deasserts $\overline{WR}$ (high).
- The ADSP-2106x latches the 1st 16-bit word.
- The host again drives the address and initiates another write cycle for the 2nd 16-bit word, by asserting $\overline{WR}$.
- When the ADSP-2106x has accepted the 2nd word it performs an internal direct write to its memory (or memory-mapped IOP register). If the ADSP-2106x's internal write has not completed by the time another host access occurs, that access will be held off with REDY.

If the ADSP-2106x is waiting for another 16-bit word from the host to complete the packed word, the HPS bits in the SYSTAT register will be non-zero. (See "SYSTAT Register Status Bits.") Because there is only one packing buffer for the host interface, the host must fully complete each packed read or write before another is begun.

8.5.4 48-Bit Instruction Packing

The host can also download and upload 48-bit instructions over its 16- or 32-bit bus. The packing sequence for downloading ADSP-2106x instructions from a 32-bit host bus (HPM=11) takes 3 cycles for every 2 words, as illustrated below. 32-bit data is transferred on data bus lines 47-16 ($DATA_{47-16}$). If an odd number of instruction words are transferred, the packing buffer must be flushed by a dummy access to remove the unused word.

32-Bit to 48-Bit Word Packing (Host Bus ↔ ADSP-2106x):

	Data Bus Lines 47-32	*Data Bus Lines 31-16*
1st transfer	Word1, bits 47-32	Word1, bits 31-16
2nd transfer	Word2, bits 15-0	Word1, bits 15-0
3rd transfer	Word2, bits 47-32	Word2, bits 31-16

The HMSWF bit of SYSCON is ignored for 32-to-48-bit packing.

The packing sequence for downloading or uploading ADSP-2106x instructions over a 16-bit host bus takes 3 cycles for every word, as shown below. The HMSWF bit in SYSCON determines whether the most significant 16-bit word or least significant 16-bit word is packed first.

16-Bit to 48-Bit Word Packing w/HMSWF=1 (Host Bus ↔ ADSP-2106x):

	Data Bus Pins 31-16
1st transfer	Word1 bits 47-32
2nd transfer	Word1 bits 31-16
3rd transfer	Word1 bits 15-0

40-bit extended precision data may be transferred using the 48-bit packing mode. Refer to the *Memory* chapter of this manual for a discussion of memory allocation for the different word widths.

8.6 SYSTAT REGISTER STATUS BITS

The SYSTAT register provides status information, primarily for multiprocessor systems. Table 8.5 shows the status bits in this register.

Bit(s) Name	*Definition*
HSTM	Host Mastership
BSYN	Bus Synchronization
CRBM	Current Bus Master (ID_{2-0} of ADSP-2106x bus master)
IDC	ID Code (ID_{2-0} of this ADSP-2106x)
DWPD	Direct Write Pending
VIPD	Vector Interrupt Pending
HPS	Host Packing Status

Table 8.5 SYSTAT Status Bits

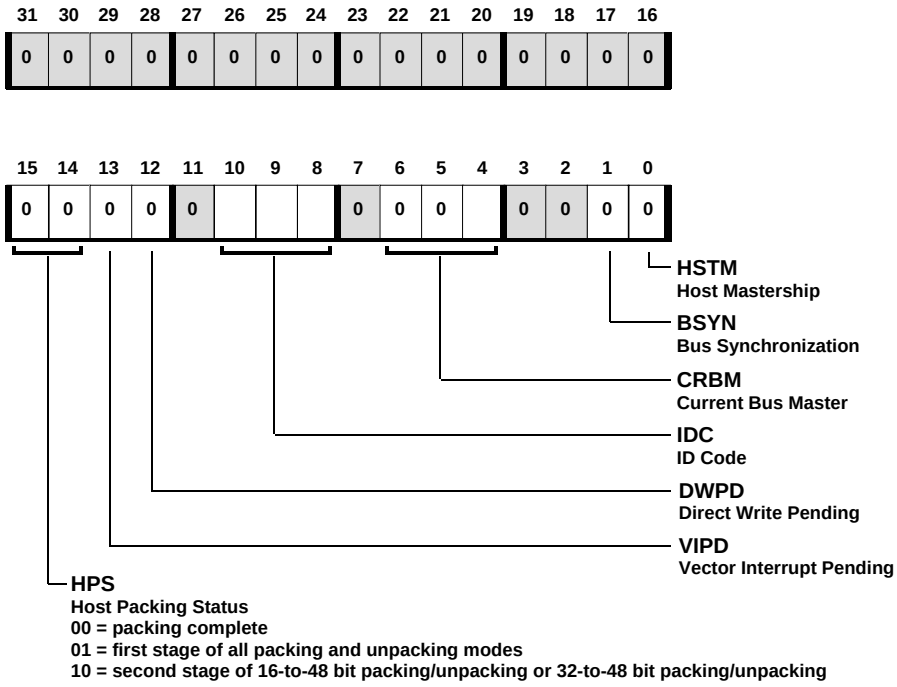


Figure 8.6 SYSTAT Register

HSTM **Host Mastership.** Indicates whether the host processor has been granted control of the bus.

1 = Host is bus master
0 = Host is not bus master

BSYN **Bus Synchronization.** Indicates when the ADSP-2106x's bus arbitration logic is synchronized after reset. (See "Bus Synchronization After Reset" in the *Multiprocessing* chapter of this manual for more information.)

1 = Bus arbitration logic is synchronized
0 = Bus arbitration logic is not synchronized

CRBM **Current Bus Master.** Indicates the ID code of the ADSP-2106x that is the current bus master. If CRBM is equal to the ID of this ADSP-2106x then it is the current bus master. CRBM is only valid for ID_{2-0} > (greater than zero). When ID_{2-0}=000, CRBM is always 1.

IDC **ID Code.** Indicates the ID_{2-0} inputs of this ADSP-2106x.

DWPD **Direct Write Pending.** Indicates when a direct write to the ADSP-2106x's internal memory is pending. The DWPD bit is cleared when the direct write has been completed. (Direct writes may be delayed for several cycles if DMA chaining is underway or if higher priority DMA requests occur. Maximum delay is 12 cycles.)

1 = Direct write pending
0 = No direct write pending

VIPD **Vector Interrupt Pending.** Indicates that a pending vector interrupt has not yet been serviced. The VIPD bit is set when the VIRPT register is written to and is cleared upon return from the interrupt service routine. The host processor (or other ADSP-2106x) that issued the vector interrupt should monitor this bit to determine when the service routine has been completed (and when a new vector interrupt may be issued).

1 = Vector interrupt pending
0 = No vector interrupt pending

HPS **Host Packing Status.** Indicates when host word packing is completed or, if not, what stage of the process is taking place. (See "Host Packing" for more information.)

00 = Packing complete
01 = 1st stage of all packing and unpacking modes.
10 = 2nd stage of 16-to-48 bit packing/unpacking or 32-to-48 bit packing/unpacking

8.7 INTERPROCESSOR MESSAGES & VECTOR INTERRUPTS

Once it has requested and been granted control of the ADSP-2106x, the host processor communicates with the ADSP-2106x by writing messages to the memory-mapped IOP registers. Asynchronous writes are the easiest way for the host to do this. In a multiprocessor system, the host can access the internal memory and IOP registers of every ADSP-2106x.

The MSGR0-MSGR7 registers are general-purpose registers that can be used for convenient message passing between the host and ADSP-2106x. They are also useful for semaphores and resource sharing between multiple ADSP-2106xs. The MSGRx and VIRPT registers can be used for message passing in the following ways:

- **Message Passing.** The host can use any of the 8 message registers, MSGR0–MSGR7, to communicate with the ADSP-2106x.

- **Vector Interrupts.** The host can issue a vector interrupt to the ADSP-2106x by writing the address of an interrupt service routine to the VIRPT register. This causes an immediate high-priority interrupt on the ADSP-2106x which, when serviced, will cause the ADSP-2106x to branch to the specified service routine.

The MSGRx and VIRPT registers also support shared-bus multiprocessing via the external port. Since these registers may be shared resources within a single ADSP-2106x, conflicts may occur—your system software must prevent this. For further discussion of IOP register access conflicts, refer to the *Control/Status Registers* appendix of this manual.

8.7.1 Message Passing (MSGRx)

Three possible software protocols by which a host can communicate with the ADSP-2106x through the MSGRx message registers are: 1) *vector-interrupt-driven*, 2) *register handshake*, and 3) *register write-back*.

For the *vector-interrupt-driven* method, the host fills predetermined MSGRx registers with data and triggers a vector interrupt by writing the address of the service routine to VIRPT. The service routine should read the data from the MSGRx registers and then write "0" into VIRPT to tell the host it is done. The service routine could also use one of the ADSP-2106x's $FLAG_{3-0}$ pins to tell the host it has finished.

For the *register handshake* method, four of the MSGRx registers should be designated as follows: a receive register (R), a receive handshake register (RH), a transmit register (T), and a transmit handshake register (TH). To pass data to the ADSP-2106x, the host would write data into T and then write a "1" into TH. When the ADSP-2106x sees a "1" in TH, it reads the data from T and then writes back a "0" into TH. When the host sees a "0" in TH, it knows that the transfer is complete. A similar sequence of events occurs when the ADSP-2106x passes data to the host through R and RH.

The *register write-back* method is similar to register handshaking, but uses only the T and R data registers. The host writes data to T. When the ADSP-2106x sees a non-zero value in T, it retrieves it and writes back a "0" to T. A similar sequence occurs when the host is receiving data. This simpler method works well as long as the data to be passed does not include "0."

8.7.2 Host Vector Interrupts (VIRPT)

Vector interrupts are used for interprocessor commands between the host and an ADSP-2106x or between two ADSP-2106xs. When the external processor writes an address to the ADSP-2106x's VIRPT register a vector interrupt is caused.

When the vector interrupt is serviced, the ADSP-2106x automatically pushes the status stack and begins executing the service routine located at the address specified in VIRPT. The lower 24 bits of VIRPT contain the address; the upper 8 bits may be optionally used as data to be read by the interrupt service routine. At reset, VIRPT is initialized to its standard address in the ADSP-2106x's interrupt vector table.

The minimum latency for vector interrupts is six cycles, five of which are NOPs. When the RTI (return from interrupt) instruction is reached in the service routine, the ADSP-2106x automatically pops the status stack.

The VIPD bit in the SYSTAT register reflects the status of the VIRPT register. If VIRPT is written while a previous vector interrupt is pending, the new vector address replaces the pending one. If VIRPT is written while a previous vector interrupt is being serviced, the new vector address is ignored and no new interrupt is generated. If the ADSP-2106x writes to its own VIRPT register, no interrupt is generated.

To use the ADSP-2106x's vector interrupt feature, the host could perform the following sequence of actions:

1. Poll the VIRPT register until it reads a certain token value (i.e. zero).
2. Write the vector interrupt service routine address to VIRPT.
3. When the service routine is finished, the ADSP-2106x should write the token back into VIRPT to indicate that it is finished and that another vector interrupt can be initiated.

The DWPD (Direct Write Pending) bit of the SYSTAT register indicates when a direct write to internal memory is pending. Direct writes and IOP register accesses may be completed in different sequences. If, for example, the host performs a direct memory write to an ADSP-2106x and then writes to an IOP register on the ADSP-2106x, the IOP register

write may complete before the direct write. Because of this, direct writes performed just before vector interrupt writes (to VIRPT) may be delayed until after the branch to the interrupt vector:

1. The host processor performs a direct write to the internal memory of an ADSP-2106x.
2. The host processor writes to the VIRPT register of the ADSP-2106x to initiate a vector interrupt. This causes the direct write to be delayed.
4. The ADSP-2106x jumps to the vector interrupt service routine.
5. The direct write is completed after the interrupt service routine is underway.

To prevent this from happening, the host should check that all direct writes have completed before writing to the ADSP-2106x's VIRPT register. This can be done by polling the ADSP-2106x's DWPD bit (in SYSTAT) after performing a direct write, waiting for it to become cleared, and then proceeding with the write to VIRPT.

8.8 SYSTEM BUS INTERFACING

An ADSP-2106x subsystem, consisting of several ADSP-2106xs with local memory, may be viewed as one of several processing elements connected together by a system bus. Examples of such systems are the EISA bus, PCI bus, or even several ADSP-2106x subarrays. The processing elements in such a system arbitrate for the system bus via an arbitration unit. Each device on the bus that wishes to become a bus master must be able to drive a bus request signal and respond to a bus grant signal. The arbitration unit determines which request it will grant in any given cycle.

8.8.1 Access To The ADSP-2106x Bus—Slave ADSP-2106x

Figure 8.7 shows an example of a basic interface to a system bus which isolates the local ADSP-2106x bus from the system bus. When the system is not accessing the ADSP-2106xs, the local bus supports transfers between other local ADSP-2106xs and / or local external memory or devices.

When the system wishes to access an ADSP-2106x, it executes a read or write to the address range of the ADSP-2106x subsystem. The external address comparator detects a local access and asserts $\overline{\text{HBR}}$ and one of the appropriate $\overline{\text{CS}}$ lines. The system bus is held off by REDY until the ADSP-2106x is ready to accept the data. The $\overline{\text{HBG}}$ signal enables the system bus buffers. The buffers' direction for data is controlled by the

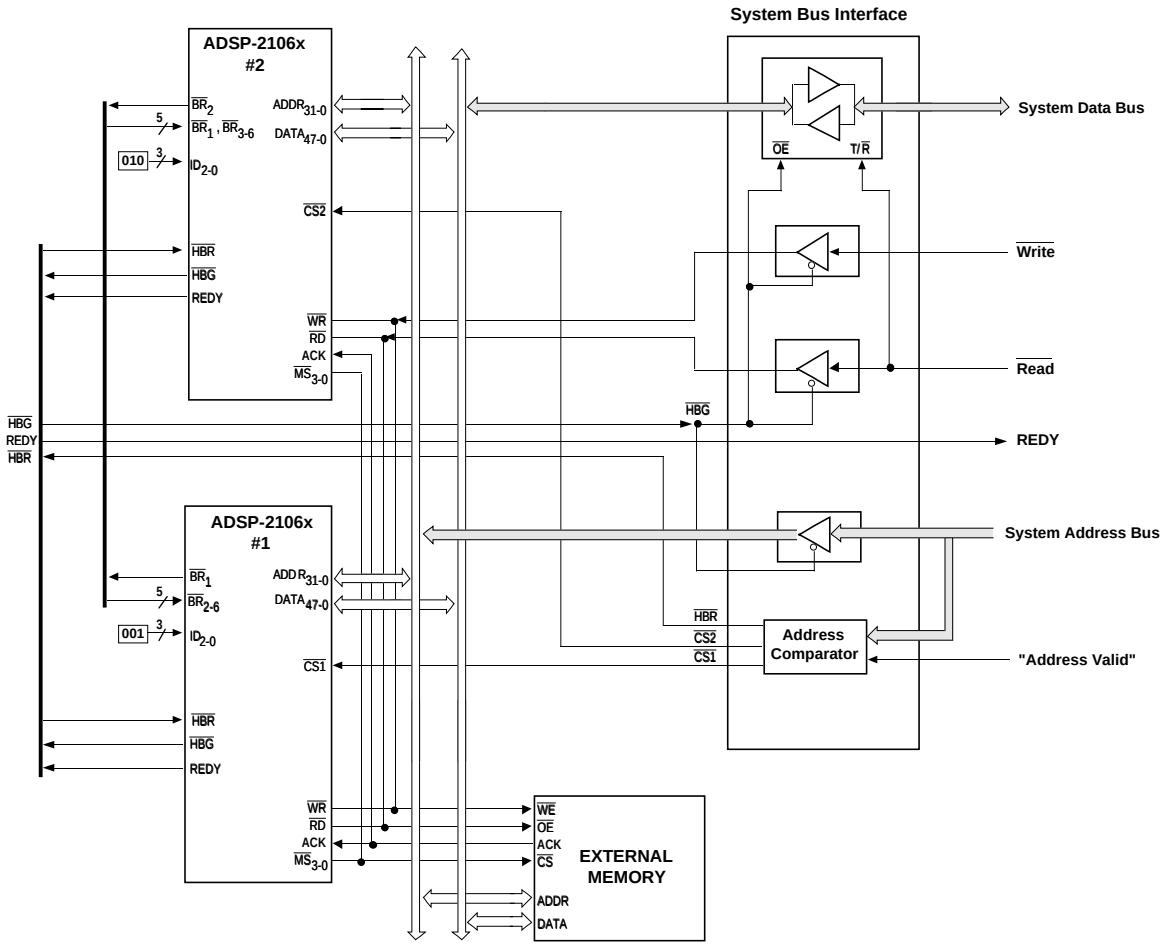


Figure 8.7 Basic System Bus Interface

read or write signals. To avoid glitching the $\overline{HBR}$ line when addresses are changing, the address comparator may be qualified by an address latch enable signal from the system or by the system read or write signals. These methods cause $\overline{HBR}$ to be deasserted each time system read or write is deasserted or the address is changed. Because these techniques deassert $\overline{HBR}$ with each access, the overhead of an HTC occurs as part of each access. One can avoid this type of overhead by latching $\overline{HBG}$ during long sequences of bus accesses.

8.8.2 Access To The System Bus—Master ADSP-2106x

Figure 8.8 shows a more complex, bidirectional system interface in which the ADSP-2106x subsystem can access the system bus by becoming a bus master. Before it begins the access, the ADSP-2106x must first request permission to become the bus master by generating the System Bus Request signal ($\overline{SBR}$). The system bus arbitration unit determines when to respond with the System Bus Grant signal ($\overline{SBG}$). In this system, each system bus master generates and responds to its own unique pair of signals.

The method an ADSP-2106x uses to arbitrate for the system bus depends on whether the access is from the ADSP-2106x processor core or from its DMA controller. These two methods, which are quite different, are described below under "Core Processor Access To System Bus" and "ADSP-2106x DMA Access To System Bus."

8.8.2.1 Core Processor Access To System Bus

The ADSP-2106x core may arbitrate for the system bus by setting a flag and waiting for $\overline{SBG}$ via another flag. This has the benefit of not tying up the local bus while waiting. If $\overline{SBG}$ is tied to an interrupt pin, then useful work can continue while waiting.

Another method is to attempt the access assuming that the system bus is available, and then either wait or abort the access if it is not available. The ADSP-2106x begins the access to the system bus by asserting one of the memory select lines, $\overline{MS3}$. This also asserts $\overline{SBR}$. If the system bus is not available, i.e. $\overline{SBG}$ is deasserted, the ADSP-2106x should be held off with ACK. This approach is simple but ties up the ADSP-2106x and the local bus whenever the system bus is accessed while it is busy. To overcome this, the Type 10 instruction

```
IF condition  JUMP(addr), ELSE compute, DM(addr)=dreg;
```

can be used. This instruction aborts the bus access if the condition ($\overline{SBG}$) is not true, and causes a branch to a "try again later" routine. This method works well if $\overline{SBG}$ is asserted most of the time. If the Type 10 instruction is not used, a deadlock condition can result if an access is attempted before the bus is granted, as described in the next section.

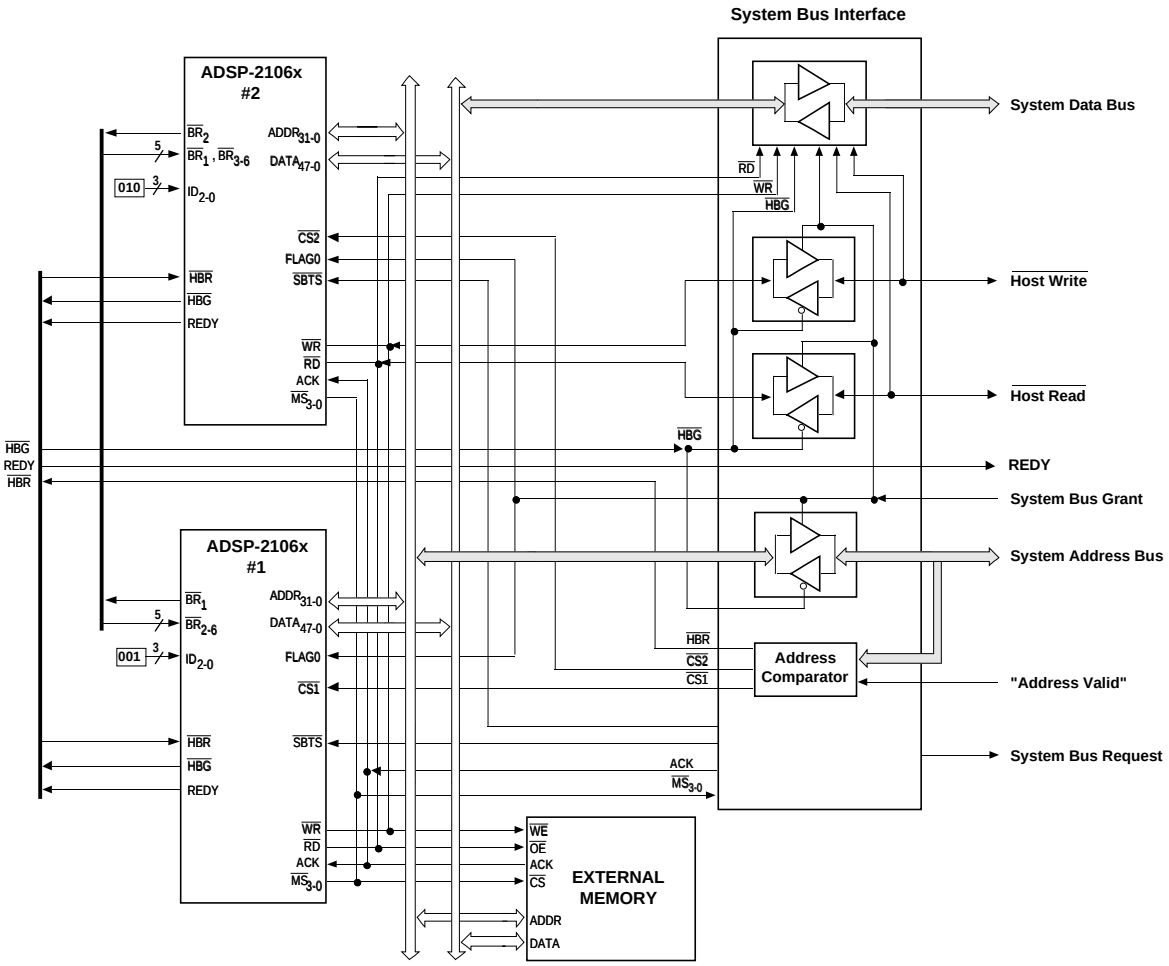


Figure 8.8 Bidirectional System Bus Interface

Note: The memory controller for shared external memory must generate wait states and REDY for host accesses to the memory.

8.8.2.2 Deadlock Resolution

In the rare case where both the ADSP-2106x subsystem and the system are trying to access each other's bus in the same cycle, a deadlock may occur in which neither access can complete; ACK stays deasserted.

Normally the master ADSP-2106x will respond to an $\overline{\text{HBR}}$ request by asserting $\overline{\text{HBG}}$ after the completion of the current access. If the ADSP-2106x is accessing the system bus at the same time, however, $\overline{\text{HBG}}$ will not be asserted because this current access cannot complete—this results in a deadlock in which neither access can complete. The deadlock may be broken by asserting the $\overline{\text{SBTS}}$ input for one or more cycles once the deadlock is detected (i.e. when the system bus to local bus buffer is enabled from both sides). $\overline{\text{SBTS}}$ is the Suspend Bus Tristate pin of the ADSP-2106x.

The combination of $\overline{\text{SBTS}}$ and $\overline{\text{HBR}}$ puts the master ADSP-2106x into slave mode, just like a normal $\overline{\text{HBR}}$ assertion, and suspends the ADSP-2106x core's external access. This allows the system access to the local bus to proceed, once the ADSP-2106x asserts $\overline{\text{HBG}}$. The combination of $\overline{\text{HBR}}$ and $\overline{\text{SBTS}}$ should only be applied when there is a deadlock caused by an ADSP-2106x access to the system bus. It should not be used when there is a local bus transfer because the $\overline{\text{WR}}$ signal will be asserted twice, once before the SBTS is asserted and once after the access resumes. For SHARC-to-SHARC transfers on the local bus, this will violate the slave timing requirements.

The following sequence of actions allows the host processor to suspend an ongoing ADSP-2106x access and gain access to its internal resources, provided that: 1) the access originates from the ADSP-2106x's core, not the DMA controller, 2) a DRAM PAGE miss is not detected for that memory access, and 3) bus lock is not enabled.

1. After $\overline{\text{HBR}}$ is asserted, the host asserts $\overline{\text{SBTS}}$ for one or more cycles. If $\overline{\text{SBTS}}$ is asserted one or more cycles after $\overline{\text{HBR}}$ is recognized, $\overline{\text{HBG}}$ is guaranteed to be asserted in the next cycle. $\overline{\text{SBTS}}$ should be deasserted before $\overline{\text{HBR}}$ is deasserted.

2. The host drives both $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes to their correct value (within the setup time specified in the data sheet) after $\overline{\text{HBG}}$ is asserted. The host may then perform as many accesses as desired.

3. The host has full control of the bus and may access any of the ADSP-2106xs or peripherals on the bus.

4. The host deasserts $\overline{\text{HBR}}$. $\overline{\text{HBG}}$ will be deasserted when the internal read buffer is empty.

5. One cycle after the ADSP-2106x deasserts $\overline{\text{HBG}}$, the ADSP-2106x restarts its suspended access.

8.8.2.3 ADSP-2106x DMA Access To System Bus

The use of the $\overline{\text{SBTS}}$ and $\overline{\text{HBR}}$ inputs to resolve a system bus deadlock, as described above, cannot be used for DMA transfers because once a DMA word transfer has begun in the ADSP-2106x, it must be completed (i.e. it must receive the ACK signal). If $\overline{\text{SBTS}}$ and $\overline{\text{HBR}}$ are asserted during a DMA access, the $\overline{\text{HBG}}$ pin will not be asserted until the access cycle has completed. If the single DMA access is not allowed to complete, a deadlock condition may result.

To prevent system bus deadlock when using DMA, you must ensure that $\overline{\text{SBG}}$ has been asserted before the DMA sequence begins. If a higher priority access is needed, the DMA sequence may be held off (by asserting $\overline{\text{HBR}}$) at any time after a word has been transferred. You must make sure that $\overline{\text{SBG}}$ is asserted before $\overline{\text{HBR}}$ is deasserted to prevent the possibility of another deadlock occurring. When the DMA sequence is complete, the DMA interrupt service routine should clear the external $\overline{\text{SBR}}$ flag.

Because the system bus is likely to be considerably slower than the ADSP-2106x local bus, performance on the local bus may be considerably improved by using handshake mode DMA. In this case, the $\overline{\text{SBG}}$ signal is tied to the DMA request line, $\overline{\text{DMAR}}$x. Thus the local bus and system bus access will only be initiated when the system bus is available.

The use of a FIFO in the system interface unit, to allow DMA data from the local bus to be posted, may also increase performance on the local bus when using a slow system bus.

8.8.3 Multiprocessing With Local Memory

Figure 8.9 shows how several ADSP-2106x subsystems may be connected together on a system bus for high throughput. The gate array implements bus arbitration when the system bus is accessed. The buffers isolate the ADSP-2106x local buses from the system bus.

This example system works in the following way:

- An ADSP-2106x requests the system bus with $\overline{\text{SBR}}$ when it asserts the $\overline{\text{MS3}}$ line (for example). The gate array arbitrates between the $\overline{\text{SBR}}$ lines and then enables the highest priority group by asserting $\overline{\text{SBG}}$, which is tied to ACK.

- The master ADSP-2106x may connect to system memory or to other ADSP-2106x groups. When the bus buffer is enabled, the read or write strobe enables should be asserted with a delay to allow the address to stabilize.

- To access an ADSP-2106x slave in another group, the master ADSP-2106x addresses that group's multiprocessor memory space. The gate array detects group multiprocessor memory space from three high-order address bits and asserts the $\overline{\text{HBR}}$ line for the selected group. When $\overline{\text{HBG}}$ is asserted, the gate array enables the slave's bus buffer. The high-order group address bits are cleared by the buffer to allow the group to decode the E, M, and S address fields as local multiprocessor memory space. The access will be synchronous because the $\overline{\text{CS}}$ line is not asserted. The single wait state option for the bus should be enabled.

- If two groups access each other in the same cycle, a deadlock may occur. The $\overline{\text{SBTS}}$ pin may be used to clear the deadlock.

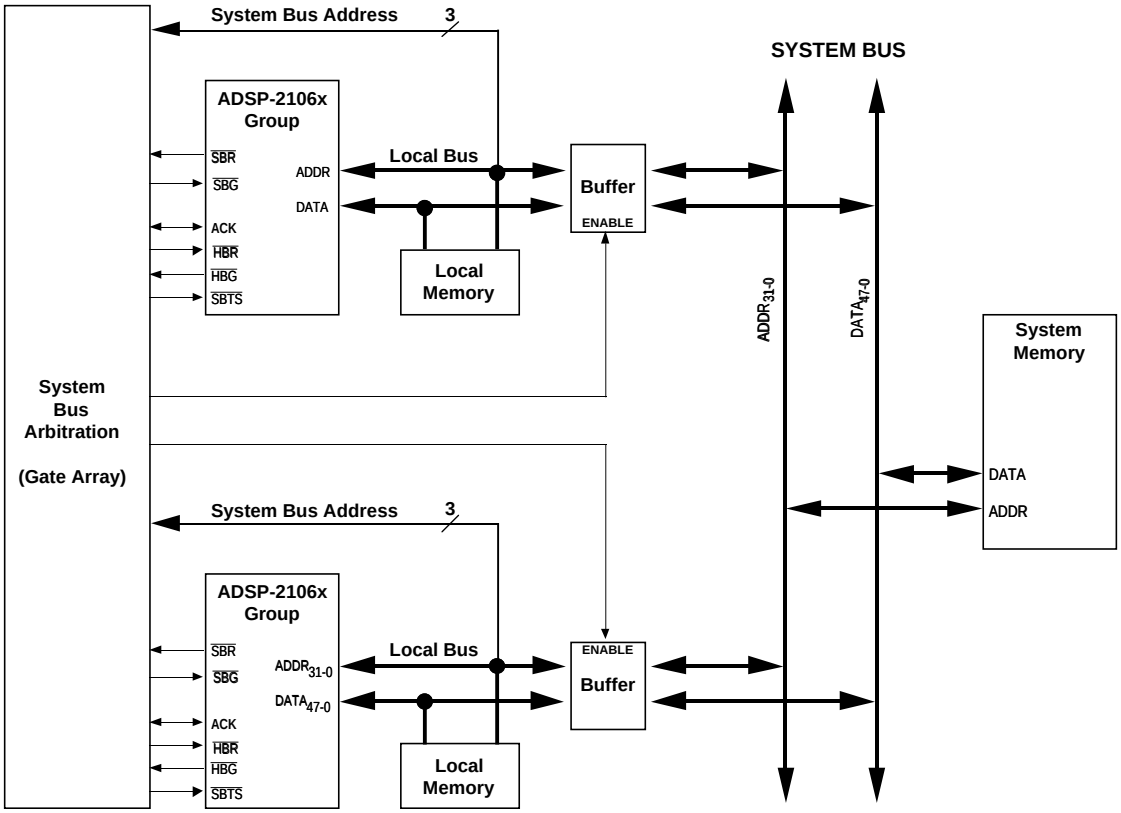


Figure 8.9 ADSP-2106x Subsystems On A System Bus

8.8.4 ADSP-2106x To Microprocessor Interface

An ADSP-2106x without external memory may connect more or less directly to a host microprocessor's bus, and the interface may not require any buffers. This type of connection assumes that the ADSP-2106x can execute its application from internal memory most of the time and only occasionally needs to request an external access. The host microprocessor should always keep the $\overline{HBR}$ request asserted unless it sees $\overline{BR1}$ asserted (i.e. the $\overline{BR}$x line of the ADSP-2106x with ID=001). It can then deassert $\overline{HBR}$ to allow the ADSP-2106x to perform an external access when the host is ready to give up its bus. Most of the time, however, the host can read or write to the ADSP-2106x at will. The host accesses the ADSP-2106x by asserting $\overline{CS}$ and handshaking with REDY. $\overline{HBG}$ need not be used in this scenario.

8 Host Interface

Link Ports 9

9.1 OVERVIEW

The ADSP-2106x SHARC provides additional I/O capability through six dedicated 4-bit link ports. Each link port consists of four bidirectional data lines, a bidirectional clock line, and a bidirectional acknowledge line. The link ports can be clocked twice per processor clock cycle, allowing each port to transfer up to 8 bits of data per cycle. Link port I/O allows a variety of interconnection schemes to I/O peripheral devices as well as coprocessing and multiprocessing schemes. Using link port I/O, it is also possible to configure multidimensional, multiprocessor arrays.

➠ **Note that the ADSP-21061 processor does not have link ports; the discussion in this chapter does not apply to the ADSP-21061.**

Link port features and functions include:

- Link ports can operate independently and simultaneously.
- Link port data is packed into 32-bit or 48-bit words, and can be directly read by the ADSP-2106x core processor or DMA-transferred to on-chip memory.
- Link port data can also be accessed by the external host processor, using direct reads and writes.
- Double-buffered transmit and receive data registers.
- Clock/acknowledge handshaking controls link port transfers which are programmable as either transmit or receive with each link port supported by a separate DMA channel.
- Link ports provide high-speed, point-to-point data transfers to other ADSP-2106x processors. This allows a variety of interconnection schemes between multiple ADSP-2106x processors and external devices, including 1-, 2- and 3-dimensional arrays.

9 Link Ports

The six pins associated with each link port are listed in Table 9.1. Each link port consists of four bidirectional data lines, $LxDAT_{3-0}$, and two handshake lines, Link Clock (LxCLK) and Link Acknowledge (LxACK). The LxCLK line allows asynchronous data transfers and the LxACK line provides handshaking. When configured as a transmitter, the port drives both the data and LxCLK lines. When configured as a receiver, the port drives the LxACK line.

Pin(s)	*Function*
$LxDAT_{3-0}$	Link Port x Data
LxCLK	Link Port x Clock
LxACK	Link Port x Acknowledge

Table 9.1 Link Port Pins
"x" denotes the link port number, 0-5.

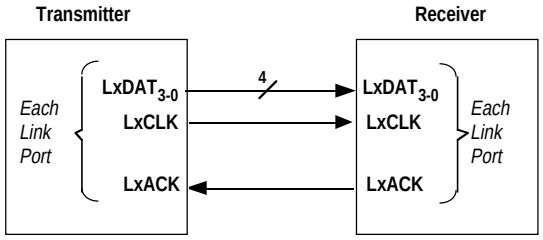


Figure 9.a Link Port Pin Connections

Figure 9.b shows examples of different link port communications schemes. See Chapter 7, *Multiprocessing*, for a discussion of these multiprocessor communications schemes.

9.1.1 Link Port To Link Buffer Assignment

There are six internal data buffer registers which are independent of the actual link ports—these *link buffers*, LBUF0-LBUF5, may be connected to any of the six *link ports*. The link ports receive and transmit data on their $LxDAT_{3-0}$ data pins, but any of the six link buffers may be assigned to handle data for a particular link port. The link buffers read from or write to internal memory under DMA control.

Remember that "Link Port x" does not automatically mean "Link Buffer x."

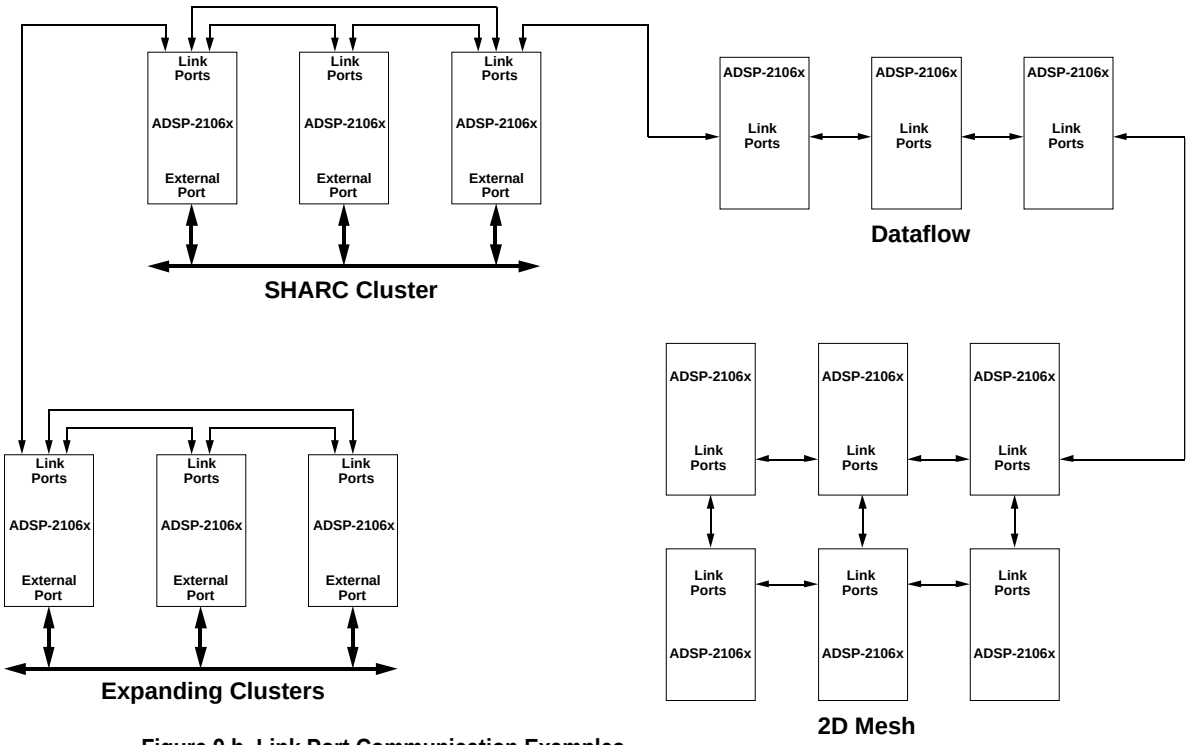


Figure 9.b Link Port Communication Examples

The Link Assignment Register (LAR) is used to assign the link buffer to link port connections. Memory-to-memory transfers may be accomplished by assigning the same link port to two buffers. Details on the LAR register can be found in the “Link Port Control Registers” section of this chapter. Figure 9.1 shows a block diagram of the link ports and link buffers.

9 Link Ports

9.1.2 Link Port DMA Channels

Link buffers 0-5 are supported by DMA channels 1, 3, 4, 5, 6, and 7 respectively:

DMA Channel 1	Link Buffer 0 *(shared with SPORT1 Receive)*
DMA Channel 3	Link Buffer 1 *(shared with SPORT1 Transmit)*
DMA Channel 4	Link Buffer 2
DMA Channel 5	Link Buffer 3
DMA Channel 6	Link Buffer 4 *(shared with Ext. Port Buffer 0)*
DMA Channel 7	Link Buffer 5 *(shared with Ext. Port Buffer 1)*

Some channels are dedicated to link ports. Some are shared with serial ports or the external port as described in the “Link Port DMA Channels” section of this chapter.

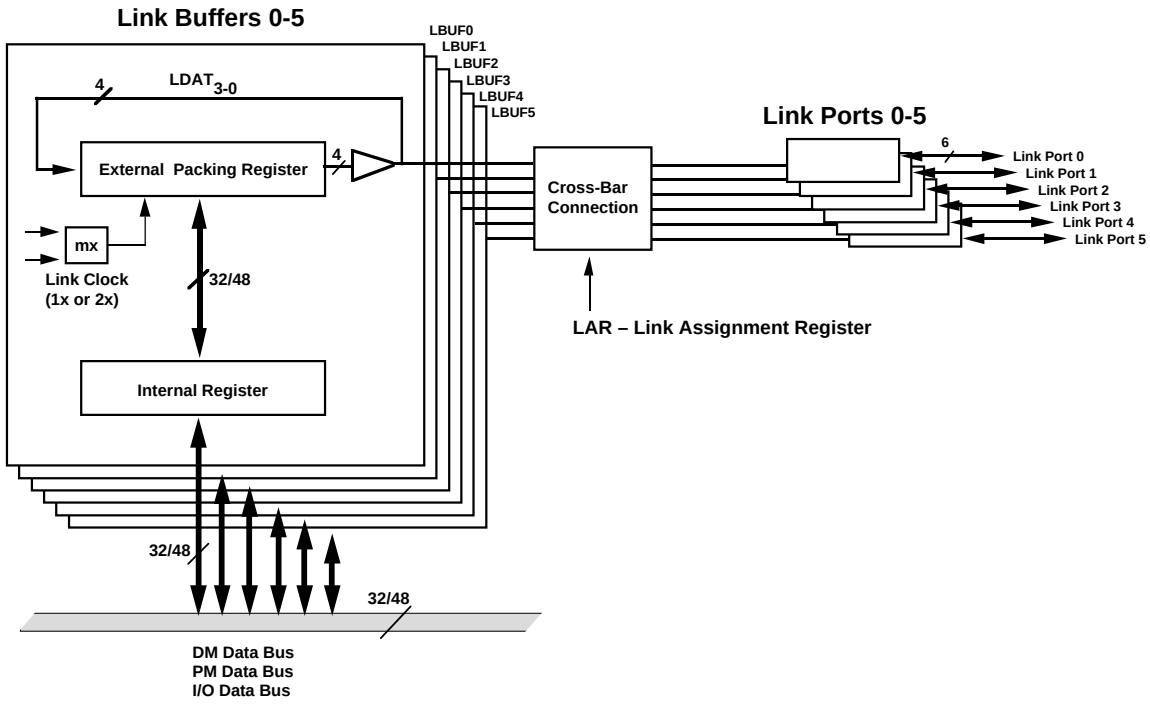


Figure 9.1 Link Ports & Buffers

9.1.3 Link Port Interrupts

Three types of interrupts are dedicated to the link ports:

- If DMA is enabled, a maskable interrupt is generated when the DMA block transfer has completed.
- If DMA is disabled, then the link buffer may be read or written by the core processor as a memory-mapped location (part of the IOP register space). A maskable interrupt is generated while DMA is disabled and the receive buffer is not empty, or if the transmit buffer is not full.
- When an external source accesses an unassigned link port (or accesses an assigned link port that has its link buffer disabled), this access causes a maskable LSRQ interrupt.

9.1.4 Link Port Booting

The link ports may be used to load internal memory at reset. Refer to the section “Booting” in the *System Design* chapter of this manual for details of this operation.

9.2 LINK PORT CONTROL REGISTERS

There are three link port control registers: the Link Buffer Control Register (LCTL), the Link Common Control Register (LCOM), and the Link Assignment Register (LAR). To configure link port operations, these registers should be set up in the following order: LAR, LCOM, then LCTL. Before reassigning a link port with the LAR register, disable the link port’s assigned buffer with the LCTL register.

The link port control registers and the serial port control registers share a common internal bus when being written or read. There is a one-cycle latency whenever one of these registers is read after one has been written.

The LCTL and LCOM control registers are initialized to 0x0000 0000 after reset. LAR is initialized to 0x0002 C688, assigning Link Port 0 to Link Buffer 0, Link Port 1 to Link Buffer 1, Link Port 2 to Link Buffer 2, Link Port 3 to Link Buffer 3, Link Port 4 to Link Buffer 4, and Link Port 5 to Link Buffer 5. For complete information about register initialization after reset, see the *Control/Status Registers* appendix of this manual.

9.2.1 Link Buffer Control Register (LCTL)

The LCTL register contains control bits unique to each link buffer. Table 9.2 describes the control bits in LCTL.

Bit(s)	*Name*	*Definition*
0-3	*	Link buffer 0 controls
4-7	*	Link buffer 1 controls
8-11	*	Link buffer 2 controls
12-15	*	Link buffer 3 controls
16-19	*	Link buffer 4 controls
20-23	*	Link buffer 5 controls
24	LEXT0	Extended word size: 1=48-bit transfers, 0=32-bit transfers
25	LEXT1	Extended word size: 1=48-bit transfers, 0=32-bit transfers
26	LEXT2	Extended word size: 1=48-bit transfers, 0=32-bit transfers
27	LEXT3	Extended word size: 1=48-bit transfers, 0=32-bit transfers
28	LEXT4	Extended word size: 1=48-bit transfers, 0=32-bit transfers
29	LEXT5	Extended word size: 1=48-bit transfers, 0=32-bit transfers
30-31	*reserved*	

Table 9.2 Link Control Register (LCTL)

* Each four-bit group includes the following control bits for each link buffer (x=0,1,2,3,4,5):

Bit#	*Name*	*Definition*
0+4x	LxEN	LBUFx enable
1+4x	LxDEN	LBUFx DMA enable
2+4x	LxCHEN	LBUFx chaining enable
3+4x	LxTRAN	LBUFx direction: 1=transmit, 0=receive

LCTL Control Bits:

LxEN Enables a link buffer. As a buffer is disabled (LxEN transitions from high to low), the LxSTAT and LRERR bits are cleared. When its buffer is disabled, an assigned link port stops receiving (driving LxACK) or transmitting (driving LxCLK). To pull the LxACK and LXCLK signals low, enable the pull down resistors with the LCOM register.

LxDEN Enables the associated DMA channel.

LxCHEN Enables DMA chaining for that channel.

LxTRAN Selects the direction of the link buffer, link port and DMA channel: 0 to receive link data, 1 to transmit link data.

LEXTx Specifies word size for each link buffer:

LEXTx=1 specifies 48-bit transfers in link buffer x
LEXTx=0 specifies 32-bit transfers in link buffer x

Link buffer data is transmitted and received MSB-first. LEXTx must not be changed while that link buffer is enabled, as this will cause the nibble packing to initialize to an incorrect value.

The LEXTx bits override the setting of the IMDW memory word width bits in SYSCON. If LEXTx=1, data to be transmitted will be read from 48-bit word space in memory, regardless of the setting of IMDW.

Note that when link buffers are enabled or disabled, it is possible to generate unwanted interrupt service requests. This can occur if Link Service Requests (LSRQ) are in use. To avoid this potential problem, the LSRQ register should be masked out while the link buffers are being enabled or disabled. See the “Link Port Interrupts” section of this chapter for more information.

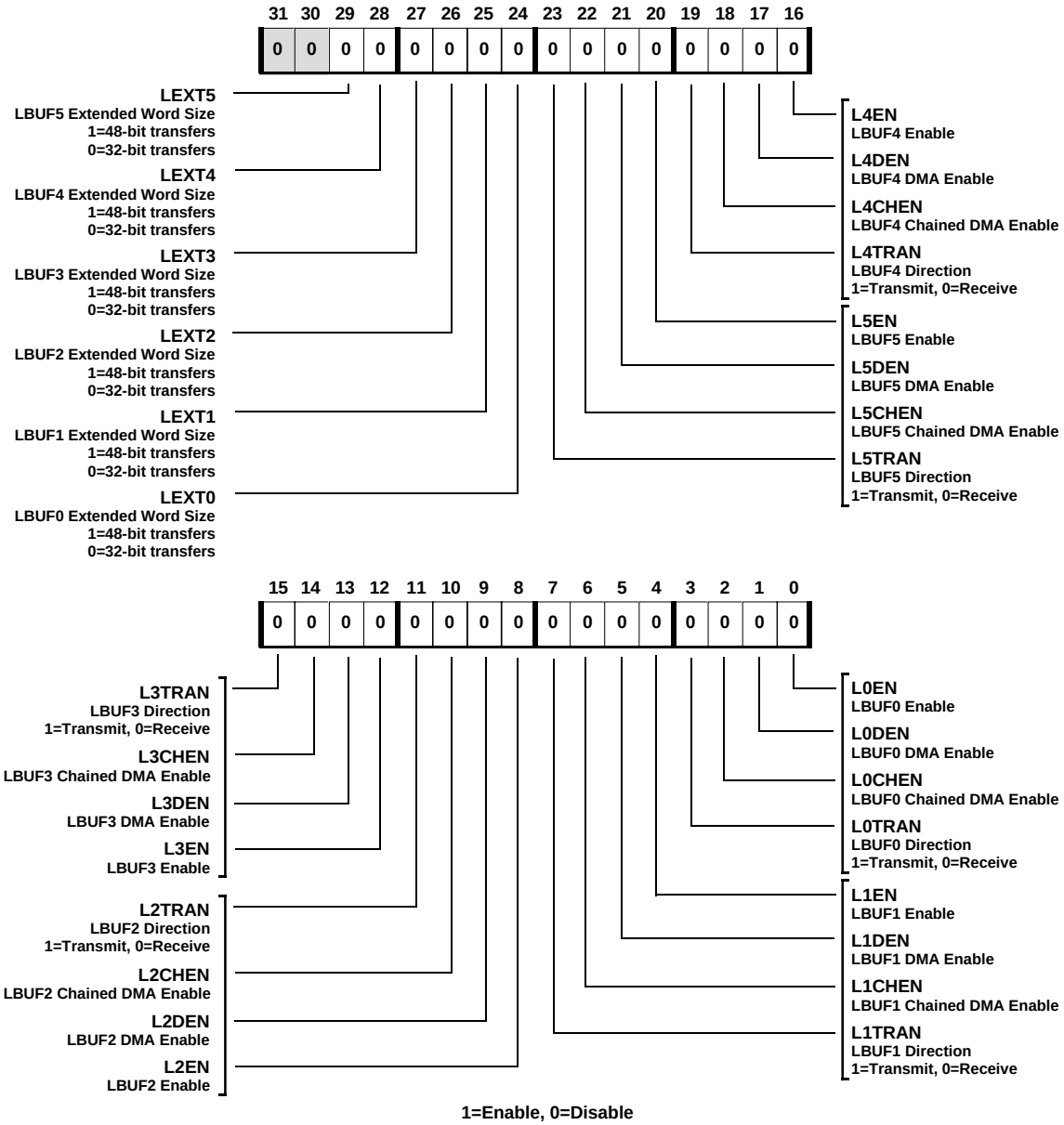


Figure 9.2 LCTL Register

9.2.2 Link Common Control Register (LCOM)

The LCOM register contains status bits, packing status bits, and 2X clock rate bits for each buffer. These bits are listed in Table 9.3.

Bit(s)	*Name*	*Definition*
0-1	L0STAT(0:1)	Link buffer 0 status: 11=full, 00=empty,10=one word *
2-3	L1STAT(0:1)	Link buffer 1 status: 11=full, 00=empty,10=one word *
4-5	L2STAT(0:1)	Link buffer 2 status: 11=full, 00=empty,10=one word *
6-7	L3STAT(0:1)	Link buffer 3 status: 11=full, 00=empty,10=one word *
8-9	L4STAT(0:1)	Link buffer 4 status: 11=full, 00=empty,10=one word *
10-11	L5STAT(0:1)	Link buffer 5 status: 11=full, 00=empty,10=one word *
12	LCLKX20	Transfer data at 2X the clock rate on Link Buffer 0
13	LCLKX21	Transfer data at 2X the clock rate on Link Buffer 1
14	LCLKX22	Transfer data at 2X the clock rate on Link Buffer 2
15	LCLKX23	Transfer data at 2X the clock rate on Link Buffer 3
16	LCLKX24	Transfer data at 2X the clock rate on Link Buffer 4
17	LCLKX25	Transfer data at 2X the clock rate on Link Buffer 5
18	L2DDMA**	Enable 2-dimensional DMA
19	LPDRD**	Disable internal pulldown resistor for LxCLK and LxACK
20	LMSP**	Mesh multiprocessing enable (set to 0 for normal operation)
21-22	LPATHD**	Mesh multiprocessing LPATH changeover delay: 00=no additional delay, 01=1 additional delay, 10=2 additional delays, 11=3 additional delays
23-25	*reserved*	
26	LRERR0	Receive pack error status for Link Buffer 0: 1=incomplete, 0=complete
27	LRERR1	Receive pack error status for Link Buffer 1: 1=incomplete, 0=complete
28	LRERR2	Receive pack error status for Link Buffer 2: 1=incomplete, 0=complete
29	LRERR3	Receive pack error status for Link Buffer 3: 1=incomplete, 0=complete
30	LRERR4	Receive pack error status for Link Buffer 4: 1=incomplete, 0=complete
31	LRERR5	Receive pack error status for Link Buffer 5: 1=incomplete, 0=complete

Table 9.3 Link Common Control Register (LCOM)

Status bits are read-only.

* The code 01 does not appear as a valid status.
** Common to all link ports.

LCOM Control Bits:

LxSTAT(0:1) When transmitting, these status bits indicate whether there is room in the buffer for more data. When receiving, these status bits indicate whether new (unread) data is available in the receive buffer. LxSTAT(1)=1 if there is data in the buffer. LxSTAT(0)=0 if there is room in the buffer. These bits are read-only. They are cleared when LxEN changes from 1 to 0. They may subsequently change state when the data buffer is read or written.

LCLKX2x This specifies link buffers to transfer at twice the ADSP-2106x clock rate. If LCLKX2x=0, transmit transfers occur at the ADSP-2106x clock frequency, and receive transfers occur at (up to) the ADSP-2106x clock frequency. Set LCLKX2x=1 for receive transfers occurring at greater than the ADSP-2106x clock frequency.

L2DDMA This directs the DMA controller to address memory as a two-dimensional array as specified in the DMA address registers. Only DMA channels 0-5 support 2D DMA. Link buffers 4 and 5 on DMA channels 6 and 7 do not support 2D DMA.

LPDRD Disables pulldown resistors on signals for unassigned link ports (or on assigned link ports that have their link buffers disabled). These pulldown resistors are 50 kΩ and apply to the LxACK, LxCLK, and $LxDAT_{3-0}$ signals. Enabling these pulldown resistors keeps the unassigned link port in an inactive state when accessed by another link port. In an application where several ADSP-2106xs share a link port, only one ADSP-2106x should have this bit cleared during operation to prevent too many pulldowns on these lines. External resistors may be used in place of these if needed. LxACK, LxCLK, and $LxDAT_{3-0}$ should never be left unconnected unless the internal pulldowns are enabled.

LMSP Enables mesh multiprocessing mode. Set LMSP=0 for normal operation.

LPATHD In a mesh multiprocessing application, these bits allow 1, 2 or 3 additional clock delays to be inserted before changing to the next LPATH register. This allows the current receive operation to complete on the current link port before a new link port is selected. In some mesh multiprocessing applications, this completion delay is significant.

LRERRx These bits reflects the status of the receive nibble packer for each link buffer. LRERRx will equal 0 when the nibble packer is set to start receiving a new word. Otherwise it will be 1. If this bit is equal to 1 after a word is received, then an error has occurred (e.g. clock glitch). The LRERRx bits are cleared when LxEN changes from 1 to 0. They may subsequently change state when the link buffer is read or written or while a word is being received.

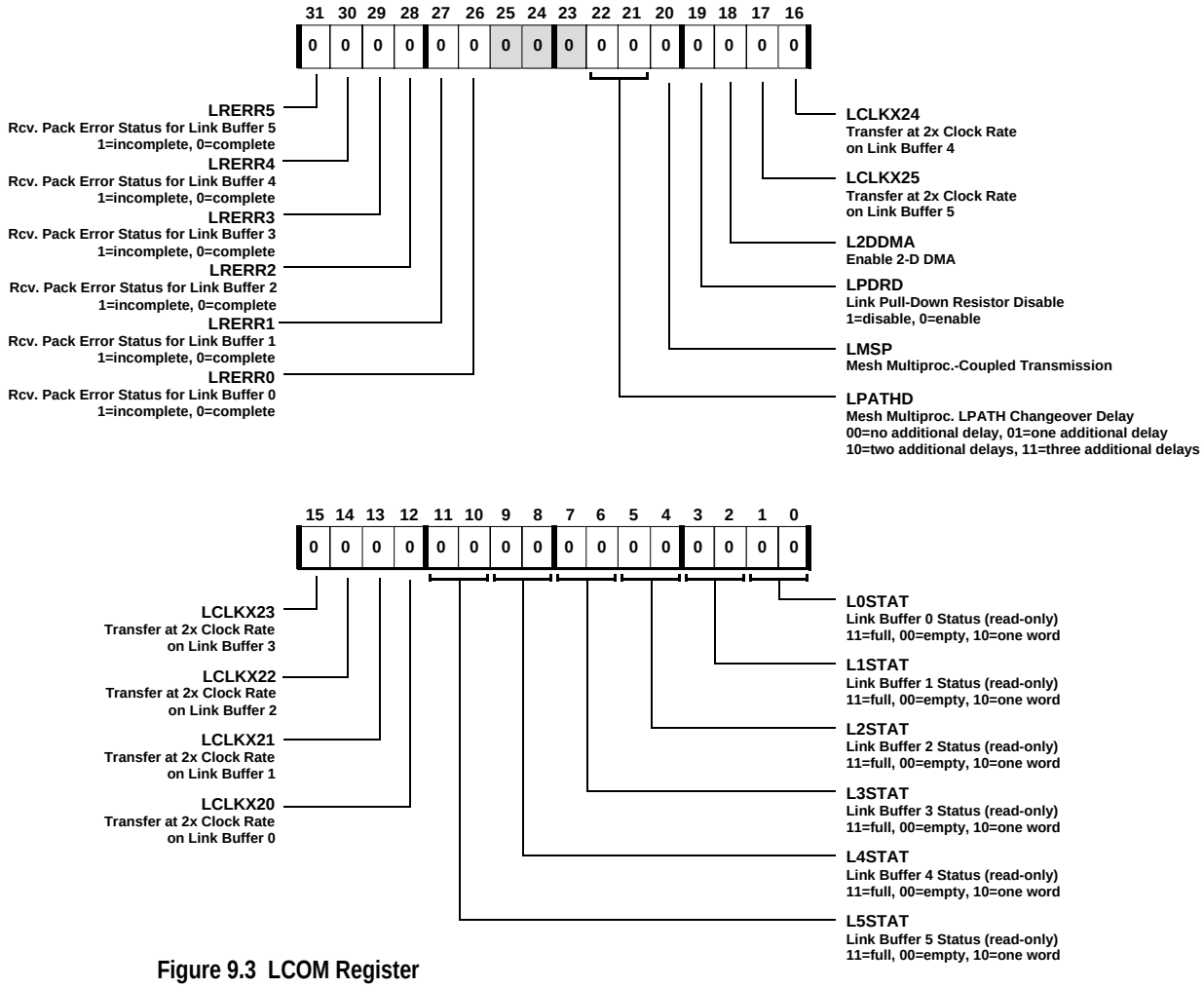


Figure 9.3 LCOM Register

9.2.3 Link Assignment Register (LAR)

Each link port is assigned to a link buffer by a 3-bit group in the Link Assignment Register (LAR). There are 6 such groups, one for each buffer, as shown in Table 9.4. The LAR can be thought of as performing a logical (i.e. the link buffer) to physical (i.e. the link port) mapping.

Bits	*Name*	*Description*
0-2	A0LB*	Link port assignment for LBUF0
3-5	A1LB*	Link port assignment for LBUF1
6-8	A2LB*	Link port assignment for LBUF2
9-11	A3LB*	Link port assignment for LBUF3
12-14	A4LB*	Link port assignment for LBUF4
15-17	A5LB*	Link port assignment for LBUF5
18-31		*reserved (must be set to 0)*

Table 9.4 Link Assignment Register (LAR)

* *AxLB*	*Link Port #*
000	Link Port 0
001	Link Port 1
010	Link Port 2
011	Link Port 3
100	Link Port 4
101	Link Port 5
110	*reserved*
111	inactive buffer

The AxLB bits assign a link port to the link buffer *x*. A link port is DISABLED if it has no buffers assigned or if the link port's assigned buffers are disabled. When a link port is disabled, its $LxDAT_{3-0}$, LxCLK, and LxACK pins are three-stated. If a buffer is intended to be inactive, the corresponding link port assignment field should be set to 7.

Memory-to-memory transfers may be accomplished by assigning the same link port to two buffers, disabling the port partially. One buffer transmits while the other receives. Up to three memory-to memory transfers may occur simultaneously, by using all six link buffers. This partially disabled mode is known as loopback mode. Using this configuration, $LxDAT_{3-0}$ and LxACK will not be driven or sensed and LxCLK will not be driven. However, LxCLK should not be driven externally in this mode, due to the fact that an LxCLK transition may be sensed and result in a nibble shift in the received data buffer.

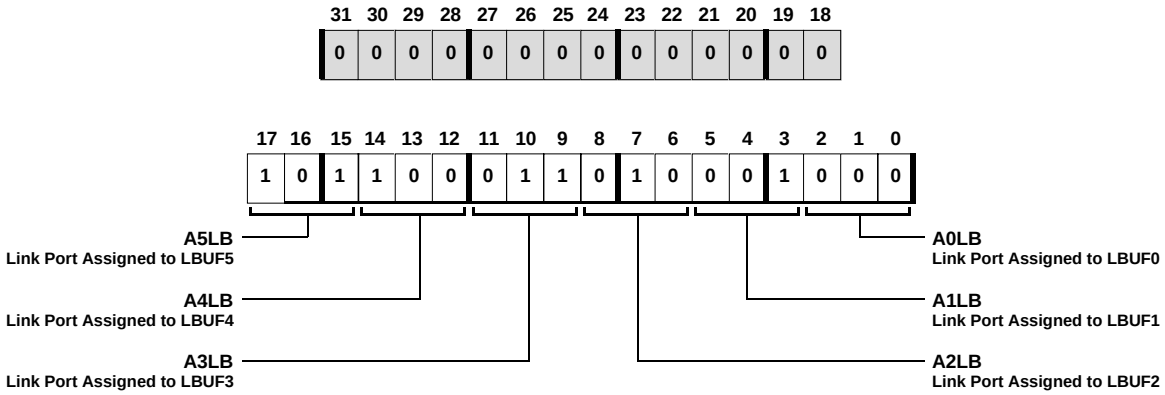


Figure 9.4 LAR Register

9.3 HANDSHAKE CONTROL SIGNALS

The LxCLK and LxACK pins of each link port allow handshaking for asynchronous data communication between ADSP-2106xs. Other devices that follow the same protocol may also communicate with these link ports.

A link-port-transmitted word consists of 8 nibbles (for a 32-bit word) or 12 nibbles (for a 48-bit word). The transmitter asserts the clock (LxCLK) high with each new nibble of data. The falling edge of LxCLK is used by the receiver to latch the nibble. The receiver asserts LxACK when it is ready to accept another word in the buffer. The transmitter samples LxACK at the beginning of each word transmission (i.e. after every 8 or 12 nibbles). If LxACK is deasserted at that time, the transmitter will not transmit the new word—see Figure 9.5. The transmitter will leave LxCLK high and continue to drive the first nibble if LxACK is deasserted. When LxACK is eventually asserted again, LxCLK will go low and begin transmission of the next word. If the transmit buffer is empty, LxCLK will remain low until the buffer is refilled, regardless of the state of LxACK.

The receive buffer may fill if a higher priority DMA or chain loading operation is occurring. LxACK may deassert when it anticipates the buffer may fill. However, LxACK will be reasserted by the receiver as soon as the internal DMA grant signal has occurred, freeing a buffer location.

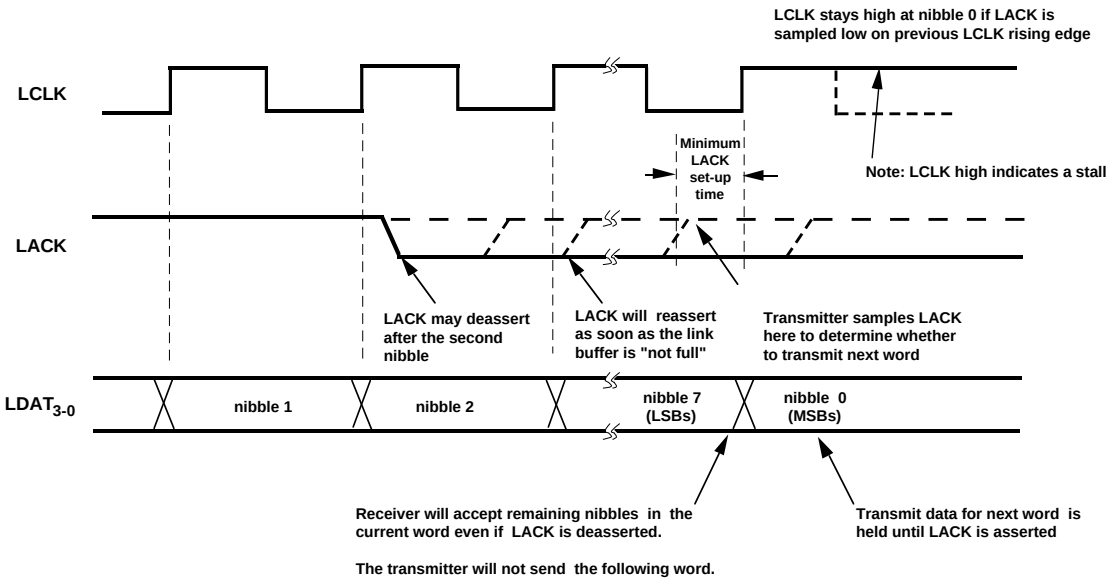


Figure 9.5 Link Port Handshake Timing

Data is latched in the receive buffer on the falling edge of LxCLK. The receive operation is purely asynchronous and can occur at any frequency up to twice CLKIN, the processor clock frequency. **If the receive clock frequency is less than or equal to CLKIN, the LCLKX2 bit for the receive buffer should be set to 0 in LCOM. If the receive clock frequency is between CLKIN and (2 ∗ CLKIN), the LCLKX2 bit should be set to 1 in LCOM.** This causes the buffer status to change, generating an internal DMA request, after the sixth nibble (of eight) or tenth nibble (of twelve) has been received. Because a preemptive DMA request is made, the entire word must be received in a single burst, with no gated clocks used during the word.

When a link port is not enabled, $LxDAT_{3-0}$, LxCLK and LxACK are tristated. When a port is enabled to transmit, the data pins will be driven with whatever data is in the output buffer, LxCLK will be driven high and LxACK will be tristated. When a port is enabled to receive, the data pins and LxCLK will be tristated and LxACK will be driven high.

To allow a transmitter and a receiver to be enabled (assigned and link buffer enabled) at different times, LxACK, LxCLK, and $LxDAT_{3-0}$ may be held low with the 50 kΩ internal pulldown resistors if LPDRD is cleared when the link port is disabled. Thus, if the transmitter is enabled before the receiver, LxACK will be low and the transmission is held off. Similarly, if the receiver is enabled before the transmitter, LxCLK will be held low and the receiver will be held off. If many link ports are bused together, one external resistor should be used to pull down each bused line instead of the internal pulldowns. This will guarantee that the bused lines are not pulled down too strongly.

LxACK, LxCLK, and $LxDAT_{3-0}$ should not be left unconnected unless external pulldown resistors or the internal pulldowns are used.

9.4 LINK BUFFERS

Each link buffer consists of an external and an internal register (see Figure 9.1). When transmitting, the internal register is used to accept the DMA data from internal memory. The external register performs the unpacking for the link port, most significant nibble first. These two registers form a two-stage FIFO, the LBUFx buffer. Two words can be written into the register (by DMA or from the core) before it signals a full condition. As each word is unpacked and transmitted, the next location in the FIFO becomes available and a new DMA request is made. If the register becomes empty, the LxCLK signal will be deasserted.

Full/empty status for the link buffer FIFOs is given by the LxSTAT bits of the LCOM register. This status is cleared for a link buffer when its LxEN enable bit is cleared in the LCTL register.

During receiving, the external buffer is used to pack the receive port data (most significant nibble first) and pass it to the internal register before DMA-transferring it to internal memory. This buffer is a two-deep FIFO. If the ADSP-2106x's DMA controller does not service it before both locations are filled, then the LxACK signal will be deasserted.

The link buffer width may be selected to be either 32 or 48 bits. This selection is made individually for each buffer with the LEXT bits in the LCTL register. For 40-bit extended precision data or 48-bit instruction transfers, the width must be set to 48 bits.

9.4.1 Core Processor Access To Link Buffers

In applications where the latency of link port DMA transfers to and from internal memory is too long, or where a process is continuous and has no block boundaries, the ADSP-2106x processor core may read or write link buffers directly using the full/empty status bit of the link buffer to automatically pace the operation. The *full or empty* status of a particular LBUFx buffer can be determined by reading the LCOM control/status register. DMA should be disabled (i.e. the LxDEN bit should be cleared) when using this capability. A programming example of core-driven transfers is shown at the end of this chapter.

If a read is attempted from an empty receive buffer, the core will hang until the link port completes transmission of a word. Similarly, if a write is attempted to a full transmit buffer, the core will hang until the external device accepts the complete word. Up to four words (2 in the receiver and 2 in the transmitter) may be sent without a hang before the receiver core must read a link buffer register. To prevent this type of hang condition from occurring, the BHD (Buffer Hang Disable) bit can be set in the SYSCON register.

9.4.2 Host Processor Access To Link Buffers

The link buffers can also be accessed by the external host processor, using direct reads and writes. When the host reads or writes to these buffers, the word width is determined *only* by the host packing mode, as selected by the HPM bits in the SYSCON register, and not by the LEXT bit in LCTL.

9.5 LINK PORT DMA CHANNELS

Link buffers 0-5 are supported by DMA channels 1, 3, 4, 5, 6, and 7 respectively. Some DMA channels are dedicated and others are shared:

- DMA channel 1 is shared by SPORT1 receive and link buffer 0 (LBUF0).
- DMA channel 3 is shared by SPORT1 transmit and link buffer 1 (LBUF1).
- DMA channel 4 is dedicated to link buffer 2 (LBUF2).
- DMA channel 5 is dedicated to link buffer 3 (LBUF3).
- DMA channel 6 is shared by ext. port buffer 0 (EPB0) and link buffer 4.
- DMA channel 7 is shared by ext. port buffer 1 (EPB1) and link buffer 5.

DMA Channels 1 and 3 are shared by link buffers 0 and 1, respectively, and by SPORT1. This has several functional implications:

- If the SPORT1 receive DMA enable bit or chaining enable bit is set, then SPORT1 receive is assigned DMA channel 1.
- If the LBUF0 DMA enable bit is set, then link buffer 0 is assigned this DMA channel.
- If both enables are set, the SPORT is selected.
- If neither the SPORT DMA enable or LBUF0 DMA enable is set, then interrupts from both buffers are ORed.

SPORT1 transmit and LBUF1 are shared and selected in the same way.

DMA Channel 6 is shared by the external port buffer EPB0 and link buffer 4 (LBUF4). Functional implications include:

- If the EPB0 DMA enable bit or chaining enable bit is set, then EPB0 is assigned DMA channel 6.
- If the LBUF4 DMA enable bit is set, then link buffer 4 is assigned this DMA channel.
- If both enables are set, EPB0 is selected.
- If neither the external port DMA enable or LBUF4 DMA enable is set, then interrupts from both buffers are ORed.

EPB1 and LBUF5 share DMA channel 7 and are selected in the same way.

A maskable interrupt is generated when the DMA block transfer has completed. A more complete discussion on interrupts can be found in the “Link Port Interrupts” section of this chapter.

If DMA is disabled for a buffer, then the buffer may be read or written by the core processor as a memory-mapped location. A maskable interrupt is generated while DMA is disabled and the receive buffer is not empty or if the transmit buffer is not full.

9.5.1 DMA Chaining For Link Ports

In chained DMA operations, the ADSP-2106x automatically sets up another DMA transfer when the contents of the current buffer have been transmitted (or received). The chain pointer register (CPx) is used to point to the next set of buffer parameters stored in memory. The ADSP-2106x's DMA controller automatically downloads these buffer parameters to set up the next DMA sequence. Refer to the *DMA* chapter of this manual for details on how to set up chaining parameters in memory.

DMA chaining is enabled on each link port by setting the LxCHEN bit in LCTL. When chaining is enabled, DMA transfers are initiated by writing a memory address to the CP register.

Six DMA parameter registers are initialized for the chained operation, in the following order:

IIx Index (start address of memory buffer)
IMx Modify (increment)
Cx Count
CPx Chain Pointer

9.6 LINK PORT INTERRUPTS

Link ports have 3 different types of interrupts:

1. nterrupts caused by the received or transmitted data when the ADSP-2106x core is accessing the buffers directly and DMA is not enabled.
2. Interrupts generated by the completion of a DMA cycle.
3. Interrupts caused by an attempted external access of a link port that is unassigned or assigned to a link buffer that is not enabled.

Types 1 and 2 are mutually exclusive and use the same interrupt.
Type 3 is independent of 1 and 2 and uses a different interrupt vector.
Details of each kind of interrupt follow below.

9.6.1 Link Port Interrupts With DMA Disabled

If DMA is disabled for a link port buffer, then the buffer may be written or read by the ADSP-2106x core as a memory-mapped IOP register.

If the DMA is disabled but the associated link buffer is enabled, then a maskable interrupt is generated whenever a receive buffer is not empty or when a transmit buffer is not full.

The interrupt latch bit in IRPTL may be masked in the corresponding IMASK register bit. When initially enabling the IMASK bit, the corresponding bit in IRPTL should be cleared first to clear out any request that may have been inadvertently latched.

The interrupt service routine should test the buffer status after each read or write to check when the buffer is empty or full, in order to determine when it should return from interrupt. This will reduce the number of interrupts it must service.

9.6.2 Link Port Interrupts With DMA Enabled

A link port interrupt is generated when the DMA operation is done—i.e. when the block transfer has completed and the DMA count register is zero.

There is an option for implementing a protocol to send additional control information at the end of a block transfer. Because the receive DMA buffer is empty when the DMA block has completed, the external bus master can send up to two additional words to the slave ADSP-2106x's buffer which has space for the two words. The slave's *DMA done* interrupt service routine could then read the buffer and use these control words to determine the next course of action.

9.6.3 Link Port Service Request Interrupts (LSRQ)

Link port service requests allow a disabled (unassigned or assigned and buffer disabled) link port to cause an interrupt when an external access is attempted. The transmit and receive request status bits of the LSRQ register (bits 20-31) allow an ADSP-2106x to determine if another ADSP-2106x is attempting to send or receive data through a particular link port. This lets two processors comunicate without prior knowledge of the transfer direction, link port number, or exactly when the transfer is to occur.

When LxACK or LxCLK is asserted externally, a link service request (LSR) is generated in a disabled (unassigned or assigned and buffer disabled) link port. LSRs will not be generated for a link port that is disabled by loopback mode. Each LSR is gated by mask bits before

being latched in the LSRQ register. The six possible receive LSRs and the six possible transmit LSRs are ORed together to generate the link service request interrupt. The LSRQ interrupt request may be masked by the LSRQI mask bit of the IMASK register. When the mask bit is set, the interrupt is allowed to pass into the interrupt priority encoder. A diagram of this logic appears in Figure 9.5a.

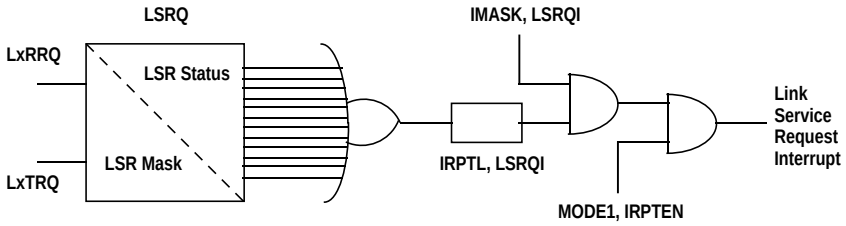


Figure 9.5a Logic For Link Port Interrupts

The interrupt routine must read the LSRQ register to determine which link port to service and whether it is a transmit or receive request.

LSR interrupts have a latency of two cycles. Note that the link service request interrupt is different from the link receive and transmit interrupt—this is also true in IMASK.

The 32-bit LSRQ register holds the masked link status of each link port and the corresponding interrupt mask bits. The link status of the port is set whenever the port is not enabled and one of LxACK or LxCLK is asserted high. The LSRQ status bits are read-only. Table 9.5 and Figure 9.6 show the individual bits of the LSRQ register.

To determine which link port to service, transfer LSRQ to a register Rx (in the register file), then use the leading 0s detect instruction, Rn=LEFTZ Rx. Rn indicates which link port is active in order of priority.

If link service requests are in use, they should be masked out when the assigned link buffers are being enabled, disabled, or when the link port is being unassigned in LAR, otherwise spurious service requests may be generated.

This need for masking is due to a delay before LxCLK or LxACK (if already asserted) signals are pulled (if pulldowns enabled) or driven externally (if pulldowns disabled) below logic threshold. During this delay, these signals are sampled asserted and generate an LSRQ.

To avoid the possiblity of spurious interrupts, mask the LSRQ interrupt or the appropriate request bit in the LSRQ register and allow an appropriate delay before unmasking. Alternatively, mask the LSRQ interrupt and poll the appropriate request status bit until it is cleared and then unmask the interrupt.

Bit	*Name*	*Description*
0-3	*reserved*	
4	L0TM	Link Port 0 transmit mask
5	L0RM	Link Port 0 receive mask
6	L1TM	Link Port 1 transmit mask
7	L1RM	Link Port 1 receive mask
8	L2TM	Link Port 2 transmit mask
9	L2RM	Link Port 2 receive mask
10	L3TM	Link Port 3 transmit mask
11	L3RM	Link Port 3 receive mask
12	L4TM	Link Port 4 transmit mask
13	L4RM	Link Port 4 receive mask
14	L5TM	Link Port 5 transmit mask
15	L5RM	Link Port 5 receive mask
16-19	*reserved*	
20	L0TRQ	Link Port 0 transmit request status *(read-only)*
21	L0RRQ	Link Port 0 receive request status *(read-only)*
22	L1TRQ	Link Port 1 transmit request status *(read-only)*
23	L1RRQ	Link Port 1 receive request status *(read-only)*
24	L2TRQ	Link Port 2 transmit request status *(read-only)*
25	L2RRQ	Link Port 2 receive request status *(read-only)*
26	L3TRQ	Link Port 3 transmit request status *(read-only)*
27	L3RRQ	Link Port 3 receive request status *(read-only)*
28	L4TRQ	Link Port 4 transmit request status *(read-only)*
29	L4RRQ	Link Port 4 receive request status *(read-only)*
30	L5TRQ	Link Port 5 transmit request status *(read-only)*
31	L5RRQ	Link Port 5 receive request status *(read-only)*

Table 9.5 Link Service Request Register (LSRQ)

For *transmit request status* bits, LxTRQ=1 means LxACK=1, LxTM=1, and LxEN=0.
For *receive request status* bits, LxRRQ=1 means LxCLK=1, LxRM=1, and LxEN=0.

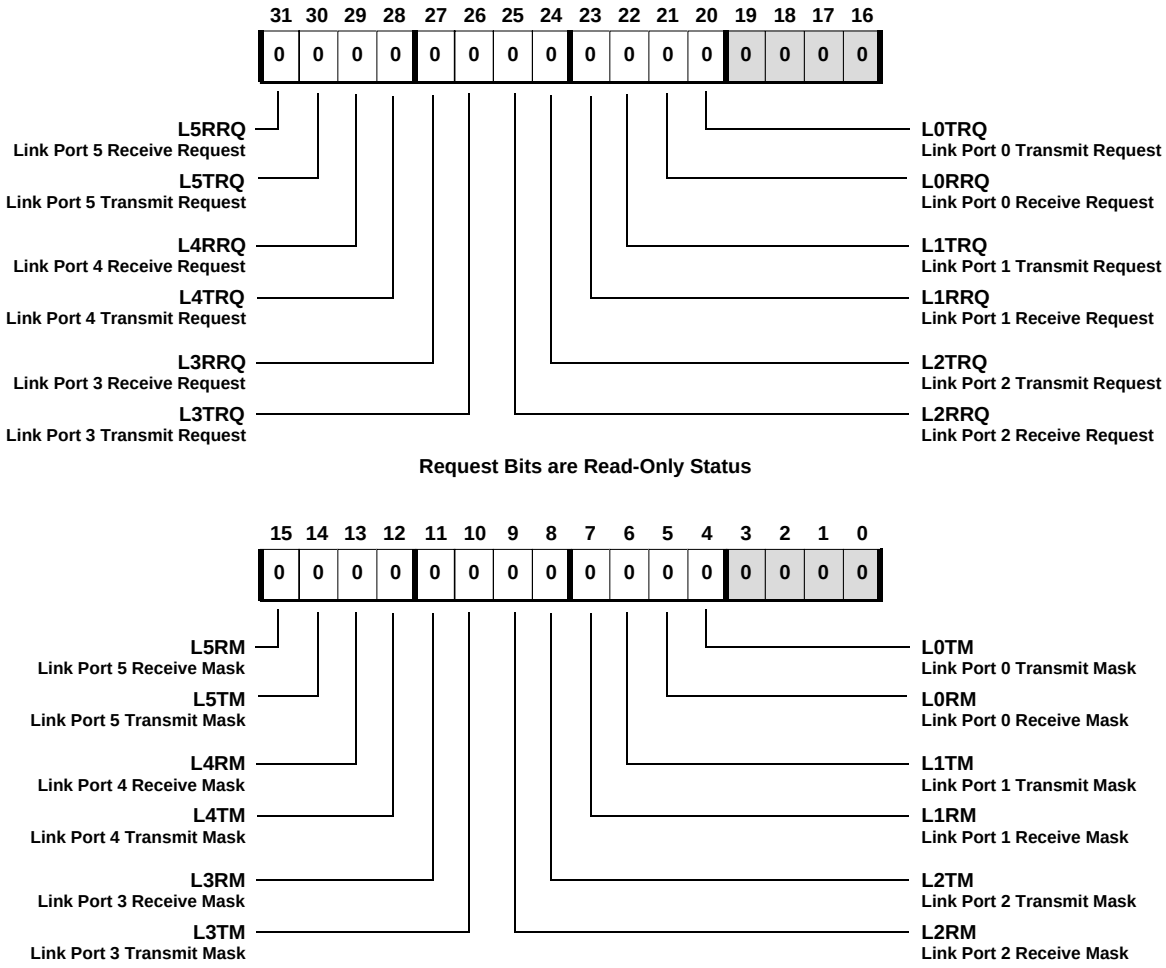


Figure 9.6 LSRQ Register

For *transmit request status* bits, LxTRQ=1 means LxACK=1, LxTM=1, and LxEN=0.
For *receive request status* bits, LxRRQ=1 means LxCLK=1, LxRM=1, and LxEN=0.

9.7 TRANSMISSION ERROR DETECTION

Transmission errors on the link ports may be detected by reading the LRERRx bits in the LCOM register. These bits reflects the status of each nibble counter. The LRERRx bit will be zero if the pack counter of the corresponding link buffer is zero (i.e. a multiple of 8 or 12 nibbles have been received). If LRERR is high when a transmission has completed, then an error occurred during transmission. (Note that the DMA word count provides an exact count of the number of words to be transferred.)

To allow checking of this status, the transmitter and receiver should follow a protocol such as the one described below:

Transmitter Protocol—To make use of the LRERRx status, one additional dummy word should always be transmitted at the end of a block transmission. The transmitter must then deselect the link port to allow the receiver to send an appropriate message back to the transmitter.

Receiver Protocol—When the receiver has received the data block, indicated by a *DMA done* interrupt, it checks that it has received an additional word in the link buffer and then reads the LRERR bit. The receiver may then clear the link buffer (LxEN=0) and transmit the appropriate message back to the transmitter on the same, or a different, link port.

9.8 TOKEN PASSING

Two processors wishing to communicate on a link port need to know which of them is currently the transmitter and which is the receiver, otherwise they might both try to transmit at the same time. Token passing is a way of accomplishing this. Figure 9.7 shows a flow chart of the token passing process.

Token passing is a way of establishing which of two ADSP-2106xs is the current transmitter. The token is a software flag, similar to a semaphore, that is passed between the processors. At reset, by convention, the token (flag) is set to reside in the link port of one device, making him the master and the transmitter. When a receiver link port (slave) wants to become the master, he may assert his LxACK line (request data) to get the master's attention. The master will know, through software protocol, whether he is supposed to respond with actual data or whether he is being asked for the token.

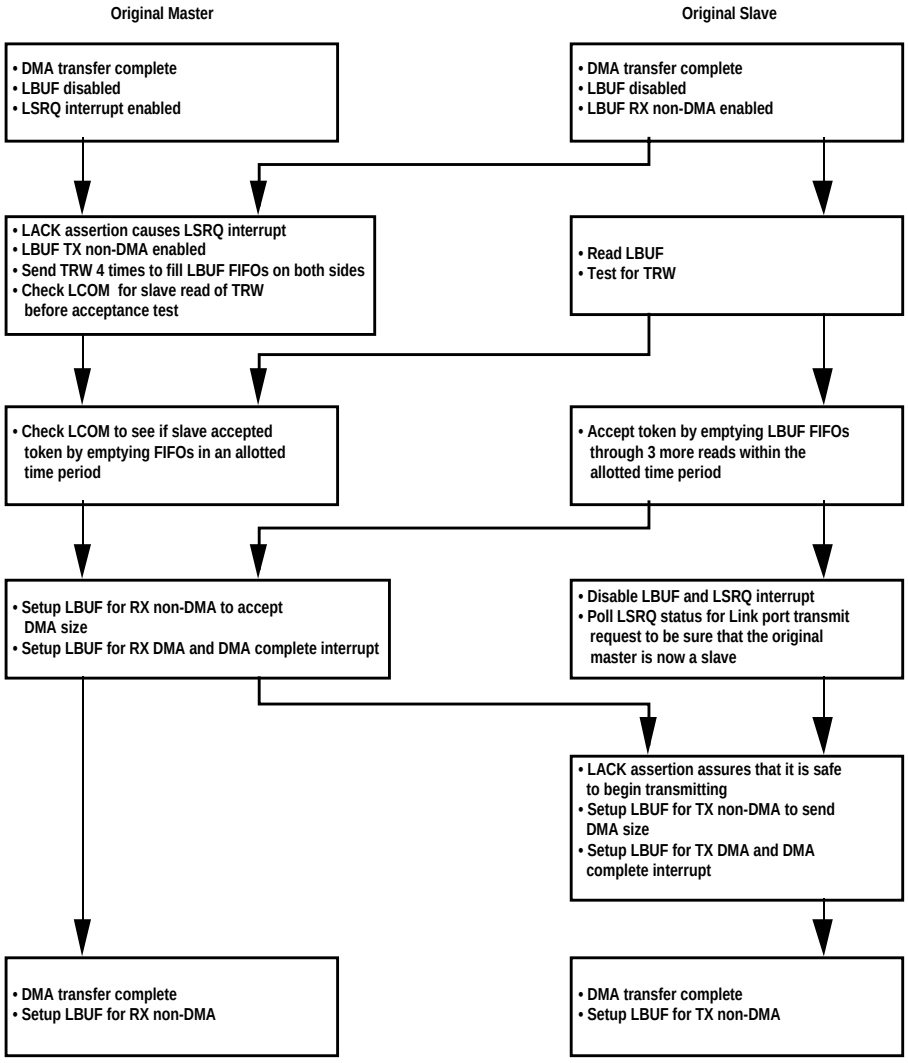


Figure 9.7 Token Passing Flow Chart

Notes:

The token release word can be any user-defined value. Since both the transmitter and receiver are expecting a code word, this need not be exclusive of normal data transmission.

If the master wishes to give up the token, he may send back a user-defined token release word and thereafter clear his token flag. Simultaneously, the slave examines the data sent back and if it is the token release word, the slave will set his token, and can thereafter transmit. If the received data is not the token release word, then the slave must assume the master was beginning a new transmission.

Through software protocol, the master can also ask to receive data by sending the token release word without the LxACK (data request) going low first.

An example of software protocol for token passing through the link ports is included at the end of this chapter. Figure 9.7 shows a flow chart of the example code and the following is a description of the process:

The example code is to be loaded on both the original master and the original slave. The code is ID intelligent for multiprocessor systems: ID1 is the original master (transmitter) and ID2 is original slave (receiver). The master transmits a buffer via DMA through LPORT0 using LBUF3 and the slave receives through LPORT0 using LBUF2. The slave then requests the token by generating an LSRQ interrupt in the disabled link port of the master (LPORT0). The master responds by sending the token release word and waiting to see if it is accepted. The slave checks to see that it is the token release word and will accept the token by emptying the master's link buffer FIFO within a predetermined amount of time. If the token is accepted the slave will become the master and transmit a buffer of data to the new slave. If the token is rejected, the master will transmit a second buffer. When complete, the original master will finish by setting up LBUF2 to receive without DMA, and the original slave will set up LBUF3 to transmit without DMA.

The following is a list of the major areas of concern when implementing a software protocol scheme for token passing:

- Ensure that both link buffers are not enabled to transmit at the same time. In the event that this is allowed, data may be transmitted and lost due to the fact that neither link port will be driving LxACK. In the example provided at the end of the chapter, the LSRQ register status bits are polled to ensure that the master becomes the slave before the slave becomes the master, thus avoiding the two transmitter conflict.

- Ensure that the link interrupt selection matches the application. If a status detection scheme using the status bits of the LSRQ register is to be used, it is important to note the following: If a link port that is configured to receive is disabled while LxACK is asserted, there will be an RC delay before the 50k ohm pulldown resistor on LxACK (if enabled) can pull the value below logic threshold. Likewise, if a link port that is configured to transmit is disabled while LxCLK is asserted, there will be an RC delay before the 50k ohm pulldown resistor on LxCLK (if enabled) can pull the value below logic threshold. If the appropriate request status bit is unmasked in the LSRQ register (in this instance), then an LSR will be latched and the LSRQ interrupt may be serviced, even though unintended, if enabled.

- Ensure that synchronization is not disrupted by unrelated influences at critical sections where timing control loops are used to synchronize parallel code execution. Disabling of nested interrupts is one of the techniques used in the example to control this.

9.9 LINK TRANSMISSION LINES

The link ports are designed to allow long distance connections to be made between the driver and the receiver. This is possible because the links are self-synchronizing, i.e. the clock and data are transmitted together. Only relative delay, not absolute delay between clock and data is of importance.

In addition, the LACK signal inhibits transmission of the next word, not of the current nibble (i.e. the current word is always allowed to complete transmission). This allows delays of 2 to 3 cycles for the LxACK signal to reach the transmitter.

The links are designed to drive transmission lines with characteristic impedances of 50Ω or greater. A higher transmission line impedance reduces the on-chip effect of driver impedance variations, for distances longer than about 6 inches. It is recommended that an external series termination resistor be used at each link port pin to absorb reflections from the open circuit at the destination. The external resistor should be selected such that its value (plus the internal resistance of the driver) be equal to the characteristic impedance of the transmission line.

Thus, if the typical internal drive resistance is 10Ω and the characteristic impedance is 50Ω, then the link port pin resistor should be 40Ω.

9.10 SYSTEM DESIGN EXAMPLE: LOCAL DRAM INTERFACE

The example shown in Figure 9.8 shows how a multiprocessing system can use link ports to connect to local memories and I/O devices. An ASIC implements the interface between the link port and DRAM or an I/O device. This minimal hardware solution frees the ADSP-2106x's external bus for other shared-bus communication. The DRAM and ASIC may be implemented on a single 10-pin SIMM module.

Accesses to the DRAM via a link is most efficient under DMA control. The ASIC receives DMA control information from the link port and sets up the access to the DRAM. It unpacks 16-bit data words from the DRAM or packs 4-bit nibbles from the link. At the end of the DMA transfer, an interrupt will allow the ADSP-2106x to send new control information to the ASIC. The ASIC always reverts to receive mode at the end of a transfer. The LxACK signal is deasserted by the ASIC whenever a page change, memory refresh cycle, or any other access to the DRAM occurs.

Memory modules may be shared by multiple ADSP-2106xs when the link port is bused. Each link port supports 40 Mbyte access throughput for either instructions or data. The ASIC is responsible for generating the 2X clock when transmitting to the ADSP-2106x. The ASIC is also responsible for generating sequential DMA addresses based on a start address and word count.

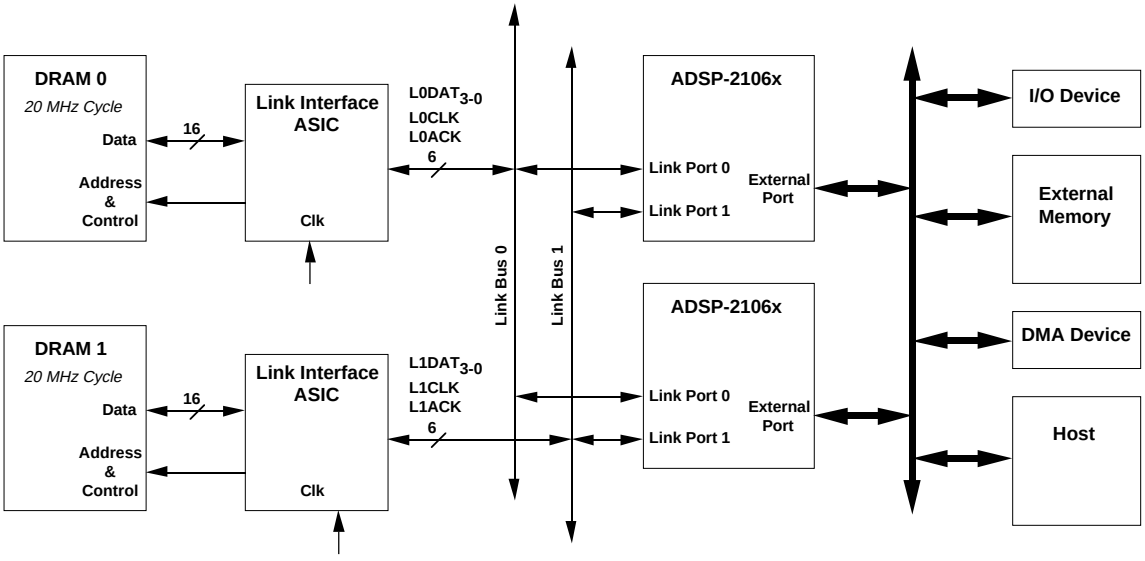


Figure 9.8 Local DRAM With Link Ports

9.11 PROGRAMMING EXAMPLES

Listings 9.1 and 9.2 illustrate two ways to perform link port data transfers.

9.11.1 Core-Driven Single-Word Transfers

Listing 9.1 shows an example of single-word data transfers controlled by the ADSP-2106x core.

9.11.2 DMA Transfers

Listing 9.2 shows an example DMA transfer program.

```
/*_____________________________________________________________
  ADSP-2106x Core-Driven LINK Loopback Example

This example shows an internally looped-back link port transfer. The core directly
writes to the transfer link buffer (LBUF3) and reads from the receive link buffer
(LBUF2). The core will hang on the read of LBUF2 until the data is ready. Loopback is
achieved by assigning the transmit and receive link buffers to the same port (Port
0).
_____________________________________________________________*/

#include “def21060.h”

.segment/pm rst_svc;    /* Reset vector from architecture file.*/
   nop;                 /* First location is used for booting. */
   jump start;
.endseg;

/*_____________________main routine_____________________*/
.segment/pm pm48_1b0;   /* Main code segment from arch file.*/

start:
   r0=0x0000c000;       /* LCOM Register: 2x rate  */
   dm(LCOM)=r0;         /* note: use r0=0x00010000 on rev. 0.X silicon */

   r0=0x0003f03f;       /* LAR Register: LBUF2->Port0, LBUF3->Port0   */
   dm(LAR)=r0;          /* All others inactive.                        */

   r0=0x00009100;       /* LCTL Register: 32-bit data, LBUF2=rx, LBUF3=tx */
   dm(LCTL)=r0;         /* Always write LCTL after LAR.                  */

   r0=0x12345678;       /* Test data to transmit.     */
   dm(LBUF3)=r0;        /* Write to LBUF3 to transmit.*/

   r1=dm(LBUF2);        /* Read—Core will hang here until data is received.*/

wait:  jump wait;

.endseg;
```

Listing 9.1 Core-Driven Example

```
/*_____________________________________________________________
  ADSP-2106x DMA-Driven LINK Loopback Example

This example shows an internally looped-back link port transfer.
Two DMA channels are used. Link buffer 3 (LBUF3) and corresponding
DMA channel 5 is used for transmit. Link buffer 2 (LBUF2) and
corresponding DMA channel 5 is used for receive. The LBUF2 interrupt
will occur when the DMA transfer is complete. Loopback is achieved
by assigning the transmit and receive link buffers to the
same port (Port 0).
_____________________________________________________________*/

#define N 8
#include “def21060.h”

.segment/dm dm32_b1;    /* Data segment name described in arch. file.*/
.var source[N] = 0x11111111, 0x22222222, 0x33333333, 0x44444444,
                 0x55555555, 0x66666666, 0x77777777, 0x88888888;
.var destination[N];
.endseg;

.segment/pm rst_svc;    /* Reset vector from arch. file.      */
   nop;                 /* First location is used for booting.*/
   jump start;
.endseg;

.segment/pm lp2_svc;    /* Link buffer 2 interrupt vector.*/
   jump lp2rx;
.endseg;

/*_____________________main routine_____________________*/

.segment/pm pm48_1b0;   /* Main code segment from arch. file.*/

start:
   r0=source;
   dm(II5)=r0;          /* Set DMA tx index to start of source buffer.*/
   r0=destination;
   dm(II4)=r0;          /* Set DMA rx index to start of destination buffer.*/
   r0=1;
   dm(IM5)=r0;          /* Set DMA modify (stride) to 1.*/
   dm(IM4)=r0;
   r0=@source;
   dm(C5)=r0;           /* Set DMA count to length of data buffer.*/
   dm(C4)=r0;

   r0=0x0000c000;       /* LCOM Register: 2x rate */
   dm(LCOM)=r0;         /* note: use r0=0x00010000 on rev. 0.X silicon */
```

(listing continues on next page)

```
   r0=0x0003f03f;       /* LAR Register: LBUF2->Port0, LBUF3->Port0 */
   dm(LAR)=r0;          /* All others inactive.                     */

   r0=0x0000b300;       /* LCTL Register: 32-bit data, LBUF2=rx, LBUF3=tx */
   dm(LCTL)=r0;         /* Enable DMA on LBUF2 and LBUF3.                 */
                           /* This will start off the DMA transfer.      */
                           /* Always write LCTL after LAR.               */

   bit set imask LP2I;      /* Enable link buffer 2 interrupt. */
   bit set mode1 IRPTEN;    /* Global interrupt enable.        */

wait: idle;             /* Wait for link buffer 2 interrupt.*/
      jump wait;        /* Will end up here after entire DMA completes.*/


/*_____________Link Buffer 2 Receive Interrupt Routine_________________*/

lp2rx: rti;             /* This interrupt will occur only once.*/

.endseg;
```

Listing 9.2 DMA Transfer Example

```
/*_____________________________________________________________________________
ADSP-2106x LINK Token Pass Example

This is an example of software protocol for token ring passing through the link
ports using LSRQ. This code is to be loaded on both the original master and the
original slave. The code is ID intelligent for multiprocessor systems: ID1 is
the original master (transmitter) and ID2 is original slave (receiver). The
master transmits a buffer via dma through LPORT0 using LBUF3 and the slave
receives through LPORT0 using LBUF2. The slave then requests the token by
generating an LSRQ interrupt in the disabled link port of the master (LPORT0).
The master responds by sending the token release word and waiting to see if it
is accepted. The slave checks to see that it is the token release word and will
accept the token by emptying the master's link buffer FIFO within a
predetermined amount of time. If the token is accepted the slave will become
the master and transmit a buffer of data to the new slave. If the token is
rejected, the master will transmit a second buffer. When complete the original
master will finish by setting up LBUF2 to receive without DMA and the original
slave will set up LBUF3 to transmit without DMA. FLAG0 is used as a software
flac to indicate the original master.
_____________________________________________________________________________*/


#include "def21060.h"

#define N 8       /* Size of buffer */

#define trw 0x0         /* Token release word   */

#define orig_master_id 1   /* ID of SHARC to be original master   */
#define orig_slave_id 2    /* ID of SHARC to be original slave */

.SEGMENT/DM     dm_data;

.var source_1[N]= 0x11111111, 0x22222222, 0x33333333, 0x44444444,
        0x11111111, 0x22222222, 0x33333333, 0x44444444;

.var source_2[N]= 0x55555555, 0x66666666, 0x77777777, 0x88888888,
        0x55555555, 0x66666666, 0x77777777, 0x88888888;
```

Listing 9.3 Link Token Passing Example (continues)

```
.var source_3[N]= 0x11111111, 0x22222222, 0x33333333, 0x44444444,
        0x55555555, 0x66666666, 0x77777777, 0x88888888;

.var destination_1[N];

.var destination_2[N];

.var destination_3;

.ENDSEG;


.SEGMENT/PM   isr_tabl; /* Interrupt Service Table    */

      NOP; NOP;NOP;NOP; /* Reserved interrupt      */
rst_svc:        nop; jump start; nop; nop;
      NOP; NOP; NOP; NOP;
sovfi_svc:      RTI; RTI; RTI; RTI;
tmzhi_svc:      rti; RTI; RTI; RTI;
vrpti_svc:      RTI; RTI; RTI; RTI;
irq2_svc:       rti; RTI; RTI; RTI;
irq1_svc:       rti; RTI; RTI; RTI;
irq0_svc:       rti; RTI; RTI; RTI;
      NOP; NOP; NOP; NOP;
spr0_svc:       RTI; RTI; RTI; RTI;
spr1_svc:       RTI; RTI; RTI; RTI;
spt0_svc:       RTI; RTI; RTI; RTI;
spt1_svc:       RTI; RTI; RTI; RTI;
lp2_svc:        nop; jump lp2; nop; nop;
lp3_svc:        nop; jump lp3; nop; nop;
ep0_svc:        RTI; RTI; RTI; RTI;
ep1_svc:        RTI; RTI; RTI; RTI;
ep2_svc:        RTI; RTI; RTI; RTI;
ep3_svc:        RTI; RTI; RTI; RTI;
lsrq_svc:       nop; jump lsrq; nop; nop;
cb7_svc:        RTI; RTI; RTI; RTI;
cb15_svc:       RTI; RTI; RTI; RTI;
tmzl_svc:       rti; RTI; RTI; RTI;

.ENDSEG;
```

Listing 9.3 Link Token Passing Example (continues)

```
/*_____________________main routine_____________________*/

.SEGMENT/PM   pm_code;


start:
   bit set mode2 FLG0O; /* Set Flag0 for output          */
   bit clr astat FLG0;  /* Clear Flag0 for use as flag to test    */
            /* if this SHARC is the original master      */
   r0=dm(SYSTAT);
   r0=FEXT r0 BY 8:3;   /* Extract Processor ID from SYSTAT    */

   r1=orig_master_id;
   r1=r0-r1;         /* Test if this SHARC is original   */
   if eq jump start_as_master;   /* master and jump appropriately */

   r1=orig_slave_id;
   r1=r0-r1;         /* Test if this SHARC is original   */
   if eq jump start_as_slave; /* slave and jump appropriately     */

   idle;
   nop;



/*_________________Start of Original Master Routine_____________________*/

start_as_master:

   bit set astat FLG0;  /* Set Flag0 to signify original master         */

   r0=source_1;
   dm(II5)=r0; /* Set DMA tx index to start of source buffer   */

   r0=1;
   dm(IM5)=r0; /* Set DMA modify (stride) to 1        */

   r0=@source_1;
   dm(C5)=r0;  /* Set DMA count to length of data buffer */
```

Listing 9.3 Link Token Passing Example (continues)

```
   r0=0xc000;     /* LCOM Register: 2x rate on LBUF3,       */
   dm(LCOM)=r0;   /* note:use r0=0x00010000 on rev. 0.X silicon   */

   r0=0x3f1ff;    /* LAR Register: LBUF3->port 0   */
   dm(LAR)=r0; /* All others inactive     */

   bit set imask LP3I;  /* Enable Link buffer 3 interrupt      */
   bit set mode1 IRPTEN;   /* Global interrupt enable       */

   r0=0x0000b000; /* LCTL Register: 32-bit data, LBUF3=tx      */
   dm(LCTL)=r0;   /* enable DMA on LBUF3           */
            /* This will start off the DMA transfer      */
            /* Always write LCTL after LAR      */


wait_1:   idle;      /* Wait for Link buffer 3 interrupt    */
          jump wait_1;     /* Will end up here after entire DMA complete
*/
     nop;         /* All master interrupts will come back to here */
     nop;




/*____________________Link buffer 3 Interrupt Routine____________________*/

lp3:
   if  NOT FLAG0_IN jump lp3_orig_slave;  /* Test for original master   */
   nop;              /* and jump appropriately  */
   nop;




/*_________________Link buffer 3 Tx finish Interrupt Routine_____________*/

lp3_orig_master:
```

Listing 9.3 Link Token Passing Example (continues)

```
/*_________________Allow for pulldown delay on LxACK of the
slave___________________*/


   bit clr imask LSRQI;    /* Disable Link port service request interrupt
*/

   r0=0x10;
   dm(LSRQ)=r0;      /* Unmask LSRQ lport 0 transmit request status     */

   r0=0x00000000;
   dm(LCTL)=r0;      /* LCTL: disable all LBUFs       */

disabled1:
   r0=dm(LSRQ);      /* Check to ensure that the pull down on LxACK     */
   r0=FEXT r0 BY 20:1;  /* has pulled the LxACK low.  Both the original */
   r0=pass r0;    /* slave and original master will be in sync.   */
   if NE jump disabled1;   /* The next assertion of LxACK will signify to  */
            /* the master that slave is requesting the token*/


/
*_____________________________________________________________________________*/



   r0=0x00000000;
   dm(LCTL)=r0;      /* disable all LBUFs */

   bit set imask LSRQI;    /* Enable Link port service request interrupt
*/

   r0=0x10;
   dm(LSRQ)=r0;      /* Unmask LSRQ lport 0 transmit request status     */

   rti;



/*_________________Link buffer 2 Tx finish Interrupt Routine_____________*/
```

Listing 9.3 Link Token Passing Example (continues)

```
lp3_orig_slave:
           /* Finish by setting up Tx without DMA    */
   bit clr imask LP3I;     /* disable Link buffer 2 interrupt     */

   r0=0x3f1ff;
   dm(LAR)=r0;    /* LAR Register:  LBUF3->port 0        */

   r0=0x00009000;
   dm(LCTL)=r0;      /* enable LBUF3 Tx, No DMA       */

   rti;



/*_______________Link Service Request Interrupt Routine_____________*/

lsrq:
   bit clr imask LP3I;  /* disable Link buffer 3 interrupt     */

   r0=0x00009000; /* LCTL Register:32-bit data, LBUF3=tx    */
   dm(LCTL)=r0;   /* disable DMA on LBUF3          */

   r0=trw;        /* Get token release word        */

   dm(LBUF3)=r0;  /* Send token release word to slave    */
   dm(LBUF3)=r0;  /* fill slave's and master's LP fifos by  */
   dm(LBUF3)=r0;  /* writing four times, leaving the fifos  */
   dm(LBUF3)=r0;  /* completely filled on both sides     */

token_read:
   r1=0x40;    /* check if slave read the token    */
   r0=dm(LCOM);      /* check if slave read the first word  */
   r0=r0 AND r1;     /* to be sure they are in sync for the    */
   if NE jump token_read;  /* reject test             */


   LCNTR=10, DO wait_for_slave UNTIL LCE;
wait_for_slave:   nop;     /* Give slave chance to accept or reject token
```

Listing 9.3 Link Token Passing Example (continues)

```
*/


   r1=0xc0;       /* check if slave wants the token   */
   r0=dm(LCOM);         /* check if slave emptied the fifos     */
   r0=r0 AND r1;        /* within 10 cycles        */
   if NE jump second_master_mode;   /* else second master mode    */


slave_mode:

/*_______________Protection to avoid two transmitting link
ports_________________*/


   bit clr imask LSRQI;    /* Disable Link port service request interrupt
*/

   r0=0x10;
   dm(LSRQ)=r0;      /* Unmask LSRQ lport 0 transmit request status     */

   r0=0x00000000;
   dm(LCTL)=r0;      /* LCTL: disable all LBUFs       */

slave_disabled:
   r0=dm(LSRQ);      /* Check to ensure that the original slave   */
   r0=FEXT r0 BY 20:1;  /* is now disabled.  If disabled, will deassert */
   r0=pass r0;    /* LxACK and stop generating LxTRQ     */
   if NE jump slave_disabled;

/
*_____________________________________________________________________________*/



   r0=0x3fe3f;
   dm(LAR)=r0;    /* LAR Register:  LBUF2->port 0        */

   r0=0x00000100;
   dm(LCTL)=r0;      /* LCTL: enable LBUF2 (Rx), non DMA    */
```

Listing 9.3 Link Token Passing Example (continues)

```
   r0=dm(LBUF2);  /* read DMA size           */
   dm(C4)=r0;  /* Set DMA count to length of data buffer */

   r0=destination_3;
   dm(II4)=r0; /* Set DMA rx index to start of destin buffer   */

   r0=1;
   dm(IM4)=r0;    /* step size               */

   r0=0x00000300;
   dm(LCTL)=r0;   /* enable LBUF2 DMA Rx           */

   bit clr irptl LP2I;  /* clear pending Link buffer 2 interrupt  */
   bit set imask LP2I;  /* Enable Link buffer 2 interrupt      */
   bit set mode1 IRPTEN;   /* Global interrupt enable       */

   rti;


second_master_mode:

token_read2:
   r1=0xC0;    /* check if slave read the tokens      */
   r0=dm(LCOM);      /* check if slave emptied fifos        */
   r0=r0 AND r1;     /* to be sure they are in sync for the    */
   if NE jump token_read2; /* the second DMA transfer       */

   r0=0x3fe3f;
   dm(LAR)=r0;    /* LAR Register:  LBUF2->port0         */

   r0=0x00000900;
   dm(LCTL)=r0;      /* LBUF2: Tx, non DMA            */

   r0=@source_2;
   dm(LBUF2)=r0;     /* Tx size of DMA to the slave         */

   r0=source_2;
   dm(II4)=r0; /* Set DMA tx index to start of source buffer.  */

   r0=1;
```

Listing 9.3 Link Token Passing Example (continues)

```
   dm(IM4)=r0; /* Set DMA modify (stride) to 1        */

   r0=@source_2;
   dm(C4)=r0;  /* Set DMA count to length of data buffer.   */

   r0=0x00000b00; /* LCTL Register:32-bit data, LBUF2=tx    */
   dm(LCTL)=r0;   /* enable DMA on LBUF2.          */
            /* This will start off the DMA transfer.  */
            /* Always write LCTL after LAR.     */

   bit clr irptl LP2I;  /* clear pending Link buffer 2 interrupt  */
   bit set imask LP2I;  /* Enable Link buffer 2 interrupt      */
   bit set mode1 IRPTEN;   /* Global interrupt enable       */

   rti;



/*_____________________Link buffer 2 Interrupt Routine_____________________*/

lp2:

   if NOT FLAG0_IN jump lp2_orig_slave;   /* Test for original master   */
   nop;              /* and jump appropriately  */
   nop;



/*_________________Link buffer 2 Rx finish Interrupt Routine_____________*/

lp2_orig_master:
            /* Finish by setting up Rx without DMA    */
   r0=0x3fe3f;
   dm(LAR)=r0;    /* LAR Register:  LBUF2->port0         */

   r0=0x00000100;
   dm(LCTL)=r0;      /* LBUF2: Rx, non DMA           */

   rti;  /* This interrupt will occur only once.      */
```

Listing 9.3 Link Token Passing Example (continues)

```
/*_________________Link buffer 2 Rx Interrupt Routine_____________________*/

lp2_orig_slave:


/*_________________Allow for pulldown delay on LxACK of the
slave___________________*/


   bit clr imask LSRQI;    /* Disable Link port service request interrupt
*/

   r0=0x10;
   dm(LSRQ)=r0;      /* Unmask LSRQ lport 0 transmit request status     */

   r0=0x00000000;
   dm(LCTL)=r0;      /* LCTL: disable all LBUFs       */

disabled2:
   r0=dm(LSRQ);      /* Check to ensure that the pull down on LxACK     */
   r0=FEXT r0 BY 20:1;  /* has pulled the LxACK low.  Both the original */
   r0=pass r0;    /* slave and original master will be in sync.   */
   if NE jump disabled2;   /* The next assertion of LxACK will signify to  */
            /* the master that slave is requesting the token*/


/
*_____________________________________________________________________________*/



   bit clr imask LP2I;     /* disable Link buffer 2 interrupt     */

   r0=0x3fe3f;
   dm(LAR)=r0; /* LAR Register:  LBUF2->port 0        */
```

Listing 9.3 Link Token Passing Example (continues)

```
   r0=0x00000100;
   dm(LCTL)=r0;      /* LBUF2=rx (slave), No DMA   */

   bit clr mode1 NESTM;    /* disable interrupt nesting        */
            /* to avoid breaking sync        */

   r0=dm(LBUF2);  /* read token permission from master      */


/* The following check if the first word after DMA is token */
/* permission.  If it is not, the slave read other dummy words */
/* and continue to be slave.  If it is token permission, */
/* continue the original flow and the slave will decide if  */
/* if will accept the permissiom.   */


   r1=trw;     /* token release word      */

   r0 = r0 - r1;        /* test received word and set ALU flag */
   if NE jump second_slave_mode;   /* not token permission     */
   nop;
   nop;


/* The following 3 lines shows how to reject token      */
/* If commented-out, this slave will change to master   */
/* if uncommented, this slave will continue to be slave */

/* LCNTR=20, DO reject_token UNTIL LCE;
   reject_token: nop;
   jump second_slave_mode;
   nop;        /* delay read of incoming message      */
   nop;


master_mode:

   r0=dm(LBUF2);  /* Read 3 times to clean the     */
   r0=dm(LBUF2);  /* ex-master's LBUF.  So, ex-master    */
   r0=dm(LBUF2);  /* will release the token     */
```

Listing 9.3 Link Token Passing Example (continues)

```
/*_________________Protection to avoid two transmitting link
ports_________________*/


   bit clr imask LSRQI;    /* Disable Link port service request interrupt
*/

   r0=0x10;
   dm(LSRQ)=r0;      /* Unmask LSRQ lport 0 transmit request status     */

   r0=0x00000000;
   dm(LCTL)=r0;      /* LCTL: disable all LBUFs       */

disabled3:
   r0=dm(LSRQ);      /* Check to ensure that the pull down on LxACK     */
   r0=FEXT r0 BY 20:1;  /* has pulled the LxACK low.  Both the original */
   r0=pass r0;    /* slave and original master will be in sync.   */
   if NE jump disabled3;   /* The next assertion of LxACK will signify the */
            /* master has become the slave.        */
slave_enabled:
   r0=dm(LSRQ);      /* Check to ensure that the original master has    */
   r0=FEXT r0 BY 20:1;  /* become the slave by observing that the */
   r0=pass r0;    /* assertion of LxACK has generated an LxRRQ */
   if EQ jump slave_enabled;


/
*_____________________________________________________________________________*/



   r0=0x3f1ff;
   dm(LAR)=r0;    /* LAR Register:  LBUF3->port 0        */

   r0=0x00009000;
   dm(LCTL)=r0;      /* LBUF3=tx,  no DMA       */

   r0=@source_3;     /* Tx DMA size          */
   dm(LBUF3)=r0;  /* send DMA size across    */
```

Listing 9.3 Link Token Passing Example (continues)

```
   r0=0xc0;
wait: r1=dm(LCOM);      /* check if LBUF3 is empty       */
   r0=r0 AND r1;
   if NE jump wait;  /* if DMA size not thru, wait       */
   nop;

   r0=source_3;
   dm(II5)=r0;    /* Tx DMA setup         */

   r0=1;
   dm(IM5)=r0;    /* Set modify to 1            */

   r0=@source_3;
   dm(C5)=r0;  /* Set DMA count to length of data buffer */

   r0=0x0000b000; /* LCTL Register:32-bit data,   LBUF3=Tx  */
   dm(LCTL)=r0;   /* enable DMA on LBUF3           */
            /* This will start off the DMA transfer      */
            /* Always write LCTL after LAR      */

   bit clr irptl LP3I;     /* clear pending Link buffer 3 interrupt.    */
   bit set imask LP3I;     /* Enable Link buffer 3 interrupt      */

   rti;


second_slave_mode:

   r0=dm(LBUF2);           /* read 3 times to clean the     */
   r0=dm(LBUF2);           /* ex-master's LBUF.  So, ex-master    */
   r0=dm(LBUF2);           /* will release the token     */

   r0=0x3f1ff;
   dm(LAR)=r0;    /* LAR Register:  LBUF3->port 0     */

   r0=0x00001000;
   dm(LCTL)=r0;      /* enable LBUF3 Rx, No DMA       */

   r1=dm(LBUF3);     /* read new DMA size          */
```

Listing 9.3 Link Token Passing Example (continues)

```
   r0=destination_2;
   dm(II5)=r0; /* Set DMA rx index to start of dest buffer  */

   r0=1;
   dm(IM5)=r0;    /* Set modify to 1            */

   r0=@destination_2;
   dm(C5)=r1;     /* real DMA Rx size should be got from master   */

   r0=0x00003000;    /* LCTL Register:32-bit data,      LBUF3=Rx  */
   dm(LCTL)=r0;   /* enable DMA on LBUF3          */
            /* This will start off the DMA transfer      */
            /* Always write LCTL after LAR         */

   bit clr irptl LP3I;     /* clear pending Link buffer 3 interrupt.    */
   bit set imask LP3I;     /* Enable Link buffer 3 interrupt      */

   rti;



/*_________________Start of Original Slave Routine_____________________*/

start_as_slave:

   r0=destination_1;
   dm(II4)=r0; /* Set DMA rx index to start of dest buffer  */

   r0=1;
   dm(IM4)=r0;    /* Set DMA modify (stride) to 1        */

   r0=@destination_1;
   dm(C4)=r0;  /* real DMA Rx size should be from master */

   r0=0xc000;     /* LCOM Register: 2x rate,       */
   dm(LCOM)=r0;   /* note:use r0=0x10000 on rev. 0.X silicon   */
            /* original :  0x0000c000        */

   r0=0x3fe3f;    /* LAR Register:  LBUF2->port 0        */
   dm(LAR)=r0; /* All others inactive          */
```

Listing 9.3 Link Token Passing Example (continues)

```
   bit set imask LP2I;     /* Enable Link buffer 2 interrupt      */
   bit set mode1 IRPTEN;   /* Global interrupt enable       */

   r0=0x00000300;    /* LCTL Register:32-bit data,   LBUF2=Rx  */
   dm(LCTL)=r0;   /* enable DMA on LBUF2           */
            /* This will start off the DMA transfer      */
            /* Always write LCTL after LAR      */


wait_2:   idle;         /* Wait for Link buffer 2 interrupt    */
          jump wait_2;     /* Will end up here after entire DMA complete
*/
     nop;         /* All slave interrupts will come back to here  */
     nop;

.ENDSEG;
```

Listing 9.3 Link Token Passing Example

9 Link Ports

Serial Ports 10

10.1 OVERVIEW

The ADSP-2106x has two independent, synchronous serial ports, SPORT0 and SPORT1, that provide an I/O interface to a wide variety of peripheral devices. Each serial port has its own set of control registers and data buffers. With a range of clock and frame synchronization options, the SPORTs allow a variety of serial communication protocols and provide a glueless hardware interface to many industry-standard data converters and CODECs.

The serial ports can operate at the full clock rate of the processor, providing each with a maximum data rate of *n* Mbit/s, where *n* equals the processor clock frequency. Independent transmit and receive functions provide greater flexibility for serial communications. Serial port data can be automatically transferred to and from on-chip memory using DMA block transfers. Each of the serial ports offers a TDM (time division multiplexed) multichannel mode.

Serial port clocks and frame syncs can be internally generated by the ADSP-2106x or received from an external source. The serial ports can operate with little-endian or big-endian transmission formats, with word lengths selectable from 3 to 32 bits. They offer selectable synchronization and transmit modes as well as optional μ-law or A-law companding in hardware.

The serial ports offer the following features and capabilities:

- Independent transmit and receive functions.
- Can transfer data words up to 32 bits in length, either MSB-first or LSB-first.
- Double-buffering of data—both receive and transmit functions have a data buffer register as well as a shift register; the double-buffering provides additional time to service the SPORT.
- A-law and μ-law hardware companding on transmitted and received words.
- Serial clock and frame sync signals can be generated internally, in a wide range of frequencies, or input from an external source.
- Interrupt-driven, single-word transfers to and from on-chip memory controlled by the ADSP-2106x core.

10 Serial Ports

- DMA transfers to and from on-chip memory—each SPORT can automatically receive and / or transmit an entire block of data.
- Chained DMA operations of multiple data blocks.
- Multichannel mode for TDM interfaces—each SPORT can receive and transmit data selectively from channels of a time-division-multiplexed serial bitstream; this mode can be useful for T1 interfaces.

Table 10.1 shows the pins of each serial port:

	SPORT0	*SPORT1*
Function	*Pins*	*Pins*
Transmit Data	DT0	DT1
Transmit Clock	TCLK0	TCLK1
Transmit Frame Sync	TFS0	TFS1
Receive Data	DR0	DR1
Receive Clock	RCLK0	RCLK1
Receive Frame Sync	RFS0	RFS1

Table 10.1 Serial Port Pins

A serial port 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.

Serial communications are synchronized to a clock signal—every data bit must be accompanied by a clock pulse. Each serial port can generate or receive its own transmit clock signal (TCLK) and receive clock signal (RCLK). Internally-generated serial clock frequencies are configured in the TDIVx and RDIVx registers.

In addition to the serial clock signal, data may be signalled by a frame synchronization signal. The framing signal can occur either at the beginning of an individual word or at the beginning of a block of words. The configuration of frame synch signals depends upon the type of serial device connected to the ADSP-2106x. Each serial port can generate or receive its own transmit frame sync signal (TFS) and receive frame sync signal (RFS). Internally-generated frame sync frequencies are configured in the TDIVx and RDIVx registers.

Figure 10.1 shows a block diagram of a serial port. Data to be transmitted is written to the TX buffer. The data is (optionally) compressed in hardware, then automatically transferred to the transmit shift register. The data in the shift register is then shifted out on the SPORT's DT pin, synchronous to the TCLK transmit clock. If

framing signals are used, the TFS signal indicates the start of the serial word transmission. The DT pin is always driven, i.e. not tristated, if the serial port is enabled (SPEN=1 in the STCTLx control register), unless it is in multichannel mode and an inactive time slot occurs. (See Section 10.7, “Multichannel Operation.”)

The receive portion of the SPORT shifts in data from the DR pin, synchronous to the RCLK receive clock. If framing signals are used, the RFS signal indicates the beginning of the serial word being received. When an entire word is shifted in, the data is (optionally) expanded, then automatically transferred to the RX buffer.

Note: The ADSP-2106x SPORTs are not UARTs and cannot be used to communicate with an RS-232 device or any other asynchronous communications protocol. One way to implement RS-232-compatible communications with the ADSP-2106x, however, is to use two of the FLAG pins as asynchronous data receive and transmit signals. For an example of how to do this, see the *Software UART* chapter of *Digital Signal Processing Applications Using The ADSP-2100 Family, Volume 2.*

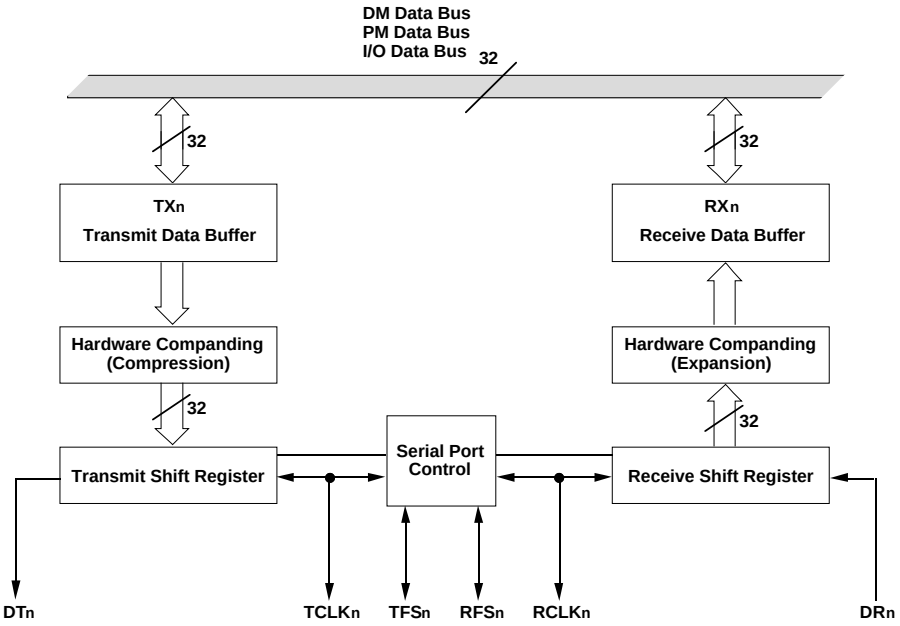


Figure 10.1 Serial Port Block Diagram

10.1.1 SPORT Interrupts

Each serial port has a transmit DMA interrupt and a receive DMA interrupt. When serial port DMA is not enabled, the interrupts occur for each data word transmitted and received. The priority of the serial port interrupts is shown in Table 10.2.

*Interrupt Name**	*Interrupt*	
SPR0I	SPORT0 Receive DMA Channel	***Highest Priority***
SPR1I	SPORT1 Receive DMA Channel	
SPT0I	SPORT0 Transmit DMA Channel	
SPT1I	SPORT1 Transmit DMA Channel	***Lowest Priority***

Table 10.2 SPORT Interrupts

* These names are defined in the `def21060.h` include file supplied with the ADSP-21000 Family Development Software.

SPORT Interrupts occur on the second system clock (CLKIN) after the last bit of the serial word is latched in or driven out.

10.2 SPORT $\overline{\text{RESET}}$

There are two ways to reset the serial ports: a hardware reset using the $\overline{\text{RESET}}$ pin of the processor, and a software reset accomplished by clearing the serial port's enable bit (SPEN) in the STCTLx and SRCTLx control registers. Each method has a different effect on the serial port.

A hardware reset disables the serial ports by clearing the STCTLx and SRCTLx control registers (including the SPEN enable bits) and the TDIVx and RDIVx frame sync divisor registers. Any ongoing operations are aborted.

A software reset of the SPEN enable bit(s) disables the serial port(s) and aborts any ongoing operations. Status bits are also cleared.

The serial ports will be ready to start transmitting or receiving data two CLKIN cycles after they are enabled (in the STCTLx or SRCTLx control register). No serial clocks will be lost from this point on.

10.3 SPORT CONTROL REGISTERS & DATA BUFFERS

The registers used to control and configure the serial ports are part of the IOP register set. Each SPORT has its own set of the following control registers and data buffers:

Register *Name**	*Function*
STCTLx	SPORT Transmit Control Register
TXx	Transmit Data Buffer
TDIVx	Transmit Clock & Frame Sync Divisors
MTCSx	Multichannel Transmit Select
MTCCSx	Multichannel Transmit Compand Select
SRCTLx	SPORT Receive Control Register
RXx	Receive Data Buffer
RDIVx	Receive Clock & Frame Sync Divisors
MRCSx	Multichannel Receive Select
MRCCSx	Multichannel Receive Compand Select
SPATHx	SPORT Path Length (for mesh multiprocessing)
KEYWDx	SPORT Receive Comparison**
KEYMASKx	SPORT Receive Comparison Mask**

* x = 0, 1
** ADSP-21061 only

Table 10.3 (on the next page) shows the memory-mapped address and reset initialization value of each SPORT register. All of the registers are 32 bits wide, except for the 16-bit SPATHx register and location 0x00FF. (Note that for standard, non-mesh-multiprocessing operation of the serial ports, the SPATHx register and location 0x00FF must remain equal to the reset initialization value, 0x0001.)

The SPORT control registers are programmed by writing to the appropriate address in memory. The symbolic names of the registers and individual control bits can be used in ADSP-2106x programs—the `#define` definitions for these symbols are contained in the file `def21060.h` which is provided in the INCLUDE directory of the ADSP-21000 Family Development Software. The `def21060.h` file is shown in the *Control/Status Registers* appendix of this manual. All control and status bits in the SPORT registers are active high unless otherwise noted.

Because the SPORT registers are memory-mapped they cannot be written with data coming directly from memory. They must instead be written

Memory Address	*Register Name*	*Initialization After RESET*	*Description*
0x00E0	STCTL0	0x0000 0000	SPORT0 Transmit Control Register
0x00E1	SRCTL0	0x0000 0000	SPORT0 Receive Control Register
0x00E2	TX0	ni	SPORT0 Transmit Data Buffer
0x00E3	RX0	ni	SPORT0 Receive Data Buffer
0x00E4	TDIV0	ni	SPORT0 Transmit Divisor
0x00E5			*reserved*
0x00E6	RDIV0	ni	SPORT0 Receive Divisor
0x00E7			*reserved*
0x00E8	MTCS0	ni	SPORT0 Multichannel Transmit Select
0x00E9	MRCS0	ni	SPORT0 Multichannel Receive Select
0x00EA	MTCCS0	ni	SPORT0 Multichannel Transmit Compand Select
0x00EB	MRCCS0	ni	SPORT0 Multichannel Receive Compand Select
0x00EC	KEYWD	ni	SPORT0 Receive Comparison (ADSP-21061)
0x00ED	KEYMASK	ni	SPORT0 Receive Comparison Mask (ADSP-21061)
0x00EE	SPATH0	0x0001	SPORT0 Path Length (Mesh Multiprocessing)
0x00EF		0x0001	*reserved*
0x00F0	STCTL1	0x0000 0000	SPORT1 Transmit Control Register
0x00F1	SRCTL1	0x0000 0000	SPORT1 Receive Control Register
0x00F2	TX1	ni	SPORT1 Transmit Data Buffer
0x00F3	RX1	ni	SPORT1 Receive Data Buffer
0x00F4	TDIV1	ni	SPORT1 Transmit Divisor
0x00F5			*reserved*
0x00F6	RDIV1	ni	SPORT1 Receive Divisor
0x00F7			*reserved*
0x00F8	MTCS1	ni	SPORT1 Multichannel Transmit Select
0x00F9	MRCS1	ni	SPORT1 Multichannel Receive Select
0x00FA	MTCCS1	ni	SPORT1 Multichannel Transmit Compand Select
0x00FB	MRCCS1	ni	SPORT1 Multichannel Receive Compand Select
0x00FC	KEYWD	ni	SPORT1 Receive Comparison (ADSP-21061)
0x00FD	KEYMASK	ni	SPORT1 Receive Comparison Mask (ADSP-21061)
0x00FE	SPATH1	0x0001	SPORT1 Path Length (Mesh Multiprocessing)
0x00FF		0x0001	*reserved*

ni= not initialized

Table 10.3 SPORT Register Addresses & Initialization

from (or read into) ADSP-2106x core registers, usually one of the general-purpose universal registers of the register file (R15–R0). The SPORT control registers can also be written or read by external devices, i.e. another ADSP-2106x or a host processor, to set up a serial port DMA operation, for example.

10.3.1 Register Writes & Effect Latency

SPORT register writes are internally completed at the end of the same CLKIN cycle in which they occur. The register will therefore read back the newly written value on the very next cycle. When a read of one of the STCTLx or SRCTLx control registers is immediately followed by a write to that register, however, the write may take two cycles to complete.

After a write to a SPORT register, control and mode bit changes generally take effect in the second CLKIN cycle after the write is completed. The serial ports will be ready to start transmitting or receiving two CLKIN cycles after they are enabled (in the STCTLx or SRCTLx control register). No serial clocks will be lost from this point on.

10.3.2 Transmit & Receive Data Buffers (TX, RX)

TX0 and TX1 are the transmit data buffers for SPORT0 and SPORT1. They are 32-bit buffers 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 ADSP-2106x core. RX0 and RX1 are the receive data buffers for SPORT0 and SPORT1. They are 32-bit buffers which are automatically loaded from the receive shifter when a complete word has been received. Word lengths of less than 32 bits are right-justified in the receive and transmit buffers.

The TX buffers act like a two-location FIFO because they have a data register plus an output shift register (see Figure 10.1); two 32-bit words may be stored in TX at any one time. When the TX buffer is loaded and any previous word has been transmitted, the buffer contents are automatically loaded into the output shifter. An interrupt is generated when the output shifter has been loaded, signifying that the TX buffer is ready to accept the next word (i.e. the TX buffer is “not full”). This interrupt will not occur if serial port DMA is enabled or if the corresponding mask bit in the IMASK register is set.

The transmit underflow status bit (TUVF) will be set in the transmit control register when a transmit frame synch occurs and no new data has been loaded into TX. The TUVF status bit is “sticky” and is only cleared by disabling the serial port.

The RX buffers act like a three-location FIFO because they have two data registers plus an input shift register. Two complete 32-bit words can be stored in RX while a third word is being shifted in. The third word will overwrite the second if the first word has not been read out (by the ADSP-2106x core or the DMA controller). When this happens, the receive overflow status bit (ROVF) will be set in the receive control register. Almost three complete words can be received without the RX buffer being read before overflow occurs. The overflow status is generated on the last bit of third word. The ROVF status bit is “sticky” and is only cleared by disabling the serial port.

An interrupt is generated when the RX buffer has been loaded with a received word (i.e. the RX buffer is “not empty”). This interrupt will be masked out if serial port DMA is enabled or if the corresponding bit in the IMASK register is set.

10.3.2.1 Reading & Writing RX, TX

If your ADSP-2106x program causes the core processor to attempt a read from an empty RX buffer or a write to a full TX buffer, the access is delayed until the buffer is accessed by the external I/O device. (This delay is called a core processor hang.) If it is not known whether the core processor can access the RX or TX buffer without a hang, the buffer’s *full or empty* status should be read first (in STCTLx or SRCTLx) to determine if the access can be made. To prevent this type of hang condition from occurring, the BHD (Buffer Hang Disable) bit can be set in the SYSCON register.

The status bits in STCTLx and SRCTLx are updated during reads and writes from the core processor even when the serial port is disabled.

The serial port should be disabled when writing to the RX buffer or reading from the TX buffer.

10.3.3 Transmit & Receive Control Registers (STCTL, SRCTL)

The main control registers for each serial port are the transmit control register, STCTLx, and the receive control register, SRCTLx. These registers are defined in Tables 10.4 and 10.5, and are pictured in Figures 10.2 and 10.3. When changing operating modes, a serial port control register should be cleared (i.e. written with all zeros) before the new mode is written to the register.

The Transmit Underflow Status bit (TUVF) is set whenever the TFS signal occurs (from either external or internal source) while the TX buffer is empty. The internally generated TFS may be suppressed whenever TX is empty by clearing the DITFS control bit (DITFS=0).

When DITFS=0, the default, the transmit frame sync signal (TFS) is dependent upon new data being present in the TX buffer—the TFS signal will only be 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 TX buffer regardless. Serial port DMA will typically keep the TX buffer full, and when the DMA operation is complete the last word in TX will be continuously transmitted.

The TXS status bits indicate whether the TX buffer is full (11), empty (00), or partially full (10). To test for space in TX, therefore, test for TXS0 (bit 30) equal to zero. To test for the presence of any data in TX, test for TXS1 (bit 31) equal to one.

Bit(s)	*Name*	*Definition*
0	SPEN*	SPORT Enable
1-2	DTYPE	Data Type (data format, companding)
3	SENDN	Serial Word Endian (1=LSB first)
4-8	SLEN	Serial Word Length – 1
9	PACK	Data Word Unpacking (32-bit to 16-bit)
10	ICLK*	Internally Generated Transmit Clock
11	–	*reserved*
12	CKRE	Data, Frame Sync Sampling on Clock Rising Edge
13	TFSR*	Transmit Frame Sync Required
14	ITFS*	Internally Generated TFS
15	DITFS	Data-Independent TFS
16	LTFS	Active Low TFS
17	LAFS*	Late TFS
18	SDEN	SPORT Transmit DMA Enable
19	SCHEN	SPORT Transmit DMA Chaining Enable
20-23	MFD	Multichannel Frame Delay
24-28	CHNL**	Current Channel Status (read-only)
29	TUVF**	Transmit Underflow Status (sticky, read-only)
30-31	TXS**	TX Buffer Status (read-only) 11=full, 00=empty, 10=partially full

Table 10.4 STCTLx Transmit Control Register Bits

* Must be set to 0 for multichannel operation.

** Status bits are read-only. They are cleared by disabling the serial port (setting SPEN=0). TXS may subsequently change state if the data is read or written by the ADSP-2106x core while the SPORT is disabled.

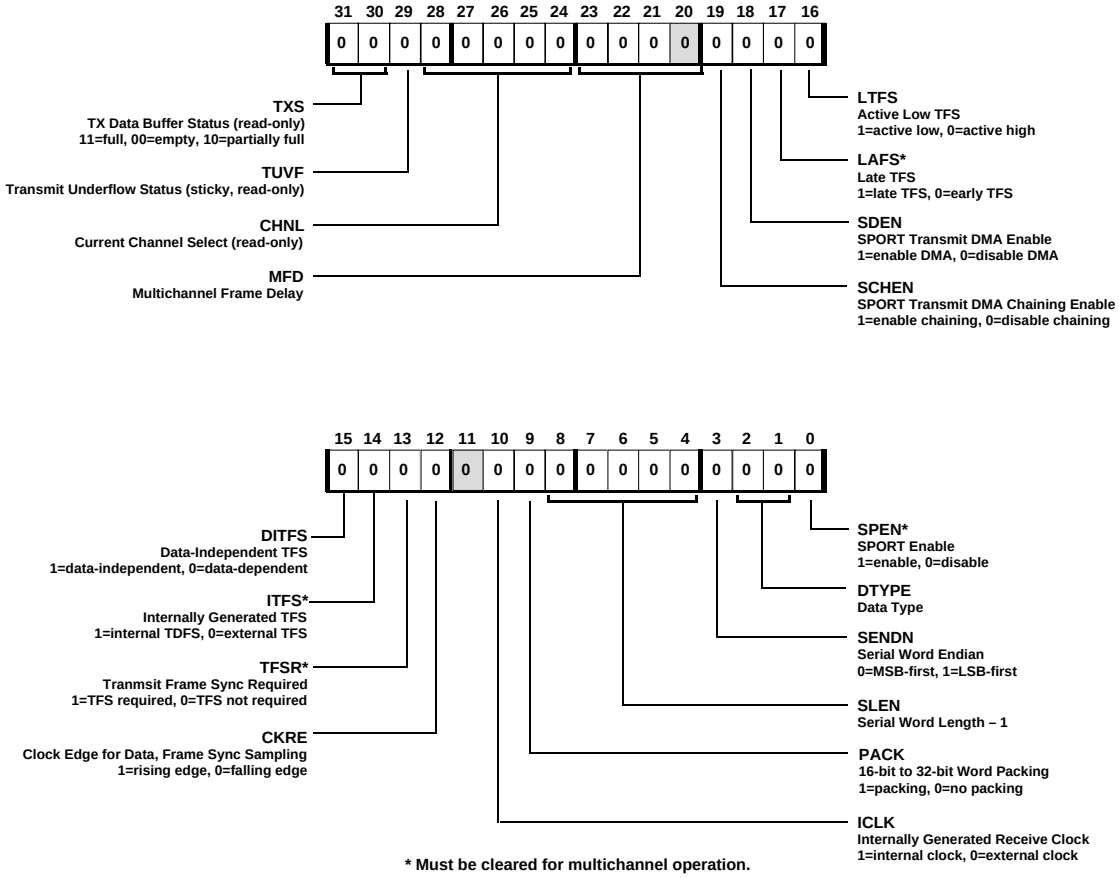


Figure 10.2 STCTL0, STCTL1 Transmit Control Registers

Bit(s)	*Name*	*Definition*
0	SPEN*	SPORT Enable
1-2	DTYPE	Data Type (data format, companding)
3	SENDN	Serial Word Endian (1=LSB first)
4-8	SLEN	Serial Word Length – 1
9	PACK	Data Word Packing (16-bit to 32-bit)
10	ICLK	Internally Generated Receive Clock
11	–	*reserved*
12	CKRE	Data, Frame Sync Sampling on Clock Rising Edge
13	RFSR*	Receive Frame Sync Required
14	IRFS	Internally Generated RFS
15	–	*reserved*
16	LRFS	Active Low RFS
17	LAFS*	Late RFS
18	SDEN	SPORT Receive DMA Enable
19	SCHEN	SPORT Receive DMA Chaining Enable
20	–	*reserved*
21	D2DMA*	2-Dimensional DMA Array Enable
22	SPL*	SPORT Loopback (test)
23	MCE	Multichannel Enable
24-28	NCH	Number of Channels – 1 (multichannel operation)
29	ROVF**	Receive Overflow Status (sticky, read-only)
30-31	RXS**	RX Buffer Status (read-only) 11=full, 00=empty, 10=partially full

Table 10.5 SRCTLx Receive Control Register Bits

* Must be cleared for multichannel operation.

** Status bits are read-only. They are cleared by disabling the serial port (setting SPEN=0). RXS may subsequently change state if the data is read or written by the ADSP-2106x core while the SPORT is disabled.

The RXS status bits indicate whether the RX buffer is full (11), empty (00), or partially full (10). To test for space in RX, therefore, test for RXS0 (bit 30) equal to zero. To test for the presence of any data in RX, test for RXS1 (bit 31) equal to one.

The Receive Overflow Status bit (ROVF) is set whenever new data is received while the RX buffer is full; the new data overwrites the existing data.

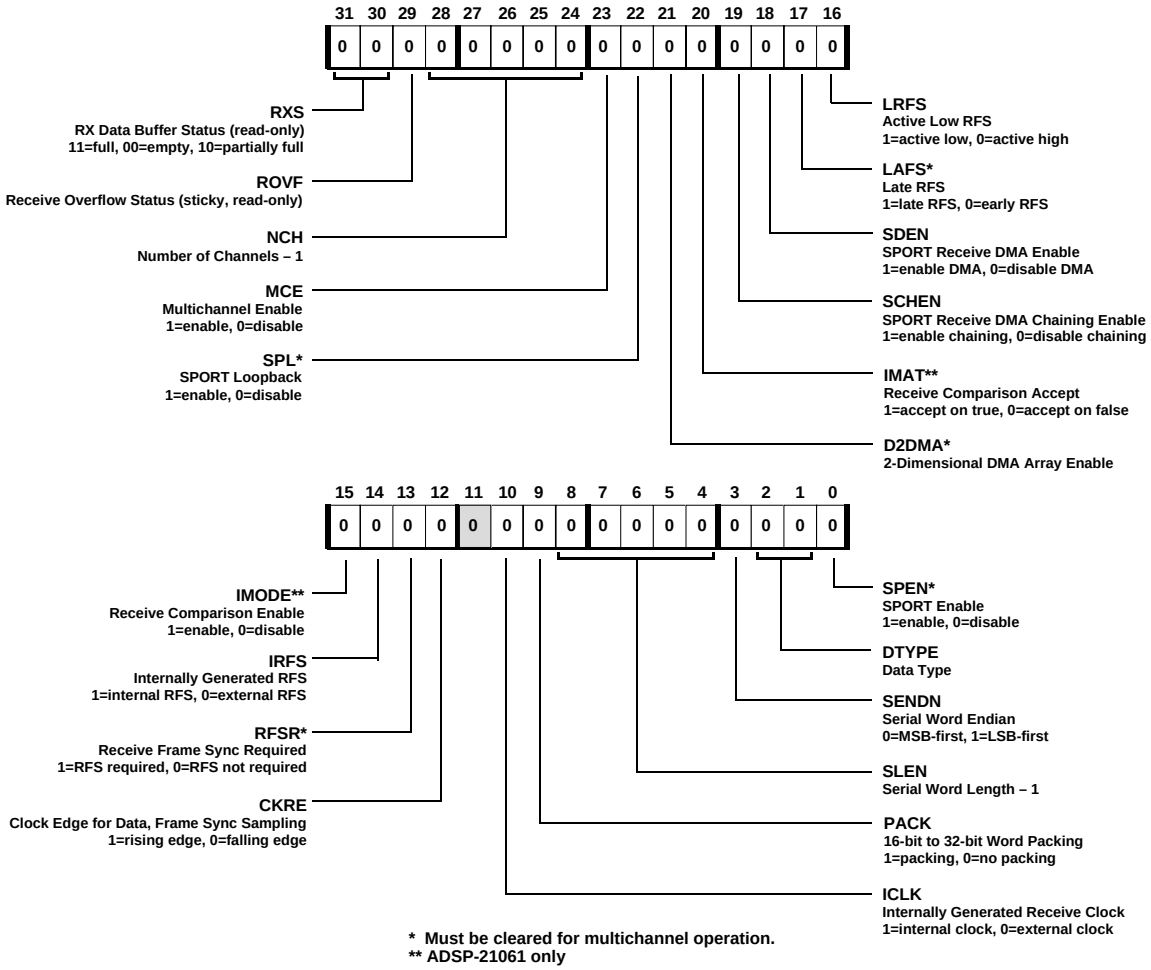


Figure 10.3 SRCTL0, SRCTL1 Receive Control Registers

10.3.4 Clock & Frame Sync Frequencies (TDIV, RDIV)

The TDIVx and RDIVx registers contain divisor values which determine the frequencies for internally generated clocks and frame syncs. These registers are defined in Tables 10.6 and 10.7, and are pictured in Figures 10.4 and 10.5.

Bits	*Name*	*Definition*
15-0	TCLKDIV	Transmit Clock Divisor
31-16	TFSDIV	Transmit Frame Sync Divisor

Table 10.6 Transmit Divisor Register Bit Fields

Bits	*Name*	*Definition*
15-0	RCLKDIV	Receive Clock Divisor
31-16	RFSDIV	Receive Frame Sync Divisor

Table 10.7 Receive Divisor Register Bit Fields

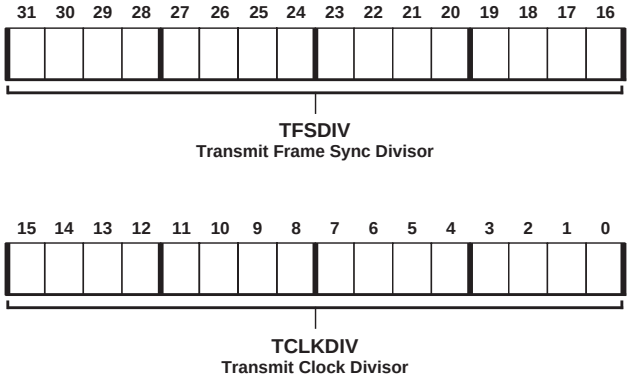


Figure 10.4 TDIV0, TDIV1 Transmit Divisor Registers

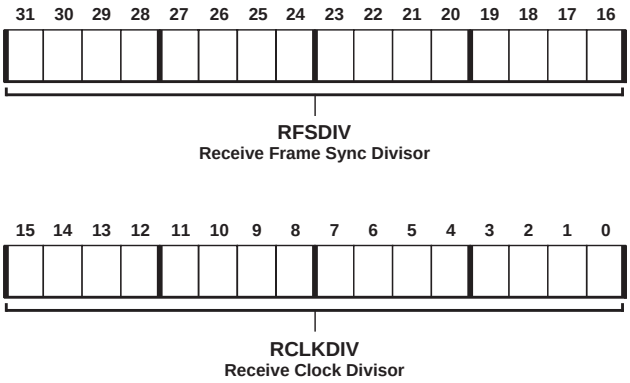


Figure 10.5 RDIV0, RDIV1 Receive Divisor Registers

TCLKDIV and RCLKDIV specify how many times the ADSP-2106x system clock (CLKIN) is divided to generate the transmit and receive clocks. The divisor is a 16-bit value, allowing a wide range of serial clock rates. The following equation is used to calculate the serial clock frequency:

$$\text{serial clock frequency} = \frac{f_{CLKIN}}{(\text{xCLKDIV} + 1)}$$

The maximum serial clock frequency is equal to the ADSP-2106x system clock frequency, which occurs when xCLKDIV is set to zero.

Use the following equation to determine the value of xCLKDIV to use, given the CLKIN frequency and desired serial clock frequency:

$$\text{xCLKDIV} = \frac{f_{CLKIN}}{\text{serial clock frequency}} - 1$$

TFSDIV and RFSDIV specify how many transmit or receive clock cycles are counted before generating a TFS or RFS pulse (when the frame synch 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 formula for the number of cycles between frame synch pulses is:

# of serial clock cycles between frame sync assertions = xFSDIV + 1

Use the following equation to determine the value of xFSDIV to use, given the serial clock frequency and desired frame sync frequency:

$$xFSDIV = \frac{\text{serial clock frequency}}{\text{frame sync frequency}} - 1$$

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), as this may cause an external device to abort the current operation or cause other unpredictable results. If the serial port 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 serial port must be enabled for this mode of operation to work.

10.3.4.1 Maximum Clock Rate Restrictions

Caution should be exercised when operating with externally generated transmit clocks near the frequency of the ADSP-2106x system clock. There is a delay between when the clock arrives at the TCLKx pin and when data is output—this delay may limit the receiver's speed of operation. Refer to the data sheet for exact timing specifications. For reliable operation, it is recommended that full-speed serial clocks only be used when *receiving with an externally generated clock and externally generated frame sync (ICLK=0, IRFS=0).*

Externally-generated *late* transmit frame syncs also experience a delay from when they arrive to when data is output—this can also limit the maximum serial clock speed. Refer to the data sheet for exact timing specifications.

The serial ports handle word lengths of 3 to 32 bits, but transmitting or receiving words smaller than 7 bits at the full clock rate of the ADSP-2106x may cause incorrect operation when DMA chaining is enabled. Chaining disables the ADSP-2106x's internal I/O bus for several cycles while the new TCB parameters are being loaded. Receive data may be lost (i.e. overwritten) during this period.

10.4 DATA WORD FORMATS

The format of the data words transmitted over the serial ports is configured by the DTYPE, SENDN, SLEN, and PACK bits of the STCTLx and SRCTLx control registers.

10.4.1 Word Length

The serial ports handle word lengths of 3 to 32 bits. The word length is configured in the 5-bit SLEN field in the STCTLx and SRCTLx control registers. The value of SLEN is equal to the word length minus one:

SLEN = Serial Word Length – 1

The SLEN value should not be set to zero or one. Words smaller than 32 bits are right-justified in the RX and TX buffers, residing in the least significant bit positions.

Transmitting or receiving words smaller than 7 bits at the full clock rate of the ADSP-2106x may cause incorrect operation when DMA chaining is enabled. Chaining disables the ADSP-2106x's internal I/O bus for several cycles while the new TCB parameters are being loaded. Receive data may be lost (i.e. overwritten) during this period.

10.4.2 Endian Format

Endian format determines whether the serial word is transmitted MSB-first or LSB-first. Endian format is selected by the SENDN bit in the STCTLx and SRCTLx control registers. When SENDN=0, serial words are transmitted (or received) MSB-first. When SENDN=1, serial words are transmitted (or received) LSB-first.

10.4.3 Data Packing & Unpacking

Received data words of 16 bits or less may be packed into 32-bit words, and 32-bit words being transmitted may be unpacked into 16-bit words. Word packing and unpacking is selected by the PACK bit in the SRCTLx and STCTLx control registers.

When PACK=1 in the receive control register (SRCTLx), two successive words received are packed into a single 32-bit word.

When PACK=1 in the transmit control register (STCTLx), each 32-bit word is unpacked and transmitted as two 16-bit words.

The first 16-bit (or smaller) word is right-justified in bits 15-0 of the packed word, and the second 16-bit (or smaller) word is right-justified in bits 31-16. This applies for both receive (packing) and transmit (unpacking) operations. Companding may be used when word packing or unpacking is being used.

When serial port data packing is enabled, the transmit and receive interrupts are generated for the 32-bit packed words, not for each 16-bit word.

Hint: When 16-bit received data is packed into 32-bit words and stored in normal word space in ADSP-2106x internal memory, the 16-bit words can be read or written with short word space addresses.

10.4.4 Data Type

The DTYPE field of the STCTLx and SRCTLx control registers specifies one of four data formats (*for non-multichannel operation*):

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 RX and TX buffers. (TX data words are not actually zero-filled or sign-extended, since only the significant bits are transmitted.)

For multichannel operation, the companding selection and MSB-fill selection is independent:

DTYPE	*Data Formatting*
x0	Right-justify, zero-fill unused MSBs
x1	Right-justify, sign-extend into unused MSBs
0x	Compand using μ-law
1x	Compand using A-law

Linear transfers will occur if the channel is active but companding is not selected for that channel. Companded transfers will occur if the channel is active and companding is selected for that channel. The multichannel compand select registers, MTCCSx and MRCCSx, are used to specify which transmit and receive channels are companded. See “Channel Selection Registers” in the “Multichannel Operation” section of this chapter.

Transmit sign extension is selected by bit 0 of DTYPE in the STCTLx register and is common to all transmit channels. Receive sign extension is selected by bit 0 of DTYPE in the SRCTLx register and is common to all receive channels. If bit 0 of DTYPE is set, sign extension will occur on selected channels that do not have companding selected. If this bit is not set, the word will contain 0s in the MSBs.

10.4.5 Companding

Companding (*com*pressing / ex*pand*ing) is the process of logarithmically encoding and decoding data to minimize the number of bits that must be sent. The ADSP-2106x serial ports support the two most widely used companding algorithms, A-law and μ-law, performed according to the CCITT G.711 specification. The type of companding can be selected independently for each SPORT. Companding is selected by the DTYPE field of the STCTLx and SRCTLx control registers.

When companding is enabled, the data in the RX0 or RX1 buffer is the right-justified, sign-extended expanded value of the eight LSBs received. Likewise, a write to TX0 or TX1 causes the 32-bit value to be compressed to eight LSBs (sign-extended to the width of the transmit word) before it is transmitted. If the 32-bit value is greater than the 13-bit A-law or 14-bit μ-law maximum, it is automatically compressed to the maximum value.

Because the values in the TX and RX buffers are actually companded in-place, the companding hardware can be used without transmitting (or receiving) any data, for example during testing or debugging. This operation requires a single cycle of overhead, as described below. To compand data in-place, without transmitting, the following sequence of operations should be used:

1. Enable companding in the DTYPE field of the STCTLx transmit control register.
2. Write a 32-bit data word to TX. *(The companding is calculated in this cycle.)*
3. Wait one cycle. A NOP instruction can be used to do this; if a NOP is not inserted, the ADSP-2106x core will be held off for one cycle anyway. *(This allows the serial port companding hardware to reload TX with the companded value.)*
4. Read the 8-bit companded value from TX.

To expand data in-place, the same sequence of operations is used but with RX rather than TX. When expanding data in this way, be sure that the serial word length (SLEN) is set appropriately in the SRCTLx control register.

With companding enabled, interfacing the ADSP-2106x serial port to a codec requires little additional programming effort. If companding is not selected, there are two formats available for received data words of fewer than 32 bits: one that fills unused MSBs with zeros, and another that sign-extends the MSB into the unused bits (see the previous section, “Data Type”).

10.5 CLOCK SIGNAL OPTIONS

Each serial port has a transmit clock signal (TCLKx) and a receive clock signal (RCLKx). The clock signals are configured by the ICLK and CKRE bits of the STCTLx and SRCTLx control registers. Serial clock frequency is configured in the TDIVx and RDIVx registers.

The receive clock pin may be tied to the transmit clock if a single clock is desired for both input and output.

10.5.1 Internal vs. External Clocks

Both transmit and receive clocks can be independently generated internally or input from an external source. The ICLK bit of the STCTLx and SRCTLx control registers determines the clock source.

When ICLK=1, the clock signal is generated internally by the ADSP-2106x and the TCLKx or RCLKx pins will be an output. The clock frequency is determined by the value of the serial clock divisor (TCLKDIV or RCLKDIV) in the TDIVx or RDIVx registers.

When ICLK=0, the clock signal is accepted as an input on the TCLKx or RCLKx pins, and the serial clock divisors in the TDIVx/RDIVx registers are ignored. The externally generated serial clock need not be synchronous with the ADSP-2106x system clock.

10.6 FRAME SYNC OPTIONS

Framing signals indicate the beginning of each serial word transfer. The framing signals for each serial port are TFS (transmit frame synchronization) and RFS (receive frame synchronization). A variety of framing options are available; these options are configured in the serial port control registers. The TFS and RFS signals of a serial port are independent and are separately configured in the control registers.

10.6.1 Framed vs. Unframed

The use of frame sync signals is optional in serial port 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 the STCTLx and SRCTLx control registers.

When TFSR=1 or RFSR=1, a frame sync signal is required for every data word. To allow continuous transmitting from the ADSP-2106x, each new data word must be loaded into the TX buffer before the previous word is shifted out and transmitted. (See “Data-Independent Frame Syncs” in this chapter.)

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. (**Caution:** 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 10.6 illustrates framed serial transfers, which have the following characteristics:

- TFSR and RFSR bits in STCTLx, SRCTLx control registers determine framed or unframed mode.
- Framed mode requires a framing signal for every word. Unframed mode ignores framing signal after first word.
- Unframed mode is appropriate for continuous reception.
- Active-low or active-high frame syncs selected with LTFS and LRFS bits of STCTLx, SRCTLx control registers.

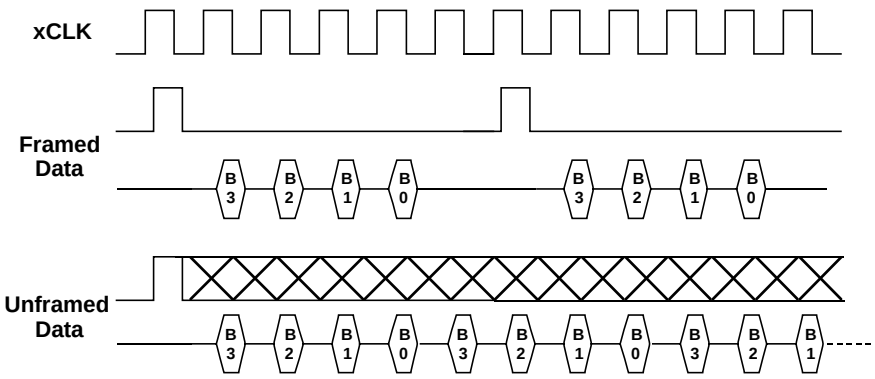


Figure 10.6 Framed vs. Unframed Data

10.6.2 Internal vs. External Frame Syncs

Both transmit and receive frame syncs can be independently generated internally or input from an external source. The ITFS and IRFS bits of the STCTLx and SRCTLx control registers determine the frame sync source.

When ITFS=1 or IRFS=1, the corresponding frame sync signal is generated internally by the ADSP-2106x and the TFSx pin or RFSx pin will be an output. The frequency of the frame sync signal is determined by the value of the frame sync divisor (TFSDIV or RFSDIV) in the TDIVx or RDIVx registers.

When ITFS=0 or IRFS=0, the corresponding frame sync signal is accepted as an input on the TFSx pin or RFSx pins, and the frame sync divisors in the TDIVx/RDIVx registers are ignored.

All of the various frame sync options are available whether the signal is generated internally or externally.

10.6.3 Active Low vs. Active High Frame Syncs

Frame sync signals may be either active high or active low (i.e. inverted). The LTFS and LRFS bits of the STCTLx and SRCTLx control registers determine the frame syncs' logic level.

When LTFS=0 or LRFS=0, the corresponding frame sync signal will be active high.

When LTFS=1 or LRFS=1, the corresponding frame sync signal will be active low.

Active high frame syncs are the default; the LTFS and LRFS bits are initialized to 0 after a processor reset.

10.6.4 Sampling Edge For Data & Frame Syncs

Data and frame syncs can be sampled on either the rising or falling edges of the serial port clock signals. The CKRE bit of the STCTLx and SRCTLx control registers selects the sampling edge.

For transmit data and frame syncs, setting CKRE=1 in STCTLx selects the rising edge of TCLKx. CKRE=0 selects the falling edge. Note that data and frame sync signals will change state on the clock edge that is not selected.

For receive data and frame syncs, setting CKRE=1 in SRCTLx selects the rising edge of RCLKx. CKRE=0 selects the falling edge.

The transmit and receive functions of two serial ports connected together, for example, should always select the same value for CKRE so that any internally generated signals are driven on one edge and any received signals are sampled on the opposite edge.

10.6.5 Early vs. Late Frame Syncs

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 LAFS bit of the STCTLx and SRCTLx control registers configures this option.

When LAFS=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 latched) 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 (i.e. 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.

When LAFS=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 latched) in the *same* serial clock cycle that the frame sync is asserted. (In multichannel operation, this is the case when frame delay is zero.) Receive data bits are latched 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 10.7 illustrates the two modes of frame signal timing:

- LAFS bits of STCTLx, SRCTLx control registers. LAFS=0 for early frame syncs, LAFS=1 for late frame syncs.
- Early framing: frame sync precedes data by one cycle. Late framing: frame sync checked on first bit only.
- Data transmitted MSB-first (SENDN=0) or LSB-first (SENDN=1).
- Frame sync and clock generated internally or externally.

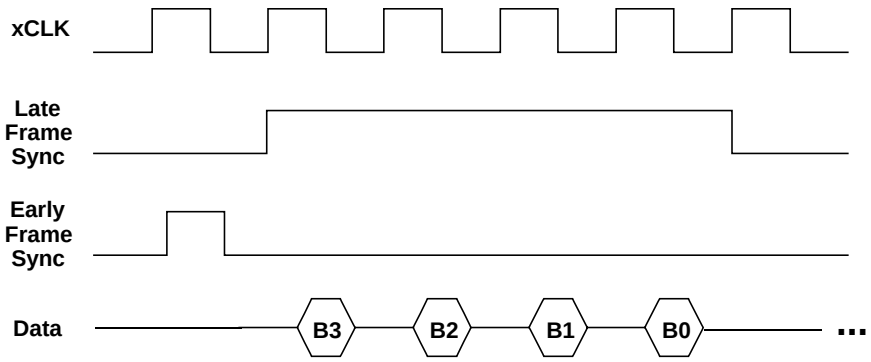


Figure 10.7 Normal vs. Alternate Framing

10.6.6 Data-Independent Transmit Frame Sync

Normally the internally generated transmit frame sync signal (TFS) is output only when the TX buffer has data ready to transmit. The DITFS mode (data-independent transmit frame sync) allows the continuous generation of the TFS signal, with or without new data. The DITFS bit of the STCTLx control register configures this option.

When DITFS=0, the internally generated TFS is only output when a new data word has been loaded into the TX buffer. Once data is loaded into TX, it is not transmitted until the next TFS is generated. This mode of operation allows data to be transmitted only at specific times.

When DITFS=1, the internally generated TFS is output at its programmed interval regardless of whether new data is available in the TX buffer. Whatever data is present in TX will be retransmitted with each assertion of TFS. The TUVF transmit underflow status bit (in the STCTLx control register) will be set when this occurs (i.e. when old data is retransmitted). The TUVF status bit is also set if the TX buffer does not have new data when an externally generated TFS occurs. Note that in this mode of operation, the first internally generated TFS will be delayed until data has been loaded into the TX buffer.

If the internally generated TFS is used, a single write to the TX data register is required to start the transfer.

10.7 MULTICHANNEL OPERATION

The ADSP-2106x serial ports 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 serial port can automatically select words for particular channels while ignoring the others. Up to 32 channels are available for transmitting or receiving—each SPORT can receive and transmit data selectively from any of the 32 channels. In other words, the SPORT can do any of the following on each channel:

1. transmit data,
2. receive data,
3. transmit and receive data, or
4. do nothing

Data companding and DMA transfers can also be used in multichannel mode.

The DT pin is always driven, i.e. not tristated, if the serial port is enabled (SPEN=1 in the STCTLx control register), unless it is in multichannel mode and an inactive time slot occurs.

Note that (in multichannel mode) the TCLKx pin is always an input and must be connected to its corresponding RCLKx pin.

Figure 10.8 shows example timing for a multichannel transfer, which have the following characteristics:

- Uses TDM method where serial data is sent or received on different channels sharing the same serial bus.
- The number of channels is selected with the NCH bits of SRCTLx: NCH=(# of channels) – 1.
- Can independently select transmit and receive channels.
- RFS signal start of frame.
- TFS is used as "Transmit Data Valid" for external logic; active only during transmit channels.
- Example: Receive on channels 0 and 2.
  Transmit on channels 1 and 2.

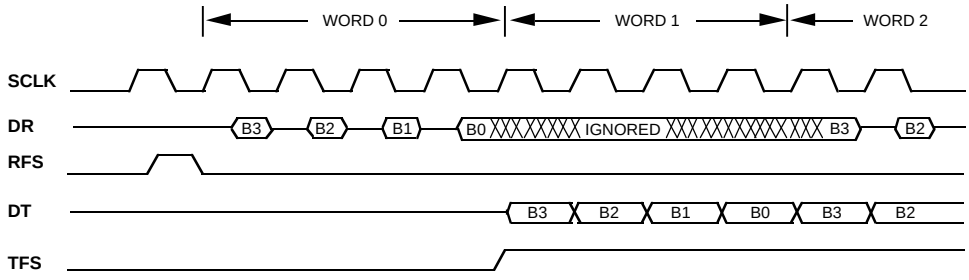


Figure 10.8 Multichannel Operation

10.7.1 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 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. RFS assertion occurs the beginning of the channel 0 data word.

TFS is used as a *transmit data valid* signal which is active during transmission of an enabled word. Since the serial port's DTx pin is tristated when the time slot is not active, the TFS signal specifies

whether or not DTx is being driven by the ADSP-2106x. The ADSP-2106x drives TFS in multichannel mode whether or not ITFS is cleared. After the TX transmit buffer is loaded, transmission begins and the TFS signal is generated. When serial port DMA is being used, this may happen several cycles after the multichannel transmission is enabled. If a deterministic start time is required, the TX buffer should be preloaded.

Note: TFS is normally left unconnected in multichannel mode, and the RFS pins of the serial port(s) are usually connected together.

10.7.2 Multichannel Control Bits In STCTL, SRCTL

The STCTLx and SRCTLx control registers contain several bits used to enable and configure multichannel operations.

10.7.2.1 Multichannel Enable

Multichannel mode is enabled by setting the MCE bit in the SRCTLx control register.

When MCE=1, multichannel operation is enabled.

When MCE=0, all multichannel operations are disabled.

Multichannel operation is activated three cycles after MCE is set. Internally generated frame sync signals activate four cycles after MCE is set.

Setting the MCE 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.

10.7.2.2 Number Of Channels

The number of channels used in multichannel operation is selected by the 5-bit NCH field in the SRCTLx control register. NCH should be set to the actual number of channels minus one:

NCH = Number of Channels – 1

10.7.2.3 Current Channel Indicator

The 5-bit CHNL field in the STCTLx control register indicates which channel is currently selected during multichannel operation. This field is a read-only status indicator. CHNL(4:0) increments modulo NCH(4:0) as each channel is serviced.

10.7.2.4 Multichannel Frame Delay

The 4-bit MFD field in the STCTLx 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 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.

A multichannel frame delay of at least one should be used when the ADSP-2106x is generating frame syncs for the multichannel system and the serial clock of the system is equal to CLKIN (the processor clock). If MFD is not set to at least one, the master ADSP-2106x in a multiprocessing system will not recognize the first frame sync after multichannel operation is enabled. All succeeding frame syncs will be recognized normally, however.

10.7.3 Channel Selection Registers

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 32 channels are available for transmitting and up to 32 channels for receiving.

The multichannel selection registers are used to enable and disable individual channels. The registers for each serial port are as follows:

Register Name	*Function*
MTCSx	Multichannel Transmit Select—specifies the active transmit channels
MRCSx	Multichannel Receive Select—specifies the active receive channels
MTCCSx	Multichannel Transmit Compand Select—specifies which active transmit channels are companded
MRCCSx	Multichannel Receive Compand Select—specifies which active receive channels are companded

Each register has 32 bits, corresponding the 32 channels. Setting a bit enables that channel so that the serial port will select 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 to 1 in the MTCSx register causes the serial port to transmit the word in that channel's position of the data stream. Clearing the bit to 0 in the MTCSx register causes the serial port's DT (data transmit) pin to tristate during the time slot of that channel.

Setting a particular bit to 1 in the 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 to 0 in the MRCSx register causes the serial port to ignore the data.

Companding may be selected on a per-channel basis. The MTCCSx and MRCCSx registers are used to specify companding for any active channels. Setting a bit to 1 in these registers causes the data to be companded. A-law or μ-law companding is selected with the DTYPE bit 1 in the STCTLx and SRCTLx control registers.

10.7.4 SPORT Receive Comparison Registers

On the ADSP-21061, two sets of registers aid multiprocessor communications when using multichannel mode (MCE=1) through the serial ports. These 32-bit registers are the Receive Comparison (KEYWDx) registers and the Receive Comparison Mask (KEYMASKx) registers.

The KEYWD0 or KEYWD1 register stores the pattern to be matched with the incoming data. The corresponding KEYMASK0 or KEYMASK1 register specifies which of the bits in the received data should be compared. Setting a KEYMASKx bit (=1) masks the corresponding bit in the KEYWDx register, disabling its comparison.

The processor receiving the data compares it with the data in the KEYWDx register. Depending on the comparison results, the received data is accepted or ignored. If accepted, the receiver requests—based on the setting to the SRCTL register—a DMA transfer to internal memory or generates an interrupt.

In addition to the MCE setting, the following bits in the SRCTL register control the operation of Receive Comparison:

IMODE (Bit 15)	IMAT (Bit 20)	Operation
0	x	Receive comparison disabled
1	0	Accept receive data if the KEYWD comparison is false
1	1	Accept receive data if the KEYWD comparison is true

When receive comparison is enabled, companding is disabled on the transmitter and receiver. The MTCCSx register, which selects multichannel companding when receive comparison is disabled, determines whether the DSP performs a KEYWD comparison for the enabled received channels. If the MTCCSx bit for a particlular channel is '0,' the processor does not perform a comparison and always accepts the receive data on that channel. If the MTCCSx bit for a particular channel is '1,' the processor performs the comparison and accepts (or rejects) the receive data, depending on the result of the comparison and IMAT setting in the SRCTLx register.

The receive comparison feature lets the ADSP-21061's SPORTS generate a DMA request or an interrupt when the received data matches a specified condition on a specified channel in multichannel mode. Without this feature, the SPORT would interrupt the processor every time data was received and the processor would be required to check if the data was meant for it or not. It is possible that most of the time the data being sent is not meant for the processor. With the receive comparison feature, the SPORT on a particular processor can be programmed to interrupt only on messages meant for that processor.

As a receive comparison example, consider four ADSP-21061s (A, B, C, and D) which use SPORT0 (in multichannel mode) for interprocessor communication. Channels 0, 1, 2, and 3 are used respectively by A, B, C, and D to transmit control information between the processors. Channels 4 through 10, 11 through 17, 18 through 24, and 25 through 31 are used respectively by A, B, C, and D to transmit data.

Because channels 0 through 3 are used to send control information between the processors, the comparisons for incoming data is enabled only for these channels. Initially, channels 4 through 31 may have receive disabled. For this example, consider communication between processors A and B only. The key word for comparison is programmable; in this example, processor B can check for the key word "START TRANSMIT TO B",

Processor B can check for this key word as follows:

1. Set the KEYWD register to "START TRANSMIT TO B"

2. Clear bits 31:16 of the KEYMASK register to 0 and set the other bits to 1
   This step enables comparison only for bits 31:16. So, assume that the code for "START TRANSMIT TO B" only uses bits 31:16 and bits 15:0 indicate the source of the transmission and the data channels.

3. Set bits 15 and 20 of the SRCTL register to 1
   This step enables the SPORT to generate an interrupt or DMA request only if the incoming data matches the KEYWD.

4. Set bits 0 through 3 of the Transmit Compand Channel Selector register to 1 and clear the remaining bits to 0
   This step enables comparison only on channels 0 through 3.

Until it receives the "START TRANSMIT TO B" keyword, processor B ignores all transmissions that it receives. When processor A wants to send data to B, it sends the "START TRANSMIT TO B" keyword on channel 0. When receive comparison on processor B recognizes the "START TRANSMIT TO B" keyword, the SPORT interrupts processor B. Then, processor B analyzes the remaining 16-bits, determining that the source is processor A and the data is on channels 4 through 10. Because processor A is using channels 4 through 10 to transmit data, processor B enables receive channels 4 through 10 and sends a "READY TO RECEIVE DATA" message to processor A, using channel 1. After processor A receives this message, it sends the data on channels 4 through 10. If the transfer protocol uses a fixed number of bytes in each message, processor B could send back a checksum message to processor A after receiving A's message, confirming that the data transferred accurately.

10.8 TRANSFERRING DATA BETWEEN SPORTS AND MEMORY

Transmit and receive data can be transferred between the ADSP-2106x serial ports 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 serial port DMA is *not* enabled in the STCTLx or SRCTLx control 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 of serial data before the interrupt is generated. The ADSP-2106x's on-chip 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.

10.8.1 DMA Block Transfers

The ADSP-2106x's on-chip DMA controller allows automatic DMA transfers between internal memory and the two serial ports. There are four DMA channels for serial port operations—each SPORT has one channel for receiving data and one for transmitting data. The serial port DMA channels are numbered as follows:

DMA Channel 0 – SPORT0 Receive
DMA Channel 1 – SPORT1 Receive (or Link Buffer 0)
DMA Channel 2 – SPORT0 Transmit
DMA Channel 3 – SPORT1 Transmit (or Link Buffer 1)

Note that channels 1 and 3 are shared between SPORT1 and link buffers 0 and 1. The SPORT DMA channels are assigned higher priority than all other DMA channels (i.e. for link ports and the external port) because of their relatively low service rate and their inability to hold off incoming data. Having higher priority causes the SPORT DMA transfers to be performed first when multiple DMA requests occur in the same cycle.

Although the DMA transfers are always performed with 32-bit words, the serial ports can handle word sizes from 3 to 32 bits. If the serial words are 16 bits or smaller, they can be packed into 32-bit words for each DMA transfer; this is configured by the PACK bit of the the STCTLx and SRCTLx control registers. When serial port data packing is enabled (PACK=1), the transmit and receive interrupts are generated for the 32-bit packed words, not for each 16-bit word.

The following sections present an overview of serial port DMA operations; some additional details are covered in the *DMA* chapter of this manual.

10.8.1.1 SPORT DMA Channel Setup

Each SPORT DMA channel has an enable bit (SDEN) in the STCTLx and SRCTLx control registers of the two serial ports. When DMA is not enabled for a particular channel, the SPORT generates an interrupt every time it has received a data word or has started to transmit a data word (see “Single-Word Transfers” later in this chapter). Each channel also has a DMA chaining enable bit (SCHEN) in the control registers (see “SPORT DMA Chaining” later in this chapter).

A serial port DMA channel is set up by writing a set of memory buffer parameters to the SPORT DMA parameter registers. The II, IM, and C registers must be loaded with a starting address for the buffer, an address modifier, and a word count, respectively. The programming of these registers can be done from the ADSP-2106x core processor or from an external processor.

Once serial port DMA is set up and enabled, data words received in the RX buffer are automatically transferred to the buffer in internal memory. Likewise, when the serial port is ready to transmit data, a word is automatically transferred from memory to the TX buffer. These transfers continue until the entire data buffer is received or transmitted (i.e. when the count register reaches zero).

When the count register of an active DMA channel reaches zero, the corresponding interrupt is generated.

10.8.1.2 SPORT DMA Parameter Registers

A DMA channel consists of a set of parameter registers that implement a data buffer in internal memory, plus hardware used by the serial port to request DMA service. The parameter registers for each SPORT DMA channel are shown in Tables 10.8 and 10.9. These registers are part of the memory-mapped IOP register set of the ADSP-2106x.

The DMA channels operate in a similar fashion as the ADSP-2106x’s Data Address Generators (DAGs). Each channel has an index register (II) and a modify register (IM) which are used to set up a data buffer in internal memory. The index register must be initialized with a starting address for the data buffer. After each serial I/O word is transferred to or from the SPORT, the DMA controller adds the modify value to the index register to generate the address for the next DMA transfer. The modify value in the IM register is a signed integer, which allows both incrementing and decrementing.

Each DMA channel has a count register (C) which must be initialized with a word count to be transferred. The count register is decremented after each DMA transfer on that channel; when the count reaches zero, the interrupt for that channel is generated and the channel is automatically disabled.

Each SPORT DMA channel also has a chain pointer register (CP), a general-purpose register (GP), and two registers used for two-dimensional array addressing in mesh multiprocessing applications (DA, DB). The CP register is used in chained DMA operations, as described below in "SPORT DMA Chaining", and the GP register can be used for any purpose. The DA and DB registers may be used as general-purpose registers in standard, non-mesh-multiprocessing DMA operations.

Register	*# of Bits*	*Function*
IIx	17	Index (starting address for data buffer)
IMx	16	Index Modifier (address increment)
Cx	16	Count (number of words to transfer)
CPx	18*	Chain Pointer (address of next set of buffer parameters)
GPx	17	General-Purpose or 2D DMA
DBx	16	General-Purpose or 2D DMA
DAx	16	General-Purpose or 2D DMA

Table 10.8 Parameter Registers For Each SPORT DMA Channel

* Lower 17 bits contains memory address of the next set of parameters for chained DMA operations. Most significant bit (bit 17) is the PCI bit (Program-Controlled Interrupts), which determines whether the DMA interrupts occur at the completion of each DMA sequence.

Memory Address	*Register*	*Channel Number & Function*
0x0060	II0	DMA Channel 0 – SPORT0 Receive
0x0061	IM0	DMA Channel 0 – SPORT0 Receive
0x0062	C0	DMA Channel 0 – SPORT0 Receive
0x0063	CP0	DMA Channel 0 – SPORT0 Receive
0x0064	GP0	DMA Channel 0 – SPORT0 Receive
0x0065	DB0	DMA Channel 0 – SPORT0 Receive
0x0066	DA0	DMA Channel 0 – SPORT0 Receive
0x0067	*reserved*	
0x0068	II1	DMA Channel 1 – SPORT1 Receive (or Link Buffer 0)
0x0069	IM1	DMA Channel 1 – SPORT1 Receive (or Link Buffer 0)
0x006A	C1	DMA Channel 1 – SPORT1 Receive (or Link Buffer 0)
0x006B	CP1	DMA Channel 1 – SPORT1 Receive (or Link Buffer 0)
0x006C	GP1	DMA Channel 1 – SPORT1 Receive (or Link Buffer 0)
0x006D	DB1	DMA Channel 1 – SPORT1 Receive (or Link Buffer 0)
0x006E	DA1	DMA Channel 1 – SPORT1 Receive (or Link Buffer 0)
0x006F	*reserved*	
0x0070	II2	DMA Channel 2 – SPORT0 Transmit
0x0071	IM2	DMA Channel 2 – SPORT0 Transmit
0x0072	C2	DMA Channel 2 – SPORT0 Transmit
0x0073	CP2	DMA Channel 2 – SPORT0 Transmit
0x0074	GP2	DMA Channel 2 – SPORT0 Transmit
0x0075	DB2	DMA Channel 2 – SPORT0 Transmit
0x0076	DA2	DMA Channel 2 – SPORT0 Transmit
0x0077	*reserved*	
0x0078	II3	DMA Channel 3 – SPORT1 Transmit (or Link Buffer 1)
0x0079	IM3	DMA Channel 3 – SPORT1 Transmit (or Link Buffer 1)
0x007A	C3	DMA Channel 3 – SPORT1 Transmit (or Link Buffer 1)
0x007B	CP3	DMA Channel 3 – SPORT1 Transmit (or Link Buffer 1)
0x007C	GP3	DMA Channel 3 – SPORT1 Transmit (or Link Buffer 1)
0x007D	DB3	DMA Channel 3 – SPORT1 Transmit (or Link Buffer 1)
0x007E	DA3	DMA Channel 3 – SPORT1 Transmit (or Link Buffer 1)
0x007F	*reserved*	

Table 10.9 SPORT DMA Parameter Registers

10.8.1.3 SPORT DMA Chaining

In chained DMA operations, the ADSP-2106x automatically sets up another DMA transfer when the contents of the current buffer have been transmitted (or received). The chain pointer register (CP) is used to point to the next set of buffer parameters stored in memory. The ADSP-2106x's DMA controller automatically downloads these buffer parameters to set up the next DMA sequence. Refer to the *DMA* chapter of this manual for details on how to set up chaining parameters in memory.

DMA chaining occurs independently for the transmit and receive channels of each serial port. Each SPORT DMA channel has a chaining enable bit (SCHEN) in the STCTLx and SRCTLx control registers. This bit must be set to 1 to enable chaining. Writing all zeros to the address field of the chain pointer register (CP) also disables chaining.

10.8.2 Single-Word Transfers

Individual data words may also be transmitted and received by the serial ports, with interrupts occurring as each 32-bit word is transmitted or received. When a serial port is enabled and DMA is disabled (in the STCTLx or SRCTLx control registers), the SPORT DMA interrupts will be generated in this way—whenever a complete 32-bit word has been received in the RX buffer, or whenever the TX buffer is not full. Single-word interrupts can be used to implement interrupt-driven I/O on the serial ports.

Whenever the ADSP-2106x core's program reads a word from a serial port's RX buffer or writes a word to its TX buffer, the buffer's *full/empty* status should first be checked in order to avoid hanging the ADSP-2106x core. (This can also happen to an external device, for example a host processor, when it is reading or writing a serial port buffer.) The *full/empty* status can be read in the RXS bits of the SRCTLx register or the TXS bits of the STCTLx register. Reading from an empty RX buffer or writing to a full TX buffer causes the ADSP-2106x (or external device) to hang, waiting for the status to change. To prevent this hang condition from occurring, the BHD (Buffer Hang Disable) bit should be set in the SYSCON register.

Multiple interrupts can occur if both SPORTs transmit or receive data in the same cycle. Any interrupt can be masked out in the IMASK register; if the interrupt is later enabled in IMASK, the corresponding interrupt latch bit in IRPTL must be cleared in case the interrupt has occurred in the meantime.

When serial port data packing is enabled (PACK=1 in the STCTLx or SRCTLx control registers), the transmit and receive interrupts are generated for the 32-bit packed words, not for each 16-bit word.

10.9 SPORT LOOPBACK

When the SPL bit (SPORT loopback) is set in the SRCTLx receive control register, the serial port is configured in an internal loopback connection. The loopback configuration allows the serial ports to be tested internally.

When loopback is configured, the DRx, RCLKx, and RFSx signals of the receive section of the SPORT are internally connected to the DTx, TCLKx, and TFSx signals of the transmit section. The DTx, TCLKx, and TFSx signals are active and are available at their respective pins, while the DRx, RCLKx, and RFSx pins are ignored by the ADSP-2106x.

Only transmit clock and transmit frame sync options may be used in loopback mode—you must ensure that the serial port is set up correctly in the STCTLx and SRCTLx control registers. Multichannel mode is not allowed.

10.10 SPORT PIN DRIVER CONCERNS

The ADSP-2106x has very fast drivers on all output pins including the serial ports. If connections on the data, clock, or frame sync lines are longer than six inches, you should 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.

10.11 SPORT PROGRAMMING EXAMPLES

There are three ways to control serial port communications and memory-to-SPORT data transfers: single-word transfers under core processor control with no interrupts, single-word transfers under core processor control with interrupts, and DMA transfers with interrupts. The three examples presented below illustrate each of these methods. In each example, the SPORT0 is used to transmit eight 32-bit words from a data buffer in internal memory.

Any of the three control schemes may be used in multichannel mode and with any of the serial clock and frame sync options.

10.11.1 Single-Word Transfers Without Interrupts

The ADSP-2106x processor core will stall (i.e. hang) when it attempts to write data to a full TX buffer or read data from an empty RX buffer. This provides a very simple method of controlling the SPORT—placing the instruction that writes data to TX or reads data from RX in a loop. Program execution will stall at this instruction, until the SPORT is ready to transmit new data or has received new data.

Listing 10.1 shows the code for this example, which sets up a loop to transmit data out of SPORT0. Although this technique provides a very simple programming solution, it prevents the ADSP-2106x processor core from handling any other tasks while waiting for the serial port. The interrupt-driven technique described in the following section alleviates this.

```
/*_____________________________________________________________

  SPORT Transmit Example: Uses the feature that the ADSP-2106x core
  will stall when attempting to write to a full TX register. This
  example sets up a loop to transmit the data in the memory buffer
  “source”.
  _____________________________________________________________*/

#define N 8
#include “def21060.h”   /* Use symbolic register names */

.segment/dm dm32_b1;    /* Data segment name described in arch file.*/
.var source[N]= 0x11111111, 0x22222222, 0x33333333, 0x44444444,
                0x55555555, 0x66666666, 0x77777777, 0x88888888;
.endseg;

.segment/pm rst_svc;    /* Reset vector from arch file. */
   nop;                 /* First location is used for booting.*/
   jump start;
.endseg;


/*_____________________Main routine_____________________*/

.segment/pm pm48_1b0;    /* Main code segment from arch file.*/

start:   r0=0x00270007;   /* TDIV0 Register: TCLKDIV=7,TFSDIV=39 */
         dm(TDIV0)=r0;    /* sclock=CLKIN/8, framerate=sclock/20 */

         r0=0x000064f1;   /* STCTL0 Register:      */
         dm(STCTL0)=r0;   /* SPEN=1,(SPORT enabled)*/
                          /* SLEN=15 (16-bit word)*/
                          /* ICLK=1, (internal tx clock)*/
                          /* TFSR=1, (require TFS)*/
                          /* ITFS=1, (internal TFS)*/
                          /* DITFS=0,(data dependent FS)*/
                          /*   all other bits = 0 */

         b0=source;      /* Pointer to source; i0=b0 automatically.*/
         l0=@source;

         lcntr=N, do tx_loop until lce;
           r0=dm(i0,1);               /* Get data from source buffer.*/
tx_loop:   dm(TX0)=r0;                /* Write transmit register,    */
                                      /* core will wait until SPORT */
                                      /* output buffer is not full.*/
         idle;
.endseg;
```

Listing 10.1 Non-Interrupt-Driven SPORT Control (Single-Word Transfers)

10.11.2 Single-Word Transfers With Interrupts

While the non-interrupt-driven solution of the previous example provides a very simple control scheme, it prevents the ADSP-2106x processor core from handling any additional tasks while it is stalled. In most real-time applications, the DSP must process data *while* new data is being received. It may also need to perform background tasks between data transfers.

In most systems, therefore, the DSP processor must be able to continue executing its program at all times. Using the serial port receive and transmit interrupts allows this to happen, by interrupting the core processor only when a new data word has been received or when a new data word can be transmitted. The interrupt service routine then performs the data transfer between internal memory and the serial port's TX or RX buffer.

Listing 10.2 below shows the code for this example. Note that the interrupt used is the *SPORT0 Transmit DMA Channel* interrupt (SPT0I)—when serial port DMA is disabled, this interrupt becomes a single-word transmit interrupt.

```
/*_____________________________________________________________

  ADSP-2106x Interrupt-Driven SPORT Transmit Example:

  This example uses interrupts to notify the core when new data
  is required for serial port 0 transmit. The buffer "source"
  is transmitted.
  _____________________________________________________________*/

#define N 8
#include "def21060.h"   /* Use symbolic register names */

.segment/dm dm32_b1;    /* Data segment name described in arch file.*/
.var source[N]= 0x11111111, 0x22222222, 0x33333333, 0x44444444,
                0x55555555, 0x66666666, 0x77777777, 0x88888888;
.endseg;

.segment/pm rst_svc;    /* Reset vector from arch file.*/
   nop;                 /* First location is used for booting.*/
   jump start;
.endseg;

.segment/pm spt0_svc;   /* Sport0 TX interrupt vector.*/
   jump s0tx;
.endseg;
```

(listing continues on next page)

```
/*_____________________________Main routine_____________________________*/

.segment/pm pm48_1b0;   /* Main code segment from arch file.*/

start:  r0=0x00270007;  /* TDIV0 Register: TCLKDIV=7,TFSDIV=39 */
        dm(TDIV0)=r0;   /* sclock=CLKIN/8, framerate=sclock/20 */

        r0=0x000064f1;  /* STCTL0 Register:        */
        dm(STCTL0)=r0;    /* SPEN=1,(SPORT enabled)*/
                           /* SLEN=15 (16-bit word)*/
                           /* ICLK=1, (internal tx clock)*/
                           /* TFSR=1, (require TFS)*/
                           /* ITFS=1, (internal TFS)*/
                           /* DITFS=0,(data dependent FS)*/
                           /*  all other bits = 0  */

        b0=source;      /* Pointer to source; i0=b0 automatically.*/
        l0=@source;

        bit set imask SPT0I;    /* Enable Sport0 TX interrupt.*/
        bit set mode1 IRPTEN;   /* Global interrupt enable.*/

        r0=dm(i0,1);    /* Write first value into TX0 to kick off sport.*/
        dm(TX0)=r0;

wait:   idle;           /* Wait for SPORT0 TX interrupts.*/
        jump wait;


/*_____________SPORT0 Transmit Interrupt Routine_____________*/

s0tx:   rti (db);
        r0=dm(i0,1);    /* Get data from source buffer */
        dm(TX0)=r0;     /* Write transmit register */
.endseg;
```

Listing 10.2 Interrupt-Driven SPORT Control (Single-Word Transfers)

10.11.3 DMA Transfers With Interrupts

This example shows how to use the ADSP-2106x's on-chip DMA controller to handle serial port I/O. The DMA controller performs the data transfers between internal memory and the SPORTs, providing the most efficient way to handle input and output of multiple-word blocks of data. Once it has been set up, the DMA controller operates independently from the ADSP-2106x processor core. It interrupts core execution only when an entire block of data has been received (or transmitted). This frees the core to continue with other tasks.

Listing 10.3 shows the code for this example, which uses the serial port's loopback mode. The program first sets up the SPORT1 DMA channels by loading values into the DMA parameter registers, then writes to the SRCTL1 and STCTL1 registers and waits to be interrupted.

```
/*_____________________________________________________________
   ADSP-2106x DMA-Driven SPORT Loopback Example:

  This example sets up a SPORT DMA transfer and receive for serial
  port 1 in the loopback mode. The buffer "source" is DMAed out of
  the sport. The loopback mode internally attaches DT1, TFS1, and
  TCLK1 to DR1, RFS1, and RCLK1. The receive DMA places the data
  in the buffer "destination".
  _____________________________________________________________*/

#define N 8
#include "def21060.h"   /* Use symbolic register names */

.segment/dm dm32_b1;    /* Data segment name described in arch. file.*/
.var source[N]= 0x11111111, 0x22222222, 0x33333333, 0x44444444,
                0x55555555, 0x66666666, 0x77777777, 0x88888888;
.var destination[N];
.endseg;

.segment/pm rst_svc;    /* Reset vector from arch. file.*/
   nop;                 /* First location is used for booting.*/
   jump start;
.endseg;

.segment/pm spr1_svc;   /* SPORT1 rx interrupt vector.*/
   jump s1rx;
.endseg;
```

(listing continues on next page)

```
/*_____________________________main routine_____________________________*/

.segment/pm pm48_1b0;      /* Main code segment from arch. file*/

start:  r0=source;
        dm(II3)=r0;        /* Set DMA tx index to start of source buffer*/
        r0=destination;
        dm(II1)=r0;        /* Set DMA rx index to start of destination buffer*/
        r0=1;
        dm(IM3)=r0;        /* Set DMA modify (stride) to 1.*/
        dm(IM1)=r0;
        r0=@source;
        dm(C3)=r0;         /* Set DMA count to length of data buffer*/
        dm(C1)=r0;

        r0=0x004421f1;     /* SRCTL1 Register:          */
        dm(SRCTL1)=r0;     /*   SPEN=1, (SPORT1 enabled)*/
                           /*   SLEN=31,(32-bit word)*/
                           /*   RFSR=1, (require RFS)*/
                           /*   SDEN=1, (rx DMA enable)*/
                           /*   SPL=1,  (loop back DT to DR & TFS to RFS)*/

        r0=0x00270007;     /* TDIV0 Register: TCLKDIV=7, TFSDIV=39*/
        dm(TDIV1)=r0;      /*   sclock=CLKIN/8, framerate=sclock/2 0 */

        r0=0x000465f1;     /* STCTL1 Register:          */
        dm(STCTL1)=r0;     /*   SPEN=1, (SPORT1 enabled)*/
                           /*   SLEN=31,(32-bit word)*/
                           /*   ICLK=1, (internal tx clock)*/
                           /*   TFSR=1, (require TFS)*/
                           /*   ITFS=1, (internal TFS)*/
                           /*   DITFS=0,(data dependent FS), all other bits=0*/
                           /*   SDEN=1, (tx dma enablea), this kicks it off*/

        bit set imask SPR1I;   /* Enable SPORT1 rx interrupt*/
        bit set mode1 IRPTEN;  /* Global interrupt enable*/

wait:   idle;              /* Wait for SPORT1 rx interrupt*/
        jump wait;         /* Will end up here after entire DMA complete*/


/*___________________SPORT1 Receive Interrupt Routine_____________________*/

s1rx:   rti;               /* This interrupt will occur only once*/

.endseg;
```

Listing 10.3 SPORT DMA Example

System Design 11

11.1 OVERVIEW

This chapter provides hardware, software, and system design information.

11.2 ADSP-2106X PINS

This section describes the pins of the ADSP-2106x and shows how these signals can be used in your system. Figure 11.1 illustrates how the pins are used in a single-processor system. Figure 7.1 in the *Multiprocessing* chapter shows a system diagram illustrating pin connections in an ADSP-2106x multiprocessor cluster.

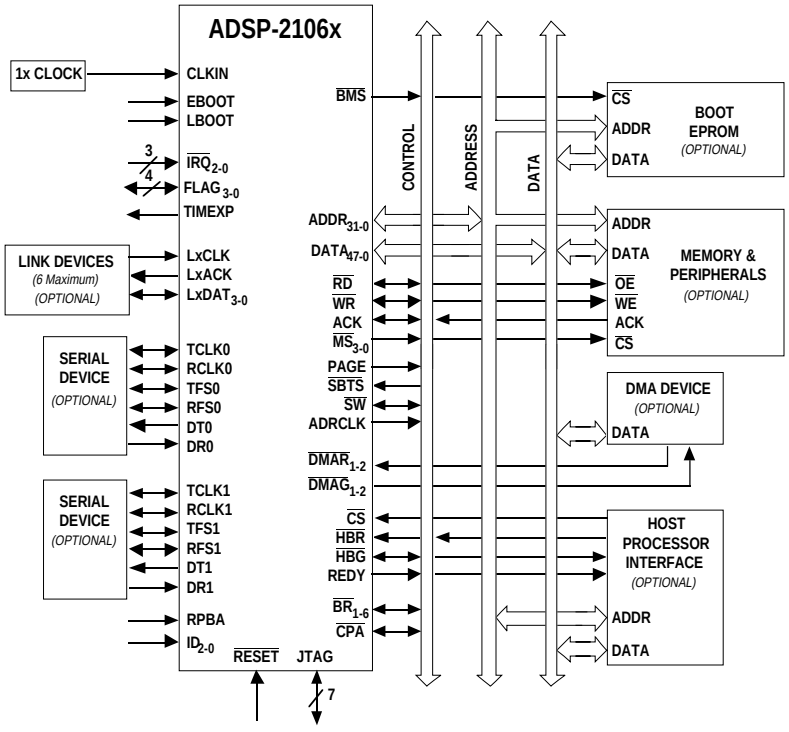


Figure 11.1 Basic ADSP-2106x System

11 System Design

11.2.1 Pin Definitions

ADSP-2106x pin definitions are listed below. All pins are identical on the ADSP-21060 and ADSP-21062. Inputs identified as synchronous (S) must meet timing requirements with respect to CLKIN (or with respect to TCK for TMS, TDI). Inputs identified as asynchronous (A) can be asserted asynchronously to CLKIN (or to TCK for $\overline{\text{TRST}}$).

Unused inputs should be tied or pulled to VDD or GND, except for $\text{ADDR}_{31\text{-}0}$, $\text{DATA}_{47\text{-}0}$, $\text{FLAG}_{3\text{-}0}$, $\overline{\text{SW}}$, and inputs that have internal pullup or pulldown resistors ($\overline{\text{CPA}}$, ACK, DTx, DRx, TCLKx, RCLKx, LxDAT3-0, LxCLK, LxACK, TMS, and TDI)—these pins should be left floating. These pins have a logic-level hold circuit that prevents the input from floating internally.

I=Input	S=Synchronous	P=Power Supply	(o/d)=Open Drain
O=Output	A=Asynchronous	G=Ground	(a/d)=Active Drive

T=Three-State (when $\overline{\text{SBTS}}$ is asserted, or when the ADSP-2106x is a bus slave)

Pin	*Type*	*Function*
$\text{ADDR}_{31\text{-}0}$	I/O/T	**External Bus Address**. The ADSP-2106x outputs addresses for external memory and peripherals on these pins. In a multiprocessor system, the bus master outputs addresses for read/writes of the internal memory or IOP registers of other ADSP-2106xs. The ADSP-2106x inputs addresses when a host processor or multiprocessing bus master is reading or writing its internal memory or IOP registers.
$\text{DATA}_{47\text{-}0}$	I/O/T	**External Bus Data**. The ADSP-2106x inputs and outputs data and instructions on these pins. 32-bit single-precision floating-point data and 32-bit fixed-point data is transferred over bits 47-16 of the bus. 40-bit extended-precision floating-point data is transferred over bits 47-8 of the bus. 16-bit short word data is transferred over bits 31-16 of the bus. In PROM boot mode, 8-bit data is transferred over bits 23-16. Pull-up resistors on unused DATA pins are not necessary.
$\overline{\text{MS}}_{3\text{-}0}$	O/T	**Memory Select Lines**. These lines are asserted (low) as chip selects for the corresponding banks of external memory. Memory bank size must be defined in the ADSP-2106x's system control register (SYSCON). The $\overline{\text{MS}}_{3\text{-}0}$ lines are decoded memory address lines that change at the same time as the other address lines. When no external memory access is occurring the $\overline{\text{MS}}_{3\text{-}0}$ lines are inactive; they are active, however, when a conditional memory access instruction is executed, whether or not the condition is true. $\overline{\text{MS}}_0$ can be used with the PAGE signal to implement a bank of DRAM memory (Bank 0). In a multiprocessing system, the $\overline{\text{MS}}_{3\text{-}0}$ lines are output by the bus master.

Pin	*Type*	*Function*
$\overline{RD}$	I/O/T	**Memory Read Strobe**. This pin is asserted (low) when the ADSP-2106x reads from external memory devices or from the internal memory of other ADSP-2106xs. External devices (including other ADSP-2106xs) must assert $\overline{RD}$ to read from the ADSP-2106x's internal memory. In a multiprocessing system, $\overline{RD}$ is output by the bus master and is input by all other ADSP-2106xs.
$\overline{WR}$	I/O/T	**Memory Write Strobe**. This pin is asserted (low) when the ADSP-2106x writes to external memory devices or to the internal memory of other ADSP-2106xs. External devices must assert $\overline{WR}$ to write to the ADSP-2106x's internal memory. In a multiprocessing system, $\overline{WR}$ is output by the bus master and is input by all other ADSP-2106xs.
PAGE	O/T	**DRAM Page Boundary**. The ADSP-2106x asserts this pin to signal that an external DRAM page boundary has been crossed. DRAM page size must be defined in the ADSP-2106x's memory control register (WAIT). DRAM can only be implemented in external memory Bank 0; the PAGE signal can only be activated for Bank 0 accesses. In a multiprocessing system, PAGE is output by the bus master.
ADRCLK	O/T	**Clock Output Reference.** In a multiprocessing system, ADRCLK is output by the bus master.
$\overline{SW}$	I/O/T	**Synchronous Write Select**. This signal is used to interface the ADSP-2106x to synchronous memory devices (including other ADSP-2106xs). The ADSP-2106x asserts $\overline{SW}$ (low) to provide an early indication of an impending write cycle, which can be aborted if $\overline{WR}$ is not later asserted (e.g. in a conditional write instruction). In a multiprocessing system, $\overline{SW}$ is output by the bus master and is input by all other ADSP-2106xs to determine if the multiprocessor memory access is a read or write. $\overline{SW}$ is asserted at the same time as the address output. A host processor using synchronous writes must assert this pin when writing to the ADSP-2106x(s).

11 System Design

Pin	*Type*	*Function*
ACK	I/O/S	**Memory Acknowledge**. External devices can deassert ACK (low) to add wait states to an external memory access. ACK is used by I/O devices, memory controllers, or other peripherals to hold off completion of an external memory access. The ADSP-2106x deasserts ACK as an output to add wait states to a synchronous access of its internal memory. In a multiprocessing system, a slave ADSP-2106x deasserts the bus master's ACK input to add wait state(s) to an access of its internal memory. The bus master has a keeper latch on its ACK pin that maintains the input at the level it was last driven to.
$\overline{\text{SBTS}}$	I/S	**Suspend Bus Tristate**. External devices can assert $\overline{\text{SBTS}}$ (low) to place the external bus address, data, selects, and strobes in a high-impedance state for the following cycle. If the ADSP-2106x attempts to access external memory while $\overline{\text{SBTS}}$ is asserted, the processor will halt and the memory access will not be completed until $\overline{\text{SBTS}}$ is deasserted. $\overline{\text{SBTS}}$ should only be used to recover from host processor/ADSP-2106x deadlock or used with a DRAM controller. The $\overline{\text{SBTS}}$ signal causes the masters to tristate during reset.
$\overline{\text{IRQ}}_{2\text{-}0}$	I/A	**Interrupt Request Lines**. May be either edge-triggered or level-sensitive.
$\text{FLAG}_{3\text{-}0}$	I/O/A	**Flag Pins**. Each is configured via control bits as either an input or output. As an input, it can be tested as a condition. As an output, it can be used to signal external peripherals.
TIMEXP	O	**Timer Expired**. Asserted for four cycles when the timer is enabled and TCOUNT decrements to zero.
$\overline{\text{HBR}}$	I/A	**Host Bus Request**. Must be asserted by a host processor to request control of the ADSP-2106x's external bus. When $\overline{\text{HBR}}$ is asserted in a multiprocessing system, the ADSP-2106x that is bus master will relinquish the bus and assert $\overline{\text{HBG}}$. To relinquish the bus, the ADSP-2106x places the address, data, select, and strobe lines in a high-impedance state. $\overline{\text{HBR}}$ has priority over all ADSP-2106x bus requests ($\overline{\text{BR}}_{6\text{-}1}$) in a multiprocessing system.
$\overline{\text{HBG}}$	I/O	**Host Bus Grant**. Acknowledges an $\overline{\text{HBR}}$ bus request, indicating that the host processor may take control of the external bus. $\overline{\text{HBG}}$ is asserted (held low) by the ADSP-2106x until $\overline{\text{HBR}}$ is released. In a multiprocessing system, $\overline{\text{HBG}}$ is output by the ADSP-2106x bus master and is monitored by all others.

Pin	*Type*	*Function*
$\overline{CS}$	I / A	**Chip Select**. Asserted by host processor to select the ADSP-2106x.
REDY (o / d)	O	**Host Bus Acknowledge**. The ADSP-2106x deasserts REDY (low) to add wait states to an asynchronous access of its internal memory or IOP registers by a host. Open drain output (o / d) by default; can be programmed in ADREDY bit of SYSCON register to be active drive (a / d). REDY will only be output if the $\overline{CS}$ and $\overline{HBR}$ inputs are asserted.
$\overline{DMAR1}$	I / A	**DMA Request 1** *(ADSP-21060/61/62 — DMA Channel 7)*
$\overline{DMAR2}$	I / A	**DMA Request 2** *(ADSP-21060/62 — DMA Channel 8) (ADSP-21061 — DMA Channel 6)*
$\overline{DMAG1}$	O / T	**DMA Grant 1** *(ADSP-21060/61/62 — DMA Channel 7)*
$\overline{DMAG2}$	O / T	**DMA Grant 2** *(ADSP-21060/62 — DMA Channel 8) (ADSP-21061 — DMA Channel 6)*
$\overline{BR}_{6-1}$	I / O / S	**Multiprocessing Bus Requests**. Used by multiprocessing ADSP-2106xs to arbitrate for bus mastership. An ADSP-2106x only drives its own $\overline{BR}$x line (corresponding to the value of its ID_{2-0} inputs) and monitors all others. In a multiprocessor system with less than six ADSP-2106xs, the unused $\overline{BR}$x pins should be pulled high; the processor's own $\overline{BR}$x line must not be pulled high or low because it is an output.
ID_{2-0}	I	**Multiprocessing ID**. Determines which multiprocessing bus request ($\overline{BR1}$ – $\overline{BR6}$) is used by ADSP-2106x. ID=001 corresponds to $\overline{BR1}$, ID=010 corresponds to $\overline{BR2}$, etc. ID=000 in single-processor systems. These lines are a system configuration selection which should be hardwired or only changed at reset.
RPBA	I / S	**Rotating Priority Bus Arbitration Select**. When RPBA is high, rotating priority for multiprocessor bus arbitration is selected. When RPBA is low, fixed priority is selected. This signal is a system configuration selection which must be set to the same value on every ADSP-2106x. If the value of RPBA is changed during system operation, it must be changed in the same CLKIN cycle on every ADSP-2106x.

Pin	*Type*	*Function*
$\overline{\text{CPA}}$ (o / d)	I / O	**Core Priority Access**. Asserting its $\overline{\text{CPA}}$ pin allows the core processor of an ADSP-2106x bus slave to interrupt background DMA transfers and gain access to the external bus. $\overline{\text{CPA}}$ is an open drain output that is connected to all ADSP-2106xs in the system. The $\overline{\text{CPA}}$ pin has an internal 5 Kohm pullup resistor. If core access priority is not required in a system, the $\overline{\text{CPA}}$ pin should be left unconnected.
DTx	O	**Data Transmit** (Serial Ports 0, 1). Each DT pin has a 50 kΩ internal pullup resistor.
DRx	I	**Data Receive** (Serial Ports 0, 1). Each DR pin has a 50 kΩ internal pullup resistor.
TCLKx	I / O	**Transmit Clock** (Serial Ports 0, 1). Each TCLK pin has a 50 kΩ internal pullup resistor.
RCLKx	I / O	**Receive Clock** (Serial Ports 0, 1). Each RCLK pin has a 50 kΩ internal pullup resistor.
TFSx	I / O	**Transmit Frame Sync** (Serial Ports 0, 1).
RFSx	I / O	**Receive Frame Sync** (Serial Ports 0, 1).
$\text{LxDAT}_{3\text{-}0}$	I / O	**Link Port Data** (Link Ports 0-5). Each LxDAT pin has a 50 kΩ internal pulldown resistor which is enabled or disabled by the LPDRD bit of the LCOM register. (NC on ADSP-21061)
LxCLK	I / O	**Link Port Clock** (Link Ports 0-5). Each LxCLK pin has a 50 kΩ internal pulldown resistor which is enabled or disabled by the LPDRD bit of the LCOM register. (NC on ADSP-21061)
LxACK	I / O	**Link Port Acknowledge** (Link Ports 0-5). Each LxACK pin has a 50 kΩ internal pulldown resistor which is enabled or disabled by the LPDRD bit of the LCOM register. (NC on ADSP-21061)
EBOOT	I	**EPROM Boot Select**. When EBOOT is high, the ADSP-2106x is configured for booting from an 8-bit EPROM. When EBOOT is low, the LBOOT and $\overline{\text{BMS}}$ inputs determine booting mode. See table below. This signal is a system configuration selection which should be hardwired.
LBOOT	I	**Link Boot – Host Boot Select**. When LBOOT is high, the ADSP-2106x is configured for link port booting. When LBOOT is low, the ADSP-2106x is configured for host processor booting or no booting. See the table with the $\overline{\text{BMS}}$ pin. This signal is a system configuration selection which should be hardwired. (Tied to GND on ADSP-21061)

Pin	*Type*	*Function*
$\overline{\text{BMS}}$	I / O / T*	**Boot Memory Select**. *Output:* Used as chip select for boot EPROM devices (when EBOOT=1, LBOOT=0). In a multiprocessor system, $\overline{\text{BMS}}$ is output by the bus master. *Input:* When low, indicates that no booting will occur and that ADSP-2106x will begin executing instructions from external memory. See table below. This input is a system configuration selection which should be hardwired.
CLKIN	I	**Clock In**. External clock input to the ADSP-2106x. The instruction cycle rate is equal to CLKIN. CLKIN may not be halted, changed, or operated below the minimum specified frequency.
$\overline{\text{RESET}}$	I / A	**Processor Reset**. Resets the ADSP-2106x to a known state and begins execution at the program memory location specified by the hardware reset vector address. This input must be asserted (low) at power-up.
TCK	I	**Test Clock (JTAG)**. Provides an asynchronous clock for JTAG boundary scan.
TMS	I / S	**Test Mode Select (JTAG)**. Used to control the test state machine. TMS has a 20 kΩ internal pullup resistor.
TDI	I / S	**Test Data Input (JTAG)**. Provides serial data for the boundary scan logic. TDI has a 20 kΩ internal pullup resistor.
TDO	O	**Test Data Output (JTAG)**. Serial scan output of the boundary scan path.
$\overline{\text{TRST}}$	I / A	**Test Reset (JTAG)**. Resets the test state machine. $\overline{\text{TRST}}$ must be asserted (pulsed low) after power-up or held low for proper operation of the ADSP-2106x. $\overline{\text{TRST}}$ has a 20 kΩ internal pullup resistor.

* Three-statable only in EPROM boot mode (when $\overline{\text{BMS}}$ is an output).

EBOOT	*LBOOT*	*BMS*	*Booting Mode*
1	0	output	EPROM (connect $\overline{\text{BMS}}$ to EPROM chip select)
0	0	1 (input)	Host processor
0	1	1 (input)	Link port
0	0	0 (input)	No booting. Processor executes from ext. memory.
0	1	0 (input)	*reserved*
1	1	x (input)	*reserved*

11 System Design

Pin	*Type*	*Function*
$\overline{\text{EMU}}$	O	**Emulation Status**. Must be connected to the ADSP-2106x EZ-ICE target board connector *only*.
ICSA	O	Reserved, leave unconnected.
VDD	P	**Power Supply**; nominally +5.0V dc for 5V devices or +3.3V dc for 3.3V devcies. (30 pins)
GND	G	**Power Supply Return**. (30 pins)
NC		**Do Not Connect**. Reserved pins which must be left open and unconnected. Note that the LxDAT, LxCLK, and LxACK pins on the ADSP-21060 and ADSP-21062 are NC on the ADSP-21061.

THE $\overline{\text{TRST}}$ INPUT OF THE JTAG INTERFACE ***MUST*** BE ASSERTED (I.E. PULSED LOW) OR HELD LOW AFTER POWER-UP FOR PROPER OPERATION OF THE ADSP-2106X. DO NOT LEAVE THIS PIN UNCONNECTED!!

Additional Notes:

• In single-processor systems, the ADSP-2106x owns the external bus during reset and does not perform bus arbitration to gain control of the bus.

• Operation of the $\overline{\text{RD}}$ and $\overline{\text{WR}}$ signals changes when $\overline{\text{CS}}$ is asserted by a host processor. See “Asynchronous Transfers” and “Synchronous Transfers” in the *Host Interface* chapter for details.

• Except during a Host Transition Cycle (HTC), the $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes should not be deasserted (low-to-high transition) while ACK or REDY are deasserted (low)—the ADSP-2106x will hang if this happens.

• In multiprocessor systems, the ACK signal is an input to the ADSP-2106x bus master and will not float when it is not being driven (because the bus master maintains a weak keeper latch on the pin). During reset, the ACK pin is pulled high internally with a 2 kΩ equivalent resistor by the ADSP-2106x bus master and is held high with the internal keeper latch. It is not necessary to use an external pullup resistor on the ACK line during booting or at any other time.

• For multiprocessor systems, PAGE is guaranteed to be asserted for the first true access after acquiring bus mastership. PAGE will not be updated or asserted for multiprocessor memory space accesses or external memory space accesses to any bank other than Bank 0.

• The $\overline{\text{HBR}}$ input is disabled during any access in which the PAGE signal is asserted. This prevents the possibility of the ADSP-2106x becoming a bus slave while a DRAM controller is servicing a page change.

Figure 11.a shows how different data word sizes are transferred over the external port.

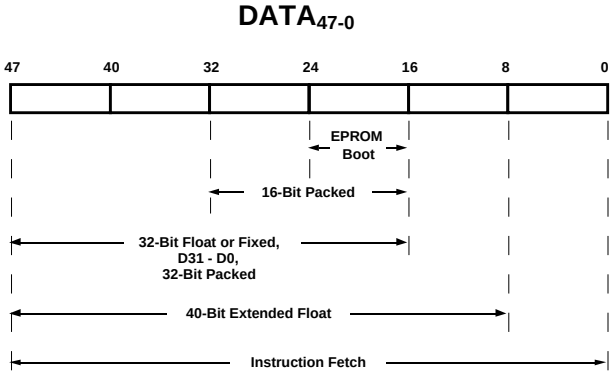


Figure 11.a External Port Data Alignment

11.2.2 Pin States At Reset

Table 11.1 shows the ADSP-2106x pin states during and immediately after reset.

Pin	Type	State During & After RESET
Driven Only By ADSP-2106x Bus Master, Otherwise Tristated:		
$ADDR_{31-0}$	I/O/T	Driven
$\overline{MS}_{3-0}$	O/T	Driven High
$\overline{RD}$	I/O/T	Driven High
$\overline{WR}$	I/O/T	Driven High
PAGE	O/T	Driven Low
ADRCLK	O/T	Driven by Clock (removes skew between each master)
$\overline{SW}$	I/O/T	Driven High
ACK	I/O/S	Pulled High by Bus Master (with 2 kΩ internal pullup resistor)
$\overline{HBG}$	I/O/ST	Driven High
$\overline{DMAG1}$	O/T	Driven High
$\overline{DMAG2}$	O/T	Driven High
$\overline{BR}_{6-1}$	I/O	$\overline{BR}_1$ Driven Low if Bus Master, Otherwise Driven High
Bus-Master-Independent:		
$DATA_{47-0}$	I/O/T	Tristate
$\overline{SBTS}$	I/S	Input; causes the master to tristate during reset
$\overline{IRQ}_{2-0}$	I/A	Inputs
$FLAG_{3-0}$	I/O/A	Inputs
TIMEXP	O	Driven Low
$\overline{HBR}$	I/A	Input
$\overline{CS}$	I	Input

Table 11.1 ADSP-2106x Pin States At $\overline{RESET}$ **(cont. on next page)**

Pin	Type	State During & After RESET
Bus-Master-Independent:		
REDY (o / d)	O	Tristate
$\overline{\text{DMAR1}}$	I	Input
$\overline{\text{DMAR2}}$	I	Input
ID_{2-0}	I	Inputs
RPBA	I / S	Input
$\overline{\text{CPA}}$ (o / d)	I / O	Tristate
EBOOT	I	Input
LBOOT	I	Input (must be tied to GND on the ADSP-21061)
$\overline{\text{BMS}}$	I / O / T	Input
CLKIN	I	Input
$\overline{\text{RESET}}$	I / A	Input
Serial Ports & Link Ports:		
DTx	O	Tristate (for multichannel)
DRx	I	Input
TCLKx	I / O	Tristate
RCLKx	I / O	Tristate
TFSx	I / O	Tristate
RFSx	I / O	Tristate
$LxDAT_{3-0}$	I / O	Tristate (NC on the ADSP-21061)
LxCLK	I / O	Tristate(NC on the ADSP-21061)
LxACK	I / O	Tristate(NC on the ADSP-21061)
JTAG Interface:		
TCK	I	Input
TMS	I / S	Input
TDI	I / S	Input
TDO	O	Tristate
$\overline{\text{TRST}}$	I / A	Input
$\overline{\text{EMU}}$	O	Tristate

Table 11.1 ADSP-2106x Pin States At $\overline{\text{RESET}}$ (cont.)

11.2.3 $\overline{\text{RESET}}$ & CLKIN

The ADSP-2106x receives its clock input on the CLKIN pin. The processor uses an on-chip phase-locked loop to generate its internal clock. Because the phase-locked loop requires some time to achieve phase lock, CLKIN must be valid for a minimum time period during reset before the $\overline{\text{RESET}}$ signal can be deasserted; this time period is specified in the *ADSP-2106x Data Sheet*.

$\overline{\text{RESET}}$ must be asserted (low) at system powerup.

11.2.3.1 Input Synchronization Delay

The ADSP-2106x has several asynchronous inputs: $\overline{\text{RESET}}$, $\overline{\text{TRST}}$, $\overline{\text{HBR}}$, $\overline{\text{CS}}$, $\overline{\text{DMAR1}}$, $\overline{\text{DMAR2}}$, $\overline{\text{IRQ}}_{2\text{-}0}$, and $\text{FLAG}_{3\text{-}0}$ (when configured as inputs). These inputs can be asserted in arbitrary phase to the processor clock, CLKIN. The ADSP-2106x synchronizes the inputs prior to recognizing them. The delay associated with recognition is called the synchronization delay.

Any asynchronous input must be valid prior to the recognition point to be recognized in a particular cycle. If an input does not meet the setup time on a given cycle, it may be recognized in the current cycle or during the next cycle.

Therefore, to ensure recognition of an asynchronous input, it must be asserted for at least one full processor cycle plus setup and hold time (except for $\overline{\text{RESET}}$, which must be asserted for at least four processor cycles). The minimum time prior to recognition (i.e. the setup and hold time) is specified in the *ADSP-2106x Data Sheet*.

11.2.4 Interrupt & Timer Pins

The ADSP-2106x's external interrupt pins, flag pins, and timer pin can be used to send and receive control signals to and from other devices in the system.

Hardware interrupt signals are received on the $\overline{\text{IRQ}}_{2\text{-}0}$ pins. Interrupts can come from devices that require the ADSP-2106x to perform some task on demand. A memory-mapped peripheral, for example, can use an interrupt to alert the processor that it has data available. Interrupts are described in detail in the *Program Sequencing* chapter.

The TIMEXP output is generated by the on-chip timer. It indicates to other devices that the programmed time period has expired. The timer is also described in detail in the *Program Sequencing* chapter.

11.2.5 Flag Pins

The $\text{FLAG}_{3\text{-}0}$ pins allow single-bit signalling between the ADSP-2106x and other devices. For example, the ADSP-2106x can raise an output flag to interrupt a host processor. Each flag pin can be programmed to be either an input or output. In addition, many ADSP-2106x instructions can be conditioned on a flag's input value, enabling efficient communication and synchronization between multiple processors or other interfaces.

The flags are bidirectional pins, each with the same functionality. To program the direction of each flag pin, the following control bits in the MODE2 register are used:

MODE2

Bit	*Name*	*Definition*
15	FLG0O	FLAG0 direction (1=output, 0=input)
16	FLG1O	FLAG1 direction (1=output, 0=input)
17	FLG2O	FLAG2 direction (1=output, 0=input)
18	FLG3O	FLAG3 direction (1=output, 0=input)

At reset, the MODE2 register is cleared, configuring all the flags as inputs.

11.2.5.1 Flag Inputs

When a flag pin is programmed as an input, its value is stored in a bit in the ASTAT register. The bit is updated in each cycle with the input value from the pin. Flag inputs can be asynchronous to the ADSP-2106x clock, so there is a one-cycle delay before a change on the pin appears in ASTAT (if the rising edge of the input misses the setup requirement for that cycle).

ASTAT

Bit	*Name*	*Definition*
19	FLG0	FLAG0 value
20	FLG1	FLAG1 value
21	FLG2	FLAG2 value
22	FLG3	FLAG3 value

An ASTAT flag bit is read-only if the flag is configured as an input. Otherwise, the bit is readable and writeable. The ASTAT flag bit states are conditions you can specify in conditional instructions.

Note that when an interrupt service routine causes ASTAT to be pushed onto the status stack, the flag bits in ASTAT are not affected; the values of these bits carry over from the main program to the service routine and from the service routine back to the main program (i.e. status stack pop).

(See “Status Stack Save & Restore” in the “Interrupts” section of the *Program Sequencing* chapter for further details.)

11.2.5.2 Flag Outputs

When a flag is configured as an output, the value on the pin follows the value of the corresponding bit in the ASTAT register. Your program can set or clear the ASTAT flag bit to provide a signal to another processor or peripheral. The timing of a flag output is shown in Figure 11.2.

The ASTAT flag bits are not changed when the ASTAT register is pushed onto or popped off of the status stack.

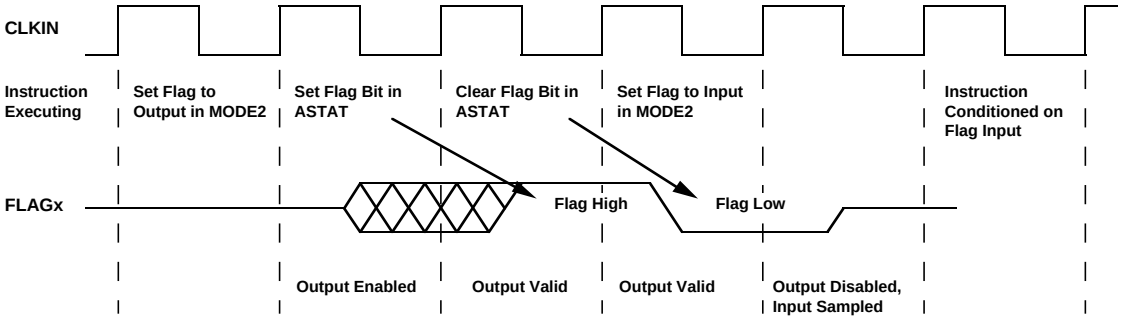


Figure 11.2 Flag Output Timing

11.2.6 JTAG Interface Pins

The JTAG test access port consists of the TCK, TMS, TDI, TDO, and $\overline{\text{TRST}}$ pins. The JTAG port can be connected to a controller that performs a boundary scan for testing purposes. This port is also used by the ADSP-2106x EZ-ICE Emulator to access on-chip emulation features. To allow the use of the emulator, a connector for its in-circuit probe must be included in your target system. See the "EZ-ICE Emulator" section of this chapter for details.

THE $\overline{\text{TRST}}$ INPUT OF THE JTAG INTERFACE ***MUST*** BE ASSERTED (I.E. PULSED LOW) OR HELD LOW AFTER POWER-UP FOR PROPER OPERATION OF THE ADSP-2106X. DO NOT LEAVE THIS PIN UNCONNECTED!!

If $\overline{\text{TRST}}$ is not asserted (or held low) at power-up, the JTAG port will be in an undefined state which may cause the ADSP-2106x to drive out on I/O pins that would normally be tristated at reset. $\overline{\text{TRST}}$ can be held low with a jumper to ground on the EZ-ICE target board connector. (See Figure 11.3 in "EZ-ICE Emulator".)

11 System Design

11.3 EZ-ICE EMULATOR

The ADSP-2106x EZ-ICE Emulator is a development tool for debugging programs running in real time on your ADSP-2106x target system hardware. The EZ-ICE provides a controlled environment for observing, debugging, and testing activities in a target system by connecting directly to the target processor through its JTAG interface. The emulator can monitor system behavior while running at full speed. It lets you examine and alter memory locations as well as processor registers and stacks.

Because the EZ-ICE controls the target system's ADSP-2106x through the processor's IEEE 1149.1 JTAG Test Access Port, non-intrusive in-circuit emulation is assured. The emulator does not impact target loading or timing. The emulator's in-circuit probe connects to an IBM PC host computer with an ISA bus plug-in board.

Target systems must have a 14-pin connector to accept the EZ-ICE's in-circuit probe, a 14-pin plug.

11.3.1 Target Board Connector For EZ-ICE Probe

The ADSP-2106x EZ-ICE Emulator uses the IEEE 1149.1 JTAG test access port of the ADSP-2106x to monitor and control the target board processor during emulation. The EZ-ICE probe requires the ADSP-2106x's CLKIN, TMS, TCK, $\overline{\text{TRST}}$, TDI, TDO, $\overline{\text{EMU}}$, and GND signals be made accessible on the target system via a 14-pin connector (a pin strip header) such as that shown in Figure 11.3. The EZ-ICE probe plugs directly onto this connector for chip-on-board emulation. You must add this connector to your target board design if you intend to use the ADSP-2106x EZ-ICE. Be sure to allow enough room in your system to fit the EZ-ICE probe onto the 14-pin connector. The length of the traces between the connector and the ADSP-2106x's JTAG pins should be as short as possible.

The 14-pin, 2-row pin strip header is keyed at the pin 3 location—you must remove pin 3 from the header. The pins must be 0.025 inch square and at least 0.20 inch in length. Pin spacing should be 0.1 x 0.1 inches. The tip of the pins must be at least 0.10 inch higher than the tallest component under the emulator's probe to allow clearance for the bottom of the probe. Pin strip headers are available from vendors such as 3M, McKenzie, and Samtec.

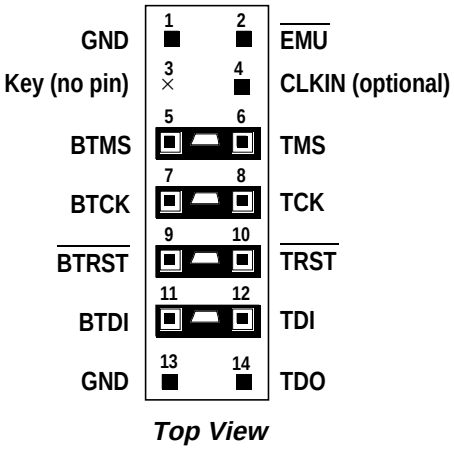


Figure 11.3 Target Board Connector For ADSP-2106x EZ-ICE Emulator (Jumpers In Place)

The BTMS, BTCK, $\overline{\text{BTRST}}$, and BTDI signals are provided so that the test access port can also be used for board-level testing. When the connector is not being used for emulation, place jumpers between the BXXX pins and the XXX pins as shown in Figure 11.3. If you are not going to use the test access port for board testing, tie $\overline{\text{BTRST}}$ to GND and tie or pullup BTCK to VDD. The $\overline{\text{TRST}}$ pin must be asserted after power-up (through $\overline{\text{BTRST}}$ on the connector) or held low for proper operation of the ADSP-2106x. None of the BXXX pins (pins 5, 7, 9, 11) are connected on the EZ-ICE probe.

The JTAG signals are terminated on the EZ-ICE probe as follows:

Signal	*Termination*
TMS	Driven through 82Ω resistor (16 mA / -3.2 mA driver)
TCK	Driven through 82Ω resistor (16 mA / -3.2 mA driver)
$\overline{\text{TRST}}$	Driven through 82Ω resistor (16 mA / -3.2 mA driver*) and pulled up by on-chip 20 kΩ resistor
TDI	Driven through 82Ω resistor (16 mA / -3.2 mA driver)
TDO	One TTL load, 92Ω Thevenin termination (160Ω / 220Ω)
CLKIN	One TTL load, 92Ω Thevenin termination (160Ω / 220Ω)
$\overline{\text{EMU}}$	4.7 kΩ pullup resistor, one TTL load (open-drain output from ADSP-2106xs)

* $\overline{\text{TRST}}$ is driven low if target board power is off, and continues to be driven low until the EZ-ICE probe is turned on by the EZ-ICE software (after the invocation command).

11 System Design

Figure 11.4 shows JTAG scan path connections for systems that contain multiple ADSP-2106x processors.

Connecting CLKIN to pin 4 of the EZ-ICE header is optional. The emulator only uses CLKIN when performing synchronous multiprocessor operations such as starting, stopping, and single-stepping multiple ADSP-2106xs. If you do not need these operations to occur synchronously on the multiple processors, simply tie pin 4 of the EZ-ICE header to ground.

If synchronous multiprocessor operations are needed and CLKIN is connected, however, clock skew between the multiple ADSP-2106x processors and the CLKIN pin on the EZ-ICE header *must be minimal.* If the skew is too large, synchronous operations may be off by one cycle between processors.

TCK, TMS, CLKIN (optional), and $\overline{\text{EMU}}$ should be treated as critical signals in terms of skew, and should be laid out as short as possible on your board. If TCK, TMS, and CLKIN are driving a large number of ADSP-2106xs (more than eight) in your system, then treat them as a “clock tree” using multiple drivers. (See “Clock Distribution” in the “High Frequency Design Considerations” section of this chapter.) If synchronous multiprocessor operations are not needed and CLKIN is not connected, just use appropriate parallel termination on TCK and TMS. TDI, TDO, and $\overline{\text{TRST}}$ are not critical signals in terms of skew.

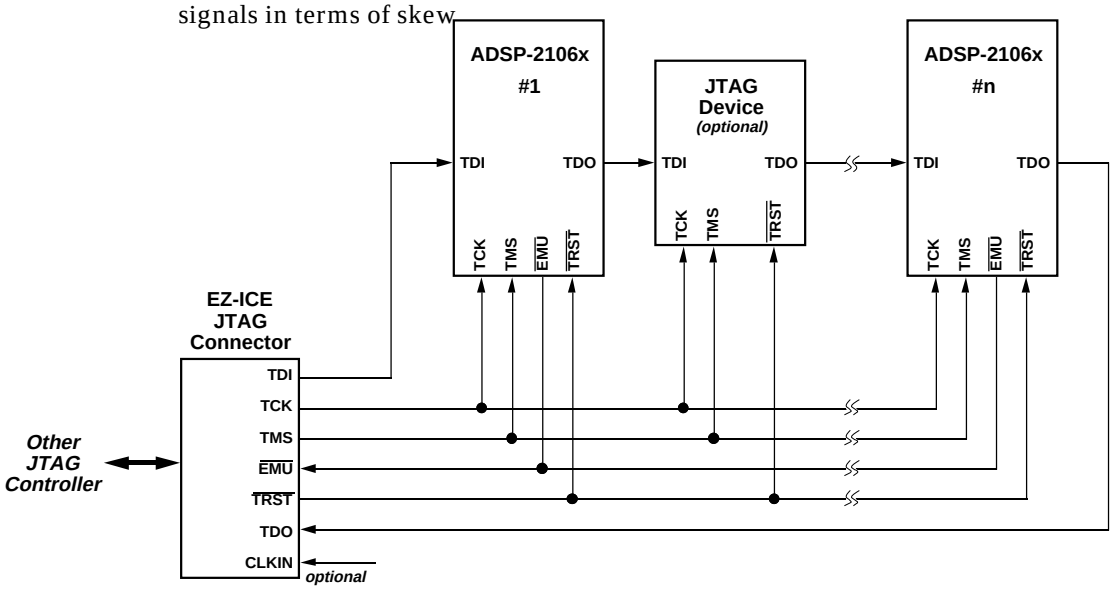


Figure 11.4 JTAG Scan Path Connections For Multiprocessor ADSP-2106x Systems

11.4 INPUT SIGNAL CONDITIONING

The ADSP-2106x SHARC processor is a CMOS device. It has input conditioning circuits which simplify system design by filtering or latching input signals to reduce susceptibility to glitches or reflections. The following sections describe why these circuits are needed and their effect on input signals.

A typical CMOS input consists of an inverter with specific N and P device sizes that cause a switching point of approximately 1.4V. This level is selected to be the midpoint of the standard TTL interface specification of V_{IL}=0.8V and V_{IH}=2.0V. This input inverter, unfortunately, has a fast response to input signals and external glitches wider than about 1 ns. Glitch rejection circuits, filter circuits, and hysteresis are therefore added after the input inverter on some ADSP-2106x inputs, as described below.

11.4.1 Glitch Rejection Circuits

The SHARC processors have on-chip glitch rejection circuits that latch certain input signals for a fixed time period after a transition has been detected. The purpose of these circuits is to make the input less sensitive to reflections and ringing once the first edge has been received. Thus, the circuits will not provide any reduced immunity to glitches that are randomly placed. The glitch rejection circuits are only used on some signals that are used as strobes. These signals are:

read and write strobes	$\overline{RD}$, $\overline{WR}$
DMA request inputs	$\overline{DMAR1}$, $\overline{DMAR2}$
serial port clock inputs	RCLK0, RCLK1, TCLK0, RCLK1

The glitch rejection circuit will cause the input signal to be latched for approximately 4 to 5 ns after a transition has been detected. Glitch rejection circuits are not implemented on the SHARC's data, address, or control lines that settle out normally before they are used. A glitch rejection circuit is used on the processor's clock input (CLKIN).

11.4.2 Link Port Input Filter Circuits

The SHARC's link port input signals have on-chip filter circuits rather than glitch rejection circuits. Filtering is not used on most signals since it delays the incoming signal and therefore the timing specifications. Filtering is implemented only on the link port data and clock inputs. This is possible because the link ports are self-synchronized, i.e. the clock and data are sent together. It is not the absolute delay but rather the relative delay between clock and data that determines performance margin.

By filtering both LxCLK and $LxDAT_{3-0}$ with identical circuits, response to LxCLK glitches and reflections are reduced but relative delay is unaffected. The filter has the effect of ignoring a full strength pulse (a glitch) narrower than approximately 2 ns. Glitches that are not full strength can be somewhat wider. The link ports do not use glitch rejection circuits because they can be used with longer, series-terminated transmission lines where the reflections do not occur near the signal transitions.

11.4.3 $\overline{\text{RESET}}$ Input Hysteresis

Hysteresis is used only on the $\overline{\text{RESET}}$ input signal. Hysteresis causes the switching point of the input inverter to be slightly above 1.4V for a rising edge and slightly below 1.4V for a falling edge. The value of the hysteresis is approximately ± 0.1V. The hysteresis is intended to prevent multiple triggering of signals which are allowed to rise slowly, as might be expected on a reset line with a delay implemented by an RC input circuit. Hysteresis is not used to reduce the effect of ringing on SHARC input signals with fast edges, because the amount of hysteresis that can be used on a CMOS chip is too small to make much difference. The small amount of hysteresis allowable is due to the restrictions on the tolerance of the V_{IL} and V_{IH} TTL input levels under worst case conditions. Refer to the *ADSP-2106x SHARC Data Sheet* for exact specifications.

11.5 HIGH FREQUENCY DESIGN CONSIDERATIONS

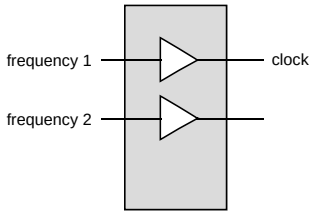
Because the ADSP-2106x processor can operate at very fast clock frequencies, signal integrity and noise problems must be considered for circuit board design and layout. The following sections discuss these topics and suggest various techniques to use when designing and debugging ADSP-2106x systems.

Initial versions of the ADSP-2106x are specified for operation at 40 MHz and 33 MHz clocks; the following information is based on these CLKIN frequencies. Refer to the most up-to-date *ADSP-2106x Data Sheet* for current clock speed specifications.

11.5.1 Clock Specifications & Jitter

The clock signal must be free of ringing and jitter. Clock jitter can easily be introduced in a system where more than one clock frequency exists. High frequency jitter on the clock to the ADSP-2106x may result in abbreviated internal cycles. The jitter should be kept to less than 0.25 ns for a 40 MHz clock and less than 0.5 ns for a 33 MHz (or slower) clock.

NEVER SHARE A CLOCK BUFFER IC WITH A SIGNAL OF A DIFFERENT CLOCK FREQUENCY. THIS WILL INTRODUCE EXCESSIVE JITTER.



Keep the portions of the system that operate at different frequencies as physically separate as possible.

The clock supplied to the ADSP-2106x must have a rise time of 3 ns or less and must meet or exceed a high and low voltage of 3V and 0.4V, respectively.

11.5.2 Clock Distribution

There must be low clock skew between ADSP-2106xs in a multiprocessor cluster when communicating synchronously on the external bus. The clock *must* be routed in a controlled-impedance transmission line that is properly terminated at either (1) the end of the line, or (2) the source. End-of-line termination (1) is illustrated in Figure 11.5. Source termination (2) is illustrated in Figure 11.6.

End-of-line termination (1) is not usually recommended unless the distance between the processors is very small, because devices that are at a different wire distance from each other will receive a skewed clock. This is due to the propagation delay of a PCB transmission line, which is typically 5 to 6 inches / ns.

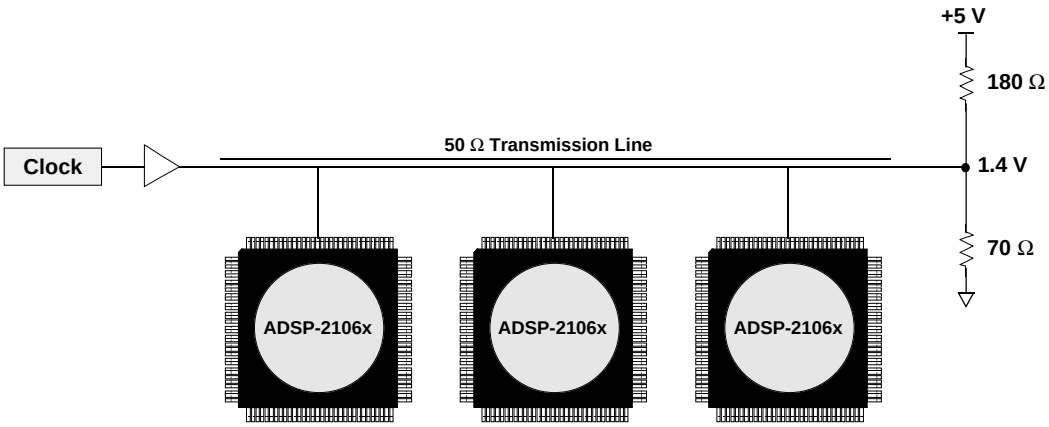


Figure 11.5 Not Recommended Clock Distribution Method (End-Of-Line Termination)

For source termination (2), Figure 11.6 shows an example of series-terminated transmission lines for clock distribution. This allows delays in each path to be identical. Each device must be at the end of the transmission line because only there does the signal have a single transition. The traces must be routed so that the delay through each is matched to the others. Line impedance higher than 50Ω may be used, but clock signal traces should be in the PCB layer closest to the ground plane to keep delays stable and crosstalk low. More than one device may be at the end of the line, but the wire length between them must be short and the impedance (capacitance) of these must be kept high. The matched inverters must be in the same IC and must be specified for a low skew (< 1 ns) with respect to each other. This skew should be as small as possible since it subtracts from the margin on most specifications.

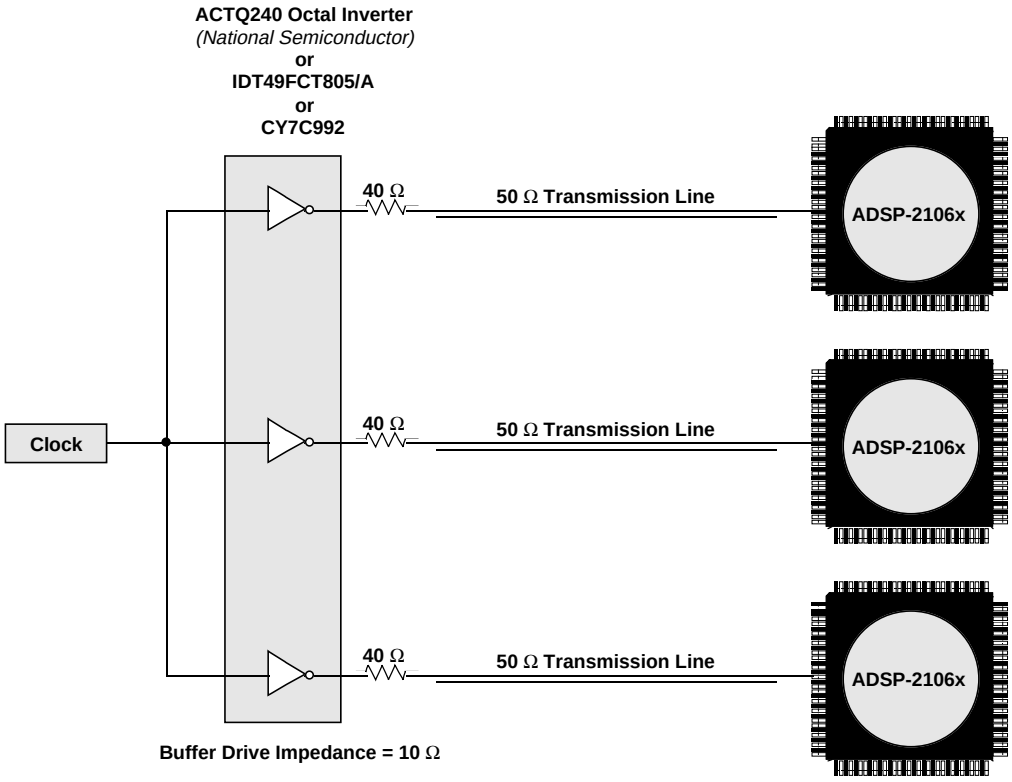


Figure 11.6 Recommended Clock Distribution Method (Source Termination)

11.5.3 Point-To-Point Connections

A series termination resistor may be added near the pin for point-to-point connections. This is typically used for link port applications when distances are greater than 6 inches. See Figure 11.7. For more specific guidence on related issues, see the reference source in the Recommended Reading section for suggestions on transmission line termination and see the ADSP-2106x Data Sheet for output drivers' rise and fall time data .

For link port operation at a 2X clock rate it is important to maintain low skew between the data ($LxDAT_{3-0}$) and clock (LxCLK). For 2X operation at CLKIN=40 MHz, a skew of less than 1.25 ns is required.

Although the ADSP-2106x's serial ports may be operated at a slow rate, the output drivers still have fast edge rates and may require source termination for longer distances.

11 System Design

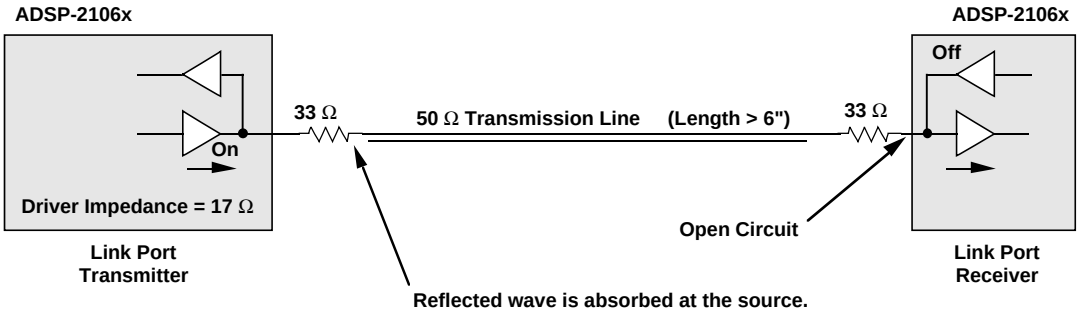


Figure 11.7 Source Termination For Long-Distance Point-To-Point Connections

11.5.4 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 the read and write strobes ($\overline{RD}$, $\overline{WR}$) and acknowledge (ACK). In a multiprocessor cluster, each ADSP-2106x can drive the read or write strobes. In this case, some damping resistance should be put in the signal path if the line length is greater than 6 inches; this will be at the expense of additional signal delay, however. The time budget for these signals should be carefully analyzed.

Two possible damping arrangements between four ADSP-2106xs are shown in Figures 11.8 and 11.9. In Figure 11.8, a star connection of resistors is used. Each ADSP-2106x can drive the signal (e.g. $\overline{RD}$ or $\overline{WR}$ strobe). Trace lengths should be minimized. Experiment with the optimal resistance value and placement, e.g. near the processor or near the common connection. This will add signal delay, however.

In the example of Figure 11.9, where processors 1 & 2 and 3 & 4 are close to each other, a single damping resistor between the processor pairs will help damp out reflections. Experiment with the resistor value. The two processor groups will have a skew with respect to each other.

Another solution to multiple drivers where longer distances are involved is to have a single transmission line that is terminated at both ends. This arrangement is shown in Figure 11.10. The stubs to the processors must be kept as short as possible. Each device driver sees

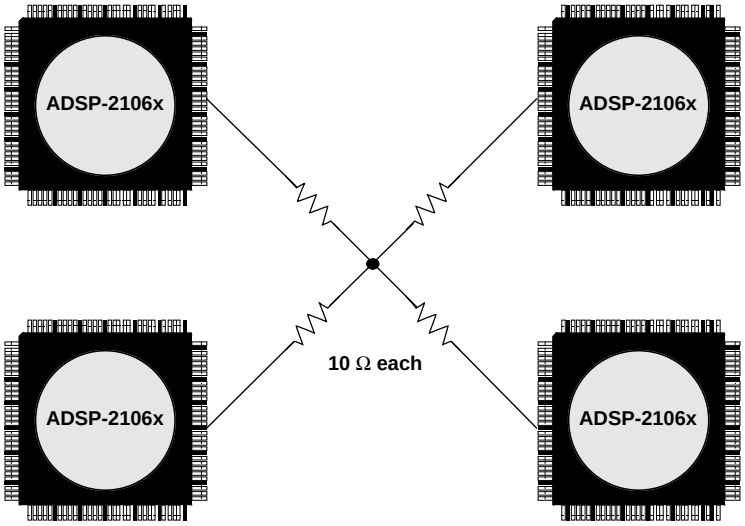


Figure 11.8 Star Connection Damping Resistors

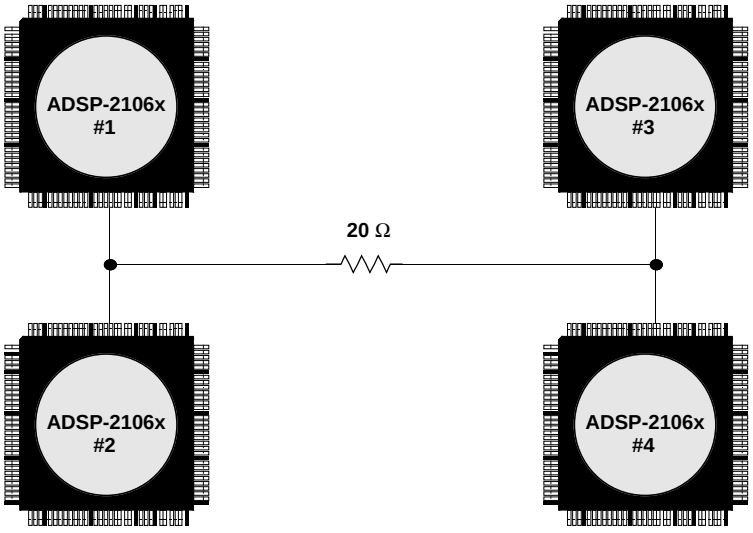
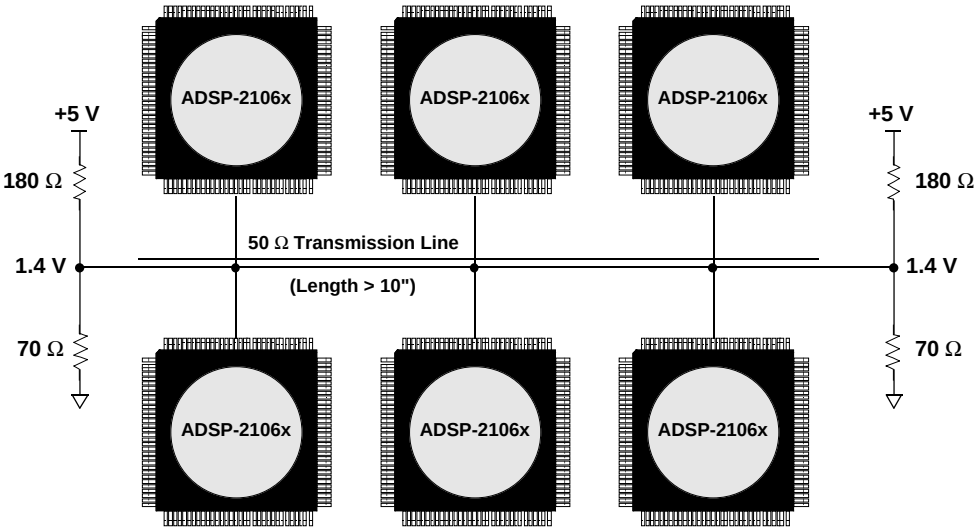


Figure 11.9 Single Damping Resistor Between Processor Groups

an impedance of 25Ω, but this resistor is biased at 1.4V so the drive from the ADSP-2106xs will be sufficient for TTL levels. To reduce power dissipation in the system and in each ADSP-2106x, this should only be used, if necessary, for signals such as the $\overline{RD}$ or $\overline{WR}$ strobe. The signals will be skewed but well behaved.

Figure 11.10 Single Transmission Line Terminated At Both Ends



11.5.5 Other Recommendations & Suggestions

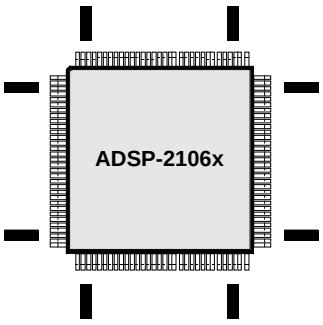
- The use of more than one ground plane on the PCB will reduce crosstalk. Be sure to use lots of vias between the ground planes. One VDD plane is sufficient. These planes should be in the center of the PCB.
- Keep critical signals such as clocks, strobes, and bus requests on a signal layer next to a ground plane and away from (or layout perpendicular to) other non-critical signals to reduce crosstalk. For example, data outputs switch at the same time that BR inputs are sampled; if your layout permits crosstalk between them, your system could have problems with bus arbitration.
- If possible, position the processors on both sides of the board to reduce area and distances.
- Lower transmission line impedances will reduce crosstalk and allow better control of impedance and delay.

- The use of 3.3V components and power supplies will help transmission line problems significantly because the receiver switching voltage of 1.5V is close to the middle of the voltage swing. In addition, ground bounce and noise coupling will be less. The ADSP-2106x is available in a 3.3V version.
- 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.

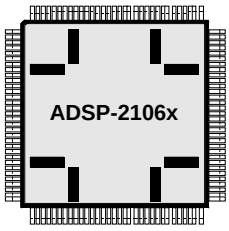
11.5.6 Decoupling Capacitors & Ground Planes

Ground planes must be used for the ground and power supplies. A minimum of eight bypass capacitors (0.02 µF ceramic) should be used, placed very close to the VDD pins of the package. See Figure 11.11. Use short and fat traces for this. The ground end of the capacitors should be tied directly to the ground plane outside the package footprint of the ADSP-2106x, not within the footprint (i.e. underneath it, on the bottom of the board). 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 will be reduced. In addition, there should be several large tantalum capacitors on the board.

Figure 11.11 Bypass Capacitor Placement



Place bypass capacitors outside of the package.



Do not put the bypass capacitors within the package footprint (on the underside of the board) if signal feedthroughs perforate the ground plane. Return current must flow to the outside ground plane, which may be poorly connected to the inside ground plane if this is perforated by feedthroughs.

11.5.7 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 will cause ringing to be seen on the displayed trace and will make the signal appear to have excessive overshoot and undershoot. A 1 GHz or better sampling oscilloscope is needed to see the signals accurately.

11.5.8 Recommended Reading

High-Speed Digital Design: A Handbook of Black Magic is recommended for further reading. This book is a technical reference that covers the problems encountered in state-of-the-art, high-frequency digital circuit design, and 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

High-Speed Digital Design: A Handbook of Black Magic
Johnson & Graham
Prentice Hall, Inc.
ISBN 0-13-395724-1

11.6 BOOTING

Programs can be automatically downloaded to the internal memory of an ADSP-2106x after power-up or after a software reset. This process is

called booting.

The ADSP-2106x supports three booting modes: EPROM, host, and link port. “No boot” mode may also be configured. Each booting mode packs boot data into 48-bit instructions and uses Channel 6 of the ADSP-2106x’s on-chip DMA controller to transfer the instructions to internal memory. For EPROM booting via the external port, the ADSP-2106x reads data from an 8-bit external EPROM. For host port booting, the ADSP-2106x accepts data from a 16-bit host microprocessor (or other external device). For link port booting, the ADSP-2106x receives 4-bit wide data in link buffer 4. If *no boot* mode is selected, the ADSP-2106x starts executing instructions from address 0x0040 0004 in external memory.

The primary configuration of DMA Channel 6, used with external port buffer EPB0, is used for EPROM and host booting. The alternate configuration of DMA Channel 6, for link buffer 4, is used for link port booting. The DMAC6 control register is specially initialized for booting in each case.

After the boot process loads 256 words into memory locations 0x20000 through 0x200FF (using any boot method), the processor begins executing instructions. Because most applications require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. Analog Devices supplies a loading routine (Loader Kernel) that can load an entire program. This routine comes with the development tools. For more information on the Loader Kernel, see the development tools documentation.

The following sections discuss the different booting modes in detail and describe additional functionality related to booting.

11.6.1 Selecting The Booting Mode

The booting mode is selected using the LBOOT, EBOOT, and $\overline{\text{BMS}}$ pins, as shown in Table 11.2.

EPROM booting is selected when the EBOOT input is high. This causes $\overline{\text{BMS}}$ to become an output, to be used as the boot EPROM chip select. When EBOOT is low, $\overline{\text{BMS}}$ becomes an input used to select between

Pin	*Type*	*Description*
EBOOT	I	**EPROM Boot Select**. When EBOOT is high, the ADSP-2106x is configured for booting from an 8-bit EPROM. When EBOOT is low, the LBOOT and $\overline{\text{BMS}}$ inputs determine booting mode. See table below. This signal is a system configuration selection which should be hardwired.
LBOOT	I	**Link Boot – Host Boot Select**. When LBOOT is high, the ADSP-2106x is configured for link port booting. When LBOOT is low, the ADSP-2106x is configured for host processor booting or no booting. See table below. This signal is a system configuration selection which should be hardwired.
$\overline{\text{BMS}}$	I / O / T*	**Boot Memory Select**. *Output:* Used as chip select for boot EPROM devices (when EBOOT=1, LBOOT=0). In a multiprocessor system, $\overline{\text{BMS}}$ is output by the bus master. *Input:* When low, indicates that no booting will occur and that ADSP-2106x will begin executing instructions from external memory. See table below. This input is a system configuration selection which should be hardwired.

* Tristatable only in EPROM boot mode (when $\overline{\text{BMS}}$ is an output).

Table 11.2 Boot Mode Selection Pins

EBOOT	*LBOOT*	*BMS*	*Booting Mode*
1	0	output	EPROM (connect $\overline{\text{BMS}}$ to EPROM chip select)
0	0	1 (input)	Host processor
0	1	1 (input)	Link port
0	0	0 (input)	No booting. Processor executes from external memory.
0	1	0 (input)	*reserved*
1	1	x (input)	*reserved*

host boot mode or *no boot* mode. In EPROM boot mode, $\overline{BMS}$ is deasserted when the ADSP-2106x is not the bus master.

Note that when using any of the power-up booting modes, address 0x0002 0004 should not contain a valid instruction since it is not executed during the booting sequence. A NOP or IDLE instruction should be placed at this location.

11.6.2 EPROM Booting

EPROM booting through the external port is selected when the EBOOT input is high. The byte-wide boot EPROM must be connected to data bus pins 23-16 ($DATA_{23-16}$). The lowest address pins of the ADSP-2106x should be connected to the EPROM's address lines. The EPROM's chip select should be connected to $\overline{BMS}$ and its output enable should be connected to $\overline{RD}$.

In a multiprocessor system, the $\overline{BMS}$ output is only driven by the ADSP-2106x bus master. This allows wire-ORing of multiple $\overline{BMS}$ signals for a single common boot EPROM.

➠ You can boot any number of ADSP-2106x's from a single EPROM, using the same code for each processor *or* differing code for each.

During reset, the ADSP-2106x's ACK line is internally pulled high with a 2 kΩ equivalent resistor and is held high with an internal keeper latch. It is not necessary to use an external pullup resistor on the ACK line during booting or at any other time.

11.6.2.1 Bootstrapping (256 Instructions)

When EPROM boot mode is configured, the External Port DMA Channel 6 (DMAC6) becomes active following reset; it is initialized to 0x02A1, which allows external port DMA enable and selects DTYPE for instruction words. The packing mode bits (PMODE) are ignored, and 8-to-48 bit packing is forced with least-significant-word first.

The UBWS and UBWM fields of the WAIT register are initialized to generate six wait states (seven cycles total) for the EPROM access in unbanked external memory space. (Note that wait states defined for unbanked memory are applied to $\overline{BMS}$-asserted accesses.)

The UBWM field's initial value selects internal wait and external acknowledge. Initially, the SHARC asserts acknowledge (high), but (if another device drives acknowledge low during EPROM boot) the

SHARC could latch acknoledge low. The SHARC responds to the deasserted (low) acknowledge as a hold off from the EPROM, inserting wait states continually and preventing completion of the EPROM boot. To avoid this type of boot holdoff, change the value in the WAIT register, setting the UBWM value to internal wait mode (01) early in the 256 word boot process.

Table 11.3 shows how the DMA Channel 6 parameter registers are initialized at reset for EPROM booting. The count register (C6) is initialized to 0x0100 for transferring 256 words to internal memory. The external count register (EC6), which is used when external addresses are generated by the DMA controller, is initialized to 0x0600 (i.e. 0x0100 words with six bytes per word).

Parameter Register	**Initialization Value**
II6	0x0002 0000
IM6	*uninitialized (increment by 1 is automatic)*
C6	0x0100 (256 instruction words)
CP6	*uninitialized*
GP6	*uninitialized*
EI6	0x0040 0000
EM6	*uninitialized (increment by 1 is automatic)*
EC6	0x0600 (256 words × 6 bytes / word)

Table 11.3 DMA Channel 6 Parameter Register Initialization For EPROM Booting

At system start-up, when the ADSP-2106x's $\overline{\text{RESET}}$ input goes inactive, the following sequence occurs:

1. The ADSP-2106x goes into an idle state, identical to that caused by the IDLE instruction. The program counter (PC) is set to address 0x0002 0004.
2. The DMA parameter registers for channel 6 are initialized (as shown in Table 11.3 above).
3. $\overline{\text{BMS}}$ becomes the boot EPROM chip select.
4. 8-bit Master Mode DMA transfers from EPROM to internal memory begin, on the external port data bus lines 23-16.
5. The external address lines ($ADDR_{31-0}$) start at 0x0040 0000 and increment after each access.
6. The $\overline{\text{RD}}$ strobe asserts as in a normal memory access, with six wait states (seven cycles).

The ADSP-2106x's DMA controller reads the 8-bit EPROM words, packs them into 48-bit instruction words, and transfers them to internal memory until 256 words have been loaded. The EPROM is automatically

selected by the $\overline{BMS}$ pin; other memory select pins are disabled. The DMA external count register (EC6) decrements after each EPROM transfer. When EC6 reaches zero, the following wake-up sequence occurs:

1. The DMA transfers stop.
2. The External Port DMA Channel 6 interrupt (EP0I) is activated.
3. $\overline{BMS}$ is deactivated and normal external memory selects are activated.
4. The ADSP-2106x vectors to the EP0I interrupt vector at 0x0002 0040.

At this point the ADSP-2106x has completed its booting mode and is executing instructions normally. The first instruction at the EP0I interrupt vector location, address 0x0002 0040, should be an RTI (Return From Interrupt). This will return execution to the reset routine at location 0x0002 0005 where normal program execution can resume. After this has occurred, your program can write a different service routine at the EP0I vector location 0x0002 0040.

Remember that when using any of the power-up booting modes, location 0x0002 0004 should not contain a valid instruction since it is not executed during the booting sequence. A NOP or IDLE instruction should be placed at this location.

11.6.2.2 Loading The Remaining EPROM Data

The EPROM boot mode only loads 256 instructions during bootstrapping. If your entire application must be loaded into internal memory from the EPROM, the ADSP-2106x must gain access to the boot EPROM after bootstrapping has completed. The BSO bit in the SYSCON register provides this capability.

The BSO bit, when set, overrides the external memory selects and causes the $\overline{BMS}$ pin to assert (low) for an external port DMA transfer. Your bootstrap program should first set the BSO bit in SYSCON and then set up an external port DMA channel to read the rest of the EPROM's contents. Any of the four external port DMA channels may be used:

Channel#	**Control Register**	**Data Buffer**
DMA Channel 6	DMAC6	EPB0 *(Ext. Port DMA Buffer 0)*
DMA Channel 7	DMAC7	EPB1 *(Ext. Port DMA Buffer 1)*
DMA Channel 8	DMAC8	EPB2 *(Ext. Port DMA Buffer 2)*
DMA Channel 9	DMAC9	EPB3 *(Ext. Port DMA Buffer 3)*

When BSO=1, the PMODE packing mode bits in the DMAC6 control register are ignored and 8-to-48 bit packing is forced for reads. (Note that 8-bit packing is only available during EPROM booting or on DMA reads

when BSO is set.) While one of the external port DMA channels is being used in conjunction with the BSO bit, none of the other three channels may be used.

When BSO=1, $\overline{\text{BMS}}$ is *not* asserted by a core processor access, only by a DMA transfer. This allows your bootstrap program (running on the ADSP-2106x core) to perform other external accesses to non-boot memory.

11.6.2.3 Writing to BMS Memory Space

You can also write to ADSP-2106x (SHARC) BMS space using the boot select override (BSO mode). The BSO (Boot Select Override) mode bit in the SYSCON register allows the $\overline{\text{BMS}}$ pin to be asserted under software control. In many systems, the boot data may need to be updated or modified. In these cases, the PROM may be substituted by a write-able EEPROM or FLASH memory.

To write to memory with the $\overline{\text{BMS}}$ asserted, use DMA channels 7, 8 or 9, but not DMA channel 6. With BSO set, DMA channel 6 should only be used for reads. This limitation of accesses appears because DMA channel 6 is hardwired for a special 8-bit boot read mode. When BSO is set, a write with DMA channel 6 with BSO set results in illegal chip operation.

When BSO is set, DMA channels 7-9 can be used with any of the modes available in the DMACx register (for read or write), any packing mode, and any data or instruction.

On DMA writes with BSO=1, 16- to 48-bit packing is forced and the PMODE bits are ignored. Because BMS space is 8-bits wide and no 8-bit packing mode is available for these writes, one must use the shifter to place data in the correct location for each write.

11.6.3 Host Booting

Booting the ADSP-2106x from a 16-bit host processor is performed via the data and address buses of the external port. The ADSP-2106x's LBOOT pin selects between link port booting and host port booting: LBOOT must be low for host booting, with EBOOT low and $\overline{\text{BMS}}$ high. When host booting is configured, the ADSP-2106x will enter slave mode after reset and wait for the host to download the boot program.

After reset the ADSP-2106x goes into an idle state, identical to that caused by the IDLE instruction, with the program counter (PC) set to address 0x0002 0004. The parameter registers for External Port DMA Channel 6 are initialized as shown below in Table 11.4, but no DMA transfers are started.

The DMA Channel 6 Control Register (DMAC6) is initialized to 0x00A1, which allows external port DMA enable and selects DTYPE for instruction

words, PMODE for 16-to-48 bit word packing, and least-significant-word first. Because the host processor is accessing the EPB0 external port buffer, the HPM host packing mode bits of the SYSCON register must be set to correspond to the external bus width specified by the PMODE bits of the DMAC6 control register. If a different packing mode is desired, the host must write to DMAC6 and SYSCON to change the PMODE and HPM settings.

Parameter Register	**Initialization Value**
II6	0x0002 0000
IM6	*uninitialized (increment by 1 is automatic)*
C6	0x0100 (256 instruction words)
CP6	*uninitialized*
GP6	*uninitialized*
EI6	*uninitialized*
EM6	*uninitialized*
EC6	*uninitialized*

Table 11.4 Ext. Port DMA Channel 6 Parameter Register Initialization For Host Booting

The host initiates the booting operation by asserting the ADSP-2106x's Host Bus Request input, $\overline{\text{HBR}}$. This tells the ADSP-2106x that the default 16-bit bus width will be used. The host may also optionally assert the $\overline{\text{CS}}$ chip select input to allow asynchronous transfers (as described in the *Host Interface* chapter).

After the host receives the $\overline{\text{HBG}}$ (Host Bus Grant) signal back from the ADSP-2106x, it can start downloading instructions by writing directly to EPB0, the external port DMA buffer 0 (which corresponds to DMA channel 6), or it can change the reset initialization conditions of the ADSP-2106x by writing to any of the IOP control registers. The host must use data bus pins 31-16 ($DATA_{31-16}$).

When 256 instructions have been downloaded, the following wake-up sequence occurs:

1. The DMA transfers stop.
2. The External Port DMA Channel 6 interrupt (EP0I) is activated.
3. The ADSP-2106x vectors to the EP0I interrupt vector at 0x0002 0040.

The first instruction at the EP0I interrupt vector location, address 0x0002 0040, should be an RTI (Return From Interrupt). This RTI will

return execution to the reset routine at location 0x0002 0005 where normal program execution can resume. After this 256 word load and RTI have occurred, your program can write a different service routine at the EP0I vector location 0x0002 0040. These 256 instructions must serve as a loader that loads the rest of your program.

Note that External Port DMA Channel 6 must be used for the initial instruction download because only this channel has its IMASK bit set to enable a *DMA done* interrupt. VIRPT vector interrupts are disabled at reset, and must be enabled by your ADSP-2106x program (in the IMASK register).

Note that a master ADSP-2106x may boot a slave ADSP-2106x by writing to its DMAC6 control register and setting the packing mode (PMODE) to 00. This allows instructions to be downloaded directly without packing. The wait state setting of 6 on the slave ADSP-2106x does not affect the speed of the download since wait states only affect bus master operation.

11.6.4 Link Port Booting

The ADSP-2106x can also be booted through Link Buffer 4 using DMA Channel 6. A four-bit-wide external device must be used to download instructions after system powerup. The output of an eight-bit-wide EPROM can be converted to nibble data by using a 2-to-1 multiplexer on the output, with the address LSB selecting the high- or low-order nibble.

The external device must provide a clock signal to the link port assigned to link buffer 4. The clock can be any frequency, up to a maximum of the ADSP-2106x clock frequency. The clock's falling edges strobe the data into the link port. The most significant 4-bit nibble of the 48-bit instruction must be downloaded first.

The link port booting operation is similar to the host booting operation; the II6 and C6 parameter registers for DMA Channel 6 are initialized to the same values. The DMA Channel 6 Control Register (DMAC6) is initialized to 0x00A0, which disables external port DMA and selects DTYPE for instruction words. The LCTL and LCOM link port control registers are overridden during link port booting to allow link buffer 4 to receive 48-bit data. Form more information on the booting process, see the *Host Booting* section.

11.6.5 Multiprocessor Booting

Multiprocessor systems can be booted from a host processor, from external EPROM, through a link port, or from external memory.

11.6.5.1 Multiprocessor Host Booting

To boot multiple ADSP-2106x processors from a host, each ADSP-2106x must have its EBOOT, LBOOT, and $\overline{\text{BMS}}$ pins configured for host booting: EBOOT=0, LBOOT=0, and $\overline{\text{BMS}}$=1. After system powerup, each ADSP-2106x will be in the idle state and the $\overline{\text{BR}}$x bus request lines will be deasserted. The host must assert the $\overline{\text{HBR}}$ input and boot each ADSP-2106x by asserting its $\overline{\text{CS}}$ pin and downloading instructions as described in "Host Booting" above.

11.6.5.2 Multiprocessor EPROM Booting

There are two methods of booting a multiprocessor system from an EPROM. Processors perform the following steps in these methods:

- Arbitrate for the bus
- DMA the 256 word boot stream, after becoming bus master
- Release the bus
- Execute the loaded instructions

All ADSP-2106xs boot in turn from a single EPROM.

The $\overline{\text{BMS}}$ signals from each ADSP-2106x may be wire-ORed together to drive the chip select pin of the EPROM. Each ADSP-2106x can boot in turn, according to its priority. When the last one has finished booting, it must inform the others (which may be in the idle state) that program execution can begin (if all SHARCs are to begin executing instructions simultaneously). An example system that uses this processors-take-turns technique appears in Figure 11.12. When multiple SHARCs boot from one EPROM, the SHARCs can boot either identical code or different code from the EPROM. If the processors load differing code, a jump table (based on processor ID) can be used to select the code for each processor.

11 System Design

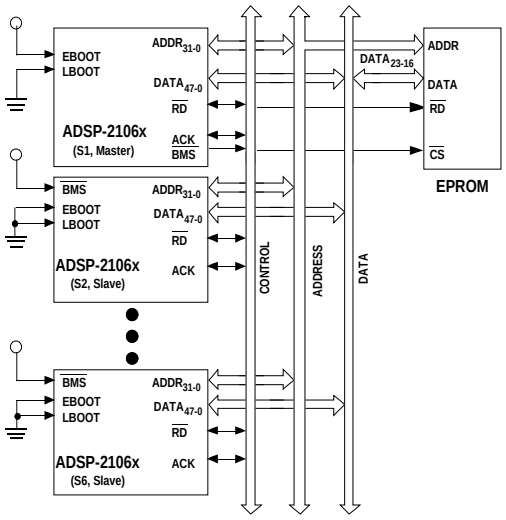


Figure 11.12 Multiple SHARCs Booting From One EPROM, Processors-Take-Turns

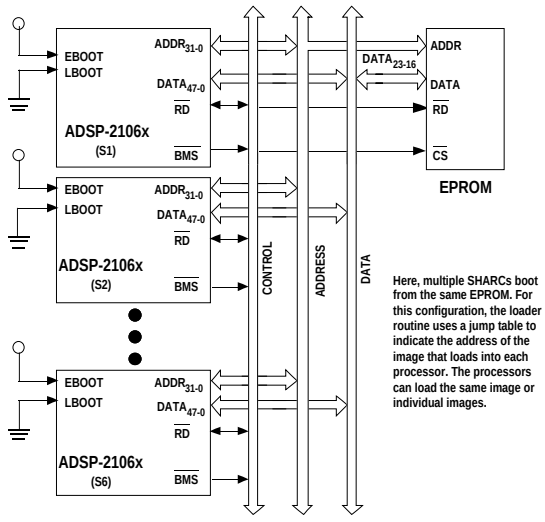


Figure 11.13 Multiple SHARCs Booting From One EPROM, One-Boots-Others

One ADSP-2106x is booted, which then boots the others.
The EBOOT pin of the ADSP-2106x with ID=1 must be set high for EPROM booting. All other ADSP-2106xs should be configured for host booting (EBOOT=0, LBOOT=0, and $\overline{\text{BMS}}$=1), which leaves them in the idle state at startup and allows the ADSP-2106x with ID=1 to become bus master and boot itself. Only the $\overline{\text{BMS}}$ pin of ADSP-2106x #1 is connected to the chip select of the EPROM. When ADSP-2106x #1 has finished booting, it can boot the remaining ADSP-2106xs by writing to their external port DMA buffer 0 (EPB0) via multiprocessor memory space. An example system that uses this one-boots-others technique appears in Figure 11.13.

11.6.5.3 Multiprocessor Link Port Booting
In systems where multiple ADSP-2106xs are not connected by the parallel external bus, booting can be accomplished from a single source through the link ports. To simultaneously boot all of the ADSP-2106xs, a parallel common connection should be made to Link Buffer 4 on each of the processors. If only a daisy chain connection exists between the processors' link ports, then each ADSP-2106x can boot the next one in turn. Link Buffer 4 must always be used for booting.

11.6.5.4 Multiprocessor Booting From External Memory
If external memory contains a program after reset, then the ADSP-2106x with ID=1 should be set up for *no boot* mode; it will begin executing from address 0x0040 0004 in external memory. When booting has completed, the other ADSP-2106xs may be booted by ADSP-2106x #1 if they are set up for host booting, or they can begin executing out of external memory if they are set up for *no boot* mode. Multiprocessor bus arbitration will allow this booting to occur in an orderly manner. The bus arbitration sequence after reset is described in the *Multiprocessing* chapter of this manual.

11.6.6 "No Boot" Mode
The *no boot* mode of the ADSP-2106x causes the processor to start fetching and executing instructions at address 0x0040 0004 in external memory space. In this mode, all DMA control and parameter registers are set equal to their default initialization values.

11.6.7 Interrupt Vector Table Location
If the ADSP-2106x is booted from an external source (i.e. EPROM, host, or link port booting), the interrupt vector table will be located in internal memory. If, however, the ADSP-2106x is not booted, and will execute from external memory, the vector table must be located in the external memory.

The IIVT bit in the SYSCON control register can be used to override the booting mode in determining where the interrupt vector table is located. If the ADSP-2106x is not booted (*no boot* mode), setting IIVT to 1 selects an internal vector table while IIVT=0 selects an external vector table. If the ADSP-2106x is booted from an external source (any mode other than *no boot* mode), then IIVT has no effect. The default initialization value of IIVT is zero.

11.7 IMPORTANT PROGRAMMING REMINDERS

This section summarizes information that you should keep in mind when writing programs.

11.7.1 Extra Cycle Conditions

All instructions can execute in a single cycle but may take longer in some cases as described below.

11.7.1.1 Nondelayed Branches

A nondelayed branch instruction (JUMP, CALL, RTS or RTI) fetches but does not execute the two instructions that follow it. Instead, these operations are aborted and the processor executes two NOPs.

This two-cycle delay can be avoided by using delayed branches, which execute the two instructions following the branch instruction. The tradeoff is that the actual program flow does not match the apparent order of operations in the program; you must remember that the two extra instructions are executed before the branch is taken.

11.7.1.2 Program Memory Data Access With Cache Miss

The ADSP-2106x checks the instruction cache on every program memory data access. If the instruction needed is in the cache, the instruction fetch from the cache happens in parallel with the PM bus data access and the instruction executes in a single cycle. However, if the instruction is not in the cache, the ADSP-2106x must wait for the PM bus data access to complete before it can fetch the next instruction. This results in a minimum one-cycle delay, more if the PM bus data access uses external memory with wait states.

This delay will occur even if the PM bus data access is conditional and the condition is false.

See “Dual Data Accesses” later in this section for additional information.

11.7.1.3 Program Memory Data Access In Loops

The ADSP-2106x caches an instruction that it needs to fetch during the execution of a PM bus data access. Because of the execution pipeline, this instruction is usually two memory locations after the PM bus data access. If the PM bus data access is in a loop, there will usually be a cache miss on the first iteration of the loop and cache hits on subsequent iterations, for a total of one extra cycle during the loop execution.

However, there are certain cases in which different instructions are needed from the cache at different iterations. In these cases the number of cache misses, and therefore extra cycles, increases. These situations are summarized below. Note that this table is based on the worst-case scenario; the actual performance of the cache for a given program may be better.

Cache Misses	*Loop Length (# instructions)*	*Address of PM Bus Data Access*
1	> 2	Not at *e* or (*e* – 1)
2	≥ 2	At *e* or (*e* – 1)
3	1	At the single loop location

e = loop end address

Two Misses: If the program memory data access occurs in the last two instructions of a loop, there will usually be cache misses on the first and the last loop iteration, for a total of two extra cycles. On the first iteration, the ADSP-2106x needs to fetch from the top of the loop (the first or second instruction). On the last iteration, the ADSP-2106x needs to fetch one of the two instructions following the loop. At each of these points there will be a cache miss the first time the code containing the loop is executed.

Three Misses: If a loop contains only one instruction, and that instruction requires a PM bus data access, there are potentially three cache misses. On the first iteration, the processor needs to fetch the loop instruction again (if the loop iterates three times or more). On the next-to-last iteration, the processor needs to fetch the instruction following the loop. On the last instruction, the processor needs to fetch the second instruction following the loop. In each case, there will be a cache miss the first time the code containing the loop is executed.

11.7.1.4 One- & Two-Instruction Loops

Counter-based loops that have only one or two instructions can cause delays if not executed a minimum number of times. The ADSP-2106x checks the termination condition two cycles before it exits the loop. In these short loops, the ADSP-2106x has already looped back when the termination condition is tested. Thus, if the termination condition tests true, the two instructions in the pipeline must be aborted and NOPs executed instead.

Specifically, a loop of length one executed one or two times or a loop of length two executed only once incurs two cycles of overhead because there are two aborted instructions after the last iteration. Note that these overhead cycles are in addition to any extra cycles caused by a PM bus data access inside the loop (see previous section). To avoid overhead, use straight-line code instead of loops in these cases.

11.7.1.5 DAG Register Writes

When an instruction that writes to a DAG register is followed by an instruction that uses any register in the same DAG for data addressing, modify instructions, or indirect jumps, the ADSP-2106x inserts an extra (NOP) cycle between the two instructions. This happens because the same bus is needed by both operations in the same cycle, therefore the second operation must be delayed. An example is:

```
L2=8;
DM(I0,M1)=R1;
```

Because L2 is in the same DAG as I0 (and M1), an extra cycle is inserted after the write to L2.

11.7.1.6 Wait States

An external memory access can be programmed to include a specific number of wait states and bus idle cycles, and (or) to wait for an external acknowledge signal (ACK) before completing. If only internally programmed wait states and bus idle cycles are used, the delay is exactly the number of wait states and bus idle cycles (1 waitstate = 1 cycle). If the external acknowledge signal is used, either alone or in combination with programmed wait states, the delay depends on the external system and can vary.

11.7.2 Delayed Branch Restrictions

A delayed branch instruction and the two instructions that follow it must be executed sequentially. Any interrupt that occurs between a

delayed branch instruction and either of the two instructions that follow is not processed until the branch is complete.

Delayed branching can be used with the JUMP, CALL, RTS, and RTI instructions. For delayed JUMPs, the following instructions may not be used in the two locations immediately after the jump:

- Other JUMP, CALL, RTS, or RTI instructions
- DO UNTIL instruction

For a delayed CALL, RTS, or RTI, the following instructions may not be used in the next two locations:

- Other JUMP, CALL, RTS, or RTI instructions
- DO UNTIL instruction
- Pushes or Pops of the PC stack
- Writes to the PC stack or PC stack pointer

11.7.3 Circular Buffer Initialization

You set up a circular buffer by initializing an L register with a positive, nonzero value and loading the corresponding (same-numbered) B register with the base address of the buffer. (The base address, or starting address, is the lowest address of the buffer.) The corresponding I register is automatically loaded with this same starting address.

11.7.4 Disallowed DAG Register Transfers

The following instructions execute on the ADSP-2106x, but cause incorrect results. These instructions are disallowed by the assembler:

- An instruction that stores a DAG register in memory using indirect addressing from the same DAG, with or without update of the index register (I). The instruction writes the wrong data to memory or updates the wrong index register.

```
DM(M2,I1)=I0;    or      DM(I1,M2)=I0;
```

- An instruction that loads a DAG register from memory using indirect addressing from the same DAG, with update of the index register. The instruction will either load the DAG register or update the index register, but not both.

```
L2=DM(I1,M0);
```

11.7.5 Two Writes To Register File

If two writes to the same register file location take place in the same cycle, only the write with higher precedence actually occurs. Precedence is determined by the source of the data being written; from highest to lowest, the precedence is:

- Data memory (DM bus) or universal register
- Program memory (PM bus)
- ALU
- Multiplier
- Shifter

11.7.6 Computation Units

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

The ALU Zero flag (AZ) signifies floating-point underflow as well as a zero result.

Transfers between MR registers and the register file are considered multiplier operations, and are listed with the multiplier operations in Appendix B, *Compute Operation Reference.*

11.7.7 Memory Space Access Restrictions

The ADSP-2106x's three internal buses, PM, DM, and I/O, can be used to access the processor's memory map according to the following rules:

- The DM bus can access all memory spaces.
- The PM bus can access only Internal Memory Space and the lowest 12 megawords of External Memory Space.
- The I/O bus can access all memory spaces except for the memory-mapped IOP registers (in Internal Memory Space).

Note that in silicon revision 1.0 and earlier, pre-modify addressing operations must not change the memory space of the address; for example, pre-modification of an address in Internal Memory Space should not generate an address in External Memory Space. The one exception to this rule is: an indirect JUMP or CALL instruction with pre-modify addressing *can* jump from internal memory to external memory. Silicon revisions 2.x and later do not have this pre-modify limitation.

11.7.8 Mixing 32-Bit & 48-Bit Words In A Memory Block

32-bit data words and 48-bit instruction words can be stored in the same memory block, with the restriction that *all instructions must reside at addresses lower than the data.* No instruction may be stored at an address higher than the lowest address of any data word. This restriction is necessary to prevent addresses for 32-bit words and 48-bit words from overlapping. Instruction storage must start at the lowest address in the block.

11.7.9 16-Bit Short Words

16-bit short words read into ADSP-2106x registers are automatically extended into 32-bit integers. The upper 16 bits can be zero-filled or sign-extended, as determined by the value of the SSE bit in the MODE1 register. If SSE=0, the upper 16 bits are zero-filled. If SSE=1, the upper 16 bits are sign-extended (except when reading a short word into the PX register, which is always zero-filled).

11.7.10 Dual Data Accesses

The ADSP-2106x's PM and DM buses allow the processor core to simultaneously access instructions and data from both memory blocks. Instructions are fetched over the PM bus or from the instruction cache. Data can be accessed over both the DM bus (using DAG1) and the PM bus (using DAG2).

The ADSP-2106x's two memory blocks can be configured to store different combinations of 48-bit instruction words and 32-bit data words. Maximum efficiency (i.e. single-cycle execution of dual-data-access instructions), though, is achieved when one block contains a mix of instructions and PM bus data while the other block contains DM bus data only. This means that for an instruction requiring two data accesses, the PM bus (and DAG2) is used to access data from the mixed block, the DM bus (and DAG1) is used to access data from the data-only block, *and the instruction must be available from the cache.* Another way to partition the data is to store one operand in external memory and the other in either block of internal memory.

In typical DSP applications such as digital filters and FFTs, two data operands must be accessed for some instructions. In a digital filter, for example, the filter coefficients can be stored in 32-bit words in the same memory block that contains the 48-bit instructions, while 32-bit data samples are stored in the other block. This provides single-cycle execution of dual-data-access instructions, with the filter coefficients being accessed by DAG2 over the PM bus and the instruction available from the cache.

To assure single-cycle, parallel accesses of two on-chip memory locations, the following conditions must be met:

- The two addresses must be located in different memory blocks (i.e. one in Block 0, one in Block 1).
- One address must be generated by DAG1 and the other by DAG2.
- The DAG1 address must not point to the same memory block that instructions are being fetched from.
- The instruction should be of the form:

  compute, `Rx=DM(I0-I7,M0-M7), Ry=PM(I8-I15,M8-M15);`

(Note that reads and writes may be intermixed.)

11.8 DATA DELAYS, LATENCIES, & THROUGHPUT

Tables 11.5 and 11.6 specify data delays, latencies, and throughput for the ADSP-2106x. *Data delay* and *latency* are defined as the number of cycles (after the first cycle) required to complete the operation. Thus a zero-wait-state memory has a data delay of zero, and a single-wait-state memory has a data delay of one. *Throughput* is the maximum rate at which the operation is performed.

Data delay and throughput are the same whether the access is from a host processor or from another ADSP-2106x.

11.9 EXECUTION STALLS

The following events can cause execution stalls in the ADSP-2106x core.

Program Sequencer Stalls
- 1 cycle on a program memory data access with instruction cache miss
- 2 cycles on non-delayed branches
- 2 cycles on normal interrupts
- 5 cycles on vector interrupt (VIRPT)
- 1-2 cycles on short loops w / small iterations
- *n* cycles on IDLE instruction

DAG Stalls
- 1 cycle hold on register conflict

Memory Stalls
- 1 cycle on PM and DM bus accesses to the same block of internal memory
- *n* cycles if conflicting accesses to external memory (PM and DM bus accesses must complete)
- *n* cycles if access to external memory (until I/O buffers are cleared out)
- *n* cycles if external access and ADSP-2106x does not own external bus
- *n* cycles until external access is complete (i.e. waitstates, idle cycles, etc.)

IOP Register Stalls
- *n* cycles if PM and DM bus access to IOP registers (both must complete)
- *n* cycles if conflict with slave access

DMA Stalls
- 1 cycle if an access to a DMA parameter register conflicts with the DMA address generation (i.e. writing to the register while a register update is taking place or reading while a DMA register read is taking place)
- 1 cycle if an access to a DMA parameter register conflicts with DMA chaining
- *n* cycles if writing (reading) to a DMA buffer and the buffer is full (empty)

Link Port & Serial Port Stalls
- 1 cycle if two link buffer reads back-to-back
- *n* cycles if write to a full buffer or read from an empty buffer

Operation		Minimum Data Delay (cycles)	Maximum Throughput (cycles/transfer)
Core Processor Access to External Memory		0	1
Synchronous Access of Slave's IOP Registers* *Delay is between data in the IOP register and at the External Port (e.g. an IOP register would be written in the 2nd cycle after the write was completed at the External Port).*	–Read (Transfer Out) –Write (Transfer In)	0 2**	2** 1
Synchronous Direct Access of Slave's Int. Memory* *(Direct Read/Write)* *Delay is between data in the IOP register and at the External Port .*	–Read (Transfer Out) –Write (Transfer In)	2 3**	4** 1
Slave Mode DMA	–Read (Transfer Out) –Write (Transfer In)	– –	2 † 1
Master Mode DMA	–Transfer Out –Transfer In	– –	1 1
Handshake Mode DMA *Delay is between DMA data and $\overline{DMAR}x$.*	–Transfer (In & Out)	3	1
External Handshake Mode DMA *Delay is between $\overline{DMAR}x$ and the external transfer.*	–Transfer (In & Out)	3	1

Table 11.5 Data Delays & Throughputs

* If MMSWS (Multiprocessor Memory Space Wait States) is enabled, add 1 cycle to the throughput of synchronous writes to MMS.
** For asynchronous accesses, add 1 cycle.
† These transfers are speed-limited by the read of the slave's DMA FIFO buffer.

Operation		Minimum Latency (cycles)	Maximum Throughput (cycles/transfer)
Interrupts ($\overline{IRQ}_{2-0}$)		3	–
Multiprocessor Bus Requests ($\overline{BR}_{1-6}$)		1	–
Host Bus Request ($\overline{HBR}$)		2	–
SYSCON Effect Latency		1	–
Host Packing Status Update (in SYSTAT register)		0	–
DMA Packing Status Update (in DMACx register)		1	–
DMA Chain Initialization		7-11	–
Vector Interrupt (VIRPT register)		6	–
Serial Ports *		35	32
Link Ports *	–1x speed	11	8
	–2x speed	7	4

Table 11.6 Latencies & Throughputs

* 32-bit words, ADSP-2106x core to ADSP-2106x core.

Note: The link port control registers LCOM, LCTL, LAR, and the serial port control registers STCTLx and SRCTLx all share the same internal bus for both reads and writes. Because of this, when a read of one of these registers is followed immediately by a write, the write will take two processor cycles to complete.

11 System Design

Instruction Set Reference A

A.1 OVERVIEW

This appendix and the next one describe the ADSP-2106x instruction set in detail. This appendix explains each instruction type, including the assembly language syntax and the opcode that the instruction assembles to. Many instruction types contain a field for specifying a compute operation (an operation that uses the ALU, multiplier or shifter). Because there are a large number of options available for this field, they are described separately in Appendix B. (Note that data moves between the MR registers and the register file are considered multiplier operations.)

Each instruction is specified in this section. The specification shows the **syntax** of the instruction, describes its **function**, gives one or two assembly-language **examples**, and specifies and describes the various fields of its **opcode**. The instructions are grouped into four categories:

I. *Compute and Move or Modify* instructions, which specify a compute operation in parallel with one or two data moves or an index register modify.

II. *Program Flow Control* instructions, which specify various types of branches, calls, returns and loops. Some of these instructions may also specify a compute operation and/or a data move.

III. *Immediate Data Move* instructions, which use immediate instruction fields as operands, or use immediate instruction fields for addressing.

IV. *Miscellaneous* instructions, such as bit modify and test, no operation and idle.

The instructions are numbered from 1 to 23. Some instructions have more than one syntactical form; for example, Instruction Type 4 has four distinct forms. The instruction number has no bearing on programming, but corresponds to the opcode recognized by the ADSP-2106x device.

Many instructions can be conditional. These instructions are prefaced by an "IF" plus a condition mnemonic. In a conditional instruction, the execution of the entire instruction is based on the specified condition.

A Instruction Set Reference

A.2 INSTRUCTION SET SUMMARY

The next few pages summarize the ADSP-2106x instruction set. The compute operations used within each instruction are specified in Appendix B.

Compute & Move or Modify Instructions

(pg. A-16)	1.		*compute,*	DM(Ia,Mb) = dreg1 dreg1 = DM(Ia,Mb)	, PM(Ic,Md) = dreg2 dreg2 = PM(Ic,Md) ;
(pg. A-17)	2.	*IF condition*	compute;		
(pg. A-18)	3a.	*IF condition*	*compute,*	DM(Ia,Mb) PM(Ic,Md)	= ureg ;
	3b.	*IF condition*	*compute,*	DM(Mb,Ia) PM(Md,Ic)	= ureg ;
	3c.	*IF condition*	*compute,*	ureg = DM(Ia,Mb) PM(Ic,Md)	;
	3d.	*IF condition*	*compute,*	ureg = DM(Mb,Ia) PM(Md,Ic)	;
(pg. A-20)	4a.	*IF condition*	*compute,*	DM(Ia,<data6>) PM(Ic,<data6>)	= dreg ;
	4b.	*IF condition*	*compute,*	DM(<data6>,Ia) PM(<data6>,Ic)	= dreg ;
	4c.	*IF condition*	*compute,*	dreg = DM(Ia,<data6>) PM(Ic,<data6>)	;
	4d.	*IF condition*	*compute,*	dreg = DM(<data6>,Ia) PM(<data6>,Ic)	;
(pg. A-22)	5.	*IF condition*	*compute,*	ureg1 = ureg2 ;	
(pg. A-24)	6a.	*IF condition*	shiftimm,	*DM(Ia,Mb)* *PM(Ic,Md)*	= *dreg* ;
	6b.	*IF condition*	shiftimm,	*dreg* = *DM(Ia,Mb)* *PM(Ic,Md)*	;
(pg. A-26)	7.	*IF condition*	*compute,*	MODIFY (Ia,Mb) (Ic,Md)	;

Items in *italics* are an optional part of the instruction.

Program Flow Control Instructions

(pg. A-28) 8.

IF condition JUMP | <addr24> / (PC, <reladdr24>) | | *(DB)* / *(LA)* / *(CI)* / *(DB,LA)* / *(DB,CI)* | ;

IF condition CALL | <addr24> / (PC, <reladdr24>) | *(DB)* ;

(pg. A-30) 9.

IF condition JUMP | (Md,Ic) / (PC, <reladdr6>) | | *(DB)* / *(LA)* / *(CI)* / *(DB,LA)* / *(DB,CI)* | , | *compute* / *ELSE compute* | ;

IF condition CALL | (Md,Ic) / (PC, <reladdr6>) | *(DB)* , | *compute* / *ELSE compute* | ;

(pg. A-32) 10.

IF condition JUMP | (Md,Ic) / (PC, <reladdr6>) | , ELSE | *compute* , DM(Ia,Mb) = dreg / *compute* , dreg = DM(Ia,Mb) | ;

(pg. A-34) 11.

IF condition RTS | *(DB)* / *(LR)* / *(DB,LR)* | , | *compute* / *ELSE compute* | ;

IF condition RTI *(DB)* , | *compute* / *ELSE compute* | ;

(pg. A-36) 12.

LCNTR = | <data16> / ureg | , DO | <addr24> / (PC, <reladdr24>) | UNTIL LCE ;

(pg. A-38) 13.

DO | <addr24> / (PC, <reladdr24>) | UNTIL termination ;

Items in *italics* are an optional part of the instruction.

Immediate Move Instructions

(pg. A-40) 14a. DM(<addr32>) / PM(<addr24>) = ureg ;

14b. ureg = DM(<addr32>) / PM(<addr24>) ;

(pg. A-41) 15a. DM(<data32>, Ia) / PM(<data24>, Ic) = ureg ;

15b. ureg = DM(<data32>, Ia) / PM(<data24>, Ic) ;

(pg. A-42) 16. DM(Ia,Mb) / PM(Ic,Md) = <data32> ;

(pg. A-43) 17. ureg = <data32> ;

Miscellaneous Instructions

(pg. A-46) 18. BIT SET / CLR / TGL / TST / XOR sreg <data32> ;

(pg. A-48) 19a. MODIFY (Ia, <data32>) / (Ic, <data24>) ;

19b. BITREV (Ia, <data32>) / (Ic, <data24>) ;

(pg. A-50) 20. *PUSH* / *POP* *LOOP* , *PUSH* / *POP* *STS* , *PUSH* / *POP* *PCSTK* , *FLUSH CACHE* ;

(pg. A-51) 21. NOP;

(pg. A-52) 22. IDLE ;

(pg. A-53) 23. IDLE16 ;

(pg. A-54) 24. CJUMP function / (PC, <reladdr24>) (DB) ;

RFRAME ;

Items in *italics* are an optional part of the instruction.

Instruction Set Reference A

Instruction Set Notation

Notation	*Meaning*
UPPERCASE	Explicit syntax—assembler keyword (notation only; assembler is case-insensitive and lowercase is the preferred programming convention)
;	Semicolon (instruction terminator)
,	Comma (separates parallel operations in an instruction)
italics	Optional part of instruction
option1 | *option2*	List of options between vertical bars (choose one)
compute	ALU, multiplier, shifter or multifunction operation (see Appendix B)
shiftimm	Shifter immediate operation (see Appendix B)
condition	Status condition (see condition codes below)
termination	Loop termination condition (see condition codes below)
ureg	Universal register
sreg	System register
dreg	Data register (register file): R15-R0 or F15-F0
Ia	I7-I0 (DAG1 index register)
Mb	M7-M0 (DAG1 modify register)
Ic	I15-I8 (DAG2 index register)
Md	M15-M8 (DAG2 modify register)
<data*n*>	*n*-bit immediate data value
<addr*n*>	*n*-bit immediate address value
<reladdr*n*>	*n*-bit immediate PC-relative address value
(DB)	Delayed branch
(LA)	Loop abort (pop loop and PC stacks on branch)
(CI)	Clear interrupt

Condition & Termination Codes (IF & DO UNTIL)

In a conditional instruction, execution of the entire instruction depends on the specified condition.

Condition	*Description*	*Condition*	*Description*
EQ	ALU equal zero	NE	ALU not equal to zero
LT	ALU less than zero	GE	ALU greater than or equal zero
LE	ALU less than or equal zero	GT	ALU greater than zero
AC	ALU carry	NOT AC	Not ALU carry
AV	ALU overflow	NOT AV	Not ALU overflow
MV	Multiplier overflow	NOT MV	Not multiplier overflow
MS	Multiplier sign	NOT MS	Not multiplier sign
SV	Shifter overflow	NOT SV	Not shifter overflow
SZ	Shifter zero	NOT SZ	Not shifter zero
FLAG0_IN	Flag 0 input	NOT FLAG0_IN	Not Flag 0 input
FLAG1_IN	Flag 1 input	NOT FLAG1_IN	Not Flag 1 input
FLAG2_IN	Flag 2 input	NOT FLAG2_IN	Not Flag 2 input
FLAG3_IN	Flag 3 input	NOT FLAG3_IN	Not Flag 3 input
TF	Bit test flag	NOT TF	Not bit test flag
BM	Bus master	NBM	Not bus master
LCE	Loop counter expired (DO UNTIL)	FOREVER	Always false (DO UNTIL)
NOT LCE	Loop counter not expired (IF)	TRUE	Always true (IF)

Universal Registers

Register	Function
Data Register File	
R15 - R0	Register file locations, fixed-point
F15 - F0	Register file locations, floating-point
Program Sequencer	
PC	Program counter (read-only)
PCSTK	Top of PC stack
PCSTKP	PC stack pointer
FADDR	Fetch address (read-only)
DADDR	Decode address (read-only)
LADDR	Loop termination address, code; top of loop address stack
CURLCNTR	Current loop counter; top of loop count stack
LCNTR	Loop count for next nested counter-controlled loop
Data Address Generators	
I7 - I0	DAG1 index registers
M7 - M0	DAG1 modify registers
L7 - L0	DAG1 length registers
B7 - B0	DAG1 base registers
I15 - I8	DAG2 index registers
M15 - M8	DAG2 modify registers
L15 - L8	DAG2 length registers
B15 - B8	DAG2 base registers
Bus Exchange	
PX1	PMD-DMD bus exchange 1 (16 bits)
PX2	PMD-DMD bus exchange 2 (32 bits)
PX	48-bit combination of PX1 and PX2
Timer	
TPERIOD	Timer period
TCOUNT	Timer counter
System Registers	
MODE1	Mode control & status
MODE2	Mode control & status
IRPTL	Interrupt latch
IMASK	Interrupt mask
IMASKP	Interrupt mask pointer (for nesting)
ASTAT	Arithmetic status flags, bit test flag, etc.
STKY	Sticky arithmetic status flags, stack status flags, etc.
USTAT1	User status register 1
USTAT2	User status register 2

Memory Addressing in Instructions

Direct:

Absolute
Instruction Types 8, 12, 13, 14
Examples:
```
dm(0x000015F0) = astat;
if ne jump label2;        {'label2' is an address label}
```

PC-relative
Instruction Types 8, 9, 10, 12, 13
Examples:
```
call(pc,10), r0=r6+r3;
do(pc,length) until sz;   {'length' is a variable}
```

Register Indirect (using DAG registers):

Post-modify with M register, update I register
Instruction Types 1, 3, 6, 16
Examples:
```
f5=pm(i9,m12);
dm(i0,m3)=r3, r1=pm(i15,m10);
```

Pre-modify with M register, no update
Instruction Types 3, 9, 10
Examples:
```
r1=pm(m10,i15);
jump(m13,i11);
```

Post-modify with immediate value, update I register
Instruction Type 4
Examples:
```
f15=dm(i0,6);
if av r1=pm(i15,0x11);
```

Pre-modify with immediate value, no update
Instruction Types 4, 15
Examples:
```
if av r1=pm(0x11,i15);
dm(127,i5)=laddr;
```

A.3 OPCODE NOTATION

In ADSP-2106x opcodes, some bits are explicitly defined to be zeros or ones. The values of other bits or fields set various parameters for the instruction. The terms in this section define these opcode bits and fields. Bits which are unspecified are ignored when the processor decodes the instruction, but are reserved for future use.

A Loop abort code

1	Pop loop, PC stacks on branch
0	Do not pop loop, PC stacks on branch

ADDR Immediate address field

AI Computation unit register

0000	MR0F
0001	MR1F
0010	MR2F
0100	MR0B
0101	MR1B
0110	MR2B

B Branch type

0	Jump
1	Call

BOP Bit Operation select codes

000	Set
001	Clear
010	Toggle
100	Test
101	XOR

COMPUTE Compute operation field (see Appendix B)

COND Status Condition codes

0 - 31

CI Clear interrupt code

1	Clear current interrupt
0	Do not clear current interrupt

CU Computation unit select codes

00	ALU
01	Multiplier
10	Shifter

DATA Immediate data field

DEC Counter decrement code

0	No counter decrement
1	Counter decrement

DMD Memory access direction

0	Read
1	Write

DMI Index (I) register numbers, DAG1

0 - 7

DMM Modify (M) register numbers, DAG1

0 - 7

DREG Register file locations

0 - 15

E ELSE clause code

0	No ELSE clause
1	ELSE clause

FC Flush cache code

0	No cache flush
1	Cache flush

G DAG / Memory select

0	DAG1 or Data Memory
1	DAG2 or Program Memory

INC Counter increment code

0	No counter increment
1	Counter increment

A Instruction Set Reference

J Jump Type

0 Non-delayed
1 Delayed

LPO Loop stack pop code

0 No stack pop
1 Stack pop

LPU Loop stack push code

0 No stack push
1 Stack push

LR Loop reentry code

0 No loop reentry
1 Loop reentry

NUM Interrupt vector

0 - 7

OPCODE Computation unit opcodes (see Appendix B)

PMD Memory access direction

0 Read
1 Write

PMI Index (I) register numbers, DAG2

8 - 15

PMM Modify (M) register numbers, DAG2

8 - 15

PPO PC stack pop code

0 No stack pop
1 Stack pop

PPU PC stack push code

0 No stack push
1 Stack push

RELADDR PC-relative address field

SPO Status stack pop code

0	No stack pop
1	Stack pop

SPU Status stack push code

0	No stack push
1	Stack push

SREG System Register code

0 - 15 (see “Universal Register Codes” on the next page)

TERM Termination Condition codes

0 - 31

U Update, index (I) register

0	Pre-modify, no update
1	Post-modify with update

UREG Universal Register code

0 - 256 (see “Universal Register Codes” on the next page)

RA, RM, RN, RS, RX, RY Register file locations for compute operands and results

0 - 15

RXA ALU x-operand register file location for multifunction operations

8 - 11

RXM Multiplier x-operand register file location for multifunction operations

0 - 3

RYA ALU y-operand register file location for multifunction operations

12 - 15

RYM Multiplier y-operand register file location for multifunction operations

4 - 7

A.4 UNIVERSAL REGISTER CODES

Map 1 Registers:

PC	program counter
PCSTK	top of PC stack
PCSTKP	PC stack pointer
FADDR	fetch address
DADDR	decode address
LADDR	loop termination address
CURLCNTR	current loop counter
LCNTR	loop counter
R15 - R0	register file locations
I15 - I0	DAG1 and DAG2 index registers
M15 - M0	DAG1 and DAG2 modify registers
L15 - L0	DAG1 and DAG2 length registers
B15 - B0	DAG1 and DAG2 base registers

System Registers:

MODE1	mode control 1
MODE2	mode control 2
IRPTL	interrupt latch
IMASK	interrupt mask
IMASKP	interrupt mask pointer
ASTAT	arithmetic status
STKY	sticky status
USTAT1	user status reg 1
USTAT2	user status reg 2

b3 b2 b1 b0 \ (b7=0) b7 b6 b5 b4	0000	0001	0010	0011	0100	0101	0110	0111 (System Registers)
0000	R0	I0	M0	L0	B0		FADDR	USTAT1
0001	R1	I1	M1	L1	B1		DADDR	USTAT2
0010	R2	I2	M2	L2	B2			
0011	R3	I3	M3	L3	B3		PC	
0100	R4	I4	M4	L4	B4		PCSTK	
0101	R5	I5	M5	L5	B5		PCSTKP	
0110	R6	I6	M6	L6	B6		LADDR	
0111	R7	I7	M7	L7	B7		CURLCNTR	
1000	R8	I8	M8	L8	B8		LCNTR	
1001	R9	I9	M9	L9	B9			IRPTL
1010	R10	I10	M10	L10	B10			MODE2
1011	R11	I11	M11	L11	B11			MODE1
1100	R12	I12	M12	L12	B12			ASTAT
1101	R13	I13	M13	L13	B13			IMASK
1110	R14	I14	M14	L14	B14			STKY
1111	R15	I15	M15	L15	B15			IMASKP

Figure A.1 Map 1 Universal Register Codes

Map 2 Registers:

PX	48-bit PX1 and PX2 combination
PX1	bus exchange 1 (16 bits)
PX2	bus exchange 2 (32 bits)
TPERIOD	timer period
TCOUNT	timer counter

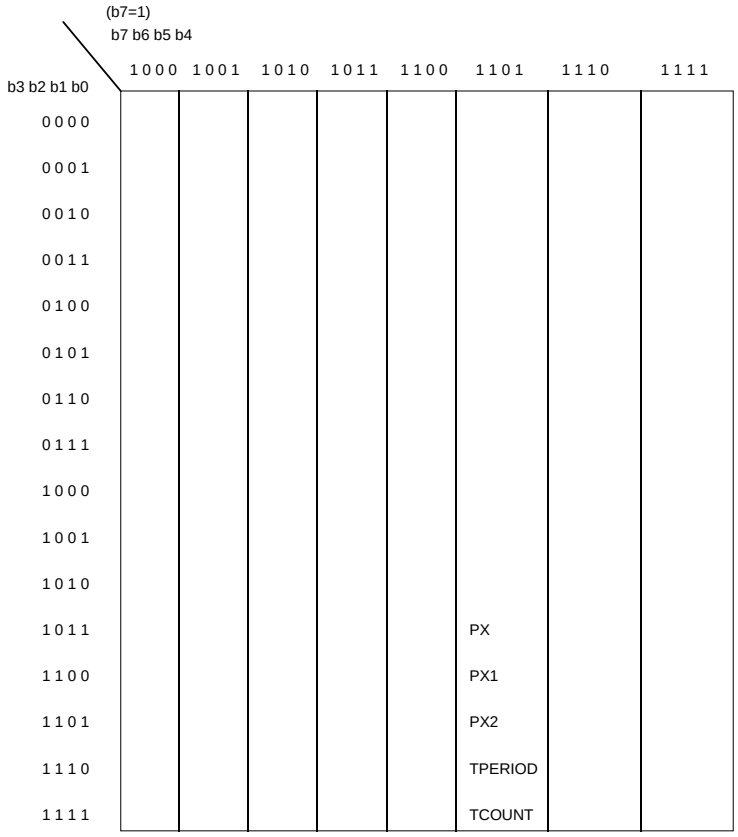


Figure A.2 Map 2 Universal Register Codes

A Instruction Set Reference

Instruction Set Reference A

Group I.
Compute and Move Instructions

compute / dreg↔DM / dreg↔PM

Syntax:

compute, | DM(Ia, Mb) = dreg1
dreg1 = DM(Ia, Mb) | , | PM(Ic, Md) = dreg2
dreg2 = PM(Ic, Md) | ;

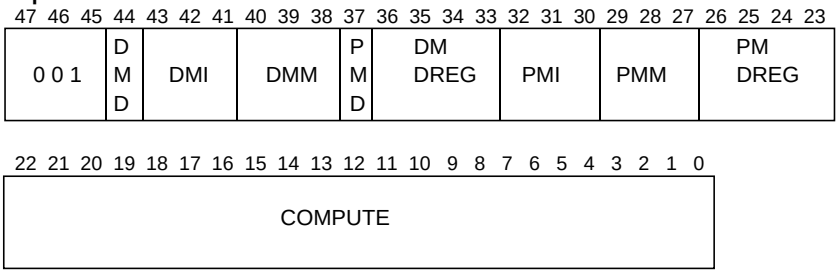
Function:
Parallel accesses to data memory and program memory from the register file. The specified I registers address data memory and program memory. The I values are post-modified and updated by the specified M registers. Pre-modify offset addressing is not supported.

Note: See Section 4.4.1, “DAG Register Transfer Restrictions”, in Chapter 4, *Data Addressing*.

Examples:

```
R7=BSET R6 BY R0, DM(I0,M3)=R5, PM(I11,M15)=R4;

R8=DM(I4,M1), PM(I12 M12)=R0;
```

Opcode:



DMD and PMD select the access types (read or write). DMDREG and PMDREG specify register file locations. DMI and PMI specify I registers for data and program memory. DMM and PMM specify M registers used to update the I registers. The COMPUTE field defines a compute operation to be performed in parallel with the data accesses; this is a NOP if no compute operation is specified in the instruction.

compute

Syntax:

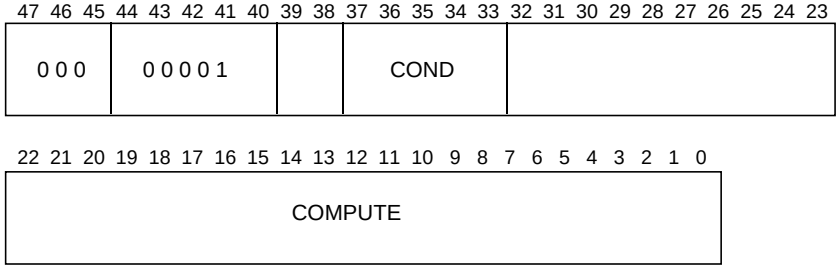
IF condition compute ;

Function:
Conditional compute instruction. The instruction is executed if the specified condition tests true.

Examples:

```
IF MS MRF=0;

F6=(F2+F3)/2;
```

Opcode:



The operation specified in the COMPUTE field is executed if the condition specified by COND is true. If no condition is specified in the instruction, COND is the TRUE condition, and the compute operation is always executed.

compute / ureg↔DM|PM , register modify

Syntax:

a. *IF condition* *compute,* | DM(Ia, Mb) / PM(Ic, Md) | = ureg ;

b. *IF condition* *compute,* | DM(Mb, Ia) / PM(Md, Ic) | = ureg ;

c. *IF condition* *compute,* ureg = | DM(Ia, Mb) / PM(Ic, Md) | ;

d. *IF condition* *compute,* ureg = | DM(Mb, Ia) / PM(Md, Ic) | ;

Function:
Access between data memory or program memory and a universal register. The specified I register addresses data memory or program memory. The I value is either pre-modified (M, I order) or post-modified (I, M order) by the specified M register. If it is post-modified, the I register is updated with the modified value. If a compute operation is specified, it is performed in parallel with the data access. If a condition is specified, it affects entire instruction.

Notes:

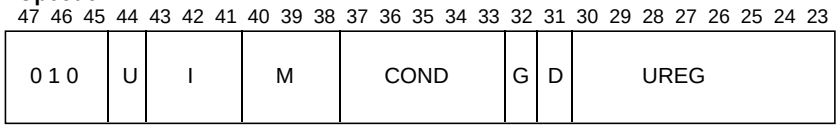
1. ureg may not be from the same DAG (i.e. DAG1 or DAG2) as Ia / Mb or Ic / Md.
2. See Section 4.4.1, “DAG Register Transfer Restrictions”, in Chapter 4, *Data Addressing.*

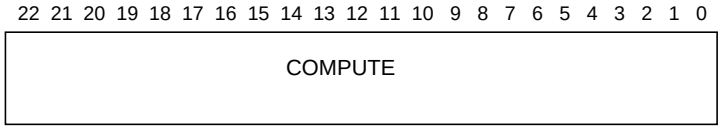
Examples:

```
R6=R3-R11, DM(I0,M1)=ASTAT;

IF NOT SV F8=CLIP F2 BY F14, PX=PM(I12,M12);
```

compute / ureg↔DM|PM , register modify

Opcode:





COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

D selects the access type (read or write). G selects data memory or program memory. UREG specifies the universal register. I specifies the I register, and M specifies the M register. U selects either pre-modify without update or post-modify with update. The COMPUTE field defines a compute operation to be performed in parallel with the data access; this is a no-operation if no compute operation is specified in the instruction.

compute / dreg↔DM|PM , immediate modify

Syntax:

a. *IF condition* *compute,* | DM(Ia, <data6>) | = dreg ;
| PM(Ic, <data6>) |

b. *IF condition* *compute,* | DM(<data6>, Ia) | = dreg ;
| PM(<data6>, Ic) |

c. *IF condition* *compute,* dreg = | DM(Ia, <data6>) | ;
| PM(Ic, <data6>) |

d. *IF condition* *compute,* dreg = | DM(<data6>, Ia) | ;
| PM(<data6>, Ic) |

Function:
Access between data memory or program memory and the register file. The specified I register addresses data memory or program memory. The I value is either pre-modified (data order, I) or post-modified (I, data order) by the specified immediate data. If it is post-modified, the I register is updated with the modified value. If a compute operation is specified, it is performed in parallel with the data access. If a condition is specified, it affects entire instruction.

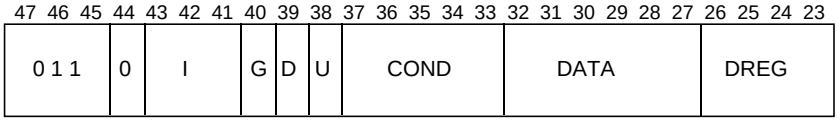
Note: See Section 4.4.1, “DAG Register Transfer Restrictions”, in Chapter 4, *Data Addressing*.

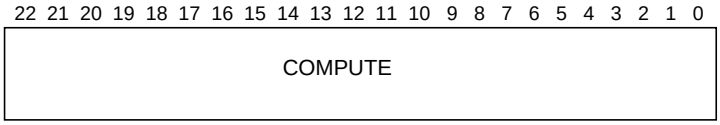
Examples:

```
IF FLAG0_IN F1=F5*F12, F11=PM(I10,40);

R12=R3 AND R1, DM(6,I1)=R6;
```

compute / dreg↔DM|PM , immediate modify

Opcode:





COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

D selects the access type (read or write). G selects data memory or program memory. DREG specifies the register file location. I specifies the I register. DATA specifies a 6-bit, twos-complement modify value. U selects either pre-modify without update or post-modify with update. The COMPUTE field defines a compute operation to be performed in parallel with the data access; this is a no-operation if no compute operation is specified in the instruction.

compute / ureg↔ureg

Syntax:

IF condition *compute,* ureg1 = ureg2 ;

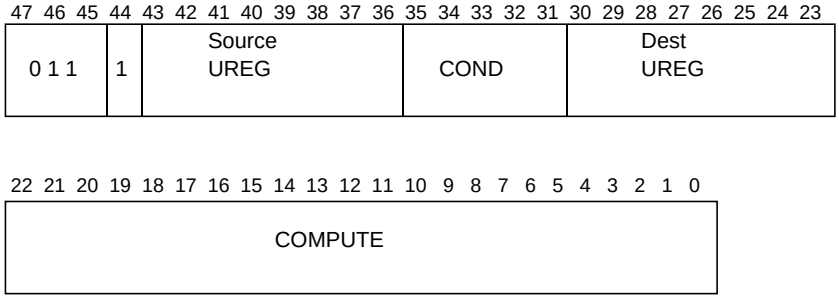
Function:
Transfer from one universal register to another. If a compute operation is specified, it is performed in parallel with the data access. If a condition is specified, it affects entire instruction.

Examples:

```
IF TF MRF=R2*R6(SSFR), M4=R0;

LCNTR=L7;
```

compute / ureg↔ureg

Opcode:



COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

Source UREG identifies the universal register source. Dest UREG identifies the universal register destination. The COMPUTE field defines a compute operation to be performed in parallel with the data transfer; this is a no-operation if no compute operation is specified in the instruction.

immediate shift / dreg↔DM|PM

Syntax:

a. *IF condition* shiftimm , | *DM(Ia, Mb)* / *PM(Ic, Md)* | = *dreg* ;

b. *IF condition* shiftimm , *dreg* = | *DM(Ia, Mb)* / *PM(Ic, Md)* | ;

Function:
An immediate shift operation is a shifter operation that takes immediate data as its Y-operand. The immediate data is one 8-bit value or two 6-bit values, depending on the operation. The x-operand and the result are register file locations.

If an access to data or program memory from the register file is specified, it is performed in parallel with the shifter operation. The I register addresses data or program memory. The I value is post-modified by the specified M register and updated with the modified value. If a condition is specified, it affects entire instruction.

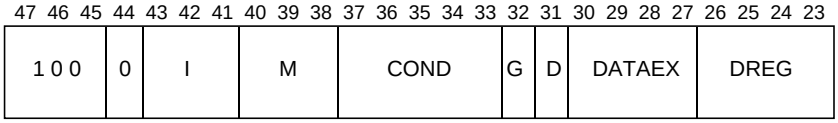
Note: See Section 4.4.1, “DAG Register Transfer Restrictions”, in Chapter 4, *Data Addressing*.

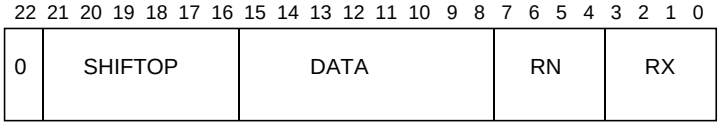
Examples:

```
IF GT R2=R6 LSHIFT BY 30, DM(I4,M4)=R0;

IF NOT SZ R3=FEXT R1 BY 8:4;
```

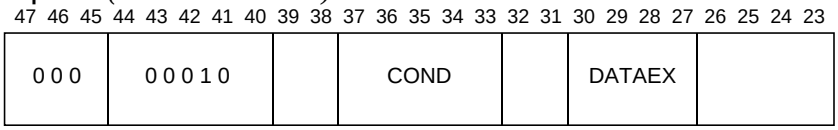
immediate shift / dreg↔DM|PM

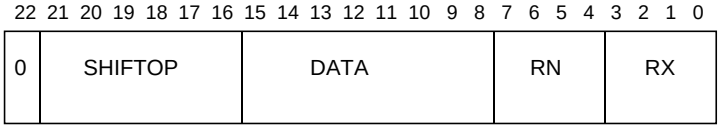
Opcode: *(with data access)*





Opcode: *(without data access)*





COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

SHIFTOP specifies the shifter operation. The DATA field specifies an 8-bit immediate shift value. For shifter operations requiring two 6-bit values (a shift value and a length value), the DATAEX field adds 4 MSBs to the DATA field, creating a 12-bit immediate value. The six LSBs are the shift value, and the six MSBs are the length value.

If a memory access is specified, D selects the access type (read or write). G selects data memory or program memory. DREG specifies the register file location. I specifies the I register, which is post-modified and updated by the M register identified by M.

The COMPUTE field defines a compute operation to be performed in parallel with the data access; this is a no-operation if no compute operation is specified in the instruction.

compute / modify

Syntax:

IF condition *compute,* MODIFY | (Ia, Mb) | ;
| (Ic, Md) |

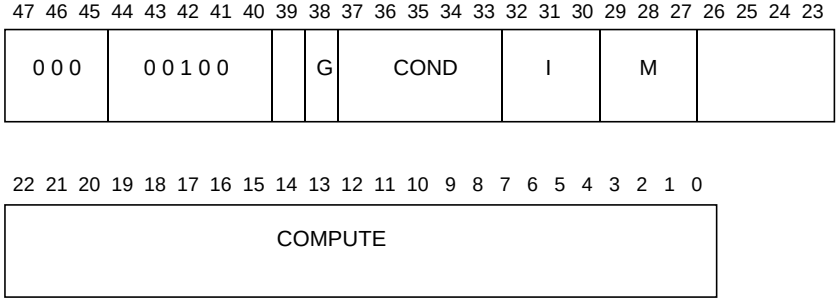
Function:
Update of the specified I register by the specified M register. If a compute operation is specified, it is performed in parallel with the data access. If a condition is specified, it affects entire instruction.

Note: See Section 4.4.1, “DAG Register Transfer Restrictions”, in Chapter 4, *Data Addressing*.

Examples:

```
IF NOT FLAG2_IN R4=R6*R12(SUF), MODIFY(I10,M8);

IF NOT LCE MODIFY(I3,M1);
```

Opcode:



COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

G selects DAG1 or DAG2. I specifies the I register, and M specifies the M register. The COMPUTE field defines a compute operation to be performed in parallel with the data access; this is a no-operation if no compute operation is specified in the instruction.

Group II.
Program Flow Control

direct jump|call

Syntax:

IF condition JUMP | <addr24> / (PC, <reladdr24>) | (| *DB* / *LA* / *CI* / *DB, LA* / *DB, CI* |) ;

IF condition CALL | <addr24> / (PC, <reladdr24>) | (*DB*) ;

Function:
A jump or call to the specified address or PC-relative address. The PC-relative address is a 24-bit, twos-complement value. If the delayed branch (DB) modifier is specified, the branch is delayed; otherwise, it is non-delayed. If the loop abort (LA) modifier is specified for a jump, the loop stacks and PC stack are popped when the jump is executed. You should use the (LA) modifier if the jump will transfer program execution outside of a loop. If there is no loop, or if the jump address is within the loop, you should not use the (LA) modifier.

The clear interrupt (CI) modifier allows the reuse of an interrupt while it is being serviced. Normally the ADSP-2106x ignores and does not latch an interrupt that reoccurs while its service routine is already executing. The JUMP (CI) instruction should be located within the interrupt service routine. JUMP (CI) clears the status of the current interrupt without leaving the interrupt service routine, reducing the interrupt routine to a normal subroutine—this allows the interrupt to occur again, as a result of a different event or task in the ADSP-2106x system. See “Clearing The Current Interrupt For Reuse” in the *Program Sequencing* chapter for further details.

The JUMP (CI) instruction reduces an interrupt service routine to a normal subroutine by clearing the appropriate bit in the interrupt latch register (IRPTL) and interrupt mask pointer (IMASKP). The ADSP-2106x then allows the interrupt to occur again.

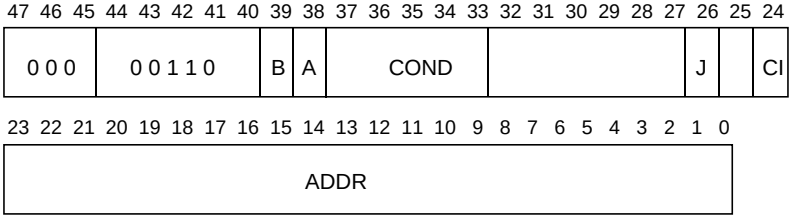
When returning from a subroutine which has been reduced from an interrupt service routine with a JUMP (CI) instruction, the (LR) modifier of the RTS instruction must be used (in case the interrupt occurred during the last two instructions of a loop). (See instruction type 11, return from subroutine).

direct jump|call

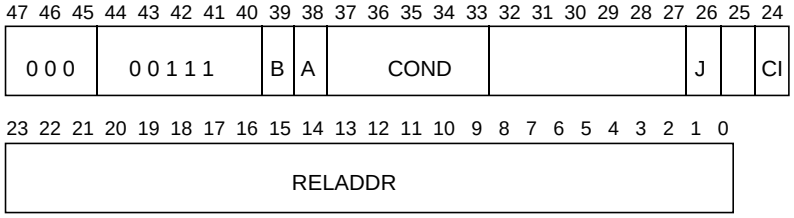
Examples:

```
IF AV JUMP(PC,0x00A4)(LA);

CALL init (DB);              {init is a program label}

JUMP (PC,2) (DB,CI);         {clear current int. for reuse}
```

Opcode: *(with direct branch)*



Opcode: *(with PC-relative branch)*



COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

B selects the branch type, jump or call. J determines whether the branch is delayed or non-delayed. The ADDR field specifies a 24-bit program memory address. RELADDR is a 24-bit, twos-complement value that is added to the current PC value to generate the branch address. The A bit activates loop abort. CI activates clear interrupt. (For calls, A and CI are ignored.)

indirect jump|call / compute

Syntax:

IF condition JUMP | (Md, Ic) / (PC, <reladdr6>) | (| *DB* / *LA* / *CI* / *DB, LA* / *DB, CI* |) , | *compute* / *ELSE compute* | ;

IF condition CALL | (Md, Ic) / (PC, <reladdr6>) | (*DB*) , | *compute* / *ELSE compute* | ;

Function:
A jump or call to the specified PC-relative address or pre-modified I register value. The PC-relative address is a 6-bit, twos-complement value. If an I register is specified, it is modified by the specified M register to generate the branch address. The I register is not affected by the modify operation.

The jump or call is executed if a condition is specified and is true. If a compute operation is specified without the ELSE, it is performed in parallel with the jump or call. If a compute operation is specified with the ELSE, it is performed only if the the condition specified is false. Note that a condition must be specified if an *ELSE compute* clause is specified.

If the delayed branch (DB) modifier is specified, the jump or call is delayed; otherwise, it is non-delayed. If the loop abort (LA) modifier is specified for a jump, the loop stacks and PC stack are popped when the jump is executed. You should use the (LA) modifier if the jump will transfer program execution outside of a loop. If there is no loop, or if the jump address is within the loop, you should not use the (LA) modifier.

The clear interrupt (CI) modifier allows the reuse of an interrupt while it is being serviced. Normally the ADSP-2106x ignores and does not latch an interrupt that reoccurs while its service routine is already executing. The JUMP (CI) instruction should be located within the interrupt service routine. JUMP (CI) clears the status of the current interrupt without leaving the interrupt service routine, reducing the interrupt routine to a normal subroutine—this allows the interrupt to occur again, as a result of a different event. See “Clearing The Current Interrupt For Reuse” in the *Program Sequencing* chapter for further details.

The JUMP (CI) instruction reduces an interrupt service routine to a normal subroutine by clearing the appropriate bit in the interrupt latch register (IRPTL) and interrupt mask pointer (IMASKP). The ADSP-2106x then allows the interrupt to occur again.

When returning from a subroutine which has been reduced from an interrupt service routine with a JUMP (CI) instruction, the (LR) modifier of the RTS instruction must be used (in case the interrupt occurred during the last two instructions of a loop). (See instruction type 11, return from subroutine).

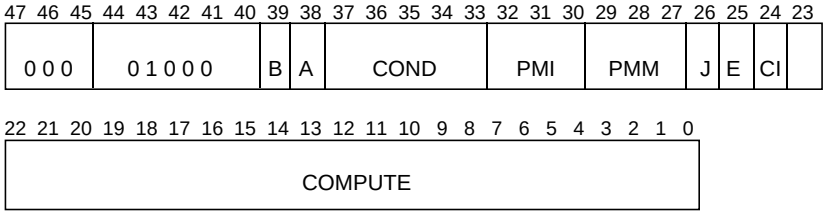
indirect jump|call / compute

Note: For indirect branches, see Section 4.4.1, “DAG Register Transfer Restrictions”, in Chapter 4, *Data Addressing*.

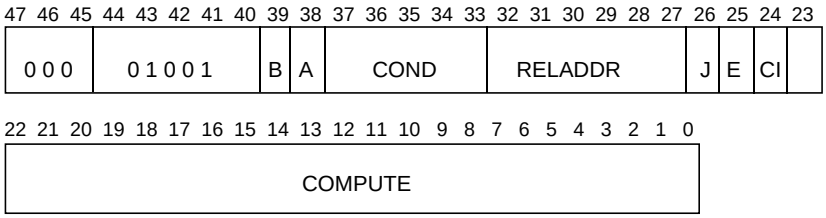
Examples:

```
JUMP(M8,I12), R6=R6-1;

IF EQ CALL(PC,17)(DB) , ELSE R6=R6-1;
```

Opcode: *(with indirect branch)*



Opcode: *(with PC-relative branch)*



COND specifies the condition to test. If no condition is specified in the instruction, COND is the true condition, and the instruction is always executed. E specifies whether or not an ELSE clause is used.

B selects the branch type, jump or call. J determines whether the branch is delayed or non-delayed. The A bit activates loop abort. CI activates clear interrupt. (For calls, A and CI are ignored.)

RELADDR is a 6-bit, twos-complement value that is added to the current PC value to generate the branch address. PMI specifies the I register for indirect branches. The I register is pre-modified but not updated by the M register specified by PMM.

The COMPUTE field defines a compute operation to be performed in parallel with the data access; this is a NOP if no compute operation is specified in the instruction.

indirect jump or compute / dreg↔DM

Syntax:

IF condition JUMP | (Md, Ic) / (PC, <reladdr6>) | , ELSE | *compute* / *compute* , DM(Ia, Mb) = dreg / *compute* , dreg = DM(Ia, Mb) | ;

Function:
Conditional jump to the specified PC-relative address or pre-modified I register value, or optional compute operation in parallel with a transfer between data memory and the register file. In this instruction, the IF condition and ELSE keyword are not optional and must be used. If the specified condition is true, the jump is executed. If the specified condition is false, the compute operation and data memory transfer are performed in parallel. Only the compute operation is optional in this instruction.

The PC-relative address for the jump is a 6-bit, twos-complement value. If an I register is specified (Ic), it is modified by the specified M register (Md) to generate the branch address. The I register is not affected by the modify operation. Note that the delayed branch (DB), loop abort (LA), and clear interrupt (CI) modifiers are not available for this jump instruction.

For the data memory access, the I register (Ia) provides the address. The I register value is post-modified by the specified M register and is updated with the modified value. Pre-modify addressing is not available for this data memory access.

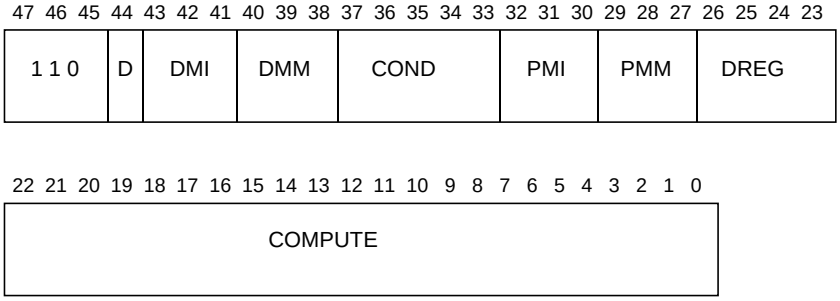
Note: For indirect branches, see Section 4.4.1, “DAG Register Transfer Restrictions”, in Chapter 4, *Data Addressing*.

Examples:

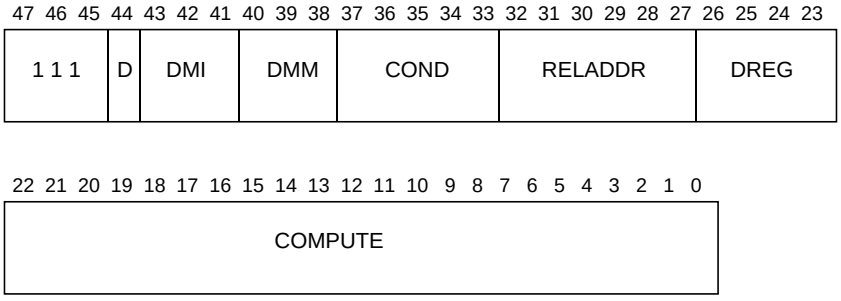
```
IF TF JUMP(M8,I8), ELSE R6=DM(I6,M1);

IF NE JUMP(PC,0x20), ELSE F12=FLOAT R10 BY R3, R6=DM(I5,M0);
```

indirect jump / compute / dreg↔DM

Opcode: *(with indirect jump)*



Opcode:*(with PC-relative jump)*



COND specifies the condition to test.

PMI specifies the I register for indirect branches. The I register is pre-modified but not updated by the M register specified by PMM. RELADDR is a 6-bit, twos-complement value that is added to the current PC value to generate the branch address.

D selects the data memory access type (read or write). DREG specifies the register file location. DMI specifies the I register, which is post-modified and updated by the M register identified by DMM.

The COMPUTE field defines a compute operation to be performed in parallel with the data access; this is a NOP if no compute operation is specified in the instruction.

return from subroutine|interrupt / compute

Syntax:

IF condition RTS (*DB* | *LR* | *DB, LR*) , *compute* | *ELSE compute* ;

IF condition RTI (*DB*) , *compute* | *ELSE compute* ;

Function:
A return from a subroutine (RTS) or return from an interrupt service routine (RTI). If the delayed branch (DB) modifier is specified, the return is delayed; otherwise, it is non-delayed.

A return causes the processor to branch to the address stored at the top of the PC stack. The difference between RTS and RTI is that the RTI instruction not only pops the return address off the PC stack, but also 1) pops the status stack if the ASTAT and MODE1 status registers have been pushed (if the interrupt was $\overline{\text{IRQ2-0}}$, the timer interrupt, or the VIRPT vector interrupt), and 2) clears the appropriate bit in the interrupt latch register (IRPTL) and the interrupt mask pointer (IMASKP).

The return is executed if a condition is specified and is true. If a compute operation is specified without the ELSE, it is performed in parallel with the return. If a compute operation is specified with the ELSE, it is performed only if the condition is false. Note that a condition must be specified if an *ELSE compute* clause is specified.

If a non-delayed call is used as one of the last three instructions of a loop, the loop reentry (LR) modifier must be used with the RTS instruction that returns from the subroutine. The (LR) modifier assures proper reentry into the loop. In counter-based loops, for example, the termination condition is checked by decrementing the current loop counter (CURLCNTR) during execution of the instruction two locations before the end of the loop. The RTS (LR) instruction prevents the loop counter from being decremented again (i.e. twice for the same loop iteration).

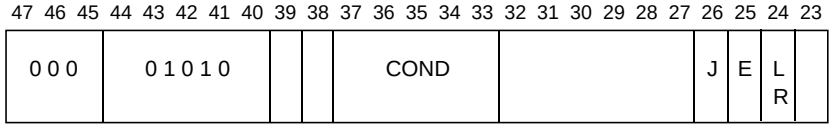
The (LR) modifier of RTS must also be used when returning from a subroutine which has been reduced from an interrupt service routine with a JUMP (CI) instruction (in case the interrupt occurred during the last two instructions of a loop). (For a description of JUMP (CI), refer to instruction Type 8, Direct Jump / Call or Type 9, Indirect Jump / Call.)

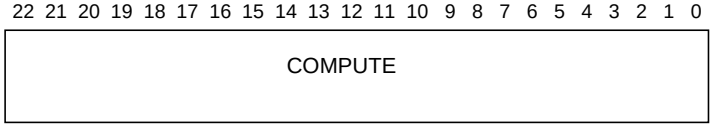
return from subroutine|interrupt / compute

Examples:

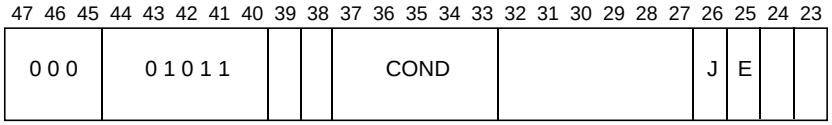
```
RTI, R6=R5 XOR R1;

IF NOT GT RTS(DB);

IF SZ RTS, ELSE R0=LSHIFT R1 BY R15;
```

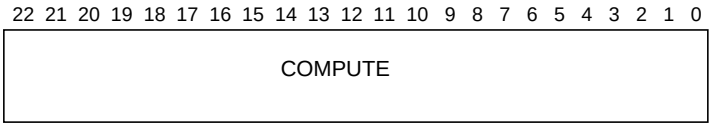
Opcode: *(return from subroutine)*





Opcode: *(return from interrupt)*





COND specifies the condition to test. If no condition is specified in the instruction, COND is the true condition, and the return is always executed. J determines whether the return is delayed or non-delayed. E specifies whether or not an ELSE clause is used.

The COMPUTE field defines the compute operation to be performed; this is a NOP if no compute operation is specified in the instruction.

LR specifies whether or not the loop reentry modifier is specified.

do until counter expired

Syntax:

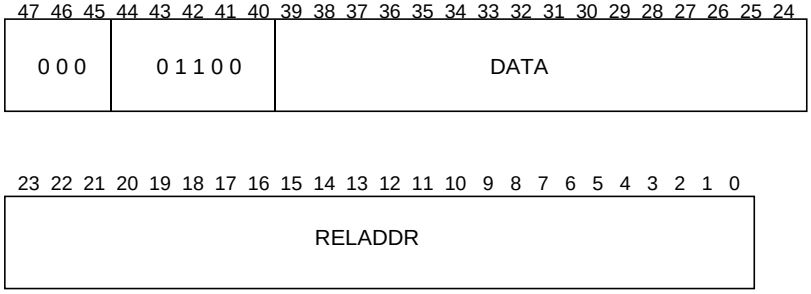
LCNTR = | <data16> / ureg | , DO | <addr24> / (<PC, reladdr24>) | UNTIL LCE ;

Function:
Sets up a counter-based program loop. The loop counter LCNTR is loaded with 16-bit immediate data or from a universal register. The loop start address is pushed on the PC stack. The loop end address and the LCE termination condition are pushed on the loop address stack. The end address can be either a label for an absolute 24-bit program memory address, or a PC-relative 24-bit twos-complement address. The LCNTR is pushed on the loop counter stack and becomes the CURLCNTR value. The loop executes until the CURLCNTR reaches zero.

Examples:

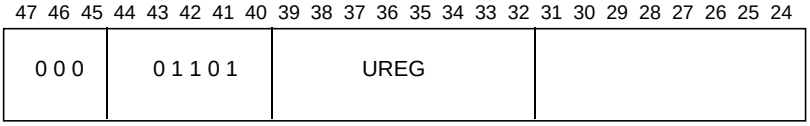
```
LCNTR=100, DO fmax UNTIL LCE;    {fmax is a program label}

LCNTR=R12, DO (PC,16) UNTIL LCE;
```

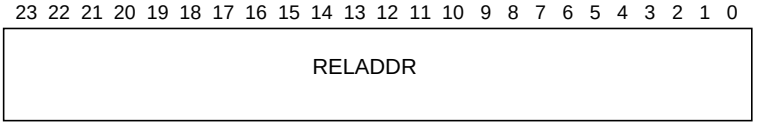
Opcode: *(with immediate loop counter load)*



do until counter expired

Opcode: *(with loop counter load from a universal register)*





RELADDR specifies the end-of-loop address relative to the DO LOOP instruction address. (The Assembler accepts an absolute address as well; it converts the absolute address to the equivalent relative address for coding.) The loop counter (LCNTR) is loaded with the 16-bit DATA value or with the contents of the register specified by UREG.

do until

Syntax:

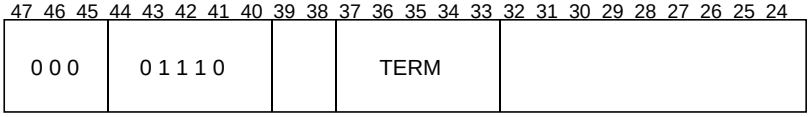
DO | <addr24> / (PC, <reladdr24>) | UNTIL termination ;

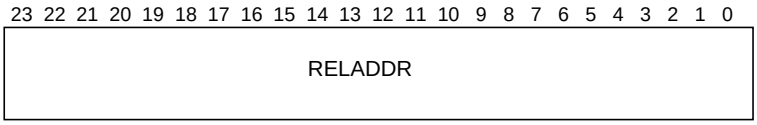
Function:
Sets up a condition-based program loop. The loop start address is pushed on the PC stack. The loop end address and the termination condition are pushed on the loop stack. The end address can be either a label for an absolute 24-bit program memory address or a PC-relative, 24-bit twos-complement address. The loop executes until the termination condition tests true.

Examples:

```
DO end UNTIL FLAG1_IN;           {end is a program label}

DO (PC,7) UNTIL AC;
```

Opcode: *(relative addressing)*





RELADDR specifies the end-of-loop address relative to the DO LOOP instruction address. (The Assembler accepts an absolute address as well; it converts the absolute address to the equivalent relative address for coding.) TERM specifies the termination condition.

Group III.
Immediate Move

Immediate Move

ureg↔DM|PM (direct addressing)

Syntax:

a. | DM(<addr32>) | = ureg ;
| PM(<addr24>) |

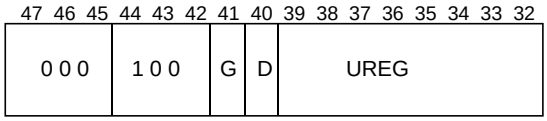
b. ureg = | DM(<addr32>) | ;
| PM(<addr24>) |

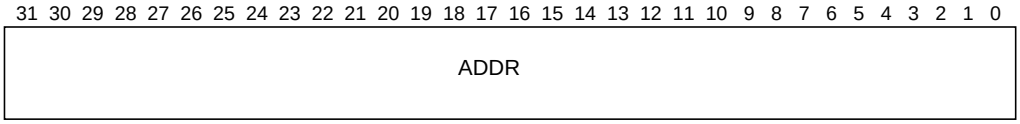
Function:
Access between data memory or program memory and a universal register, with direct addressing. The entire data memory or program memory address is specified in the instruction. Data memory addresses are 32 bits wide (0 to 2^{32}–1). Program memory addresses are 24 bits wide (0 to 2^{24}–1).

Examples:

```
DM(temp)=MODE1;                    {temp is a program label}

DMWAIT=PM(0x489060);
```

Opcode:





D selects the access type (read or write). G selects the memory type (data or program). UREG specifies the number of a universal register. ADDR contains the immediate address value.

ureg↔DM|PM (indirect addressing)

Syntax:

a. | DM(<data32>, Ia) | = ureg ;
| PM(<data24>, Ic) |

b. ureg = | DM(<data32>, Ia) | ;
| PM(<data24>, Ic) |

Function:
Access between data memory or program memory and a universal register, with indirect addressing using I registers. The I register is pre-modified with an immediate value specified in the instruction. The I register is not updated. Data memory address modifiers are 32 bits wide (0 to 2^{32}–1). Program memory address modifiers are 24 bits wide (0 to 2^{24}–1).

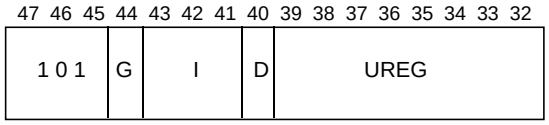
Notes:

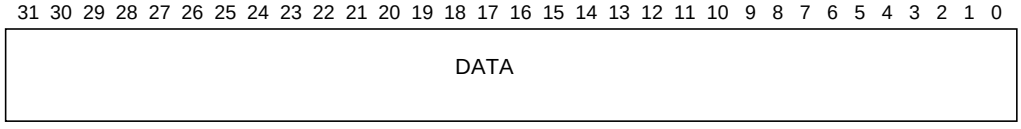
1. ureg may not be from the same DAG (i.e. DAG1 or DAG2) as Ia/Mb or Ic/Md.
2. See Section 4.4.1, “DAG Register Transfer Restrictions”, in Chapter 4, *Data Addressing*.

Examples:

```
DM(24,I5)=TCOUNT;

USTAT1=PM(offs,I13);       {"offs" is a defined constant}
```

Opcode:





D selects the access type (read or write). G selects the memory type (data or program). UREG specifies the number of a universal register. ADDR contains the immediate address value. The I field specifies the I register. The DATA field specifies the immediate modify value for the I register.

A

Immediate Move

immediate data → DM|PM

Syntax:

| DM(Ia, Mb) | = <data32> ;
| PM(Ic, Md) |

Function:
A write of 32-bit immediate data to data or program memory, with indirect addressing. The data is placed in the most significant 32 bits of the 40-bit memory word. The least significant 8 bits are loaded with 0s. The I register is post-modified and updated by the specified M register.

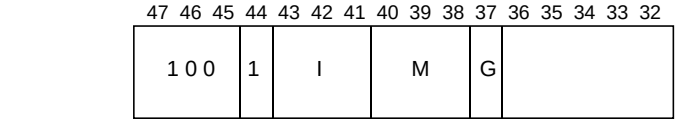
Notes:

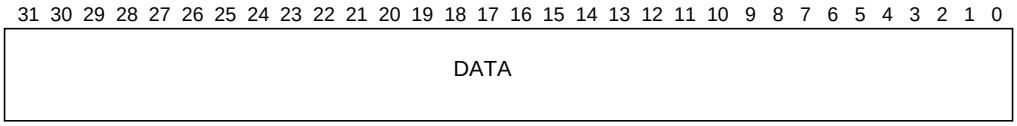
1. ureg may not be from the same DAG (i.e. DAG1 or DAG2) as Ia / Mb or Ic / Md.
2. See Section 4.4.1, “DAG Register Transfer Restrictions”, in Chapter 4, *Data Addressing*.

Examples:

```
DM(I4,M0)=19304;

PM(I14,M11)=count;          {count is user-defined constant}
```

Opcode:





I selects the I register, and M selects the M register. G selects the memory (data or program). DATA specifies the 32-bit immediate data.

immediate data → ureg

Syntax:

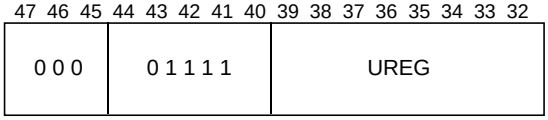
ureg = <data32> ;

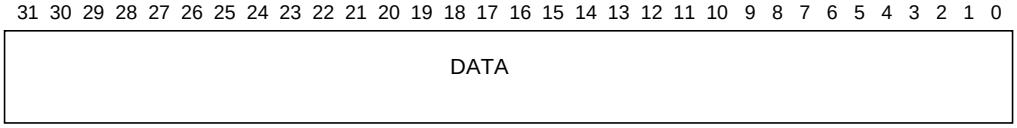
Function:
A write of 32-bit immediate data to a universal register. If the register is 40 bits wide, the data is placed in the most significant 32 bits, and the least significant 8 bits are loaded with 0s.

Examples:

```
IMASK=0xFFFC0060;

M15=mod1;                  {mod1 is user-defined constant}
```

Opcode:





UREG specifies the number of a universal register. The DATA field specifies the immediate data value.

A Instruction Set Reference

Group IV. Miscellaneous

system register bit manipulation

Syntax:

BIT	SET CLR TGL TST XOR	sreg <data32> ;

Function:

A bit manipulation operation on a system register. This instruction can set, clear, toggle or test specified bits, or compare (XOR) the system register with a specified data value. In the first four operations, the immediate data value is a mask. The set operation sets all the bits in the specified system register that are also set in the specified data value. The clear operation clears all the bits that are set in the data value. The toggle operation toggles all the bits that are set in the data value. The test operation sets the bit test flag (BTF in ASTAT) if all the bits that are set in the data value are also set in the system register. The XOR operation sets the bit test flag (BTF in ASTAT) if the system register value is the same as the data value.

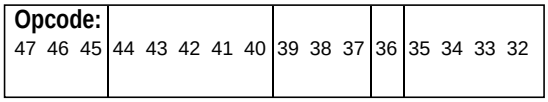
See shifter instructions for bit manipulation of data in the register file. See Appendix E for more information on system registers.

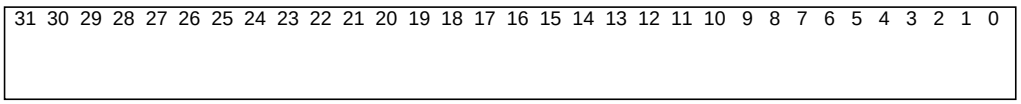
Examples:

```
BIT SET MODE2 0x00000070;
```

system register bit manipulation

```
BIT TST ASTAT 0x00002000;
```





BOP selects one of the five bit operations. SREG specifies the system register. DATA specifies the data value.

I register modify / bit-reverse

Syntax:

a. MODIFY | (Ia, <data32>) | ;
(Ic, <data24>)

b. BITREV | (Ia, <data32>) | ;
(Ic, <data24>)

Function:
Modifies and updates the specified I register by an immediate 32-bit (DAG1) or 24-bit (DAG2) data value. If the address is to be bit-reversed, you must specify a DAG1 register (I0-I7) or DAG2 register (I8-I15), and the modified value is bit-reversed before being written back to the I register. No address is output in either case.

Note: See Section 4.4.1, “DAG Register Transfer Restrictions”, in Chapter 4, *Data Addressing*.

Examples:

```
MODIFY (I4,304);

BITREV (I7,space);          {space is a defined constant}
```

Opcode: *(without bit-reverse)*

47 46 45	44 43 42 41 40	39	38	37 36 35	34 33 32
0 0 0	1 0 1 1 0		G		I

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
DATA

I register modify / bit-reverse

Opcode: *(with bit-reverse)*

47 46 45	44 43 42 41 40	39	38	37 36 35	34 33 32
0 0 0	1 0 1 1 0	1	G		I

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
DATA

G selects the data address generator:

G=0 for DAG1
G=1 for DAG2

I selects the I register:

I=0-7 for I0-I7 (for DAG1)
I=0-7 for I8-I15 (for DAG2)

DATA specifies the immediate modifier.

push|pop stacks / flush cache

Syntax:

PUSH / *POP*	*LOOP* ,	*PUSH* / *POP*	*STS* ,	*PUSH* / *POP*	*PCSTK* ,	*FLUSH CACHE* ;

Function:
Pushes or pops the loop address and loop counter stacks, the status stack, and / or the PC stack, and / or clear the instruction cache. Any of these options may be combined in a single instruction.

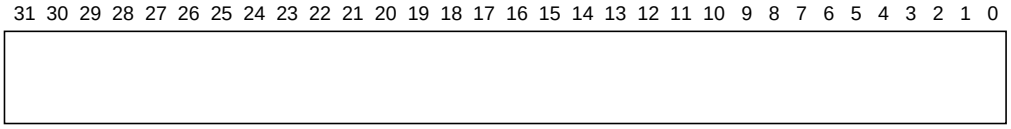
Flushing the instruction cache invalidates all entries in the cache, with no latency—the cache is cleared at the end of the cycle.

Examples:

```
PUSH LOOP, PUSH STS;

POP PCSTK, FLUSH CACHE;
```

Opcode:

47 46 45	44 43 42 41 40	39	38	37	36	35	34	33	32
		LPU	LPO	SPU	SPO	PPU	PPO	FC	
0 0 0	1 0 1 1 1								



LPU pushes the loop stacks. LPO pops the loop stacks. SPU pushes the status stack. SPO pops the status stack. PPU pushes the PC stack. PPO pops the PC stack. FC causes a cache flush.

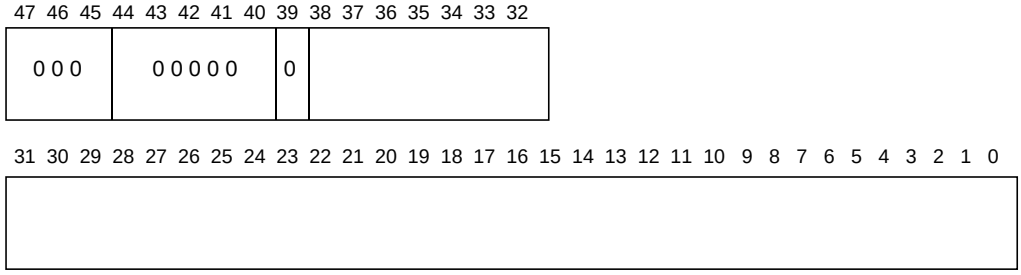
nop

Syntax:

NOP;

Function:
A null operation; only increments the fetch address.

Opcode:



idle

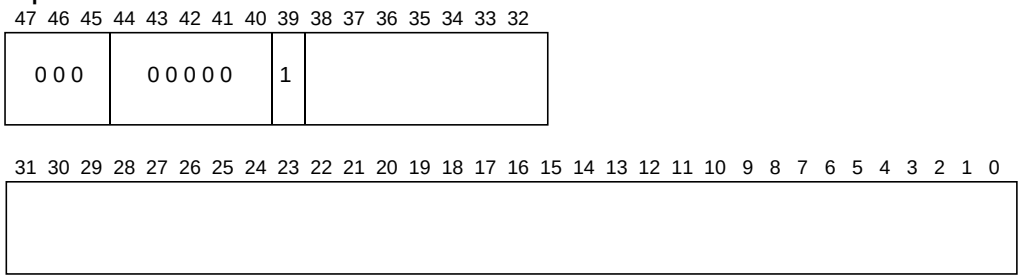
Syntax:

IDLE ;

Function:
Executes a NOP and puts the processor in a low power state. The processor remains in the low power state until an interrupt occurs.

On return from the interrupt, execution continues at the instruction following the IDLE instruction.

Opcode:



idle16

Syntax:

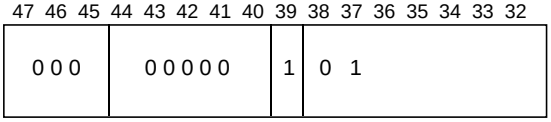
IDLE16 ;

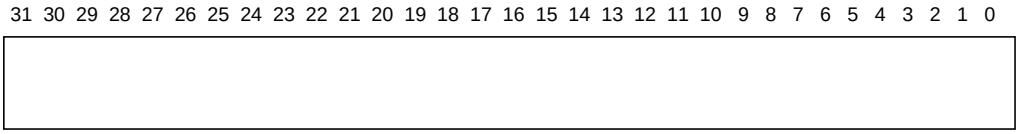
Function:
On the ADSP-21061 only, this instruction executes a NOP and puts the processor in a low power state. IDLE16 is a lower power version of the IDLE instruction. This instruction halts the processor like the IDLE instruction; in this case, the internal clock runs at 1/16th the rate of CLKIN. The ADSP-21061's I/O processor continues to function, but all operations occur at 1/16th the rate. All internal memory transfers require an extra 15 cycles. The serial clocks and frame syncs (if being sourced by the ADSP-21061) are divided down by a factor of 16 during IDLE16. Similarly, all Host accesses take 16 times longer to complete. The processor remains in the low power state until an interrupt occurs.

The processor exits from the IDLE16 state when it detects an external (edge-sensitive) or timer interrupt. Tor information on the minimum pulse width of the external interrupt, see the ADSP-21061 data sheet. The period of the timer interrupt for IDLE16 is TPERIOD x TCK x 16.

After recognizing the interrupt, the processor requires two cycles to exit. Execution continues at the instruction following the IDLE16 instruction.

Opcode:





cjump / rframe

Syntax:

```
CJUMP  | function          |  (DB) ;
       | (PC, <reladdr24>) |

RFRAME ;
```

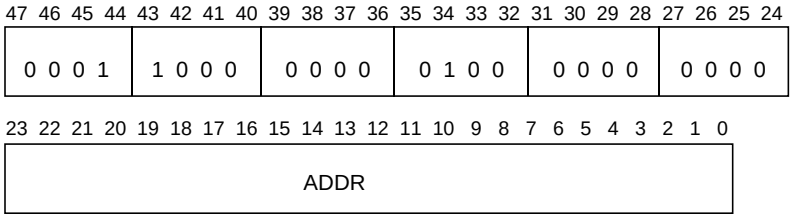
Function:
The CJUMP instruction is generated by the C compiler for function calls, and is not intended for use in assembly language programs. CJUMP combines a direct or PC-relative jump with register transfer operations that save the frame and stack pointers. The RFRAME instruction reverses the register transfers to restore the frame and stack pointers.

The symbol “function” is a 24-bit immediate address for direct jumps. The PC-relative address is a 24-bit, twos-complement value. The (DB) modifier causes the jump to be delayed.

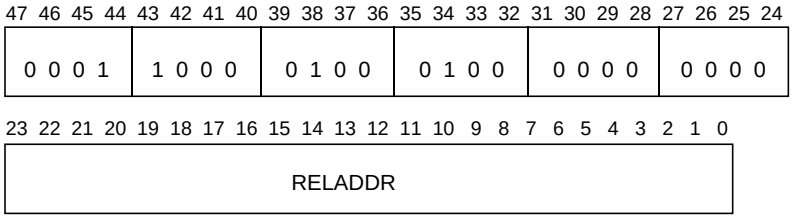
The different forms of this instruction perform the following operations:

Compiler-Generated Instruction	*Operations Performed*
CJUMP function (DB);	JUMP function (DB), R2=I6, I6=I7;
CJUMP (PC,<reladdr24>) (DB);	JUMP (PC,function) (DB), R2=I6, I6=I7;
RFRAME;	I7=I6, I6=DM(0,I6);

Opcode: *(with direct branch)*



Opcode: *(with PC-relative branch)*



cjump / rframe

The ADDR field specifies a 24-bit program memory address for "function." RELADDR is a 24-bit, twos-complement value that is added to the current PC value to generate the branch address.

Opcode: *(RFRAME)*

47 46 45 44	43 42 41 40	39 38 37 36	35 34 33 32	31 30 29 28	27 26 25 24
0 0 0 1	1 0 0 1	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0

23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0

A Instruction Set Reference

Compute Operation Reference

B

B.1 OVERVIEW

Compute operations execute in the multiplier, the ALU and the shifter. The 23-bit compute field is like a mini-instruction within the ADSP-21000 instruction and can be specified for a variety of compute operations. This appendix describes each compute operation in detail, including its assembly language syntax and opcode field.

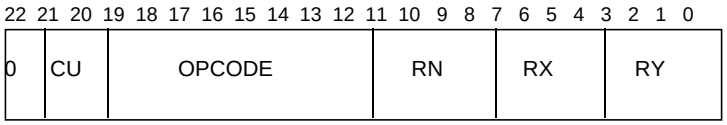
A compute operation is one of the following:

- Single-function operations involve a single computation unit.
- Multifunction operations specify parallel operation of the multiplier and the ALU or two operations in the ALU.
- The MR register transfer is a special type of compute operation used to access the fixed-point accumulator in the multiplier. (See p. B-52).

The operations in each category are described in the following sections. For each operation, the assembly language syntax, the function, and the opcode format and contents are specified. Refer to the beginning of Appendix A for an explanation of the notation and abbreviations used.

B.2 SINGLE-FUNCTION OPERATIONS

The compute field of a single-function operation looks like:



An operation determined by OPCODE is executed in the computation unit specified by CU. The x- and the y-operands are received from data registers RX and RY. The result operand is returned to data register RN.

B Compute Operations

The CU (computation unit) field is defined as follows:

CU=00 ALU operations
CU=01 Multiplier operations
CU=10 Shifter operations

In some shifter operations, data register RN is used both as a destination for a result operand and as source for a third input operand.

The available operations and their 8-bit OPCODE values are listed in the following sections, organized by computation unit: ALU, multiplier and shifter. In each section, the syntax and opcodes for the operations are first summarized and then the operations are described in detail.

B.2.1 ALU Operations

The ALU operations are described in this section. Tables B.1 and B.2 summarize the syntax and opcodes for the fixed-point and floating-point ALU operations, respectively. The rest of this section contains detailed descriptions of each operation.

Syntax	*Opcode*
Rn = Rx + Ry	0000 0001
Rn = Rx – Ry	0000 0010
Rn = Rx + Ry + CI	0000 0101
Rn = Rx – Ry + CI – 1	0000 0110
Rn = (Rx + Ry) / 2	0000 1001
COMP(Rx, Ry)	0000 1010
Rn = Rx + CI	0010 0101
Rn = Rx + CI – 1	0010 0110
Rn = Rx + 1	0010 1001
Rn = Rx – 1	0010 1010
Rn = –Rx	0010 0010
Rn = ABS Rx	0011 0000
Rn = PASS Rx	0010 0001
Rn = Rx AND Ry	0100 0000
Rn = Rx OR Ry	0100 0001
Rn = Rx XOR Ry	0100 0010
Rn = NOT Rx	0100 0011
Rn = MIN(Rx, Ry)	0110 0001
Rn = MAX(Rx, Ry)	0110 0010
Rn = CLIP Rx BY Ry	0110 0011

Table B.1 Fixed-Point ALU Operations

Syntax	*Opcode*
Fn = Fx + Fy	1000 0001
Fn = Fx – Fy	1000 0010
Fn = ABS (Fx + Fy)	1001 0001
Fn = ABS (Fx – Fy)	1001 0010
Fn = (Fx + Fy)/2	1000 1001
COMP(Fx, Fy)	1000 1010
Fn = –Fx	1010 0010
Fn = ABS Fx	1011 0000
Fn = PASS Fx	1010 0001
Fn = RND Fx	1010 0101
Fn = SCALB Fx BY Ry	1011 1101
Rn = MANT Fx	1010 1101
Rn = LOGB Fx	1100 0001
Rn = FIX Fx BY Ry	1101 1001
Rn = FIX Fx	1100 1001
Rn = TRUNC Fx BY Ry	1101 1101
Rn = TRUNC Fx	1100 1101
Fn = FLOAT Rx BY Ry	1101 1010
Fn = FLOAT Rx	1100 1010
Fn = RECIPS Fx	1100 0100
Fn = RSQRTS Fx	1100 0101
Fn = Fx COPYSIGN Fy	1110 0000
Fn = MIN(Fx, Fy)	1110 0001
Fn = MAX(Fx, Fy)	1110 0010
Fn = CLIP Fx BY Fy	1110 0011

Table B.2 Floating-Point ALU Operations

The individual registers of the register file are prefixed with an “F” when used in floating-point computations. The registers are prefixed with an “R” when used in fixed-point computations. The following instructions, for example, use the same registers:

```
F0=F1*F2;        floating-point multiply
R0=R1*R2;        fixed-point multiply
```

The F and R prefixes do not affect the 32-bit (or 40-bit) data transfer; they only determine how the ALU, multiplier, or shifter treat the data. The F and R may be either uppercase or lowercase; the assembler is case-insensitive.

Rn = Rx + Ry

Syntax:

Rn = Rx + Ry

Function:
Adds the fixed-point fields in registers Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status flags:

AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = Rx – Ry

Syntax:

Rn = Rx – Ry

Function:
Subtracts the fixed-point field in register Ry from the fixed-point field in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = Rx + Ry + CI

Syntax:

Rn = Rx + Ry + CI

Function:
Adds with carry (AC from ASTAT) the fixed-point fields in registers Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = Rx – Ry + CI – 1

Syntax:

Rn = Rx – Ry + CI – 1

Function:
Subtracts with borrow (AC – 1 from ASTAT) the fixed-point field in register Ry from the fixed-point field in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status flags:

AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = (Rx + Ry)/2

Syntax:

Rn = (Rx + Ry) / 2

Function:
Adds the fixed-point fields in registers Rx and Ry and divides the result by 2. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit in the MODE1 register.

Status flags:

AZ	Is set if the fixed-point output is all 0s, otherwise cleared
AU	Is cleared
AN	Is set if the most significant output bit is 1, otherwise cleared
AV	Is cleared
AC	Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS	Is cleared
AI	Is cleared

COMP(Rx, Ry)

Syntax:

COMP(Rx, Ry)

Function:
Compares the fixed-point field in register Rx with the fixed-point field in register Ry. Sets the AZ flag if the two operands are equal, and the AN flag if the operand in register Rx is smaller than the operand in register Ry.

The ASTAT register stores the results of the previous eight ALU compare operations in bits 24-31. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed. The MSB of ASTAT is set if the X operand is greater than the Y operand (its value is the AND of ~AZ and ~AN); it is otherwise cleared.

Status flags:
AZ Is set if the operands in registers Rx and Ry are equal, otherwise cleared
AU Is cleared
AN Is set if the operand in the Rx register is smaller than the operand in the Ry register, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = Rx + CI

Syntax:

Rn = Rx + CI

Function:
Adds the fixed-point field in register Rx with the carry flag from the ASTAT register (AC). The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF).

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = Rx + CI – 1

Syntax:

Rn = Rx + CI – 1

Function:
Adds the fixed-point field in register Rx with the borrow from the ASTAT register (AC – 1). The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF).

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = Rx + 1

Syntax:

Rn = Rx + 1

Function:
Increments the fixed-point operand in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

Status flags:

AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder, stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = Rx – 1

Syntax:

Rn = Rx – 1

Function:
Decrements the fixed-point operand in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set), underflow causes the minimum negative number (0x8000 0000) to be returned.

Status flags:

AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = –Rx

Syntax:

Rn = –Rx

Function:

Negates the fixed-point operand in Rx by twos complement. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. Negation of the minimum negative number (0x8000 0000) causes an overflow. In saturation mode (the ALU saturation mode bit in MODE1 set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

Status flags:

AZ Is set if the fixed-point output is all 0s
AU Is cleared
AN Is set if the most significant output bit is 1
AV Is set if the XOR of the carries of the two most significant adder stages is 1
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = ABS Rx

Syntax:

Rn = ABS Rx

Function:
Determines the absolute value of the fixed-point operand in Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. ABS of the minimum negative number (0x8000 0000) causes an overflow. In saturation mode (the ALU saturation mode bit in MODE1 set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is set if the fixed-point operand in Rx is negative, otherwise cleared
AI Is cleared

Rn = PASS Rx

Syntax:

Rn = PASS Rx

Function:
Passes the fixed-point operand in Rx through the ALU to the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = Rx AND Ry

Syntax:

Rn = Rx AND Ry

Function:
Logically ANDs the fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = Rx OR Ry

Syntax:

Rn = Rx OR Ry

Function:
Logically ORs the fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = Rx XOR Ry

Syntax:

Rn = Rx XOR Ry

Function:
Logically XORs the fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = NOT Rx

Syntax:

Rn = NOT Rx

Function:
Logically complements the fixed-point operand in Rx. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = MIN(Rx, Ry)

Syntax:

Rn = MIN(Rx, Ry)

Function:
Returns the smaller of the two fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = MAX(Rx, Ry)

Syntax:

Rn = MAX(Rx, Ry)

Function:
Returns the larger of the two fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = CLIP Rx BY Ry

Syntax:

Rn = CLIP Rx BY Ry

Function:
Returns the fixed-point operand in Rx if the absolute value of the operand in Rx is less than the absolute value of the fixed-point operand in Ry. Otherwise, returns |Ry| if Rx is positive, and –|Ry| if Rx is negative. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Fn = Fx + Fy

Syntax:

Fn = Fx + Fy

Function:
Adds the floating-point operands in registers Fx and Fy. The normalized result is placed in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal returns ±Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:

AZ	Is set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Is set if the post-rounded result is a denormal, otherwise cleared
AN	Is set if the floating-point result is negative, otherwise cleared
AV	Is set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC	Is cleared
AS	Is cleared
AI	Is set if either of the input operands is a NAN, or if they are opposite-signed Infinities, otherwise cleared

Fn = Fx – Fy

Syntax:

Fn = Fx – Fy

Function:
Subtracts the floating-point operand in register Fy from the floating-point operand in register Fx. The normalized result is placed in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal returns ±Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:

AZ	Is set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Is set if the post-rounded result is a denormal, otherwise cleared
AN	Is set if the floating-point result is negative, otherwise cleared
AV	Is set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC	Is cleared
AS	Is cleared
AI	Is set if either of the input operands is a NAN, or if they are like-signed Infinities, otherwise cleared

Fn = ABS (Fx + Fy)

Syntax:

Fn = ABS (Fx + Fy)

Function:
Adds the floating-point operands in registers Fx and Fy, and places the absolute value of the normalized result in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns +Infinity (round-to-nearest) or +NORM.MAX (round-to-zero). Post-rounded denormal returns +Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:

AZ	Is set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Is set if the post-rounded result is a denormal, otherwise cleared
AN	Is cleared
AV	Is set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC	Is cleared
AS	Is cleared
AI	Is set if either of the input operands is a NAN, or if they are opposite-signed Infinities, otherwise cleared

Fn = ABS (Fx – Fy)

Syntax:

Fn = ABS (Fx – Fy)

Function:
Subtracts the floating-point operand in Fy from the floating-point operand in Fx and places the absolute value of the normalized result in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns +Infinity (round-to-nearest) or +NORM.MAX (round-to-zero). Post-rounded denormal returns +Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:

AZ	Is set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Is set if the post-rounded result is a denormal, otherwise cleared
AN	Is cleared
AV	Is set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC	Is cleared
AS	Is cleared
AI	Is set if either of the input operands is a NAN, or if they are like-signed Infinities, otherwise cleared

Fn = (Fx + Fy)/2

Syntax:

Fn = (Fx + Fy) / 2

Function:
Adds the floating-point operands in registers Fx and Fy and divides the result by 2, by decrementing the exponent of the sum before rounding. The normalized result is placed in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal results return ±Zero. A denormal input is flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:

AZ	Is set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Is set if the post-rounded result is a denormal, otherwise cleared
AN	Is set if the floating-point result is negative, otherwise cleared
AV	Is set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC	Is cleared
AS	Is cleared
AI	Is set if either of the input operands is a NAN, or if they are opposite-signed Infinities, otherwise cleared

COMP(Fx, Fy)

Syntax:

COMP(Fx, Fy)

Function:
Compares the floating-point operand in register Fx with the floating-point operand in register Fy. Sets the AZ flag if the two operands are equal, and the AN flag if the operand in register Fx is smaller than the operand in register Fy.

The ASTAT register stores the results of the previous eight ALU compare operations in bits 24-31. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed. The MSB of ASTAT is set if the X operand is greater than the Y operand (its value is the AND of ~AZ and ~AN); it is otherwise cleared.

Status flags:
AZ Is set if the operands in registers Fx and Fy are equal, otherwise cleared
AU Is cleared
AN Is set if the operand in the Fx register is smaller than the operand in the Fy register, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, otherwise cleared

Fn = –Fx

Syntax:

Fn = –Fx

Function:
Complements the sign bit of the floating-point operand in Fx. The complemented result is placed in register Fn. A denormal input is flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if the result operand is a ±Zero, otherwise cleared
AU Is cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if the input operand is a NAN, otherwise cleared

Fn = ABS Fx

Syntax:

Fn = ABS Fx

Function:
Returns the absolute value of the floating-point operand in register Fx by setting the sign bit of the operand to 0. Denormal inputs are flushed to +Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if the result operand is +Zero, otherwise cleared.
AU Is cleared
AN Is cleared
AV Is cleared
AC Is cleared
AS Is set if the input operand is negative, otherwise cleared
AI Is set if the input operand is a NAN, otherwise cleared

Fn = PASS Fx

Syntax:

Fn = PASS Fx

Function:
Passes the floating-point operand in Fx through the ALU to the floating-point field in register Fn. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if the result operand is a ±Zero, otherwise cleared
AU Is cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if the input operand is a NAN, otherwise cleared

Fn = RND Fx

Syntax:

Fn = RND Fx

Function:
Rounds the floating-point operand in register Fx to a 32 bit boundary. Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit in MODE1. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). A denormal input is flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:

AZ	Is set if the result operand is a ±Zero, otherwise cleared
AU	Is cleared
AN	Is set if the floating-point result is negative, otherwise cleared
AV	Is set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC	Is cleared
AS	Is cleared
AI	Is set if the input operand is a NAN, otherwise cleared

Fn = SCALB Fx BY Ry

Syntax:

Fn = SCALB Fx BY Ry

Function:
Scales the exponent of the floating-point operand in Fx by adding to it the fixed-point twos-complement integer in Ry. The scaled floating-point result is placed in register Fn. Overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Denormal returns ±Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if the result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU Is set if the post-rounded result is a denormal, otherwise cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is set if the result overflows (unbiased exponent > +127), otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if the input is a NAN, an otherwise cleared

Rn = MANT Fx

Syntax:

Rn = MANT Fx

Function:
Extracts the mantissa (fraction bits with explicit hidden bit, excluding the sign bit) from the floating-point operand in Fx. The unsigned-magnitude result is left-justified (1.31 format) in the fixed-point field in Rn. Rounding modes are ignored and no rounding is performed because all results are inherently exact. Denormal inputs are flushed to ±Zero. A NAN or an Infinity input returns an all 1s result (–1 in signed fixed-point format).

Status flags:

AZ	Is set if the result is zero, otherwise cleared
AU	Is cleared
AN	Is cleared
AV	Is cleared
AC	Is cleared
AS	Is set if the input is negative, otherwise cleared
AI	Is set if the input operands is a NAN or an Infinity, otherwise cleared

Rn = LOGB Fx

Syntax:

Rn = LOGB Fx

Function:
Converts the exponent of the floating-point operand in register Fx to an unbiased twos-complement fixed-point integer. The result is placed in the fixed-point field in register Rn. Unbiasing is done by subtracting 127 from the floating-point exponent in Fx. If saturation mode is not set, a ±Infinity input returns a floating-point +Infinity and a ±Zero input returns a floating-point –Infinity. If saturation mode is set, a ±Infinity input returns the maximum positive value (0x7FFF FFFF) and a ±Zero input returns the maximum negative value (0x8000 0000). Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if the fixed-point result is zero, otherwise cleared
AU Is cleared
AN Is set if the result is negative, otherwise cleared
AV Is set if the input operand is an Infinity or a Zero, otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if the input is a NAN, otherwise cleared

Rn = FIX Fx
Rn = TRUNC Fx

Syntax:

Rn = FIX Fx	Rn = TRUNC Fx
Rn = FIX Fx BY Ry	Rn = TRUNC Fx BY Ry

Function:
Converts the floating-point operand in Fx to a twos-complement 32-bit fixed-point integer result. If the MODE1 register TRUNC bit=1, the FIX operation truncates the mantissa towards –Infinity. If the TRUNC bit=0, the FIX operation rounds the mantissa towards the nearest integer. The TRUNC operation always truncates toward 0. Note that the TRUNC bit does not influence operation of the TRUNC instruction.

If a scaling factor (Ry) is specified, the fixed-point twos-complement integer in Ry is added to the exponent of the floating-point operand in Fx before the conversion. The result of the conversion is right-justified (32.0 format) in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows and +Infinity return the maximum positive number (0x7FFF FFFF), and negative overflows and –Infinity return the minimum negative number (0x8000 0000).

For the FIX operation, rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit in MODE1. A NAN input returns a floating-point all 1s result. If saturation mode is not set, an Infinity input or a result that overflows returns a floating-point all 1s result. All positive underflows return zero. Negative underflows that are rounded-to-nearest return zero, and negative underflows that are rounded by truncation return –1 (0xFF FFFF FF00).

Status flags:

AZ Is set if the fixed-point result is Zero, otherwise cleared
AU Is set if the pre-rounded result is a denormal, otherwise cleared
AN Is set if the fixed-point result is negative, otherwise cleared
AV Is set if the conversion causes the floating-point mantissa to be shifted left, i.e. if the floating-point exponent + scale bias is > 157 (127 + 31 – 1) or if the input is ±Infinity, otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if the input operand is a NAN or, when saturation mode is not set, either input is an Infinity or the result overflows, otherwise cleared

Fn = FLOAT Rx BY Ry / Fn = FLOAT Rx

Syntax:

Fn = FLOAT Rx BY Ry
Fn = FLOAT Rx

Function:
Converts the fixed-point operand in Rx to a floating-point result. If a scaling factor (Ry) is specified, the fixed-point twos-complement integer in Ry is added to the exponent of the floating-point result. The final result is placed in register Fn.

Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode, to a 40-bit boundary, regardless of the values of the rounding boundary bits in MODE1. The exponent scale bias may cause a floating-point overflow or a floating-point underflow. Overflow causes a ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero) to be returned; underflow causes a ±Zero to be returned.

Status flags:
AZ Is set if the result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU Is set if the post-rounded result is a denormal, otherwise cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is set if the result overflows (unbiased exponent >127)
AC Is cleared
AS Is cleared
AI Is cleared

Fn = RECIPS Fx

Syntax:

Fn = RECIPS Fx

Function:
Creates an 8-bit accurate seed for 1/Fx, the reciprocal of Fx. The mantissa of the seed is determined from a ROM table using the 7 MSBs (excluding the hidden bit) of the Fx mantissa as an index. The unbiased exponent of the seed is calculated as the twos complement of the unbiased Fx exponent, decremented by one; i.e., if *e* is the unbiased exponent of Fx, then the unbiased exponent of Fn = $-e-1$. The sign of the seed is the sign of the input. ±Zero returns ±Infinity and sets the overflow flag. If the unbiased exponent of Fx is greater than +125, the result is ±Zero. A NAN input returns an all 1s result.

The following code performs floating-point division using an iterative convergence algorithm.* The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set (32-bit only for ADSP-21010). The following inputs are required: F0=numerator, F12=denominator, F11=2.0. The quotient is returned in F0. (The two highlighted instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

```
F0=RECIPS F12, F7=F0;           {Get 8 bit seed R0=1/D}
F12=F0*F12;                     {D' = D*R0}
F7=F0*F7, F0=F11-F12;           {F0=R1=2-D', F7=N*R0}
F12=F0*F12;                     {F12=D'-D'*R1}
F7=F0*F7, F0=F11-F12;           {F7=N*R0*R1, F0=R2=2-D'}
 F12=F0*F12;                     {F12=D'=D'*R2}
 F7=F0*F7, F0=F11-F12;           {F7=N*R0*R1*R2, F0=R3=2-D'}
F0=F0*F7;                       {F7=N*R0*R1*R2*R3}
```

Note that this code segment can be made into a subroutine by adding an `RTS(DB)` clause to the third-to-last instruction.

Status flags:

AZ Is set if the floating-point result is ±Zero (unbiased exponent of Fx is greater than +125), otherwise cleared
AU Is cleared
AN Is set if the input operand is negative, otherwise cleared
AV Is set if the input operand is ±Zero, otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if the input operand is a NAN, otherwise cleared

* Cavanagh, J. 1984. *Digital Computer Arithmetic*. McGraw-Hill. Page 284.

Fn = RSQRTS Fx

Syntax:

Fn = RSQRTS Fx

Function: Creates a 4-bit accurate seed for $1/\sqrt{Fx}$, the reciprocal square root of Fx. The mantissa of the seed is determined from a ROM table using the LSB of the biased exponent of Fx concatenated with the 6 MSBs (excluding the hidden bit) of the mantissa of Fx as an index. The unbiased exponent of the seed is calculated as the twos complement of the unbiased Fx exponent, shifted right by one bit and decremented by one; i.e., if *e* is the unbiased exponent of Fx, then the unbiased exponent of Fn = $-\text{INT}[e/2] - 1$. The sign of the seed is the sign of the input. ±Zero returns ±Infinity and sets the overflow flag. +Infinity returns +Zero. A NAN input or a negative nonzero input returns an all 1s result.

The following code calculates a floating-point reciprocal square root ($1/\sqrt{x}$) using a Newton-Raphson iteration algorithm.* The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set (32-bit only for ADSP-21010). To calculate the square root, simply multiply the result by the original input. The following inputs are required: F0=input, F8=3.0, F1=0.5. The result is returned in F4. (The four highlighted instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

```
F4=RSQRTS F0;                    {Fetch 4-bit seed}
F12=F4*F4;                       {F12=X0^2}
F12=F12*F0;                      {F12=C*X0^2}
F4=F1*F4, F12=F8-F12;            {F4=.5*X0, F12=3-C*X0^2}
F4=F4*F12;                       {F4=X1=.5*X0(3-C*X0^2)}
F12=F4*F4;                       {F12=X1^2}
F12=F12*F0;                      {F12=C*X1^2}
F4=F1*F4, F12=F8-F12;            {F4=.5*X1, F12=3-C*X1^2}
 F4=F4*F12;                       {F4=X2=.5*X1(3-C*X1^2)}
 F12=F4*F4;                       {F12=X2^2}
 F12=F12*F0;                      {F12=C*X2^2}
 F4=F1*F4, F12=F8-F12;            {F4=.5*X2, F12=3-C*X2^2}
F4=F4*F12;                       {F4=X3=.5*X2(3-C*X2^2)}
```

Note that this code segment can be made into a subroutine by adding an `RTS(DB)` clause to the third-to-last instruction.

Status flags:

AZ Is set if the floating-point result is +Zero (Fx = +Infinity), otherwise cleared
AU Is cleared
AN Is set if the input operand is –Zero, otherwise cleared
AV Is set if the input operand is ±Zero, otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if the input operand is negative and nonzero, or a NAN, otherwise cleared

* Cavanagh, J. 1984. *Digital Computer Arithmetic.* McGraw-Hill. Page 278.

Fn = Fx COPYSIGN Fy

Syntax:

Fn = Fx COPYSIGN Fy

Function:
Copies the sign of the floating-point operand in register Fy to the floating-point operand from register Fx without changing the exponent or the mantissa. The result is placed in register Fn. A denormal input is flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if the floating-point result is ±Zero, otherwise cleared
AU Is cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, otherwise cleared

Fn = MIN(Fx, Fy)

Syntax:

Fn = MIN(Fx, Fy)

Function:
Returns the smaller of the floating-point operands in register Fx and Fy. A NAN input returns an all 1s result. MIN of +Zero and –Zero returns –Zero. Denormal inputs are flushed to ±Zero.

Status flags:
AZ Is set if the floating-point result is ±Zero, otherwise cleared.
AU Is cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, otherwise cleared

Fn = MAX(Fx, Fy)

Syntax:

Fn = MAX(Fx, Fy)

Function:
Returns the larger of the floating-point operands in registers Fx and Fy. A NAN input returns an all 1s result. MAX of +Zero and –Zero returns +Zero. Denormal inputs are flushed to ±Zero.

Status flags:
AZ Is set if the floating-point result is ±Zero, otherwise cleared.
AU Is cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, otherwise cleared

Fn = CLIP Fx BY Fy

Syntax:

Fn = CLIP Fx BY Fy

Function:
Returns the floating-point operand in Fx if the absolute value of the operand in Fx is less than the absolute value of the floating-point operand in Fy. Else, returns | Fy | if Fx is positive, and –| Fy | if Fx is negative. A NAN input returns an all 1s result. Denormal inputs are flushed to ±Zero.

Status flags:
AZ Is set if the floating-point result is ±Zero, otherwise cleared.
AU Is cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, otherwise cleared

B.2.2 Multiplier Operations

The multiplier operations are described in this section. Table B.3 summarizes the syntax and opcodes for the fixed-point and floating-point multiplier operations. The rest of this section contains detailed descriptions of each operation.

Fixed-point:

Syntax	*Opcode*
Rn = Rx ∗ Ry *mod2*†	01yx f00r
MRF = Rx ∗ Ry *mod2*†	01yx f10r
MRB = Rx ∗ Ry *mod2*†	01yx f11r
Rn = MRF + Rx ∗ Ry *mod2*†	10yx f00r
Rn = MRB + Rx ∗ Ry *mod2*†	10yx f01r
MRF = MRF + Rx ∗ Ry *mod2*†	10yx f10r
MRB = MRB + Rx ∗ Ry *mod2*†	10yx f11r
Rn = MRF – Rx ∗ Ry *mod2*†	11yx f00r
Rn = MRB – Rx ∗ Ry *mod2*†	11yx f01r
MRF = MRF – Rx ∗ Ry *mod2*†	11yx f10r
MRB = MRB – Rx ∗ Ry *mod2*†	11yx f11r
Rn = SAT MRF *mod1*††	0000 f00x
Rn = SAT MRB *mod1*††	0000 f01x
MRF = SAT MRF *mod1*††	0000 f10x
MRB = SAT MRB *mod1*††	0000 f11x
Rn = RND MRF *mod1*††	0001 100x
Rn = RND MRB *mod1*††	0001 101x
MRF = RND MRF *mod1*††	0001 110x
MRB = RND MRB *mod1*††	0001 111x
MRF = 0	0001 0100
MRB = 0	0001 0110
MR = Rn	
Rn = MR	

Floating-point:

Syntax	*Opcode*
Fn = Fx ∗ Fy	0011 0000

† See Table B.4
†† See Table B.5

y	y-input; 1=signed, 0=unsigned
x	x-input; 1=signed, 0=unsigned
f	format; 1=fractional, 0=integer
r	rounding; 1=yes, 0=no

Table B.3 Multiplier Operations

Mod2 in Table B.3 is an optional modifier, enclosed in parentheses, consisting of three or four letters that indicate whether the x-input is signed (S) or unsigned (U), whether the y-input is signed or unsigned, whether the inputs are in integer (I) or fractional (F) format and whether the result when written to the register file is to be rounded-to-nearest (R). The options for *mod2* and the corresponding opcode values are listed in Table B.4.

Mod2	*Opcode*
(SSI)	--11 0--0
(SUI)	--01 0--0
(USI)	--10 0--0
(UUI)	--00 0--0
(SSF)	--11 1--0
(SUF)	--01 1--0
(USF)	--10 1--0
(UUF)	--00 1--0
(SSFR)	--11 1--1
(SUFR)	--01 1--1
(USFR)	--10 1--1
(UUFR)	--00 1--1

Table B.4 Multiplier Mod2 Options

Similarly, *mod1* in Table B.3 is an optional modifier, enclosed in parentheses, consisting of two letters that indicate whether the input is signed (S) or unsigned (U) and whether the input is in integer (I) or fractional (F) format. The options for *mod1* and the corresponding opcode values are listed in Table B.5.

Mod1	*Opcode*
(SI) (for SAT only)	---- 0--1
(UI) (for SAT only)	---- 0--0
(SF)	---- 1--1
(UF)	---- 1--0

Table B.5 Multiplier Mod1 Options

Rn|MR = Rx * Ry

Syntax:

```
Rn    = Rx * Ry   mod2
MRF   = Rx * Ry   mod2
MRB   = Rx * Ry   mod2
```

Function:
Multiplies the fixed-point fields in registers Rx and Ry. If rounding is specified (fractional data only), the result is rounded. The result is placed either in the fixed-point field in register Rn or one of the MR accumulation registers. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31-0 for integers, bits 63-32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status flags:
MN Is set if the result is negative, otherwise cleared
MV Is set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result). Number of upper bits depends on format. For a signed result, fractional=33, integer=49. For an unsigned result, fractional=32, integer=48.
MU Is set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros. Integer results do not underflow.
MI Is cleared

Multiplier Fixed-Point

Rn|MR = MR + Rx * Ry

Syntax:

Rn	= MRF	+ Rx * Ry	*mod2*
Rn	= MRB	+ Rx * Ry	*mod2*
MRF	= MRF	+ Rx * Ry	*mod2*
MRB	= MRB	+ Rx * Ry	*mod2*

Function:
Multiplies the fixed-point fields in registers Rx and Ry, and adds the product to the specified MR register value. If rounding is specified (fractional data only), the result is rounded. The result is placed either in the fixed-point field in register Rn or one of the MR accumulation registers, which must be the same MR register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31-0 for integers, bits 63-32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status flags:

MN Is set if the result is negative, otherwise cleared

MV Is set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result). Number of upper bits depends on format. For a signed result, fractional=33, integer=49. For an unsigned result, fractional=32, integer=48.

MU Is set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros. Integer results do not underflow.

MI Is cleared

Rn|MR = MR – Rx * Ry

Syntax:

```
Rn    = MRF  – Rx * Ry  mod2
Rn    = MRB  – Rx * Ry  mod2
MRF   = MRF  – Rx * Ry  mod2
MRB   = MRB  – Rx * Ry  mod2
```

Function:
Multiplies the fixed-point fields in registers Rx and Ry, and subtracts the product from the specified MR register value. If rounding is specified (fractional data only), the result is rounded. The result is placed either in the fixed-point field in register Rn or one of the MR accumulation registers, which must be the same MR register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31-0 for integers, bits 63-32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status flags:
MN Is set if the result is negative, otherwise cleared
MV Is set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result). Number of upper bits depends on format. For a signed result, fractional=33, integer=49. For an unsigned result, fractional=32, integer=48.
MU Is set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros. Integer results do not underflow.
MI Is cleared

Rn|MR = SAT MR

Syntax:

Rn	= SAT MRF	*mod1*
Rn	= SAT MRB	*mod1*
MRF	= SAT MRF	*mod1*
MRB	= SAT MRB	*mod1*

Function:
If the value of the specified MR register is greater than the maximum value for the specified data format, the multiplier sets the result to the maximum value. Otherwise, the MR value is unaffected. The result is placed either in the fixed-point field in register Rn or one of the MR accumulation registers, which must be the same MR register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31-0 for integers, bits 63-32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status flags:
MN Is set if the result is negative, otherwise cleared
MV Is cleared
MU Is set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros. Integer results do not underflow.
MI Is cleared

Rn|MR = RND MR

Syntax:

Rn	= RND MRF	*mod1*
Rn	= RND MRB	*mod1*
MRF	= RND MRF	*mod1*
MRB	= RND MRB	*mod1*

Function:
Rounds the specified MR value to nearest at bit 32 (the MR1-MR0 boundary). The result is placed either in the fixed-point field in register Rn or one of the MR accumulation registers, which must be the same MR register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31-0 for integers, bits 63-32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status flags:
MN Is set if the result is negative, otherwise cleared
MV Is set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result). Number of upper bits depends on format. For a signed result, fractional=33, integer=49. For an unsigned result, fractional=32, integer=48.
MU Is set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros. Integer results do not underflow.
MI Is cleared

MR=0

Syntax:
MRF = 0
MRB = 0

Function: Sets the value of the specified MR register to zero. All 80 bits (MR2, MR1, MR0) are cleared.

Status flags:
MN Is cleared
MV Is cleared
MU Is cleared
MI Is cleared

MR=Rn / Rn=MR

Function: A transfer to an MR register places the fixed-point field of register Rn in the specified MR register. The floating-point extension field in Rn is ignored. A transfer from an MR register places the specified MR register in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Syntax:

MR0F = Rn	Rn = MR0F
MR1F = Rn	Rn = MR1F
MR2F = Rn	Rn = MR2F
MR0B = Rn	Rn = MR0B
MR1B = Rn	Rn = MR1B
MR2B = Rn	Rn = MR2B

Compute Field:

22	21 20 19 18 17	16	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1	0 0 0 0 0	T	AI	RK	

The MR register is specified by Ai and the data register by Rk. The direction of the transfer is determined by T (0=to register file, 1=to MR register).

Ai	MR Register
0000	MR0F
0001	MR1F
0010	MR2F
0100	MR0B
0101	MR1B
0110	MR2B

Status flags:
MN Is cleared
MV Is cleared
MU Is cleared
MI Is cleared

Fn = Fx * Fy

Syntax:

Fn = Fx * Fy

Function:
Multiplies the floating-point operands in registers Fx and Fy. The result is placed in the register Fn.

Status flags:
MN Is set if the result is negative, otherwise cleared
MV Is set if the unbiased exponent of the result is greater than 127, otherwise cleared
MU Is set if the unbiased exponent of the result is less than –126, otherwise cleared
MI Is set if either input is a NAN or if the inputs are ±Infinity and ±Zero, otherwise cleared

Reminder: The individual registers of the register file are prefixed with an "F" when used in floating-point computations. The registers are prefixed with an "R" when used in fixed-point computations. The following instructions, for example, use the same registers:

```
F0=F1*F2;        floating-point multiply
R0=R1*R2;        fixed-point multiply
```

The F and R prefixes do not affect the 32-bit (or 40-bit) data transfer; they only determine how the ALU, multiplier, or shifter treat the data. The F or R may be either uppercase or lowercase; the assembler is case-insensitive.

B.2.3 Shifter Operations

Shifter operations are described in this section. Table B.6 summarizes the syntax and opcodes for the shifter operations. The succeeding pages provide detailed descriptions of each operation.

The shifter operates on the register file's 32-bit fixed-point fields (bits 39-8). Two-input shifter operations can take their y-input from the register file or from immediate data provided in the instruction. Either form uses the same opcode. However, the latter case, called an **immediate shift** or **shifter immediate** operation, is allowed only with instruction type 6, which has an immediate data field in its opcode for this purpose. All other instruction types must obtain the y-input from the register file when the compute operation is a two-input shifter operation.

Syntax	*Opcode*
Rn = LSHIFT Rx BY Ry | <data8>	0000 0000
Rn = Rn OR LSHIFT Rx BY Ry | <data8>	0010 0000
Rn = ASHIFT Rx BY Ry | <data8>	0000 0100
Rn = Rn OR ASHIFT Rx BY Ry | <data8>	0010 0100
Rn = ROT Rx BY RY | <data8>	0000 1000
Rn = BCLR Rx BY Ry | <data8>	1100 0100
Rn = BSET Rx BY Ry | <data8>	1100 0000
Rn = BTGL Rx BY Ry | <data8>	1100 1000
BTST Rx BY Ry | <data8>	1100 1100
Rn = FDEP Rx BY Ry | <bit6>:<len6>	0100 0100
Rn = Rn OR FDEP Rx BY Ry | <bit6>:<len6>	0110 0100
Rn = FDEP Rx BY Ry | <bit6>:<len6> (SE)	0100 1100
Rn = Rn OR FDEP Rx BY Ry | <bit6>:<len6> (SE)	0110 1100
Rn = FEXT Rx BY Ry | <bit6>:<len6>	0100 0000
Rn = FEXT Rx BY Ry | <bit6>:<len6> (SE)	0100 1000
Rn = EXP Rx	1000 0000
Rn = EXP Rx (EX)	1000 0100
Rn = LEFTZ Rx	1000 1000
Rn = LEFTO Rx	1000 1100
Rn = FPACK Fx	1001 0000
Fn = FUNPACK Rx	1001 0100

Instruction modifiers:
(SE) Sign extension of deposited or extracted field
(EX) Extended exponent extract

Table B.6 Shifter Operations

Rn = LSHIFT Rx BY Ry|<data8>

Syntax:

Rn = LSHIFT Rx BY Ry
Rn = LSHIFT Rx BY <data8>

Function:
Logically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are twos-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status flags:
SZ Is set if the shifted result is zero, otherwise cleared
SV Is set if the input is shifted to the left by more than 0, otherwise cleared
SS Is cleared

Rn = Rn OR LSHIFT Rx BY Ry|<data8>

Syntax:

Rn = Rn OR LSHIFT Rx BY Ry
Rn = Rn OR LSHIFT Rx BY <data8>

Function:
Logically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is logically ORed with the fixed-point field of register Rn and then written back to register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are twos-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status flags:
SZ Is set if the shifted result is zero, otherwise cleared
SV Is set if the input is shifted left by more than 0, otherwise cleared
SS Is cleared

Rn = ASHIFT Rx BY Ry|<data8>

Syntax:

Rn = ASHIFT Rx BY Ry
Rn = ASHIFT Rx BY <data8>

Function:
Arithmetically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are twos-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status flags:
SZ Is set if the shifted result is zero , otherwise cleared
SV Is set if the input is shifted left by more than 0, otherwise cleared
SS Is cleared

Rn = Rn OR ASHIFT Rx BY Ry|<data8>

Syntax:

Rn = Rn OR ASHIFT Rx BY Ry
Rn = Rn OR ASHIFT Rx BY <data8>

Function:
Arithmetically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is logically ORed with the fixed-point field of register Rn and then written back to register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are twos-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status flags:
SZ Is set if the shifted result is zero, otherwise cleared
SV Is set if the input is shifted left by more than 0, otherwise cleared
SS Is cleared

Rn = ROT Rx BY Ry|<data8>

Syntax:

Rn = ROT Rx BY Ry
Rn = ROT Rx BY <data8>

Function:
Rotates the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The rotated result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are twos-complement numbers. Positive values select a rotate left; negative values select a rotate right. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a rotate of a 32-bit field from full right wrap around to full left wrap around.

Status flags:
SZ Is set if the rotated result is zero, otherwise cleared
SV Is cleared
SS Is cleared

Rn = BCLR Rx BY Ry|<data8>

Syntax:

Rn = BCLR Rx BY Ry
Rn = BCLR Rx BY <data8>

Function:
Clears a bit in the fixed-point operand in register Rx. The result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be cleared. If the bit position value is greater than 31 or less than 0, no bits are cleared.

Status flags:
SZ Is set if the output operand is 0, otherwise cleared
SV Is set if the bit position is greater than 31, otherwise cleared
SS Is cleared

Note: This compute operation affects a bit in a register file location. There is also a bit manipulation *instruction* that affects one or more bits in a system register. This BIT CLR instruction should not be confused with the BCLR shifter operation. See Appendix E for more information on BIT CLR.

Rn = BSET Rx BY Ry|<data8>

Syntax:

Rn = BSET Rx BY Ry
Rn = BSET Rx BY <data8>

Function:
Sets a bit in the fixed-point operand in register Rx. The result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be set. If the bit position value is greater than 31 or less than 0, no bits are set.

Status flags:
SZ Is set if the output operand is 0, otherwise cleared
SV Is set if the bit position is greater than 31, otherwise cleared
SS Is cleared

Note: This compute operation affects a bit in a register file location. There is also a bit manipulation *instruction* that affects one or more bits in a system register. This BIT SET instruction should not be confused with the BSET shifter operation. See Appendix E for more information on BIT SET.

Rn = BTGL Rx BY Ry|<data8>

Syntax:

Rn = BTGL Rx BY Ry
Rn = BTGL Rx BY <data8>

Function:
Toggles a bit in the fixed-point operand in register Rx. The result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be toggled. If the bit position value is greater than 31 or less than 0, no bits are toggled.

Status flags:
SZ Is set if the output operand is 0, otherwise cleared
SV Is set if the bit position is greater than 31, otherwise cleared
SS Is cleared

Note: This compute operation affects a bit in a register file location. There is also a bit manipulation *instruction* that affects one or more bits in a system register. This BIT TGL instruction should not be confused with the BTGL shifter operation. See Appendix E for more information on BIT TGL.

BTST Rx BY Ry|<data8>

Syntax:

BTST Rx BY Ry
BTST Rx BY <data8>

Function:
Tests a bit in the fixed-point operand in register Rx. The SZ flag is set if the bit is a 0 and cleared if the bit is a 1. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be tested. If the bit position value is greater than 31 or less than 0, no bits are tested.

Status flags:
SZ Is cleared if the tested bit is a 1, is set if the tested bit is a 0 or if the bit position is greater than 31
SV Is set if the bit position is greater than 31, otherwise cleared
SS Is cleared

Note: This compute operation tests a bit in a register file location. There is also a bit manipulation *instruction* that tests one or more bits in a system register. This BIT TST instruction should not be confused with the BTST shifter operation. See Appendix E for more information on BIT TST.

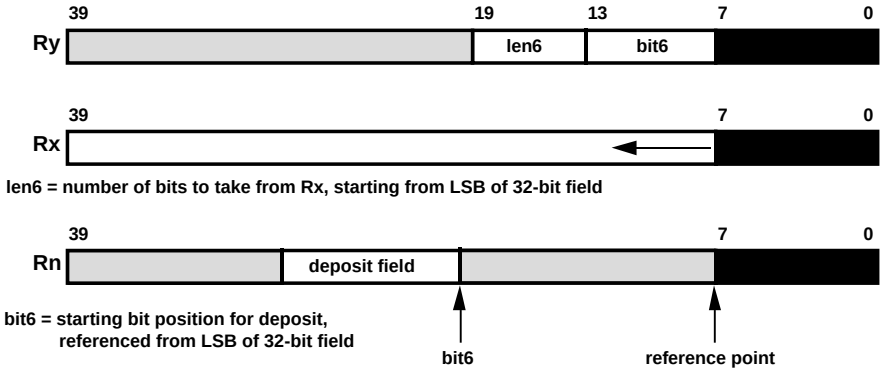
Rn = FDEP Rx BY Ry|<bit6>:<len6>

Syntax:

Rn = FDEP Rx BY Ry
Rn = FDEP Rx BY <bit6>:<len6>

Function:

Deposits a field from register Rx to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bits to the left and to the right of the deposited field are set to 0. The floating-pt. extension field of Rn (bits 7-0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits, and to bit positions ranging from 0 to off-scale left.



Example: If len6=14 and bit6=13, then the 14 bits of Rx are deposited in Rn bits 34-21 (of the 40-bit word).

```
39       31       23       15       7        0
|--------|--------|--abcdef|ghijklmn|--------|        Rx
                     \-------------/
                          14 bits

39       31       23       15       7        0
|00000abc|defghijk|lmn00000|00000000|00000000|        Rn
      \--------------/
                     |
                   bit position 13 (from reference point)
```

Status flags:

SZ Is set if the output operand is 0, otherwise cleared
SV Is set if any bits are deposited to the left of the 32-bit fixed-point output field (i.e., if len6 + bit6 > 32), otherwise cleared
SS Is cleared

Rn = Rn OR FDEP Rx BY Ry|<bit6>:<len6>

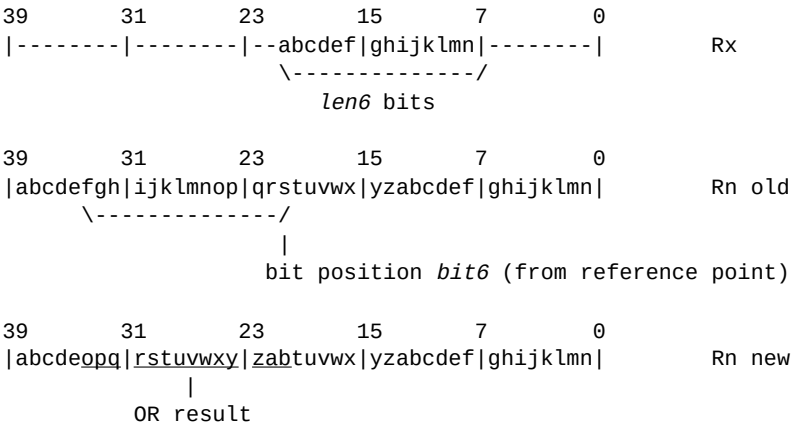
Syntax:

Rn = Rn OR FDEP Rx BY Ry
Rn = Rn OR FDEP Rx BY <bit6>:<len6>

Function:

Deposits a field from register Rx to register Rn. The field value is logically ORed bitwise with the specified field of register Rn and the new value is written back to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits, and to bit positions ranging from 0 to off-scale left.

Example:



Status flags:

SZ Is set if the output operand is 0, otherwise cleared
SV Is set if any bits are deposited to the left of the 32-bit fixed-point output field (i.e., if len6 + bit6 > 32), otherwise cleared
SS Is cleared

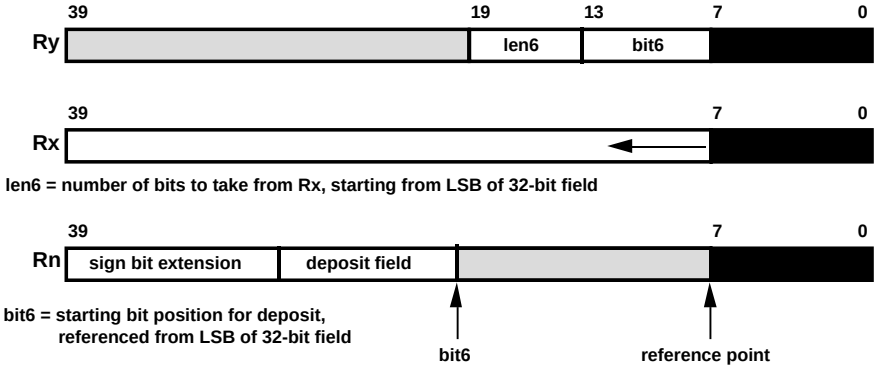
Rn = FDEP Rx BY Ry|<bit6>:<len6> (SE)

Syntax:

Rn = FDEP Rx BY Ry (SE)
Rn = FDEP Rx BY <bit6>:<len6> (SE)

Function:

Deposits and sign-extends a field from register Rx to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. The MSBs of Rn are sign-extended by the MSB of the deposited field, unless the MSB of the deposited field is off-scale left. Bits to the right of the deposited field are set to 0. The floating-point extension field of Rn (bits 7-0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits into bit positions ranging from 0 to off-scale left.



Example:

```
39       31       23       15       7        0
|--------|--------|--abcdef|ghijklmn|--------|        Rx
                     \---------------/
                        len6 bits

39       31       23       15       7        0
|aaaaaabc|defghijk|lmn00000|00000000|00000000|        Rn
\----/\--------------/
sign                |
extension           bit position bit6 (from reference point)
```

Status flags:

SZ Is set if the output operand is 0, otherwise cleared
SV Is set if any bits are deposited to the left of the 32-bit fixed-point output field (i.e., if len6 + bit6 > 32), otherwise cleared
SS Is cleared

Rn = Rn OR FDEP Rx BY Ry|<bit6>:<len6> (SE)

Syntax:

Rn = Rn OR FDEP Rx BY Ry (SE)
Rn = Rn OR FDEP Rx BY <bit6>:<len6> (SE)

Function:

Deposits and sign-extends a field from register Rx to register Rn. The sign-extended field value is logically ORed bitwise with the value of register Rn and the new value is written back to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits into bit positions ranging from 0 to off-scale left.

Example:

```
39       31       23       15       7        0
|--------|--------|--abcdef|ghijklmn|--------|        Rx
                     \-------------/
                        len6 bits

39       31       23       15       7        0
|aaaaaabc|defghijk|lmn00000|00000000|00000000|
\----/\--------------/
sign                |
extension           bit position bit6 (from reference point)

39       31       23       15       7        0
|abcdefgh|ijklmnop|qrstuvwx|yzabcdef|ghijklmn|        Rn old

39       31       23       15       7        0
|vwxyzabc|defghijk|lmntuvwx|yzabcdef|ghijklmn|        Rn new
          |
      OR result
```

Status flags:

SZ Is set if the output operand is 0, otherwise cleared
SV Is set if any bits are deposited to the left of the 32-bit fixed-point output field (i.e., if len6 + bit6 > 32), otherwise cleared
SS Is cleared

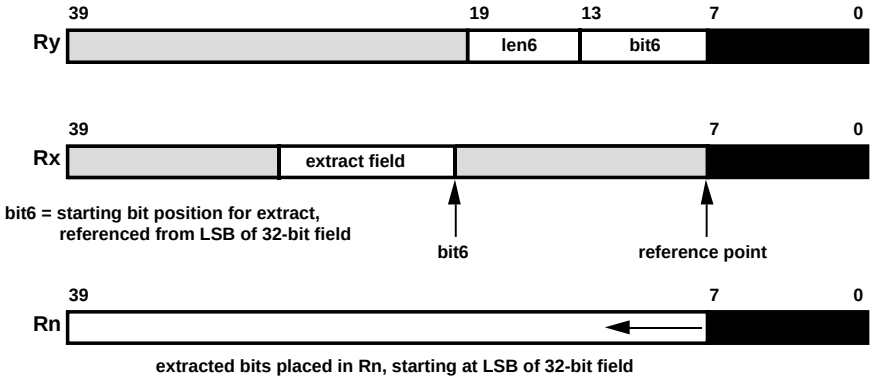
Rn = FEXT Rx BY Ry|<bit6>:<len6>

Syntax:

Rn = FEXT Rx BY Ry
Rn = FEXT Rx BY <bit6>:<len6>

Function:

Extracts a field from register Rx to register Rn. The output field is placed right-aligned in the fixed-point field of Rn. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is extracted from the fixed-point field of Rx starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bits to the left of the extracted field are set to 0 in register Rn. The floating-point extension field of Rn (bits 7-0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for extraction of fields ranging in length from 0 to 32 bits, and from bit positions ranging from 0 to off-scale left.



Example:

```
39       31       23       15       7        0
|-----abc|defghijk|lmn-----|--------|--------|        Rx
      \--------------/
         len6 bits   |
                   bit position bit6 (from reference point)

39       31       23       15       7        0
|00000000|00000000|00abcdef|ghijklmn|00000000|        Rn
```

Status flags:

SZ Is set if the output operand is 0, otherwise cleared
SV Is set if any bits are extracted from the left of the 32-bit fixed-point, input field (i.e., if len6 + bit6 > 32), otherwise cleared
SS Is cleared

Rn = FEXT Rx BY Ry|<bit6>:<len6> (SE)

Syntax:

Rn = FEXT Rx BY Ry (SE)
Rn = FEXT Rx BY <bit6>:<len6> (SE)

Function:

Extracts and sign-extends a field from register Rx to register Rn. The output field is placed right-aligned in the fixed-point field of Rn. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is extracted from the fixed-point field of Rx starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. The MSBs of Rn are sign-extended by the MSB of the extracted field, unless the MSB is extracted from off-scale left. The floating-point extension field of Rn(bits 7-0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for extraction of fields ranging in length from 0 to 32 bits and from bit positions ranging from 0 to off-scale left.

Example:

```
39       31       23       15       7        0
|-----abc|defghijk|lmn-----|--------|--------|        Rx
      \--------------/
         len6 bits   |
                     bit position bit6 (from reference point)

39       31       23       15       7        0
|aaaaaaaa|aaaaaaaa|aaabcdef|ghijklmn|00000000|        Rn
\-------------------/
    sign extension
```

Status flags:

SZ Is set if the output operand is 0, otherwise cleared
SV Is set if any bits are extracted from the left of the 32-bit fixed-point input field (i.e., if len6 + bit6 > 32), otherwise cleared
SS Is cleared

Rn = EXP Rx

Syntax:

Rn = EXP Rx

Function:
Extracts the exponent of the fixed-point operand in Rx. The exponent is placed in the shf8 field in register Rn. The exponent is calculated as the twos complement of:

leading sign bits in Rx – 1

Status flags:
SZ Is set if the extracted exponent is 0, otherwise cleared
SV Is cleared
SS Is set if the fixed-point operand in Rx is negative (bit 31 is a 1), otherwise cleared

Rn = EXP Rx (EX)

Syntax:

Rn = EXP Rx (EX)

Function:
Extracts the exponent of the fixed-point operand in Rx, assuming that the operand is the result of an ALU operation. The exponent is placed in the shf8 field in register Rn. If the AV status bit is set, a value of +1 is placed in the shf8 field to indicate an extra bit (the ALU overflow bit). If the AV status bit is not set, the exponent is calculated as the twos complement of:

leading sign bits in Rx – 1

Status flags:
SZ Is set if the extracted exponent is 0, otherwise cleared
SV Is cleared
SS Is set if the exclusive OR of the AV status bit and the sign bit (bit 31) of the fixed-point operand in Rx is equal to 1, otherwise cleared

Rn = LEFTZ Rx

Syntax:

Rn = LEFTZ Rx

Function:
Extracts the number of leading 0s from the fixed-point operand in Rx. The extracted number is placed in the bit6 field in Rn.

Status flags:
SZ Is set if the MSB of Rx is 1, otherwise cleared
SV Is set if the result is 32, otherwise cleared
SS Is cleared

Rn = LEFTO Rx

Syntax:

Rn = LEFTO Rx

Function:
Extracts the number of leading 1s from the fixed-point operand in Rx. The extracted number is placed in the bit6 field in Rn.

Status flags:
SZ Is set if the MSB of Rx is 0, otherwise cleared
SV Is set if the result is 32, otherwise cleared
SS Is cleared

Rn = FPACK Fx

Syntax:

Rn = FPACK Fx

Function:
Converts the IEEE 32-bit floating-point value in Fx to a 16-bit floating-point value stored in Rn. The short float data format has an 11-bit mantissa with a four-bit exponent plus sign bit. The 16-bit floating-point numbers reside in the lower 16 bits of the 32-bit floating-point field.

The result of the FPACK operation is as follows:

Condition	*Result*
135 < exp	Largest magnitude representation.
120 < exp ≤ 135	Exponent is MSB of source exponent concatenated with the three LSBs of source exponent. The packed fraction is the rounded upper 11 bits of the source fraction.
109 < exp ≤ 120	Exponent=0. Packed fraction is the upper bits (source exponent – 110) of the source fraction prefixed by zeros and the “hidden” 1. The packed fraction is rounded.
exp < 110	Packed word is all zeros.

exp = source exponent
sign bit remains the same in all cases

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

Status flags:
SZ Is cleared
SV Is set if overflow occurs, cleared otherwise
SS Is cleared

Fx = FUNPACK Rn

Syntax:

Fn = FUNPACK Rx

Function:
Converts the 16-bit floating-point value in Rx to an IEEE 32-bit floating-point value stored in Fx.

The result of the FUNPACK operation is as follows:

Condition	*Result*
0 < exp ≤ 15	Exponent is the 3 LSBs of the source exponent prefixed by the MSB of the source exponent and four copies of the complement of the MSB. The unpacked fraction is the source fraction with 12 zeros appended.
exp = 0	Exponent is (120 – N) where N is the number of leading zeros in the source fraction. The unpacked fraction is the remainder of the source fraction with zeros appended to pad it and the “hidden” 1 stripped away.

exp = source exponent
sign bit remains the same in all cases

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

Status flags:
SZ Is cleared
SV Is cleared
SS Is cleared

B.3 MULTIFUNCTION COMPUTATIONS

Multifunction computations are of three types, each of which has a different format for the 23-bit compute field:

- Dual add / subtract
- Parallel multiplier / ALU
- Parallel multiplier and add / subtract

See "Multifunction Computations" in the *Computation Units* chapter for a summary of the multifunction operations.

Each of the four input operands for multifunction computations are constrained to a different set of four register file locations, as shown below in Figure B.1. For example, the X-input to the ALU can only be R8, R9, R10 or R11. In all other compute operations, the input operands may be any register file locations.

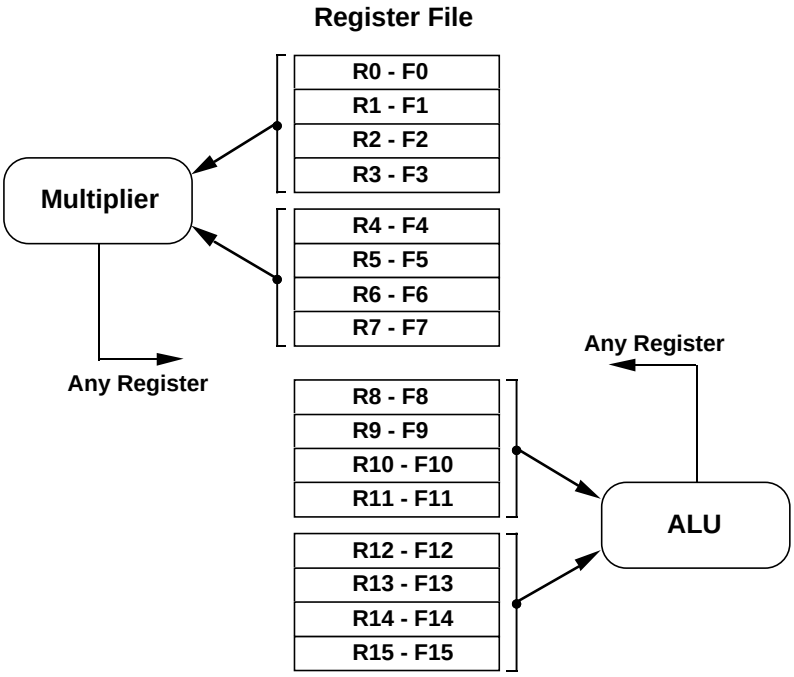


Figure B.1 Allowed Input Registers For Multifunction Computations

Dual Add/Subtract (Fixed-Pt.)

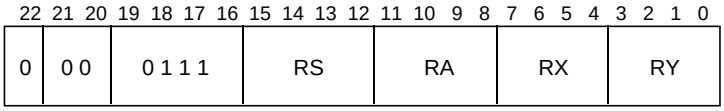
The dual add / subtract operation computes the sum and the difference of two inputs and returns the two results to different registers. There are fixed-point and floating-point versions of this operation.

Fixed-Point:

Syntax:

Ra = Rx + Ry, Rs = Rx – Ry

Compute Field:



Function:
Does a dual add / subtract of the fixed-point fields in registers Rx and Ry. The sum is placed in the fixed-point field of register Ra and the difference in the fixed-point field of Rs. The floating-point extension fields of Ra and Rs are set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status flags:
AZ Is set if either of the fixed-point outputs is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1 of either of the outputs, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages of either of the outputs is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage of either of the outputs is 1, otherwise cleared
AS Is cleared
AI Is cleared

Dual Add/Subtract (Floating-Pt.)

Floating-Point:

Syntax:

Fa = Fx + Fy, Fs = Fx – Fy

Compute Field:

22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0	0 0	1 1 1 1	FS	FA	FX	FY

Function:
Does a dual add / subtract of the floating-point operands in registers Fx and Fy. The normalized results are placed in registers Fa and Fs: the sum in Fa and the difference in Fs. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal returns ±Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if either of the post-rounded results is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU Is set if either post-rounded result is a denormal, otherwise cleared
AN Is set if either of the floating-point results is negative, otherwise cleared
AV Is set if either of the post-rounded results overflows (unbiased exponent > +127), otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, or if both of the input operands are Infinities, otherwise cleared

Parallel Multiplier & ALU (Fixed-Pt.)

The parallel multiplier / ALU operation performs a multiply or multiply / accumulate and one of the following ALU operations: add, subtract, average, fixed-point to floating-point or floating-point to fixed-point conversion, or floating-point ABS, MIN or MAX.

For detailed information about a particular operation, see the individual descriptions under Single-Function Operations.

Fixed-Point:

Syntax: See Table B.7

Compute Field:

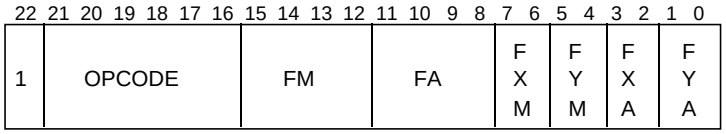
22	21 20 19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2	1 0
1	OPCODE	RM	RA	RXM	RYM	RXA	RYA

Parallel Multiplier & ALU (Floating-Pt.)

Floating-Point:

Syntax: See Table B.7

Compute Field:



The multiplier and ALU operations are determined by OPCODE. The selections for the 6-bit OPCODE field are listed in Table B.7. The multiplier x- and y-operands are received from data registers RXM (FXM) and RYM (FYM). The multiplier result operand is returned to data register RM (FM). The ALU x- and y-operands are received from data registers RXA (FXA) and RYA (FYA). The ALU result operand is returned to data register RA (FA).

The result operands can be returned to any registers within the register file. Each of the four input operands is restricted to a particular set of four data registers.

Input	*Allowed Sources*
Multiplier X:	R3-R0 (F3-F0)
Multiplier Y:	R7-R4 (F7-F4)
ALU X:	R11-R8 (F11-F8)
ALU Y:	R15-R12 (F15-F12)

Syntax	*Opcode*
Rm=R3-0 * R7-4 (SSFR), Ra=R11-8 + R15-12	000100
Rm=R3-0 * R7-4 (SSFR), Ra=R11-8 – R15-12	000101
Rm=R3-0 * R7-4 (SSFR), Ra=(R11-8 + R15-12) / 2	000110
MRF=MRF + R3-0 * R7-4 (SSF), Ra=R11-8 + R15-12	001000
MRF=MRF + R3-0 * R7-4 (SSF), Ra=R11-8 – R15-12	001001
MRF=MRF + R3-0 * R7-4 (SSF), Ra=(R11-8 + R15-12) / 2	001010
Rm=MRF + R3-0 * R7-4 (SSFR), Ra=R11-8 + R15-12	001100
Rm=MRF + R3-0 * R7-4 (SSFR), Ra=R11-8 – R15-12	001101
Rm=MRF + R3-0 * R7-4 (SSFR), Ra=(R11-8 + R15-12) / 2	001110
MRF=MRF – R3-0 * R7-4 (SSF), Ra=R11-8 + R15-12	010000
MRF=MRF – R3-0 * R7-4 (SSF), Ra=R11-8 – R15-12	010001
MRF=MRF – R3-0 * R7-4 (SSF), Ra=(R11-8 + R15-12) / 2	010010
Rm=MRF – R3-0 * R7-4 (SSFR), Ra=R11-8 + R15-12	010100
Rm=MRF – R3-0 * R7-4 (SSFR), Ra=R11-8 – R15-12	010101
Rm=MRF – R3-0 * R7-4 (SSFR), Ra=(R11-8 + R15-12) / 2	010110
Fm=F3-0 * F7-4, Fa=F11-8 + F15-12	011000
Fm=F3-0 * F7-4, Fa=F11-8 – F15-12	011001
Fm=F3-0 * F7-4, Fa=FLOAT R11-8 by R15-12	011010
Fm=F3-0 * F7-4, Fa=FIX F11-8 by R15-12	011011
Fm=F3-0 * F7-4, Fa=(F11-8 + F15-12) / 2	011100
Fm=F3-0 * F7-4, Fa=ABS F11-8	011101
Fm=F3-0 * F7-4, Fa=MAX (F11-8, F15-12)	011110
Fm=F3-0 * F7-4, Fa=MIN (F11-8, F15-12)	011111

Table B.7 Parallel Multiplier/ALU Computations

Parallel Multiplier & Dual Add/Subtract

The parallel multiplier and dual add / subtract operation performs a multiply or multiply / accumulate and computes the sum and the difference of the ALU inputs. For detailed information on the multiplier operations, see the individual descriptions under Single-Function Operations. For information on the dual add / subtract operation, see the Dual Add / Subtract section.

Fixed-Point:

Syntax:

Rm=R3-0 * R7-4 (SSFR), Ra=R11-8 + R15-12, Rs=R11-8 – R15-12

Compute Field:

22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2	1 0
1	1 0	RS	RM	RA	RXM	RYM	RXA	RYA

Floating-Point:

Syntax:

Fm=F3-0 * F7-4, Fa=F11-8 + F15-12, Fs=F11-8 – F15-12

Compute Field:

22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2	1 0
1	1 1	FS	FM	FA	FXM	FYM	FXA	FYA

The multiplier x- and y-operands are received from data registers RXM (FXM) and RYM (FYM). The multiplier result operand is returned to data register RM (FM). The ALU x- and y-operands are received from data registers RXA (FXA) and RYA (FYA). The ALU result operands are returned to data register RA (FA) and RS (FS).

Parallel Multiplier & Dual Add/Subtract

The result operands can be returned to any registers within the register file. Each of the four input operands is restricted to a different set of four data registers.

Input	*Allowed Sources*
Multiplier X:	R3-R0 (F3-F0)
Multiplier Y:	R7-R4 (F7-F4)
ALU X:	R11-R8 (F11-F8)
ALU Y:	R15-R12 (F15-F12)

B Compute Operations

Numeric Formats C

C.1 OVERVIEW

The ADSP-2106x supports the 32-bit single-precision floating-point data format defined in the IEEE Standard 754/854. In addition, the ADSP-2106x supports an extended-precision version of the same format with eight additional bits in the mantissa (40 bits total). The ADSP-2106x also supports 32-bit fixed-point formats—fractional and integer—which can be signed (twos-complement) or unsigned.

C.2 IEEE SINGLE-PRECISION FLOATING-POINT DATA FORMAT

IEEE Standard 754/854 specifies a 32-bit single-precision floating-point format, shown in Figure C.1. A number in this format consists of a sign bit *s*, a 24-bit significand, and an 8-bit unsigned-magnitude exponent *e*. For normalized numbers, the significand consists of a 23-bit fraction *f* and a "hidden" bit of 1 that is implicitly presumed to precede f_{22} in the significand. The binary point is presumed to lie between this hidden bit and f_{22}. The least significant bit (LSB) of the fraction is f_0; the LSB of the exponent is e_0. The hidden bit effectively increases the precision of the floating-point significand to 24 bits from the 23 bits actually stored in the data format. It also insures that the significand of any number in the IEEE normalized-number format is always greater than or equal to 1 and less than 2.

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

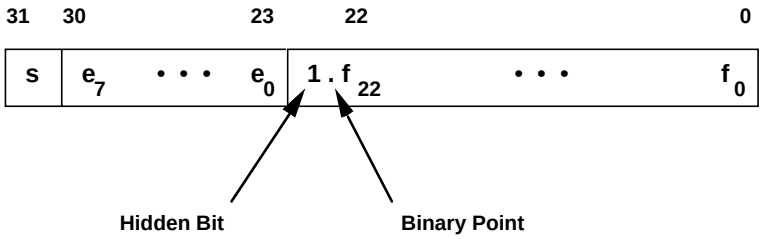


Figure C.1 IEEE 32-Bit Single-Precision Floating-Point Format

C Numeric Formats

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

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

The IEEE single-precision floating-point data types supported by the ADSP-2106x and their interpretations are summarized in Table C.1.

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

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

C.3 EXTENDED PRECISION FLOATING-POINT FORMAT

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

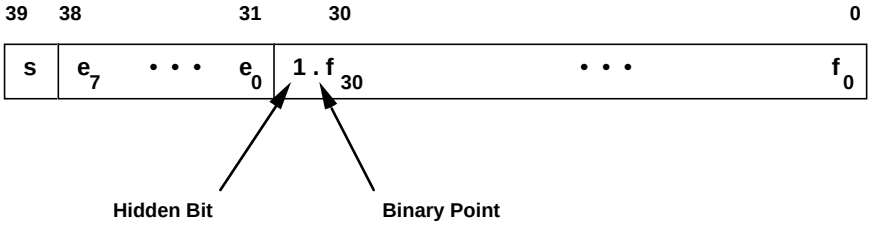


Figure C.2 40-Bit Extended-Precision Floating-Point Format

C.4 SHORT WORD FLOATING-POINT FORMAT

The ADSP-2106x supports a 16-bit floating-point data type and provides conversion instructions for it. The short float data format has an 11-bit mantissa with a four-bit exponent plus sign bit, as shown in Figure C.3. The 16-bit floating-point numbers reside in the lower 16 bits of the 32-bit floating-point field.

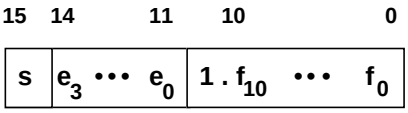


Figure C.3 16-Bit Floating-Point Format

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

The results of the FPACK and FUNPACK operations are as follows:

FPACK

Condition	*Result*
135 < exp	Largest magnitude representation.
120 < exp ≤ 135	Exponent is MSB of source exponent concatenated with the three LSBs of source exponent. The packed fraction is the rounded upper 11 bits of the source fraction.
109 < exp ≤ 120	Exponent=0. Packed fraction is the upper bits (source exponent – 110) of the source fraction prefixed by zeros and the “hidden” 1. The packed fraction is rounded.
exp < 110	Packed word is all zeros.

exp = source exponent
sign bit remains the same in all cases

FUNPACK

Condition	*Result*
$0 < exp \leq 15$	Exponent is the 3 LSBs of the source exponent prefixed by the MSB of the source exponent and four copies of the complement of the MSB. The unpacked fraction is the source fraction with 12 zeros appended.
exp = 0	Exponent is (120 – N) where N is the number of leading zeros in the source fraction. The unpacked fraction is the remainder of the source fraction with zeros appended to pad it and the "hidden" 1 stripped away.

exp = source exponent
sign bit remains the same in all cases

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

During the FPACK operation, an overflow will set the SV condition and non-overflow will clear it. During the FUNPACK operation, the SV condition will be cleared. The SZ and SS conditions are cleared by both instructions.

C.5 FIXED-POINT FORMATS

The ADSP-2106x supports two 32-bit fixed-point formats: fractional and integer. In both formats, numbers can be signed (twos-complement) or unsigned. The four possible combinations are shown in Figure C.4. In the fractional format, there is an implied binary point to the left of the most significant magnitude bit. In integer format, the binary point is understood to be to the right of the LSB. Note that the sign bit is negatively weighted in a twos-complement format.

ALU outputs always have the same width and data format as the inputs. The multiplier, however, produces a 64-bit product from two 32-bit inputs. If both operands are unsigned integers, the result is a 64-bit unsigned integer. If both operands are unsigned fractions, the result is a 64-bit unsigned fraction. These formats are shown in Figure C.5.

If one operand is signed and the other unsigned, the result is signed. If both inputs are signed, the result is signed and automatically shifted left one bit. The LSB becomes zero and bit 62 moves into the sign bit position. Normally bit 63 and bit 62 are identical when both operands are signed. (The only exception is full-scale negative multiplied by itself.) Thus, the left shift normally removes a redundant sign bit, increasing the precision of the most significant product. Also, if the data format is fractional, a single-bit left shift renormalizes the MSP to a fractional format. The signed formats with and without left shifting are shown in Figure C.6.

The multiplier has an 80-bit accumulator to allow the accumulation of 64-bit products. The multiplier and accumulator are described in detail in the *Computation Units* chapter.

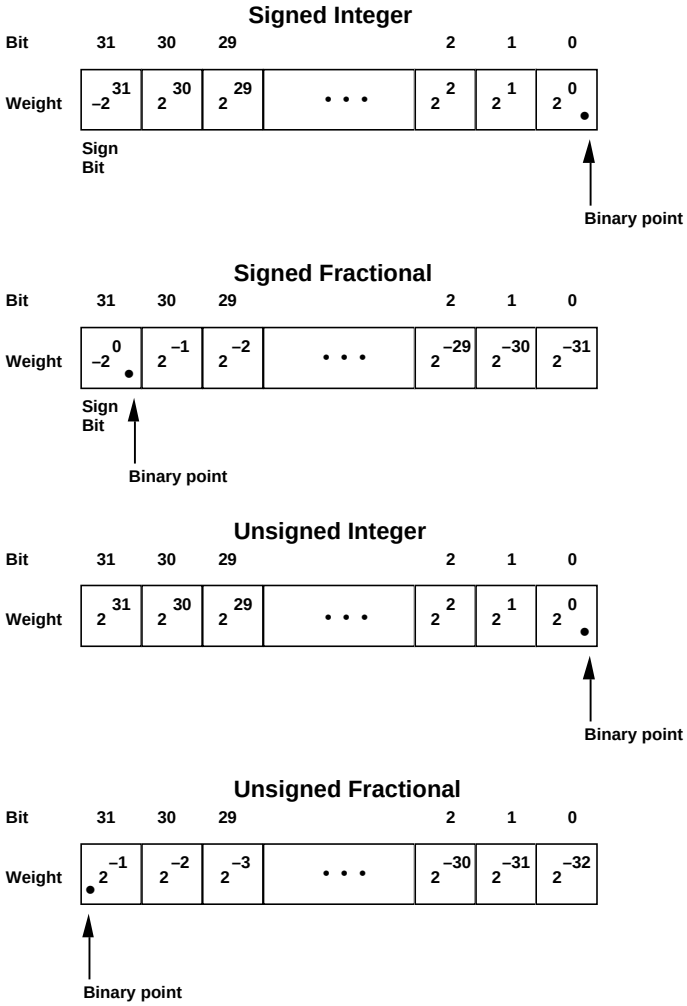


Figure C.4 32-Bit Fixed-Point Formats

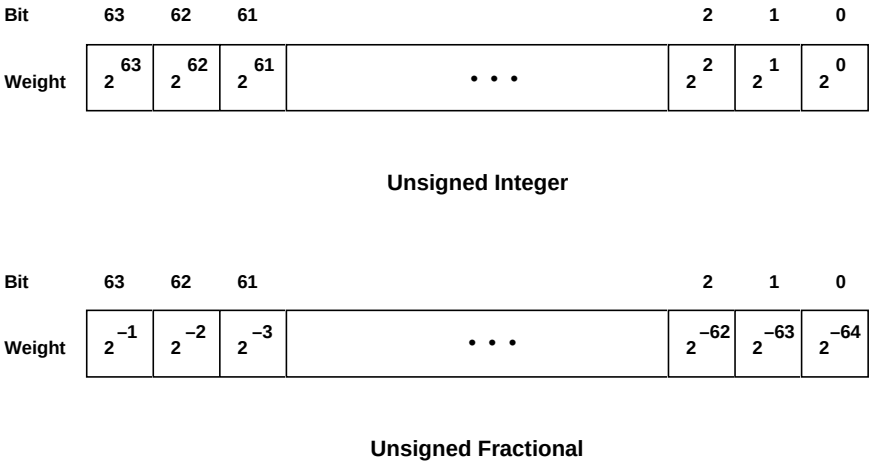


Figure C.5 64-Bit Unsigned Fixed-Point Product

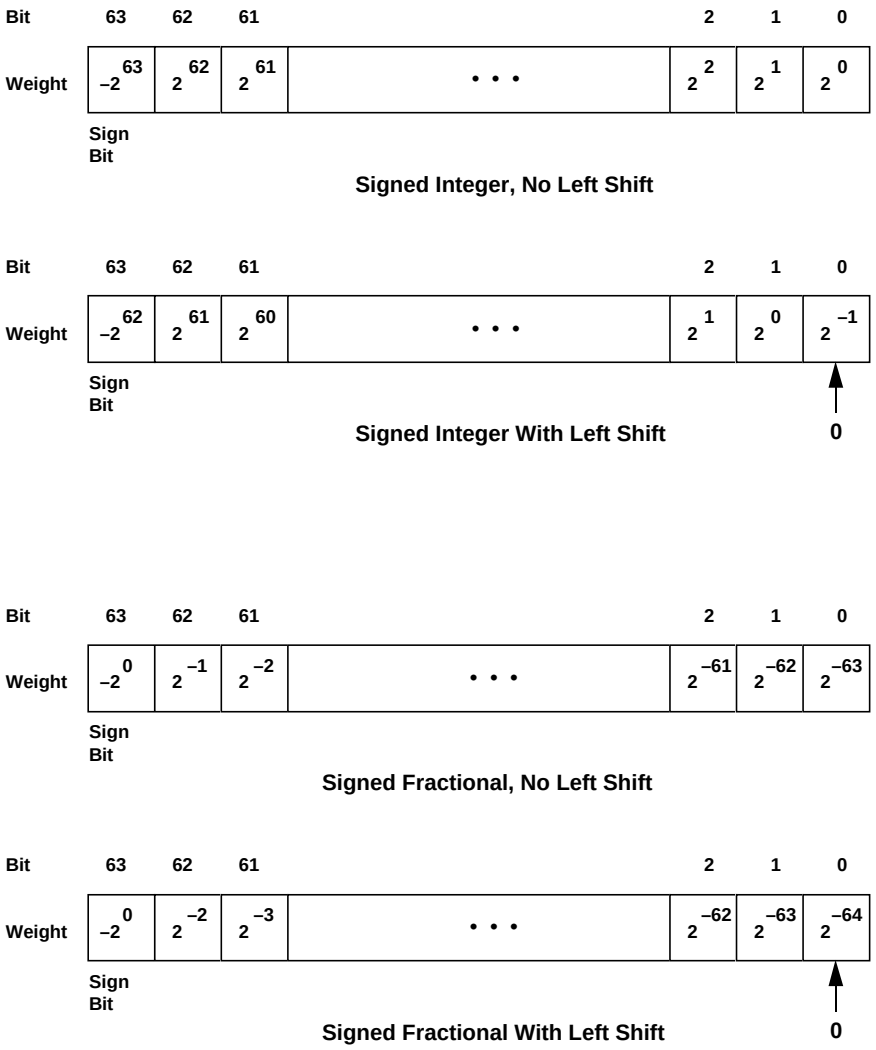


Figure C.6 64-Bit Signed Fixed-Point Product

JTAG Test Access Port D

D.1 OVERVIEW

A boundary scan allows a system designer to test interconnections on a printed circuit board with minimal test-specific hardware. The scan is made possible by the ability to control and monitor each input and output pin on each chip through a set of serially scannable latches. Each input and output is connected to a latch, and the latches are connected as a long shift register so that data can be read from or written to them through a serial test access port (TAP). The ADSP-2106x contains a test access port compatible with the industry-standard IEEE 1149.1 (JTAG) specification.

Only the IEEE 1149.1 features specific to the ADSP-2106x are described here. For more information, see the IEEE 1149.1 specification and the references listed at the end of this appendix.

The boundary scan allows a variety of functions to be performed on each input and output signal of the ADSP-2106x. Each input has a latch that monitors the value of the incoming signal and can also drive data into the chip in place of the incoming value. Similarly, each output has a latch that monitors the outgoing signal and can also drive the output in place of the outgoing value. For bidirectional pins, the combination of input and output functions is available.

Every latch associated with a pin is part of a single serial shift register path. Each latch is a master/slave type latch with the controlling clock provided externally. This clock (TCK) is asynchronous to the ADSP-2106x system clock (CLKIN).

D.2 TEST ACCESS PORT

The test access port (TAP) of the ADSP-2106x controls the operation of the boundary scan. The TAP consists of five pins that control a state machine, including the boundary scan. The state machine and pins conform to the IEEE 1149.1 specification.

TCK (input)	**Test Clock**. Used to clock serial data into scan latches and control sequencing of the test state machine. TCK can be asynchronous with CLKIN.
TMS (input)	**Test Mode Select**. Primary control signal for the state machine. Synchronous with TCK. A sequence of values on TMS adjusts the current state of the TAP.
TDI (input)	**Test Data Input**. Serial input data to the scan latches. Synchronous with TCK.
TDO (output)	**Test Data Output**. Serial output data from the scan latches. Synchronous with TCK.
$\overline{\text{TRST}}$ (input)	**Test Reset**. Resets the test state machine. Can be asynchronous with TCK.

A BSDL file for the ADSP-2106x is available on Analog Devices' BBS and Internet ftp site. The BBS can be reached at:

(617) 461-4258	8 data bits, no parity, 1 stop bit, 300 / 1200 / 2400 / 9600 / 14400 baud

To connect to the ftp site, login as anonymous using your email address for your password and type (from the Unix prompt):

ftp ftp.analog.com (*or* ftp 137.71.23.11)

D.3 INSTRUCTION REGISTER

The instruction register allows an instruction to be shifted into the processor. This instruction selects the test to be performed and/or the test data register to be accessed. The instruction register is 5 bits long with no parity bit. A value of 10000 binary is loaded (LSB nearest TDI) into the instruction register whenever the TAP reset state is entered.

Table D.1 lists the binary code for each instruction. Bit 0 is nearest TDI and bit 4 is nearest TDO. An "x" specifies a "don't-care" state. No data registers are placed into test modes by any of the public instructions. The instructions affect the ADSP-2106x as defined in the 1149.1 specification. The optional instructions RUNBIST, IDCODE and USERCODE are not supported by the ADSP-2106x.

Instruction Bits 4 3 2 1 0	**Instruction Name**	**Register (Serial Path)**	**Type**
1 x x x x	BYPASS	Bypass	Public
0 0 0 0 0	EXTEST	Boundary	Public
0 0 0 0 1	SAMPLE/PRELOAD	Boundary	Public
0 0 0 1 0	*reserved for emulation*	—	*Private*
0 0 0 1 1	INTEST	Boundary	Public
0 0 1 0 0	*reserved for emulation*	—	*Private*
0 0 1 0 1	*reserved for emulation*	—	*Private*
0 0 1 1 0	*reserved for emulation*	—	*Private*
0 0 1 1 1	*reserved for emulation*	—	*Private*
0 1 x x x	*reserved for emulation*	—	*Private*

Table D.1 Test Instructions

The entry under "Register" is the serial scan path, either Boundary or Bypass in this case, enabled by the instruction. Figure D.1 (on the next page) shows these register paths. The 1-bit Bypass register is fully defined in the 1149.1 specification. The Boundary register is described in the next section.

No special values need be written into any register prior to selection of any instruction. As Table D.1 shows, certain instructions are reserved for emulator use. See Section D.7 for more information.

D JTAG Test Access Port

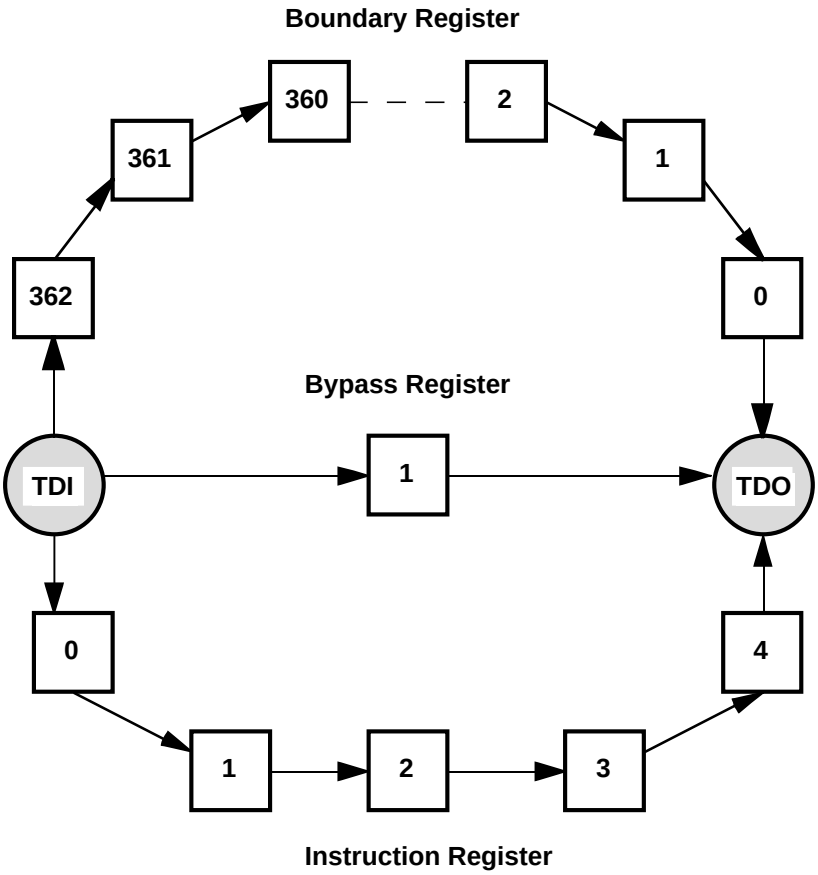


Figure D.1 Serial Scan Paths

D.4 BOUNDARY REGISTER

The Boundary register is 363 bits long. This section defines the latch type and function of each position in the scan path. The positions are numbered with 0 being the first bit output (closest to TDO) and 362 being the last (closest to TDI).

Scan Position	Latch Type	Signal Name	
0	input	$\overline{\text{IRQ0}}$	*this end closest to TDO (scan in first)*
1	input	$\overline{\text{IRQ1}}$	
2	input	$\overline{\text{IRQ2}}$	
3	input	EBOOT	
4	input	$\overline{\text{RESET}}$	
5	input	RPBA	
6	input	LBOOT	
7	input	IDO	
8	input	ID1	
9	input	ID2	
10	output	L5ACK (NC on the ADSP-21061)	
11	input	L5ACK (NC on the ADSP-21061)	
12	output	L5CLK (NC on the ADSP-21061)	
13	input	L5CLK (NC on the ADSP-21061)	
14	output	L5DAT0 (NC on the ADSP-21061)	
15	input	L5DAT0 (NC on the ADSP-21061)	
16	output	L5DAT1 (NC on the ADSP-21061)	
17	input	L5DAT1 (NC on the ADSP-21061)	
18	output	L5DAT2 (NC on the ADSP-21061)	
19	input	L5DAT2 (NC on the ADSP-21061)	
20	output	L5DAT3 (NC on the ADSP-21061)	
21	input	L5DAT3 (NC on the ADSP-21061)	
22	output enable	L5ACK output enable (NC on the ADSP-21061)	
23	output enable	L5DATx, L5CLK output enable (NC on the ADSP-21061)	
24	output	L4ACK (NC on the ADSP-21061)	
25	input	L4ACK (NC on the ADSP-21061)	
26	output	L4CLK (NC on the ADSP-21061)	
27	input	L4CLK (NC on the ADSP-21061)	
28	output	L4DAT0 (NC on the ADSP-21061)	
29	input	L4DAT0 (NC on the ADSP-21061)	
30	output	L4DAT1 (NC on the ADSP-21061)	
31	input	L4DAT1 (NC on the ADSP-21061)	
32	output	L4DAT2 (NC on the ADSP-21061)	
33	input	L4DAT2 (NC on the ADSP-21061)	
34	output	L4DAT3 (NC on the ADSP-21061)	
35	input	L4DAT3 (NC on the ADSP-21061)	
36	output enable	L4ACK output enable (NC on the ADSP-21061)	
37	output enable	L4DATx, L4CLK output enable (NC on the ADSP-21061)	
38	output	L3ACK (NC on the ADSP-21061)	
39	input	L3ACK (NC on the ADSP-21061)	

Output Enables:

1 = Drive the associated signals during the EXTEST and INTEST instructions
0 = Tristate the associated signals during the EXTEST and INTEST instructions

Scan Position	Latch Type	Signal Name
40	output	L3CLK (NC on the ADSP-21061)
41	input	L3CLK (NC on the ADSP-21061)
42	output	L3DAT0 (NC on the ADSP-21061)
43	input	L3DAT0 (NC on the ADSP-21061)
44	output	L3DAT1 (NC on the ADSP-21061)
45	input	L3DAT1 (NC on the ADSP-21061)
46	output	L3DAT2 (NC on the ADSP-21061)
47	input	L3DAT2 (NC on the ADSP-21061)
48	output	L3DAT3 (NC on the ADSP-21061)
49	input	L3DAT3 (NC on the ADSP-21061)
50	output enable	L3ACK output enable (NC on the ADSP-21061)
51	output enable	L3DATx, L3CLK output enable (NC on the ADSP-21061)
52	output	NC (Do Not Connect)
53	input	NC (Do Not Connect)
54	output	L2ACK (NC on the ADSP-21061)
55	input	L2ACK (NC on the ADSP-21061)
56	output	L2CLK (NC on the ADSP-21061)
57	input	L2CLK (NC on the ADSP-21061)
58	output	L2DAT0 (NC on the ADSP-21061)
59	input	L2DAT0 (NC on the ADSP-21061)
60	output	L2DAT1 (NC on the ADSP-21061)
61	input	L2DAT1 (NC on the ADSP-21061)
62	output	L2DAT2 (NC on the ADSP-21061)
63	input	L2DAT2 (NC on the ADSP-21061)
64	output	L2DAT3 (NC on the ADSP-21061)
65	input	L2DAT3 (NC on the ADSP-21061)
66	output enable	L2ACK output enable (NC on the ADSP-21061)
67	output enable	L2DATx, L2CLK output enable (NC on the ADSP-21061)
68	output	L1ACK (NC on the ADSP-21061)
69	input	L1ACK (NC on the ADSP-21061)
70	output	L1CLK (NC on the ADSP-21061)
71	input	L1CLK (NC on the ADSP-21061)
72	output	L1DAT0 (NC on the ADSP-21061)
73	input	L1DAT0 (NC on the ADSP-21061)
74	output	L1DAT1 (NC on the ADSP-21061)
75	input	L1DAT1 (NC on the ADSP-21061)
76	output	L1DAT2 (NC on the ADSP-21061)
77	input	L1DAT2 (NC on the ADSP-21061)
78	output	L1DAT3 (NC on the ADSP-21061)
79	input	L1DAT3 (NC on the ADSP-21061)
80	output enable	L1ACK output enable (NC on the ADSP-21061)
81	output enable	L1DATx, L1CLK output enable (NC on the ADSP-21061)
82	output	L0ACK (NC on the ADSP-21061)
83	input	L0ACK (NC on the ADSP-21061)
84	output	L0CLK (NC on the ADSP-21061)
85	input	L0CLK (NC on the ADSP-21061)
86	output	L0DAT0 (NC on the ADSP-21061)
87	input	L0DAT0 (NC on the ADSP-21061)
88	output	L0DAT1 (NC on the ADSP-21061)
89	input	L0DAT1 (NC on the ADSP-21061)

Scan Position	Latch Type	Signal Name
90	output	L0DAT2 (NC on the ADSP-21061)
91	input	L0DAT2 (NC on the ADSP-21061)
92	output	L0DAT3 (NC on the ADSP-21061)
93	input	L0DAT3 (NC on the ADSP-21061)
94	output enable	L0ACK output enable (NC on the ADSP-21061)
95	output enable	L0DATx, L0CLK output enable (NC on the ADSP-21061)
96	output	DATA0
97	input	DATA0
98	output	DATA1
99	input	DATA1
100	output	DATA2
101	input	DATA2
102	output	DATA3
103	input	DATA3
104	output	DATA4
105	input	DATA4
106	output	DATA5
107	input	DATA5
108	output	DATA6
109	input	DATA6
110	output	DATA7
111	input	DATA7
112	output	DATA8
113	input	DATA8
114	output	DATA9
115	input	DATA9
116	output	DATA10
117	input	DATA10
118	output	DATA11
119	input	DATA11
120	output	DATA12
121	input	DATA12
122	output	DATA13
123	input	DATA13
124	output	DATA14
125	input	DATA14
126	output	DATA15
127	input	DATA15
128	output	DATA16
129	input	DATA16
130	output	DATA17
131	input	DATA17
132	output	DATA18
133	input	DATA18
134	output	DATA19
135	input	DATA19
136	output	DATA20
137	input	DATA20
138	output	DATA21
139	input	DATA21

Scan Position	Latch Type	Signal Name
140	output	DATA22
141	input	DATA22
142	output	DATA23
143	input	DATA23
144	output	DATA24
145	input	DATA24
146	output	DATA25
147	input	DATA25
148	output	DATA26
149	input	DATA26
150	output enable	DATAx output enable
151	output	DATA27
152	input	DATA27
153	output	DATA28
154	input	DATA28
155	output	DATA29
156	input	DATA29
157	output	DATA30
158	input	DATA30
159	output	DATA31
160	input	DATA31
161	output	DATA32
162	input	DATA32
163	output	DATA33
164	input	DATA33
165	output	DATA34
166	input	DATA34
167	output	DATA35
168	input	DATA35
169	output	NC (Do Not Connect)
170	input	NC (Do Not Connect)
171	output	DATA36
172	input	DATA36
173	output	DATA37
174	input	DATA37
175	output	DATA38
176	input	DATA38
177	output	DATA39
178	input	DATA39
179	output	DATA40
180	input	DATA40
181	output	DATA41
182	input	DATA41
183	output	DATA42
184	input	DATA42
185	output	DATA43
186	input	DATA43
187	output	DATA44
188	input	DATA44
189	output	DATA45

Scan Position	Latch Type	Signal Name
190	input	DATA45
191	output	DATA46
192	input	DATA46
193	output	DATA47
194	input	DATA47
195	output enable	$\overline{BR1}$ output enable
196	output enable	$\overline{BR2}$ output enable
197	output enable	$\overline{BR3}$ output enable
198	output	$\overline{BR1}$
199	input	$\overline{BR1}$
200	output	$\overline{BR2}$
201	input	$\overline{BR2}$
202	output	$\overline{BR3}$
203	input	$\overline{BR3}$
204	output	$\overline{BR4}$
205	input	$\overline{BR4}$
206	output	$\overline{BR5}$
207	input	$\overline{BR5}$
208	output	$\overline{BR6}$
209	input	$\overline{BR6}$
210	output enable	$\overline{BR4}$ output enable
211	output enable	$\overline{BR5}$ output enable
212	output enable	$\overline{BR6}$ output enable
213	output	PAGE
214	input	PAGE
215	output	$\overline{DMAG1}$
216	output	$\overline{DMAG2}$
217	output	ACK
218	input	ACK
219	clock*	CLKIN
220	output enable	ACK output enable
221	output enable	$\overline{RD}$, $\overline{WR}$, PAGE, ADRCLK, $\overline{DMAG}$x output enable
222	output	$\overline{WR}$
223	input	$\overline{WR}$
224	output	$\overline{RD}$
225	input	$\overline{RD}$
226	input	$\overline{CS}$
227	output	$\overline{HBG}$
228	input	$\overline{HBG}$
229	output	REDY
230	output	ADRCLK
231	output enable	$\overline{HBG}$ output enable
232	output enable	REDY output enable
233	output enable	RFS0 output enable
234	output enable	RCLK0 output enable
235	output enable	TFS0 output enable
236	output enable	TCLK0 output enable
237	output enable	DT0 output enable
238	output	RFS0
239	input	RFS0

* CLKIN can be sampled but not controlled (read-only). CLKIN continues to clock the ADSP-2106x no matter which instruction is enabled.

Scan Position	Latch Type	Signal Name
240	output	RCLK0
241	input	RCLK0
242	input	DR0
243	output	TFS0
244	input	TFS0
245	output	TCLK0
246	input	TCLK0
247	output	DT0
248	output	$\overline{\text{CPA}}$
249	input	$\overline{\text{CPA}}$
250	output enable	RFS1, $\overline{\text{CPA}}$ output enable
251	output enable	RCLK1 output enable
252	output enable	TFS1 output enable
253	output enable	TCLK1 output enable
254	output enable	DT1 output enable
255	output	RFS1
256	input	RFS1
257	output	RCLK1
258	input	RCLK1
259	input	DR1
260	output	TFS1
261	input	TFS1
262	output	TCLK1
263	input	TCLK1
264	output	DT1
265	input	$\overline{\text{HBR}}$
266	input	$\overline{\text{DMAR1}}$
267	input	$\overline{\text{DMAR2}}$
268	input	$\overline{\text{SBTS}}$
269	output	ADDR31
270	input	ADDR31
271	output	ADDR30
272	input	ADDR30
273	output	ADDR29
274	input	ADDR29
275	output enable	$\overline{\text{BMS}}$ output enable
276	output	ADDR28
277	input	ADDR28
278	output	$\overline{\text{BMS}}$
279	input	$\overline{\text{BMS}}$
280	output	$\overline{\text{SW}}$
281	input	$\overline{\text{SW}}$
282	output	$\overline{\text{MS0}}$
283	input	$\overline{\text{MS0}}$
284	output	$\overline{\text{MS1}}$
285	input	$\overline{\text{MS1}}$
286	output	$\overline{\text{MS2}}$
287	input	$\overline{\text{MS2}}$
288	output	$\overline{\text{MS3}}$
289	input	$\overline{\text{MS3}}$

Scan Position	Latch Type	Signal Name
290	output	ADDR27
291	input	ADDR27
292	output	ADDR26
293	input	ADDR26
294	output	ADDR25
295	input	ADDR25
296	output	ADDR24
297	input	ADDR24
298	output	ADDR23
299	input	ADDR23
300	output	ADDR22
301	input	ADDR22
302	output	ADDR21
303	input	ADDR21
304	output	ADDR20
305	input	ADDR20
306	output	ADDR19
307	input	ADDR19
308	output enable	ADDRx, $\overline{MS}$x, $\overline{SW}$ output enable
309	output	ADDR18
310	input	ADDR18
311	output	ADDR17
312	input	ADDR17
313	output	ADDR16
314	input	ADDR16
315	output	ADDR15
316	input	ADDR15
317	output	ADDR14
318	input	ADDR14
319	output	ADDR13
320	input	ADDR13
321	output	ADDR12
322	input	ADDR12
323	output	ADDR11
324	input	ADDR11
325	output	ADDR10
326	input	ADDR10
327	output	ADDR9
328	input	ADDR9
329	output	ADDR8
330	input	ADDR8
331	output	ADDR7
332	input	ADDR7
333	output	ADDR6
334	input	ADDR6
335	output	ADDR5
336	input	ADDR5
337	output	ADDR4
338	input	ADDR4
339	output	ADDR3

Scan Position	Latch Type	Signal Name	
340	input	ADDR3	
341	output	ADDR2	
342	input	ADDR2	
343	output	ADDR1	
344	input	ADDR1	
345	output	ADDR0	
346	input	ADDR0	
347	output enable	FLAG0 output enable	
348	output enable	FLAG1 output enable	
349	output enable	FLAG2 output enable	
350	output enable	FLAG3 output enable	
351	output	FLAG0	
352	input	FLAG0	
353	output	FLAG1	
354	input	FLAG1	
355	output	FLAG2	
356	input	FLAG2	
357	output	FLAG3	
358	input	FLAG3	
359	output	ICSA	
360	output	$\overline{\text{EMU}}$	
361	output	TIMEXP	
362	output enable	$\overline{\text{EMU}}$ output enable	*this end closest to TDI (scan in last)*

Output Enables:

1 = Drive the associated signals during the EXTEST and INTEST instructions
0 = Tristate the associated signals during the EXTEST and INTEST instructions

D.5 DEVICE IDENTIFICATION REGISTER

No device identification register is included in the ADSP-2106x.

D.6 BUILT-IN SELF-TEST OPERATION (BIST)

No self-test functions are supported by the ADSP-2106x.

D.7 PRIVATE INSTRUCTIONS

Loading a value of 001xx into the instruction register enables the private instructions reserved for emulation. The ADSP-2106x EZ-ICE emulator uses the TAP and boundary scan as a way to access the processor in the target system. The EZ-ICE emulator requires a target board connector for access to the TAP. See "EZ-ICE Emulator" in Chapter 11, *System Design*, for information on this connector.

D.8 REFERENCES

IEEE Standard 1149.1-1990. *Standard Test Access Port and Boundary-Scan Architecture.* To order a copy, contact IEEE at 1-800-678-IEEE.

Maunder, C.M. & R. Tulloss. *Test Access Ports and Boundary Scan Architectures.* IEEE Computer Society Press, 1991.

Parker, Kenneth. *The Boundary Scan Handbook.* Kluwer Academic Press, 1992.

Bleeker, Harry, P. van den Eijnden, & F. de Jong. *Boundary-Scan Test—A Practical Approach.* Kluwer Academic Press, 1993.

Hewlett-Packard Co. *HP Boundary-Scan Tutorial and BSDL Reference Guide.* (HP part# E1017-90001.) 1992.

D JTAG Test Access Port

Control/Status Registers E

E.1 OVERVIEW

This appendix provides bit definitions for the ADSP-2106x's control and status registers. Some of the registers are located in the processor core; these are called *system registers*, a subset of the processor's universal register set. The core processor system registers are MODE1, MODE2, ASTAT, STKY, IRPTL, IMASK, IMASKP, USTAT1, and USTAT2.

The remaining control registers are located in the ADSP-2106x's I/O Processor (IOP). These include the SYSCON and SYSTAT registers. These registers are memory-mapped in ADSP-2106x internal memory.

Register	*Function*	*Initialization After Reset*
MODE1	Mode Control 1	0x0000 (cleared)
MODE2	Mode Control 2	0x*n*000 0000 *
ASTAT	Arithmetic Status	0x00*nn* 0000 **
STKY	Sticky Status	0x0540 0000
IRPTL	Interrupt Latch	0x0000 (cleared)
IMASK	Interrupt Mask	0x0003
IMASKP	Interrupt Mask Pointer	0x0000 (cleared)
USTAT1	User Status 1	0x0000 (cleared)
USTAT2	User Status 2	0x0000 (cleared)

Table E.1 System Registers (Core Processor)

* MODE2 bits 28-31 are the processor ID and silicon revision #.
** ASTAT bits 19-22 are equal to the values of the FLAG0-3 input pins after reset; the flag pins are configured as inputs after reset

Register	*Function*	*Initialization After Reset*
SYSCON	System Configuration	0x0000 0010
SYSTAT	System Status	0x0000 0*nn*0 *

Table E.2 IOP Registers (I/O Processor)

* SYSTAT bits 4-11 depend on the value of the ID_{2-0} inputs.

E Control/Status Registers

All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits must always be written with zeros.*

E.2 SYSTEM REGISTERS (CORE PROCESSOR)

The system registers of the ADSP-2106x core processor are listed in Table E.1. The system registers are a subset of the universal register set. They can be written from an immediate field in an instruction or they can be loaded from or stored to data memory. They can also be transferred to or from any other universal register in one cycle.

E.2.1 Effect Latency & Read Latency

A write to any system register other than USTAT1 or USTAT2 has one cycle of latency before any changes are effective. This delay is called effect latency. Also, if a write to a system register is immediately followed by a read, the value read is always the new one, except for IMASKP which requires an extra cycle before the value is updated. This delay is called read latency.

Effect latency and read latency for the ADSP-2106x system registers are listed below. A "0" indicates that the write takes effect on the cycle immediately after the write instruction is executed, and a "1" indicates one cycle of latency.

Register	*Contents*	*Read Latency*	*Effect Latency*
MODE1	mode control bits	0	1
MODE2	mode control bits	0	1
IRPTL	interrupt latch	0	1
IMASK	interrupt mask	0	1
IMASKP	interrupt mask pointer (for nesting)	1	1
ASTAT	arithmetic status flags	0	1
STKY	sticky status flags	0	1
USTAT1	user-defined status flags	0	0
USTAT2	user-defined status flags	0	0

E.2.2 System Register Bit Operations

The *system register bit manipulation* instruction can be used to set, clear, toggle, or test specific bits in the system registers. An immediate field in the bit manipulation instruction specifies the affected bits. This instruction is described in Appendix A, *Instruction Set Reference, Group IV–Miscellaneous.*

Examples:
```
BIT SET MODE2 0x00000070;
BIT TST ASTAT 0x00002000; {result in BTF flag}
```

Although the shifter and ALU have bit manipulation capabilities, these computations operate on register file locations only. System register bit manipulation instructions eliminate the overhead of transferring system registers to and from the register file:

Bit Instruction (System Registers)	*Shifter Operation (Data Register File)*
BIT SET register data	Rn = BSET Rx BY Ry | data
BIT CLR register data	Rn = BCLR Rx BY Ry | data
BIT TGL register data	Rn = BTGL Rx BY Ry | data
BIT TST register data (result in BTF flag)	BTST Rx BY Ry | data (result in SZ status flag)

E.2.2.1 Bit Test Flag

The test and XOR operations of the system register bit manipulation instruction store the result in the bit test flag (BTF, bit 18 in the ASTAT register). The state of BTF is a condition that you can use in conditional instructions. The test operation sets BTF if all specified bits in the system register are set. The XOR operation sets BTF if all bits in the system register match the specified bit pattern.

E.2.3 User-Defined Status Registers

Two undefined 32-bit status registers, USTAT1 and USTAT2, can be user-defined. Bits in these registers can be set and tested using system register instructions. You can use these registers for low-overhead, general-purpose software flags or for temporary storage of data.

E Control/Status Registers

E.3 IOP REGISTERS (I/O PROCESSOR)

The ADSP-2106x's I/O Processor (IOP) registers are a separate set of memory-mapped control and data registers. The IOP registers are used to configure system-level functions including serial port I/O, link port I/O, and DMA transfers. I/O operations are handled by the ADSP-2106x's on-chip I/O processor, independently from and transparent to the processor core.

The IOP registers are programmed by writing to the appropriate address in memory. They can be programmed by the code executing on the ADSP-2106x core *or* by an external device such as a host processor or another ADSP-2106x. The symbolic names of the registers and individual bits can be used in ADSP-2106x programs—the `#define` definitions for these symbols are contained in the file `def21060.h` which is provided in the INCLUDE directory of the ADSP-21000 Family Development Software. The `def21060.h` file is shown at the end of this appendix.

E.3.1 IOP Registers Summary

Tables E.3, E.4, E.5, and E.6 list the IOP registers used for processor and system control, DMA operations, link port operations, and serial port operations. Table E.7 (on pages 10-13) shows the memory-mapped address, functional group, and reset initialization value of each IOP register.

The memory-mapped IOP registers can be accessed by any external device that is the bus master, either another ADSP-2106x or a host processor. This allows, for example, an external device to set up a DMA transfer to the ADSP-2106x's internal memory without the ADSP-2106x's intervention. A conflict occurs if both the ADSP-2106x core processor and the external bus master try to access the same IOP register group at the same time. In this case, the following rule applies:

> *When both the ADSP-2106x core processor and the external bus master simultaneously attempt to access the same group of IOP registers, the external device always has priority. The ADSP-2106x core will be forced to wait until the external device is finished.*

Table E.7 shows the different IOP register groups.

The IOP registers are arranged to allow a host processor (or other bus master) to easily access the most important registers by reading or writing to the smallest amount of memory. The host only needs to control a small number of address lines to access a set of 16, 32, or 64 IOP registers including SYSCON, SYSTAT, VIRPT, WAIT, MSGR0–MSGR7, and one or two full DMA channels.

Register Name	*Width*	*Description*
SYSCON	32	System Configuration Register
SYSTAT	32	System Status Register
WAIT	32	Memory Wait State Configuration Register
VIRPT	32	Multiprocessor Vector Interrupt Register
MSGR0	32	Message Register 0
MSGR1	32	Message Register 1
MSGR2	32	Message Register 2
MSGR3	32	Message Register 3
MSGR4	32	Message Register 4
MSGR5	32	Message Register 5
MSGR6	32	Message Register 6
MSGR7	32	Message Register 7
BMAX	16	Bus Timeout Maximum
BCNT	16	Bus Timeout Counter
ELAST	32	Address of last external access in memory bank 0 (used for detection of PAGE changes)

Table E.3 IOP Registers (System Control)

Register Name(s)	*Width*	*Description*
EPB0	48	External Port FIFO Buffer 0
EPB1	48	External Port FIFO Buffer 1
EPB2	48	External Port FIFO Buffer 2
EPB3	48	External Port FIFO Buffer 3
DMAC6	16	DMA Channel 6 Control Register (Ext. Port Buffer 0 or Link Buffer 4)[1, 2]
DMAC7	16	DMA Channel 7 Control Register (Ext. Port Buffer 1 or Link Buffer 5)[1, 2]
DMAC8	16	DMA Channel 8 Control Register (Ext. Port Buffer 2)[3]
DMAC9	16	DMA Channel 9 Control Register (Ext. Port Buffer 3)[3]
DMASTAT	32	DMA Channel Status Register
II0, IM0, C0, CP0 GP0, DB0, DA0	16-18	DMA Channel 0 Parameter Registers (SPORT0 Receive)[4]
II1, IM1, C1, CP1 GP1, DB1, DA1	16-18	DMA Channel 1 Parameter Registers (SPORT1 Receive or Link Buffer 0)[1, 2, 4, 5]
II2, IM2, C2, CP2 GP2, DB2, DA2	16-18	DMA Channel 2 Parameter Registers (SPORT0 Transmit)[4, 5]
II3, IM3, C3, CP3 GP3, DB3, DA3	16-18	DMA Channel 3 Parameter Registers (SPORT1 Transmit or Link Buffer 1)[1, 2, 4, 5]
II4, IM4, C4, CP4 GP4, DB4, DA4	16-18	DMA Channel 4 Parameter Registers (Link Buffer 2)[1, 5]
II5, IM5, C5, CP5 GP5, DB5, DA5	16-18	DMA Channel 5 Parameter Registers (Link Buffer 3)[1, 5]
II6, IM6, C6, CP6 GP6, EI6, EM6, EC6	16-32	DMA Channel 6 Parameter Registers (Ext. Port Buffer 0 or Link Buffer 4)[1, 2]
II7, IM7, C7, CP7 GP7, EI7, EM7, EC7	16-32	DMA Channel 7 Parameter Registers (Ext. Port Buffer 1 or Link Buffer 5)[1, 2]
II8, IM8, C8, CP8 GP8, EI8, EM8, EC8	16-32	DMA Channel 8 Parameter Registers (Ext. Port Buffer 2)[3]
II9, IM9, C9, CP9 GP9, EI9, EM9, EC9	16-32	DMA Channel 9 Parameter Registers (Ext. Port Buffer 3)[3]

Table E.4 IOP Registers (DMA)

1. DMA control, buffer, and parameter registers associated with the link ports are not applicable to the ADSP-21061.
2. There are no shared DMA channels on the ADPS-21061.
3. DMA control, buffer, and parameter registers associated wiht DMA channels 8 and 9 are not applicalbe to the ADSP-21061.
4. The IM0, IM1, IM2, and IM3 registers contain the fixed value of 1 on the ADSP-21061.
5. The DBx and DAx registers are not available on the ADSP-21061 because there is no 2-D DMA on the ADSP-21061.

Register Name	*Width*	*Description*
LBUF0	48/32	Link Data Buffer 0
LBUF1	48/32	Link Data Buffer 1
LBUF2	48/32	Link Data Buffer 2
LBUF3	48/32	Link Data Buffer 3
LBUF4	48/32	Link Data Buffer 4
LBUF5	48/32	Link Data Buffer 5
LCTL	32	Link Buffer Control Register
LCOM	32	Link Common Control Register
LAR	18	Link Assignment Register
LSRQ	32	Link Service Request & Mask Register
LPATH1	32	Link Path 1 Register (Mesh Multiprocessing)
LPATH2	32	Link Path 2 Register (Mesh Multiprocessing)
LPATH3	32	Link Path 3 Register (Mesh Multiprocessing)
LPCNT	32	Link Path Counter (Mesh Multiprocessing)
CNST1	40/32	Link Port Constant 1 (Mesh Multiprocessing)
CNST2	40/32	Link Port Constant 2 (Mesh Multiprocessing)

Table E.5 IOP Registers (Link Ports)

Note: The IOP registers that support the link ports are not available on the ADSP-21061.

Register Name	*Width*	*Description*
STCTL0	32	SPORT0 Transmit Control Register
SRCTL0	32	SPORT0 Receive Control Register
TX0	32	SPORT0 Transmit Data Buffer
RX0	32	SPORT0 Receive Data Buffer
TDIV0	32	SPORT0 Transmit Divisors
RDIV0	32	SPORT0 Receive Divisors
MTCS0	32	SPORT0 Multichannel Transmit Selector
MRCS0	32	SPORT0 Multichannel Receive Selector
MTCCS0	32	SPORT0 Multichannel Transmit Compand Selector
MRCCS0	32	SPORT0 Multichannel Receive Compand Selector
SPATH0	16	SPORT0 Path Length (Mesh Multiprocessing)[1]
KEYWD0	32	SPORT0 Receive Comparison[2]
KEYMASK0	32	SPORT0 Receive Comparison Mask[2]
STCTL1	32	SPORT1 Transmit Control Register
SRCTL1	32	SPORT1 Receive Control Register
TX1	32	SPORT1 Transmit Data Buffer
RX1	32	SPORT1 Receive Data Buffer
TDIV1	32	SPORT1 Transmit Divisors
RDIV1	32	SPORT1 Receive Divisors
MTCS1	32	SPORT1 Multichannel Transmit Selector
MRCS1	32	SPORT1 Multichannel Receive Selector
MTCCS1	32	SPORT1 Multichannel Transmit Compand Selector
MRCCS1	32	SPORT1 Multichannel Receive Compand Selector
SPATH1	16	SPORT1 Path Length (Mesh Multiprocessing)[1]
KEYWD1	32	SPORT1 Receive Comparison[2]
KEYMASK1	32	SPORT1 Receive Comparison Mask[2]

Table E.6 IOP Registers (Serial Ports)

1. Not available on the ADSP-21061.
2. Only available on the ADSP-21061.

E.3.2 IOP Register Access Restrictions

Because the IOP registers are memory-mapped they cannot be written with data coming directly from memory. They must instead be written from (or read into) ADSP-2106x core registers, usually one of the general-purpose registers of the register file (R15–R0). The IOP registers can also be written or read by external devices, usually other ADSP-2106xs and/or a host processor.

IOP registers other than the DMA buffers cannot be the target of DMA transfers. During DMA transfers the DMA buffer registers are written and read to internal memory over the I/O data bus—these transfers are directly controlled by the ADSP-2106x's DMA controller, however, not with addresses generated over the I/O address bus. The DMA buffer registers on the ADSP-21060 and ADSP-21062 include EPB0–EPB3 (external port data buffers), LBUF0–LBUF5 (link port data buffers), and TX0, RX0, TX1, and RX1 (serial port data buffers). On the ADSP-21061, the DMA buffer registers include EBP0, EBP1, TX0, RX0, TX1, and RX1.

E.3.3 IOP Register Group Access Contention

The ADSP-2106x has four separate on-chip buses that can independently access the memory-mapped IOP registers: the PM bus, DM bus, I/O bus, and external port bus. The external port bus connects the off-chip $DATA_{47-0}$ bus to all on-chip buses. The I/O bus connects the external port's data buffers to memory and to the on-chip I/O processor. The I/O bus carries data being transferred to and from the DMA buffer IOP registers.

Any of these buses can attempt to read or write an IOP register at any time. Access contention occurs when more than one of the buses attempts to access the same *group* of IOP registers (see Table E.7). One exception to this access contention rule exists: the I/O bus and external port bus can simultaneously access the DB (DMA buffer) group of registers, allowing DMA transfers to internal memory at full speed.

IOP register group access conflicts are resolved on a fixed priority basis. External port to IOP register accesses occur first, then PM and/or DM bus, then I/O bus:

External port ↔ IOP register accesses	*1st priority*
PM-DM bus ↔ IOP register accesses	*2nd priority*

I/O bus ↔ IOP register accesses *3rd priority*

The bus with the highest priority will gain access to the IOP registers first and any lower priority accesses are held off (by extra cycles generated by the core processor and/or I/O processor). If a DMA grant has been given for an I/O access, that access will be completed before an access from any another bus is allowed.

The external port DMA data buffers (EPB0–EPB3) are 6-word deep FIFOs. An input to the buffers can occur in the same cycle as an output. The external port bus has its own independent access to these buffers. Contention occurs when the PM bus, DM bus, and/or I/O bus try to access the data buffers at the same time. In this case the I/O bus access proceeds first, but subsequent I/O bus accesses are held off until after the PM and/or DM bus accesses.

E.3.4 IOP Register Write Latencies

IOP register writes are internally completed at the end of the cycle in which they occur. The IOP register will therefore read back the newly written value on the very next cycle.

Not all writes take effect in the next cycle, however. Control and mode bits generally take effect in the second cycle after completion of the write, with the exception of the external port packing control bits and buffer flush bits which take effect in the third cycle after completion of the write.

The external port and core processor may conflict if they attempt to access the same IOP register group. In this case the core processor access is delayed until all external port accesses have completed.

E Control/Status Registers

Address	*Register Name*	*Initialization After RESET*	*Register Group*	*Description*
0x0000	SYSCON	0x0000 0010	SC	System Configuration
0x0001	VIRPT	0x0002 0014	SC	Multiprocessor Vector Interrupt
0x0002	WAIT	0x21AD 6B5A	SC	External Memory Wait State Configuration
0x0003	SYSTAT	0x0000 0*nn*0 *	SC	System Status
0x0004	EPB0	ni	DB	External Port DMA FIFO Buffer 0
0x0005	EPB1	ni	DB	External Port DMA FIFO Buffer 1
0x0006	EPB2	ni	DB	External Port DMA FIFO Buffer 2[1]
0x0007	EPB3	ni	DB	External Port DMA FIFO Buffer 3[1]
0x0008	MSGR0	ni	SC	Message Register 0
0x0009	MSGR1	ni	SC	Message Register 1
0x000A	MSGR2	ni	SC	Message Register 2
0x000B	MSGR3	ni	SC	Message Register 3
0x000C	MSGR4	ni	SC	Message Register 4
0x000D	MSGR5	ni	SC	Message Register 5
0x000E	MSGR6	ni	SC	Message Register 6
0x000F	MSGR7	ni	SC	Message Register 7
0x0010–0x0017	*reserved*			
0x0018	BMAX	0x0000 0000	SC	Bus Timeout Maximum
0x0019	BCNT	0x0000 0000	SC	Bus Timeout Counter
0x001A	*reserved*			
0x001B	ELAST	ni	SC	Address of Last External Access in Bank 0
0x001C	DMAC6	†	DB	DMA Channel 6 Control Reg. (Ext. Port Buffer 0 *or* Link Buffer 4)
0x001D	DMAC7	0x0000 0000	DB	DMA Channel 7 Control Reg. (Ext. Port Buffer 1 *or* Link Buffer 5)
0x001E	DMAC8	0x0000 0000	DB	DMA Channel 8 Control Reg. (Ext. Port Buffer 2)[1]
0x001F	DMAC9	0x0000 0000	DB	DMA Channel 9 Control Reg. (Ext. Port Buffer 3)[1]
0x0020–0x002F	*reserved*			
0x0030	II4	ni	DA	DMA Channel 4 Index (Link Buffer 2)[1]
0x0031	IM4	ni	DA	DMA Channel 4 Modifier (Link Buffer 2)[1]
0x0032	C4	ni	DA	DMA Channel 4 Count (Link Buffer 2)[1]
0x0033	CP4	ni	DA	DMA Channel 4 Chain Pointer (Link Buffer 2)[1]
0x0034	GP4	ni	DA	DMA Channel 4 General-Purpose / 2-D DMA (Link Buffer 2)[1]
0x0035	DB4	ni	DA	DMA Channel 4 General-Purpose / 2-D DMA (Link Buffer 2)[1]
0x0036	DA4	ni	DA	DMA Channel 4 General-Purpose / 2-D DMA (Link Buffer 2)[1]
0x0037	DMASTAT	ni	SC	DMA Channel Status Register
0x0038	II5	ni	DA	DMA Channel 5 Index (Link Buffer 3)[1]
0x0039	IM5	ni	DA	DMA Channel 5 Modifier (Link Buffer 3)[1]
0x003A	C5	ni	DA	DMA Channel 5 Count (Link Buffer 3)[1]
0x003B	CP5	ni	DA	DMA Channel 5 Chain Pointer (Link Buffer 3)[1]
0x003C	GP5	ni	DA	DMA Channel 5 General-Purpose / 2-D DMA (Link Buffer 3)[1]
0x003D	DB5	ni	DA	DMA Channel 5 General-Purpose / 2-D DMA (Link Buffer 3)[1]
0x003E	DA5	ni	DA	DMA Channel 5 General-Purpose / 2-D DMA (Link Buffer 3)[1]
0x003F	*reserved*			
0x0040	II6	†	DA	DMA Channel 6 Index (Ext. Port Buffer 0 *or* Link Buffer 4)
0x0041	IM6	†	DA	DMA Channel 6 Modifier (Ext. Port Buffer 0 *or* Link Buffer 4)
0x0042	C6	†	DA	DMA Channel 6 Count (Ext. Port Buffer 0 *or* Link Buffer 4)
0x0043	CP6	†	DA	DMA Channel 6 Chain Pointer (Ext. Port Buffer 0 *or* Link Buffer 4)
0x0044	GP6	†	DA	DMA Channel 6 Gen-Pur. Reg. (Ext. Port Buffer 0 *or* Link Buffer 4)
0x0045	EI6	†	DA	DMA Channel 6 Ext. Index (Ext. Port Buffer 0 *or* Link Buffer 4)
0x0046	EM6	†	DA	DMA Channel 6 Ext. Modifier (Ext. Port Buffer 0 *or* Link Buffer 4)
0x0047	EC6	†	DA	DMA Channel 6 Ext. Count (Ext. Port Buffer 0 *or* Link Buffer 4)

Address	Register Name	Initialization After RESET	Register Group	Description
0x0048	II7	ni	DA	DMA Channel 7 Index (Ext. Port Buffer 1 *or* Link Buffer 5)
0x0049	IM7	ni	DA	DMA Channel 7 Modifier (Ext. Port Buffer 1 *or* Link Buffer 5)
0x004A	C7	ni	DA	DMA Channel 7 Count (Ext. Port Buffer 1 *or* Link Buffer 5)
0x004B	CP7	ni	DA	DMA Channel 7 Chain Pointer (Ext. Port Buffer 1 *or* Link Buffer 5)
0x004C	GP7	ni	DA	DMA Channel 7 Gen-Pur. Reg. (Ext. Port Buffer 1 *or* Link Buffer 5)
0x004D	EI7	ni	DA	DMA Channel 7 Ext. Index (Ext. Port Buffer 1 *or* Link Buffer 5)
0x004E	EM7	ni	DA	DMA Channel 7 Ext. Modifier (Ext. Port Buffer 1 *or* Link Buffer 5)
0x004F	EC7	ni	DA	DMA Channel 7 Ext. Count (Ext. Port Buffer 1 *or* Link Buffer 5)
0x0050	II8	ni	DA	DMA Channel 8 Index (Ext. Port Buffer 2)[1]
0x0051	IM8	ni	DA	DMA Channel 8 Modifier (Ext. Port Buffer 2)[1]
0x0052	C8	ni	DA	DMA Channel 8 Count (Ext. Port Buffer 2)[1]
0x0053	CP8	ni	DA	DMA Channel 8 Chain Pointer (Ext. Port Buffer 2)[1]
0x0054	GP8	ni	DA	DMA Channel 8 Gen-Purpose Register (Ext. Port Buffer 2)[1]
0x0055	EI8	ni	DA	DMA Channel 8 Ext. Index (Ext. Port Buffer 2)[1]
0x0056	EM8	ni	DA	DMA Channel 8 Ext. Modifier (Ext. Port Buffer 2)[1]
0x0057	EC8	ni	DA	DMA Channel 8 Ext. Count (Ext. Port Buffer 2)[1]
0x0058	II9	ni	DA	DMA Channel 9 Index (Ext. Port Buffer 3)[1]
0x0059	IM9	ni	DA	DMA Channel 9 Modifier (Ext. Port Buffer 3)[1]
0x005A	C9	ni	DA	DMA Channel 9 Count (Ext. Port Buffer 3)[1]
0x005B	CP9	ni	DA	DMA Channel 9 Chain Pointer (Ext. Port Buffer 3)[1]
0x005C	GP9	ni	DA	DMA Channel 9 Gen-Purpose Register (Ext. Port Buffer 3)[1]
0x005D	EI9	ni	DA	DMA Channel 9 Ext. Index (Ext. Port Buffer 3)[1]
0x005E	EM9	ni	DA	DMA Channel 9 Ext. Modifier (Ext. Port Buffer 3)[1]
0x005F	EC9	ni	DA	DMA Channel 9 Ext. Count (Ext. Port Buffer 3)[1]
0x0060	II0	ni	DA	DMA Channel 0 Index (SPORT0 Receive)
0x0061	IM0	ni	DA	DMA Channel 0 Modifier (SPORT0 Receive)[2]
0x0062	C0	ni	DA	DMA Channel 0 Count (SPORT0 Receive)
0x0063	CP0	ni	DA	DMA Channel 0 Chain Pointer (SPORT0 Receive)
0x0064	GP0	ni	DA	DMA Channel 0 General-Purpose / 2-D DMA (SPORT0 Receive)
0x0065	DB0	ni	DA	DMA Channel 0 General-Purpose / 2-D DMA (SPORT0 Receive)[1]
0x0066	DA0	ni	DA	DMA Channel 0 General-Purpose / 2-D DMA (SPORT0 Receive)[1]
0x0067	*reserved*			
0x0068	II1	ni	DA	DMA Channel 1 Index (SPORT1 Receive *or* Link Buffer 0)
0x0069	IM1	ni	DA	DMA Channel 1 Modifier (SPORT1 Receive *or* Link Buffer 0)[2]
0x006A	C1	ni	DA	DMA Channel 1 Count (SPORT1 Receive *or* Link Buffer 0)
0x006B	CP1	ni	DA	DMA Channel 1 Chain Pointer (SPORT1 Receive *or* Link Buffer 0)
0x006C	GP1	ni	DA	DMA Channel 1 Gen-Pur / 2D DMA (SPORT1 Rcv *or* Link Buffer 0)
0x006D	DB1	ni	DA	DMA Channel 1 Gen-Pur / 2D DMA (SPORT1 Rcv *or* Link Buffer 0)[1]
0x006E	DA1	ni	DA	DMA Channel 1 Gen-Pur / 2D DMA (SPORT1 Rcv *or* Link Buffer 0)[1]
0x006F	*reserved*			

ni – not initialized
* SYSTAT bits 4-11 depend on the value of the ID_{2-0} inputs.
† Initialized during booting (see "Booting" in the *System Design* chapter)

IOP Register Groups: SC – System Control register group; DA – DMA Address register group; DB – DMA Buffer register group; LSP – Link / Serial Port register group

Table E.7 IOP Register Addresses, RESET Initialization, & Grouping **(cont. on next page)**

E Control/Status Registers

Address	*Register Name*	*Initialization After RESET*	*Register Group*	*Description*
0x0070	II2	ni	DA	DMA Channel 2 Index (SPORT0 Transmit)
0x0071	IM2	ni	DA	DMA Channel 2 Modifier (SPORT0 Transmit)[2]
0x0072	C2	ni	DA	DMA Channel 2 Count (SPORT0 Transmit)
0x0073	CP2	ni	DA	DMA Channel 2 Chain Pointer (SPORT0 Transmit)
0x0074	GP2	ni	DA	DMA Channel 2 Gen-Pur / 2D DMA (SPORT0 Transmit)
0x0075	DB2	ni	DA	DMA Channel 2 Gen-Pur / 2D DMA (SPORT0 Transmit)[1]
0x0076	DA2	ni	DA	DMA Channel 2 Gen-Pur / 2D DMA (SPORT0 Transmit)[1]
0x0077	*reserved*			
0x0078	II3	ni	DA	DMA Channel 3 Index (SPORT1 Transmit *or* Link Buffer 1)
0x0079	IM3	ni	DA	DMA Channel 3 Modifier (SPORT1 Transmit *or* Link Buffer 1)[2]
0x007A	C3	ni	DA	DMA Channel 3 Count (SPORT1 Transmit *or* Link Buffer 1)
0x007B	CP3	ni	DA	DMA Channel 3 Chain Pointer (SPORT1 Transmit *or* Link Buffer 1)
0x007C	GP3	ni	DA	DMA Channel 3 Gen-Pur / 2D DMA (SPORT1 Transmit *or* Link Buffer 1)
0x007D	DB3	ni	DA	DMA Channel 3 Gen-Pur / 2D DMA (SPORT1 Transmit *or* Link Buffer 1)[1]
0x007E	DA3	ni	DA	DMA Channel 3 Gen-Pur / 2D DMA (SPORT1 Transmit *or* Link Buffer 1)[1]
0x007F	*reserved*			
0x00A0–0x00BF	*reserved*			
0x00C0	LBUF0	ni	LSP	Link Buffer 0[1]
0x00C1	LBUF1	ni	LSP	Link Buffer 1[1]
0x00C2	LBUF2	ni	LSP	Link Buffer 2[1]
0x00C3	LBUF3	ni	LSP	Link Buffer 3[1]
0x00C4	LBUF4	ni	LSP	Link Buffer 4[1]
0x00C5	LBUF5	ni	LSP	Link Buffer 5[1]
0x00C6	LCTL	0x0000 0000	LSP	Link Buffer Control Register[1]
0x00C7	LCOM	0x0000 0000	LSP	Link Common Control Register[1]
0x00C8	LAR	0x2C688	LSP	Link Buffer Assignment Register[1]
0x00C9	LSRQ	0x0000 0000	LSP	Link Port Service Request & Mask Register[1]
0x00CA	LPATH1	ni	LSP	Link Path 1 Register (Mesh Multiprocessing)[1]
0x00CB	LPATH2	ni	LSP	Link Path 2 Register (Mesh Multiprocessing)[1]
0x00CC	LPATH3	ni	LSP	Link Path 3 Register (Mesh Multiprocessing)[1]
0x00CD	LPCNT	ni	LSP	Link Path Counter (Mesh Multiprocessing)[1]
0x00CE	CNST1	ni	LSP	Link Port Constant 1 (Mesh Multiprocessing)[1]
0x00CF	CNST2	ni	LSP	Link Port Constant 2 (Mesh Multiprocessing)[1]
0x00D0–0x00DF	*reserved*			
0x00E0	STCTL0	0x0000 0000	LSP	SPORT0 Transmit Control Register
0x00E1	SRCTL0	0x0000 0000	LSP	SPORT0 Receive Control Register
0x00E2	TX0	ni	LSP	SPORT0 Transmit Data Buffer
0x00E3	RX0	ni	LSP	SPORT0 Receive Data Buffer
0x00E4	TDIV0	ni	LSP	SPORT0 Transmit Divisors
0x00E5	*reserved*			
0x00E6	RDIV0	ni	LSP	SPORT0 Receive Divisors
0x00E7	*reserved*			
0x00E8	MTCS0	ni	LSP	SPORT0 Multichannel Transmit Selector
0x00E9	MRCS0	ni	LSP	SPORT0 Multichannel Receive Selector
0x00EA	MTCCS0	ni	LSP	SPORT0 Multichannel Transmit Compand Selector
0x00EB	MRCCS0	ni	LSP	SPORT0 Multichannel Receive Compand Selector
0x00EC	KEYWD0	ni	LSP	SPORT0 Receive Comparison[3]
0x00ED	KEYMASK0	ni	LSP	SPORT0 Receive Comparison Mask[3]
0x00EE	SPATH0	0x0001	LSP	SPORT0 Path Length (Mesh Multiprocessing)
0x00EF	*reserved*	0x0001		

Address	Register Name	Initialization After RESET	Register Group	Description
0x00F0	STCTL1	0x0000 0000	LSP	SPORT1 Transmit Control Register
0x00F1	SRCTL1	0x0000 0000	LSP	SPORT1 Receive Control Register
0x00F2	TX1	ni	LSP	SPORT1 Transmit Data Buffer
0x00F3	RX1	ni	LSP	SPORT1 Receive Data Buffer
0x00F4	TDIV1	ni	LSP	SPORT1 Transmit Divisors
0x00F5	*reserved*			
0x00F6	RDIV1	ni	LSP	SPORT1 Receive Divisors
0x00F7	*reserved*			
0x00F8	MTCS1	ni	LSP	SPORT1 Multichannel Transmit Selector
0x00F9	MRCS1	ni	LSP	SPORT1 Multichannel Receive Selector
0x00FA	MTCCS1	ni	LSP	SPORT1 Multichannel Transmit Compand Selector
0x00FB	MRCCS1	ni	LSP	SPORT1 Multichannel Receive Compand Selector
0x00FC	KEYWD1	ni	LSP	SPORT1 Receive Comparison[3]
0x00FD	KEYMASK1	ni	LSP	SPORT1 Receive Comparison Mask[3]
0x00FE	SPATH1	0x0001	LSP	SPORT1 Path Length (Mesh Multiprocessing)
0x00FF	*reserved*	0x0001		

ni – not initialized

IOP Register Groups: SC – System Control register group; DA – DMA Address register group; DB – DMA Buffer register group; LSP – Link / Serial Port register group

Table E.7 IOP Register Addresses, RESET Initialization, & Grouping (cont.)

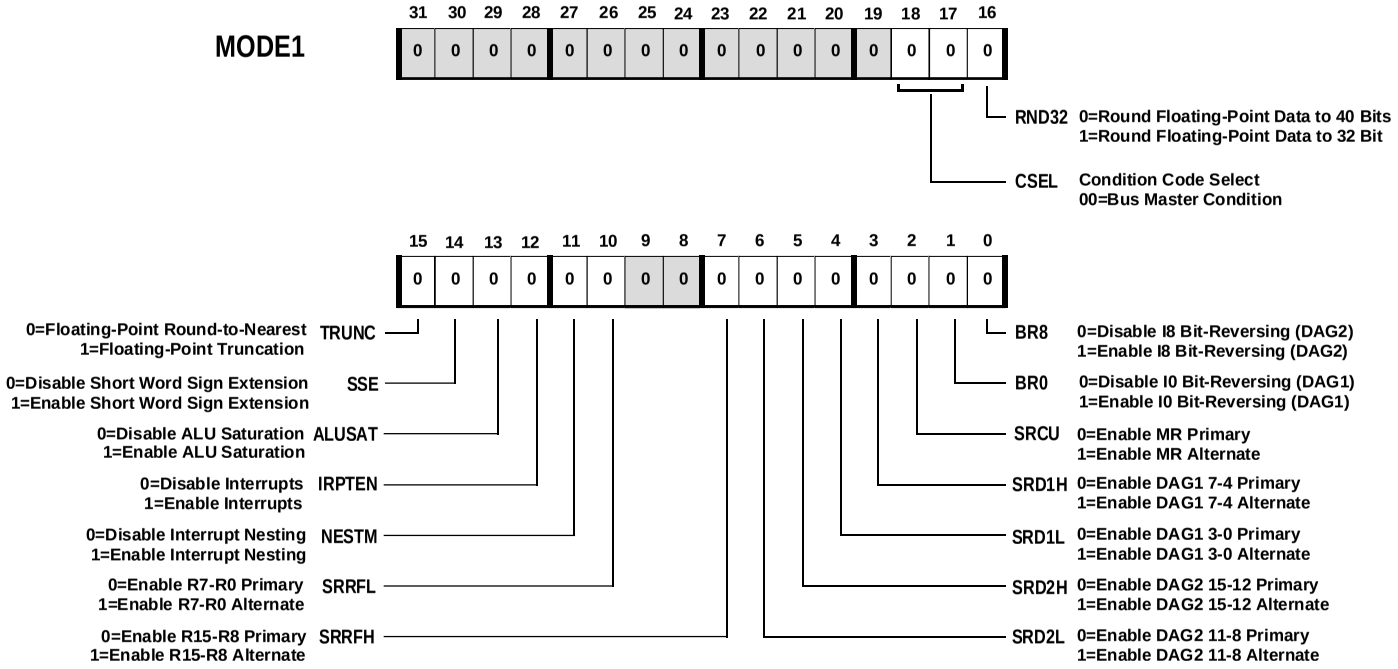
1. Not available on the ADSP-21061.
2. The contents of the IM0-3 registers are fixed to '1' on the ADSP-21061; on this DSP, these registers may be written, but the writes are ignored and reads of these registers return the value '1.'
3. Only available on the ADSP-21061. This register location is reserved on the ADSP-21060 and ADSP-21062.

E.4 MODE1 REGISTER

Bit	*Name*	*Definition*
0	BR8	Bit-reversing for I8 (DAG2)
1	BR0	Bit-reversing for I0 (DAG1)
2	SRCU	Alternate register select for computation units
3	SRD1H	DAG1 alternate register select (7-4)
4	SRD1L	DAG1 alternate register select (3-0)
5	SRD2H	DAG2 alternate register select (15-12)
6	SRD2L	DAG2 alternate register select (11-8)
7	SRRFH	Register file alternate select for R15-R8
8-9	–	*reserved*
10	SRRFL	Register file alternate select for R7-R0
11	NESTM	Interrupt nesting enable
12	IRPTEN	Global interrupt enable
13	ALUSAT	Enable ALU saturation (full scale in fixed-point)
14	SSE*	Enable short word sign extension
15	TRUNC	1=Floating-point truncation; 0=Round to nearest
16	RND32	1=Round floating-point data to 32 bits; 0=Round to 40 bits
17-18	CSEL	Spare condition code select (00 selects bus master condition)**
19-31	–	*reserved*

* Does not apply to PX register writes.
** The bus master condition (BM) indicates whether the ADSP-2106x is the current bus

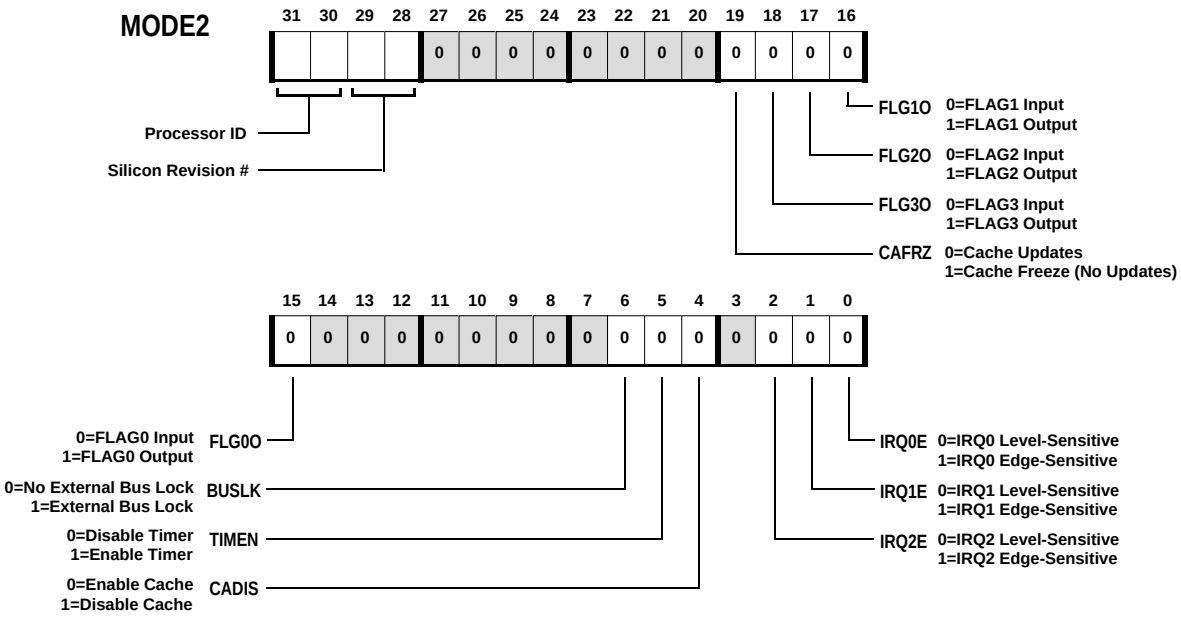
All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*


MODE1
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RND32 0=Round Floating-Point Data to 40 Bits
1=Round Floating-Point Data to 32 Bit
CSEL Condition Code Select
00=Bus Master Condition
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0=Floating-Point Round-to-Nearest
1=Floating-Point Truncation
TRUNC
0=Disable Short Word Sign Extension
1=Enable Short Word Sign Extension
SSE
0=Disable ALU Saturation
1=Enable ALU Saturation
ALUSAT
0=Disable Interrupts
1=Enable Interrupts
IRPTEN
0=Disable Interrupt Nesting
1=Enable Interrupt Nesting
NESTM
0=Enable R7-R0 Primary
1=Enable R7-R0 Alternate
SRRFL
0=Enable R15-R8 Primary
1=Enable R15-R8 Alternate
SRRFH
BR8 0=Disable I8 Bit-Reversing (DAG2)
1=Enable I8 Bit-Reversing (DAG2)
BR0 0=Disable I0 Bit-Reversing (DAG1)
1=Enable I0 Bit-Reversing (DAG1)
SRCU 0=Enable MR Primary
1=Enable MR Alternate
SRD1H 0=Enable DAG1 7-4 Primary
1=Enable DAG1 7-4 Alternate
SRD1L 0=Enable DAG1 3-0 Primary
1=Enable DAG1 3-0 Alternate
SRD2H 0=Enable DAG2 15-12 Primary
1=Enable DAG2 15-12 Alternate
SRD2L 0=Enable DAG2 11-8 Primary
1=Enable DAG2 11-8 Alternate

master in a multiprocessor system. To enable the use of this condition, bits 17 and 18 of MODE1 must both be zeros; otherwise the condition is always evaluated as false.

E.5 MODE2 REGISTER

Bit	*Name*	*Definition*
0	IRQ0E	$\overline{\text{IRQ0}}$ 1=edge sensitive; 0=level-sensitive
1	IRQ1E	$\overline{\text{IRQ1}}$ 1=edge sensitive; 0=level-sensitive
2	IRQ2E	$\overline{\text{IRQ2}}$ 1=edge sensitive; 0=level-sensitive
3		*reserved*
4	CADIS	Cache disable
5	TIMEN	Timer enable
6	BUSLK	External bus lock (multiprocessor systems)
7-14		*reserved*
15	FLG0O	FLAG0 1=output; 0=input
16	FLG1O	FLAG1 1=output; 0=input
17	FLG2O	FLAG2 1=output; 0=input
18	FLG3O	FLAG3 1=output; 0=input
19	CAFRZ	Cache freeze
20-27		*reserved*



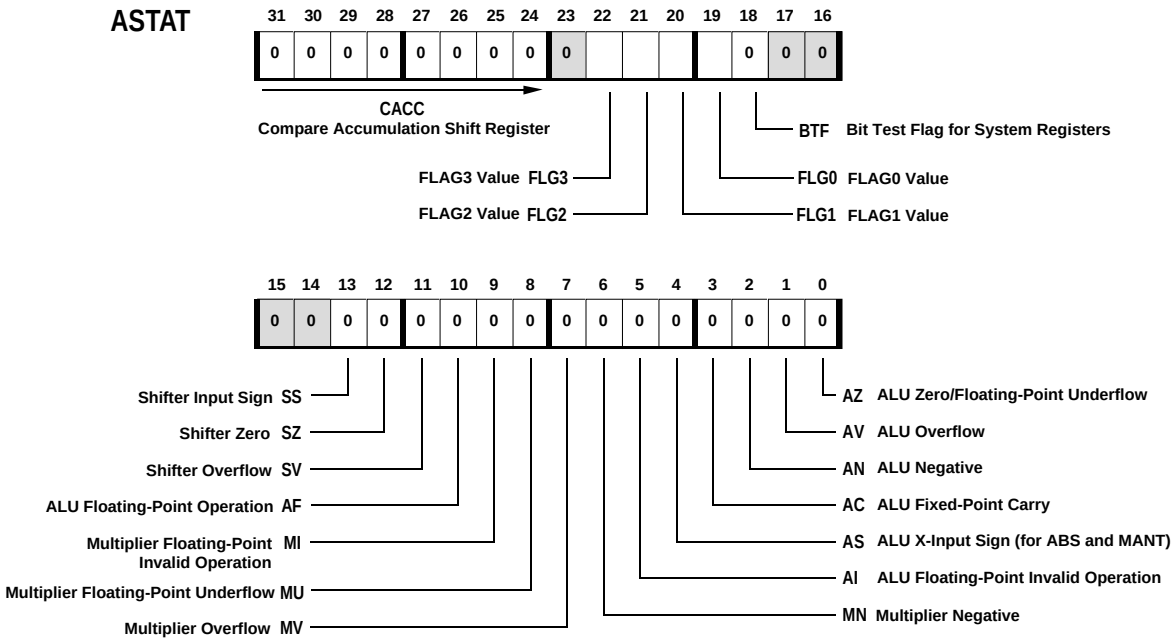
All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

28-29	Silicon revision #
30-31	Processor ID (ID=01 for ADSP-21060, ID=10 for ADSP-21062)

E.6 ARITHMETIC STATUS REGISTER (ASTAT)

Bit	*Name*	*Definition*
0	AZ	ALU result zero or floating-point underflow
1	AV	ALU overflow
2	AN	ALU result negative
3	AC	ALU fixed-point carry
4	AS	ALU X input sign (ABS and MANT operations)
5	AI	ALU floating-point invalid operation
6	MN	Multiplier result negative
7	MV	Multiplier overflow
8	MU	Multiplier floating-point underflow
9	MI	Multiplier floating-point invalid operation
10	AF	ALU floating-point operation
11	SV	Shifter overflow
12	SZ	Shifter result zero
13	SS	Shifter input sign
14-17		*reserved*
18	BTF	Bit test flag for system registers
19	FLG0	FLAG0 value
20	FLG1	FLAG1 value
21	FLG2	FLAG2 value
22	FLG3	FLAG3 value
23		*reserved*
24-31	CACC	Compare accumulation bits

ASTAT



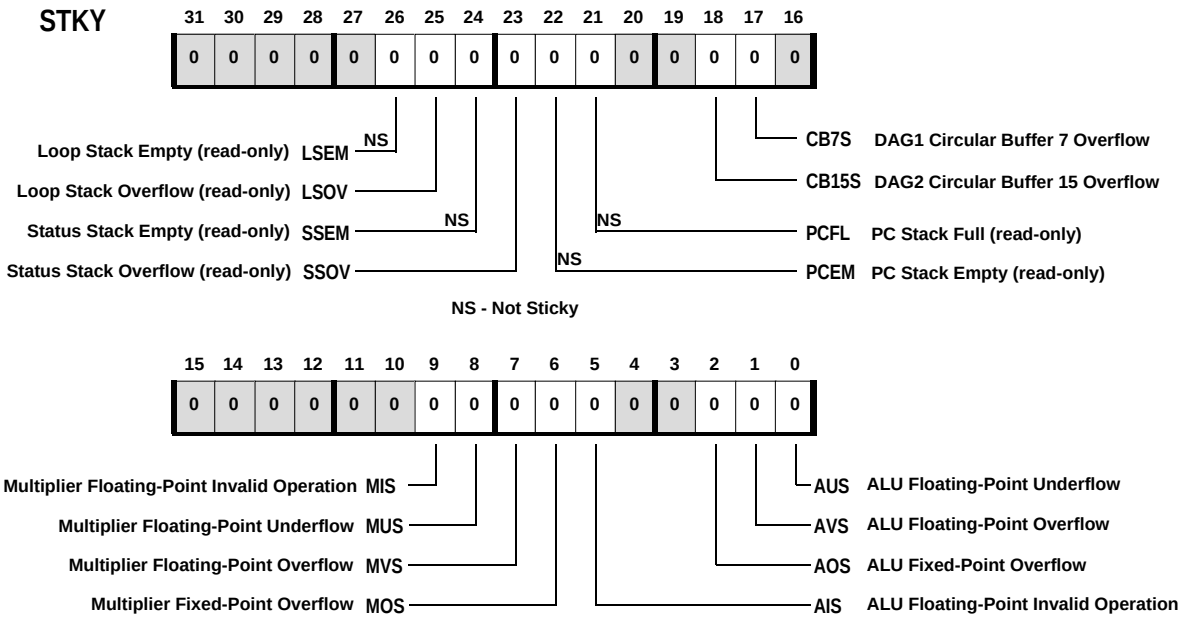
All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

E.7 STICKY STATUS (STKY)

Bit	*Name*	*Definition*
0	AUS	ALU floating-point underflow
1	AVS	ALU floating-point overflow
2	AOS	ALU fixed-point overflow
3-4		*reserved*
5	AIS	ALU floating-point invalid operation
6	MOS	Multiplier fixed-point overflow
7	MVS	Multiplier floating-point overflow
8	MUS	Multiplier floating-point underflow
9	MIS	Multiplier floating-point invalid operation
10-16		*reserved*
17	CB7S	DAG1 circular buffer 7 overflow
18	CB15S	DAG2 circular buffer 15 overflow
19-20		*reserved*
21	PCFL	PC stack full (not sticky)
22	PCEM	PC stack empty (not sticky)
23	SSOV	Status stack overflow (MODE1 and ASTAT)
24	SSEM	Status stack empty (not sticky)
25	LSOV	Loop stack overflow (Loop Address and Loop Counter)
26	LSEM	Loop stack empty (not sticky)
27-31		*reserved*

Bits 21-26 are read-only. Writes to the STKY register have no effect on these bits.

All bits except 21, 22, 24, 26 are sticky (see “Stack Flags” in the *Program Sequencing* chapter). Once a sticky bit is set, it remains set until explicitly cleared.



All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

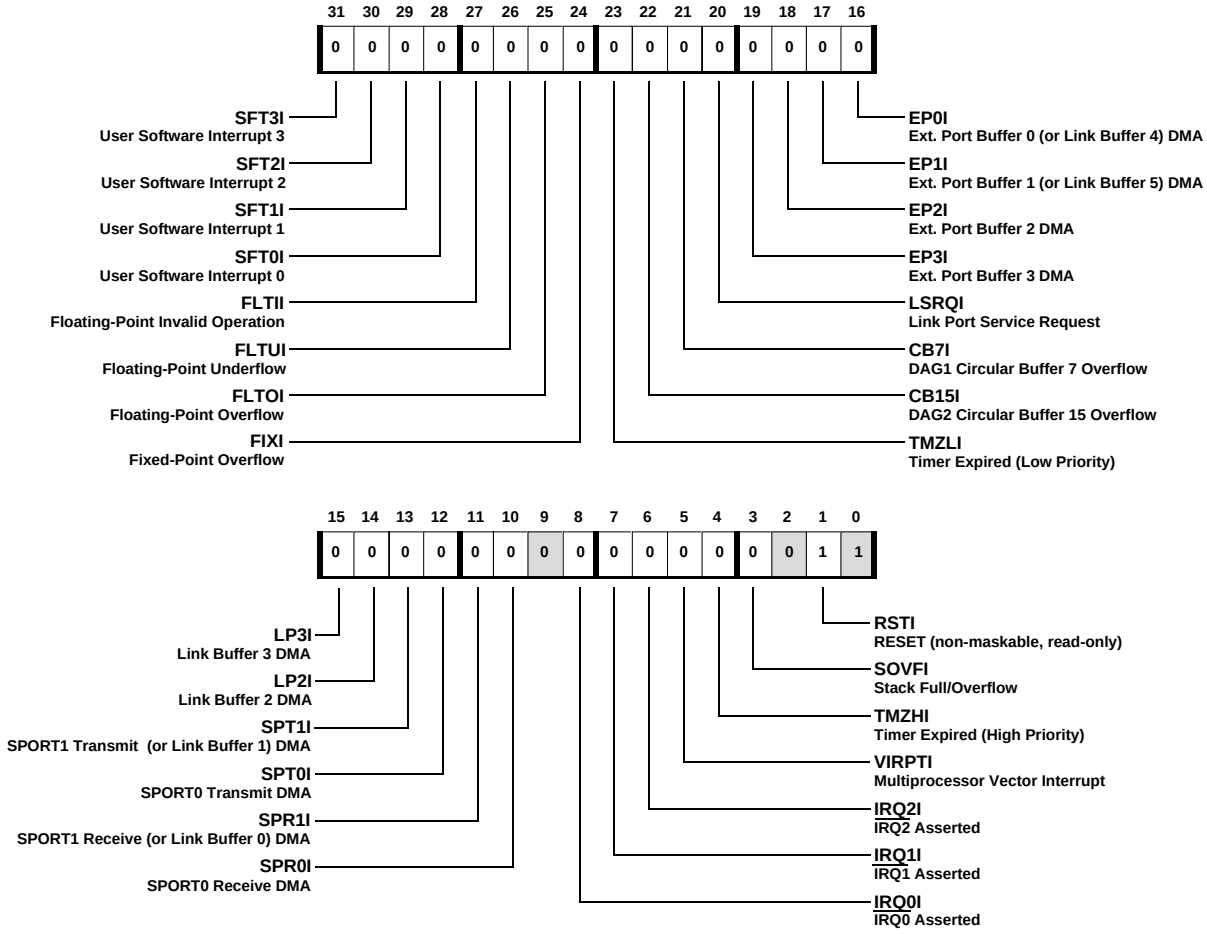
E.8 INTERRUPT LATCH (IRPTL) & INTERRUPT MASK (IMASK)

IRPTL and IMASK have the exact same bit positions, corresponding to the ADSP-2106x interrupts in order of priority.

Bit	*Vector Address**	*Interrupt Name*	*Function*	
0	0x00	–	*reserved*	
1	0x04	RSTI	Reset (read-only)**	**HIGHEST PRIORITY**
2	0x08	–	*reserved*	
3	0x0C	SOVFI	Status stack or loop stack overflow or PC stack full	
4	0x10	TMZHI	Timer=0 (high priority option)	
5	0x14	VIRPTI	Vector Interrupt	
6	0x18	IRQ2I	$\overline{\text{IRQ2}}$ asserted	
7	0x1C	IRQ1I	$\overline{\text{IRQ1}}$ asserted	
8	0x20	IRQ0I	$\overline{\text{IRQ0}}$ asserted	
9	0x24	–	*reserved*	
10	0x28	SPR0I	DMA Channel 0 – SPORT0 Receive	
11	0x2C	SPR1I	DMA Channel 1 – SPORT1 Receive (or Link Buffer 0)	
12	0x30	SPT0I	DMA Channel 2 – SPORT0 Transmit	
13	0x34	SPT1I	DMA Channel 3 – SPORT1 Transmit (or Link Buffer 1)	
14	0x38	LP2I	DMA Channel 4 – Link Buffer 2	
15	0x3C	LP3I	DMA Channel 5 – Link Buffer 3	
16	0x40	EP0I	DMA Channel 6 – Ext. Port Buffer 0 (or Link Buffer 4)	
17	0x44	EP1I	DMA Channel 7 – Ext. Port Buffer 1 (or Link Buffer 5)	
18	0x48	EP2I	DMA Channel 8 – Ext. Port Buffer 2	
19	0x4C	EP3I	DMA Channel 9 – Ext. Port Buffer 3	
20	0x50	LSRQI	Link Port Service Request	
21	0x54	CB7I	Circular Buffer 7 overflow	
22	0x58	CB15I	Circular Buffer 15 overflow	
23	0x5C	TMZLI	Timer=0 (low priority option)	
24	0x60	FIXI	Fixed-point overflow	
25	0x64	FLTOI	Floating-point overflow exception	
26	0x68	FLTUI	Floating-point underflow exception	
27	0x6C	FLTII	Floating-point invalid exception	
28	0x70	SFT0I	User software interrupt 0	
29	0x74	SFT1I	User software interrupt 1	
30	0x78	SFT2I	User software interrupt 2	
31	0x7C	SFT3I	User software interrupt 3	**LOWEST PRIORITY**

* Offset from base address: 0x0002 0000 for interrupt vector table in internal memory, 0x0040 0000 for interrupt vector table in external memory
** Non-maskable

IRPTL & IMASK



Default values for IMASK only; IRPTL is cleared after reset.
For IMASK: 1=unmasked (enabled), 0=masked (disabled)

All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

E.9 SYSTEM CONFIGURATION (SYSCON)

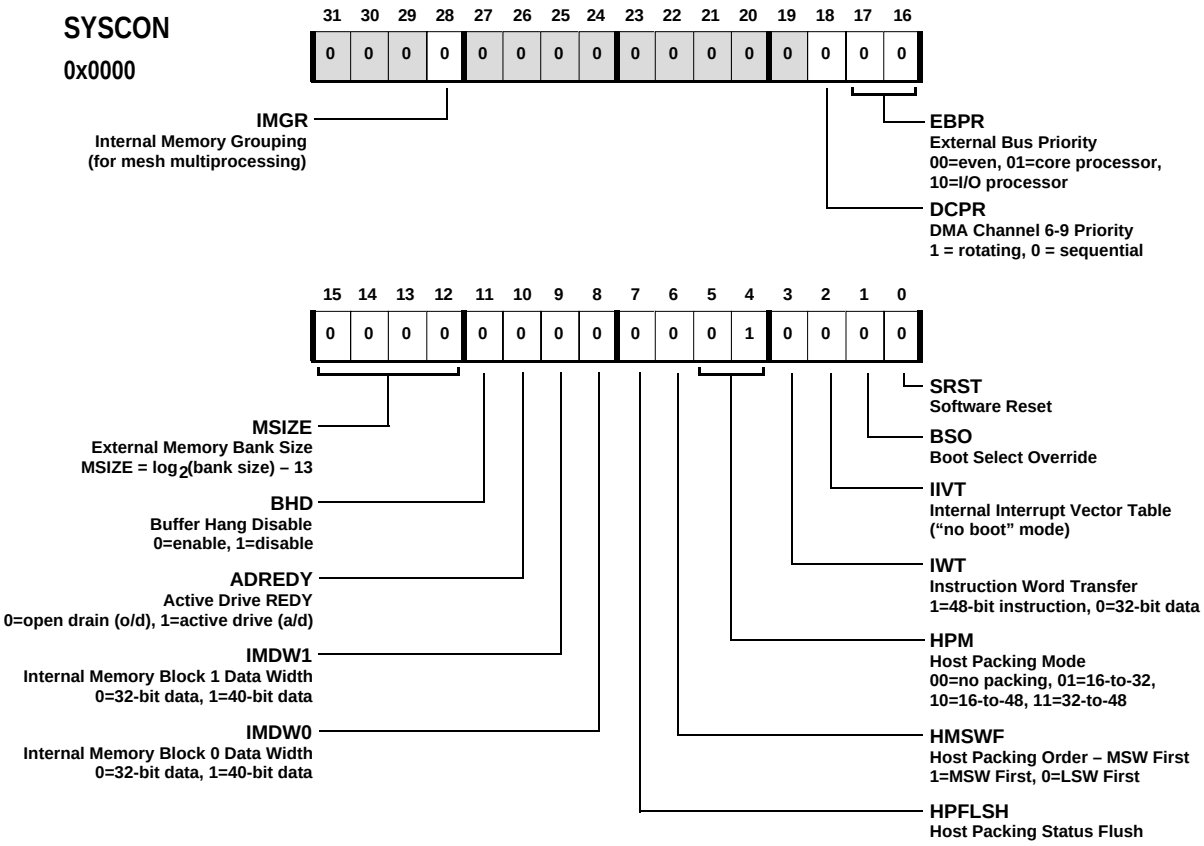
The SYSCON register is used to set up system configuration selections. SYSCON is memory-mapped in internal memory at address 0x0000. After reset the SYSCON register is initialized to 0x0000 0010. This causes the ADSP-2106x to assume a 16-bit bus for any host processor; two 16-bit words must be written to SYSCON to change the setting of HPM, even if the host bus is 32 bits wide.

Bit	*Name*	*Definition*
0	SRST	Software Reset
1	BSO	Boot Select Override
2	IIVT	Internal Interrupt Vector Table (for "no booting" mode)
3	IWT	Instruction Word Transfer (1=48-bit instruction, 0=32-bit data)
4-5	HPM	Host Packing Mode (00=none, 01=16-to-32, 10=16-to-48, 11=32-to-48)
6	HMSWF	Host Packing Order – MSW First (1=MSW first, 0=LSW first)
7	HPFLSH	Host Packing Status Flush
8	IMDW0	Internal Memory Block 0 Data Width (0=32-bit data, 1=40-bit data)
9	IMDW1	Internal Memory Block 1 Data Width (0=32-bit data, 1=40-bit data)
10	ADREDY	Active Drive REDY (1=active drive, 0=open drain)
11	BHD	Buffer Hang Disable (1=prevent hangs, 0=allow hangs)
12-15	MSIZE	External Memory Bank Size (MSIZE = $\log_2$(bank size) – 13)
16-17	EBPR	External Bus Priority (01=core processor, 10=I/O processor, 00=even)
18	DCPR	DMA Channel 6-9 Priority (1=rotating priority, 0=sequential priority)
19-27		*reserved*
28	IMGR	Internal Memory Grouping (Mesh Multiprocessing)
29-31		*reserved*

SRST **Software Reset**—Causes a software reset. This has the same effect as the $\overline{\text{RESET}}$ pin.

BSO **Boot Select Override**—Deactivates the $\overline{\text{MS}}_{3\text{-}0}$ memory select lines and activates the $\overline{\text{BMS}}$ boot memory select output. This allows the ADSP-2106x to read from its boot EPROM when it is no longer in boot mode. Since only 256 instructions can be loaded during EPROM booting, setting the BSO bit lets the ADSP-2106x read additional code or data from its boot EPROM after booting is completed. (See "Booting" in Chapter 11, *System Design*.)

1=Activate $\overline{\text{BMS}}$ to read from boot EPROM



IIVT **Internal Interrupt Vector Table ("no booting" mode)**—Locates the interrupt vector table at address 0x0002 0000 in internal memory for *no booting* mode (EBOOT=0, LBOOT=0, $\overline{\text{BMS}}$(input)=0). When IIVT=0 in *no booting* mode, the interrupt vector table is located at address 0x0040 0000 in external memory. Note that IIVT is initialized to zero after reset, locating the interrupt table in external memory for *no booting* mode. (When the ADSP-2106x is configured for any booting mode—EPROM, host, or link port—the interrupt vector table is always located in internal memory, regardless of the value of IIVT.)

1=Locate interrupt table in internal memory for "no booting" mode
0=Locate interrupt table in external memory for "no booting" mode

All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

IWT **Instruction Word Transfer**—Specifies the word width for direct reads and direct writes of the ADSP-2106x's internal memory (by other ADSP-2106xs or by the host). IWT=1 overrides the IMDW bits (see below) and forces a 48-bit (3-column) memory transfer. IWT=0 defers to the data word setting of the IMDW bits in the SYSCON register. IWT should be set whenever the ADSP-2106x bus master or host processor is reading or writing instructions from (this) ADSP-2106x.

1=48-bit words for direct read/writes
0=32-bit words for direct read/writes

HPM(1:0) **Host Packing Mode**—Specifies the internal word width and external host bus width for host processor accesses of the ADSP-2106x's internal memory or IOP registers. If the host access is a read or write to the external port data buffers (EPB0, EPB1, EPB2, or EPB3), the external host bus width selected by HPM must correspond to the external word width selected in the PMODE bits of the DMACx control register (DMAC6, DMAC7, DMAC8, and DMAC9):

HPM and PMODE must select the same external bus width for host data transfers to and from the ADSP-2106x!!

If the host access is a read or write of any IOP register other than the external port buffers or link port buffers (LBUF0-LBUF5), the word width will always be 32 bits no matter what the host bus width is. If the host access is a read or write of the link port buffers, the word width is determined *only* by HPM, and not by the LEXT bit in LCTL.

00=No packing. Maximum bus width is 32 bits for asynchronous transfers. The lower 16 bits of the 48-bit data bus will be written and read as zeros, even when reading 48-bit words. For synchronous transfers, the host bus should be 32 bits wide for data transfers or 48 bits wide for instruction word transfers. (Note: To read and write 48-bit words from internal memory, the IWT bit must be set to 1 or the IMDW bit for the block of memory being accessed must be set to 1.)

Default → at Reset **01=16-bit host bus, 32-bit words.** The host bus will be 16 bits wide; any memory access will be 32-bit words.

(Note: If the memory access is made to a block of ADSP-2106x internal memory for which the IMDW bit is set, the access will read or write the upper 32 bits of the 48-bit word.)

10=16-bit host bus, 48-bit words. The host bus will be 16 bits wide; any memory access will be 48-bit words.

11=32-bit host bus, 48-bit words. The host bus will be 32 bits wide; any memory access will be 48-bit words.

To change the host packing mode, the following sequence must occur:

1. Write to the SYSCON register, changing the setting of HPM.
2. Read SYSCON (and ignore data) to ensure that the write was completed.
3. Repeat the write to SYSCON (to flush the read, since it may have occurred in the old packing mode).
4. Wait 4 cycles.

HMSWF **Host Packing Order – Most Significant Word First**—Specifies the order in which host-accessed words are packed, for 16-to-32 bit and 16-to-48 bit packing. HMSWF is ignored for 32-to-48 bit packing. When HMSWF=1, packing is done MSW first (most significant 16-bit word first). When HMSWF=0, packing is is done LSW first.

1=MSW first
0=LSW first

HPFLSH **Host Packing Status Flush**—Resets the host packing status. Host accesses must not occur while the HPFLSH bit is being written by the ADSP-2106x processor core. There is a two cycle latency before the reset takes effect, after which the host may resume normal operations. (Note: HPFLSH is always read as a zero.)

1=Flush packing status

IMDWx **Internal Memory Block Data Width**—Selects the data word width for each block of internal memory. For 32-bit data words, set IMDWx to 0. For 40-bit data (transferred within 48-bit words), set IMDWx to 1. IMDW0 (bit 8 of SYSCON) selects the data word width for memory block 0 and IMDW1 (bit 9) selects the data word width for memory block 1. (Note: 48-bit instructions can be stored in a memory block regardless of the setting of the IMDWx bit. See “Configuring Memory For 32-Bit or 40-Bit Data” in the *Memory* chapter for more information.)

0=32-bit data
1=40-bit data

BHD **Buffer Hang Disable**—Disables the hang condition that occurs when the ADSP-2106x core tries to read from an empty (or write to a full) SPORT buffer (RX/TX), link port buffer (LBUFx), or external port buffer (EPBx). The hang condition also occurs when an external device—either another ADSP-2106x or a host processor—tries to read from an empty buffer or write to a full buffer.

After reset, BHD=0 so that buffer hang is enabled; this is the normal mode of operation for the buffers. Setting BHD=1 to disable buffer hang is useful for debugging purposes.

(Note: The *full or empty* status of a particular buffer can be determined by reading the appropriate control/status register—DMACx for the external port buffers, LCOM for the link port buffers, and SRCTLx/STCTLx for the SPORT buffers.)

1=prevent hangs
0=allow hangs

ADREDY **Active Drive REDY**—Changes REDY signal to an active drive output.

1=active drive (a/d)
0=open drain (o/d)

MSIZE(3:0) **External Memory Bank Size**—Selects the size of the external memory banks (each memory bank is the same size). The value of MSIZE is calculated with the following equation:

MSIZE=$\log_2$(bank size) – 13

(See "External Memory Banks" in the *Memory* chapter for more information.)

EBPR(1:0) **External Bus Priority**—Determines priority for use of external bus (DATA47-0 , ADDR31-0) for conflicts between the ADSP-2106x's processor core and on-chip I/O processor. This type of contention occurs when the processor core attempts an off-chip read or write at the same time that a DMA transfer is in progress (independently controlled by the I/O processor).

00=Even priority—both accesses alternate with each other
01=Core processor has priority
10=I/O processor has priority (for DMA transfer)

Note: The setting of EPBR is not related to the CPA pin function (core priority access).

(*Additional Details*: The ADSP-2106x has three on-chip buses that are multiplexed at the external port: the PM bus (instructions or data), DM bus (data), and I/O bus (DMA data). The PM bus and DM bus are controlled by the ADSP-2106x processor core. The I/O bus is controlled by the on-chip I/O processor. The I/O bus connects the external port's DMA buffers to the ADSP-2106x's internal memory and IOP registers.

Contention for use of the external bus occurs when both the processor core and I/O processor attempt an off-chip read or write during the same cycle. The contention occurs at the ADSP-2106x's external port, where the three internal buses are multiplexed together.

PM vs. DM bus conflicts are resolved with a fixed priority if the accesses are both reads or both writes—the DM bus access occurs first. An extra cycle is generated in the following cycle to allow the PM bus access to occur.

For core processor priority, the I/O processor access will be delayed until neither the PM bus nor DM bus is carrying an access. For I/O bus priority, the PM and/or DM bus accesses will be delayed until all pending I/O bus accesses are completed.)

Note: For even priority, if both the core and I/O processor try to use the external bus continuously, they will each get a bus slot every other cycle.

DCPR **DMA Channel 6-9 Priority**—Selects rotating or sequential priority for DMA channels 6-9. When DCPR is set to 1, a rotating priority scheme is implemented in which priority moves to the next highest numbered channel (modulo 4). When DCPR is 0, highest priority is assigned to channel 6 and lowest priority to channel 9.

1=Rotating priority
0=Sequential priority (ch6 high – ch9 low)

E.10 SYSTEM STATUS (SYSTAT)

The SYSTAT register provides status information, primarily for multiprocessor systems. SYSTAT is memory-mapped in internal

memory at address 0x0003. After reset, all bits in SYSTAT are initialized to zero except for IDC(2:0) and CRBM(2:0). IDC(2:0) will be equal to the value of the ADSP-2106x's ID_{2-0} inputs. CRBM(2:0) is equal to the ID of the current bus master, for ID>0. For ID=0, CRBM=1.

Bit	*Name*	*Definition*
0	HSTM	Host Mastership
1	BSYN	Bus Synchronization
2-3		*reserved*
4-6	CRBM	Current Bus Master (ID_{2-0} of ADSP-2106x bus master)
7		*reserved*
8-10	IDC	ID Code (ID_{2-0} of this ADSP-2106x)
11		*reserved*
12	DWPD	Direct Write Pending
13	VIPD	Vector Interrupt Pending
14-15	HPS	Host Packing Status
16-31		*reserved*

HSTM **Host Mastership**—Indicates whether the host processor has been granted control of the bus.

1=Host is bus master
0=Host is not bus master

BSYN **Bus Synchronization**—Indicates when the ADSP-2106x's bus arbitration logic is synchronized after reset. (See "Bus Synchronization After Reset" in the *Multiprocessing* chapter of this manual for details.)

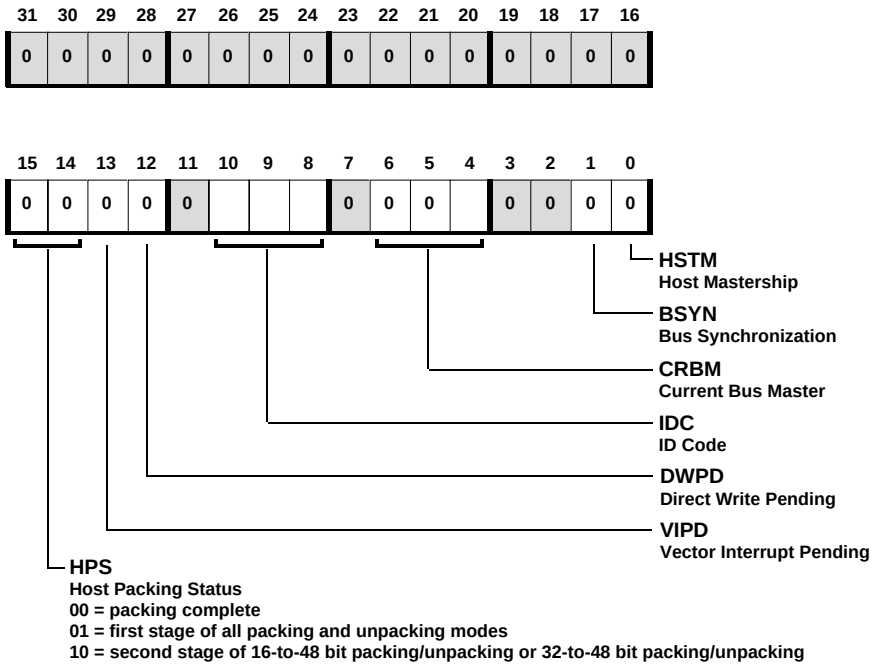
1=Bus arbitration logic is synchronized
0=Bus arbitration logic is not synchronized

CRBM(2:0) **Current Bus Master**—Indicates the ID of the ADSP-2106x that is the current bus master. If CRBM is equal to the ID of this ADSP-2106x then it is the current bus master. CRBM is only valid for $ID_{2-0} > 0$ (greater than zero). When ID_{2-0}=000, CRBM is always 1.

IDC(2:0) **ID Code**—Indicates the ID_{2-0} inputs of this ADSP-2106x.

DWPD **Direct Write Pending**—Indicates when a direct write to the ADSP-2106x's internal memory is pending. The DWPD bit is cleared when the direct write has been completed. (Direct writes may be delayed for several cycles if DMA chaining is underway or if higher

SYSTAT
0x0003



All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

priority DMA requests occur. Maximum delay is 12 cycles.)

1=Direct write pending
0=No direct write pending

VIPD **Vector Interrupt Pending**—Indicates that a pending vector interrupt has not yet been serviced. The VIPD bit is set when the VIRPT register is written to and is cleared upon return from the interrupt service routine. The host processor (or other ADSP-2106x) that issued the vector interrupt should monitor this bit to determine when the service routine has been completed (and when a new vector interrupt may be issued).

1=Vector interrupt pending
0=No vector interrupt pending

HPS(1:0) **Host Packing Status**—Indicates when host word packing is completed or, if not, what stage of the process is taking place.

00=Packing complete

01=First stage of all packing and unpacking modes.
10=Second stage of 16-to-48 bit packing/unpacking or 32-to-48 bit packing/unpacking

E.11 EXTERNAL MEMORY WAIT STATE CONTROL (WAIT)

The WAIT register is used to set up external memory wait states and response to the ACK signal. WAIT is memory-mapped in internal memory at address 0x0002. The WAIT register is initialized to 0x21AD 6B5A after a processor reset; this configures the ADSP-2106x for 1) no idle state on PAGE boundary crossings, 2) six internal wait states, 3) dependence on ACK for all memory banks and for unbanked memory, and 4) multiprocessor memory space wait state enabled.

Bit	*Name*	*Definition*
0-1	EB0WM	External Bank 0 wait state mode
2-4	EB0WS	External Bank 0 number of wait states
5-6	EB1WM	External Bank 1 wait state mode
7-9	EB1WS	External Bank 1 number of wait states
10-11	EB2WM	External Bank 2 wait state mode
12-14	EB2WS	External Bank 2 number of wait states
15-16	EB3WM	External Bank 3 wait state mode
17-19	EB3WS	External Bank 3 number of wait states
20-21	UBWM	Unbanked memory wait state mode*
22-24	UBWS	Unbanked memory number of wait states*
25-27	PAGSZ	Page size for DRAM (only in Bank 0)
28	PAGEIS	Single idle cycle on DRAM page boundary crossing
29	MMSWS	Single wait state for Multiprocessor Memory Space access
30	HIDMA	Single idle cycle for DMA handshake
31		*reserved*

* Unbanked memory wait states and wait state mode are applied to $\overline{\text{BMS}}$-asserted accesses.

Wait State Mode	
EBxWM	*Wait State Mode*
00	External acknowledge only (ACK)
01	Internal wait states only
10	Both internal and external acknowledge required
11	Either internal or external acknowledge sufficient

Number of Wait States			
EBxWS	*# of Wait States*	*Bus Idle Cycle?*	*Hold Time Cycle?*
000	0	no	no
001	1	yes	no
010	2	yes	no
011	3	yes	no
100	4	no	yes
101	5	no	yes
110	6	no	yes
111	0	yes	no

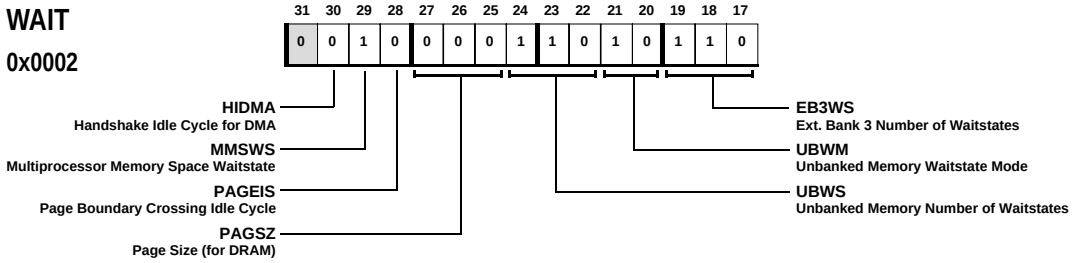
Bus Idle Cycle – inactive bus cycle automatically generated to avoid bus driver conflicts; devices with slow disable times should enable bus idle cycle generation by using *# of wait states* code 001, 010, or 011.

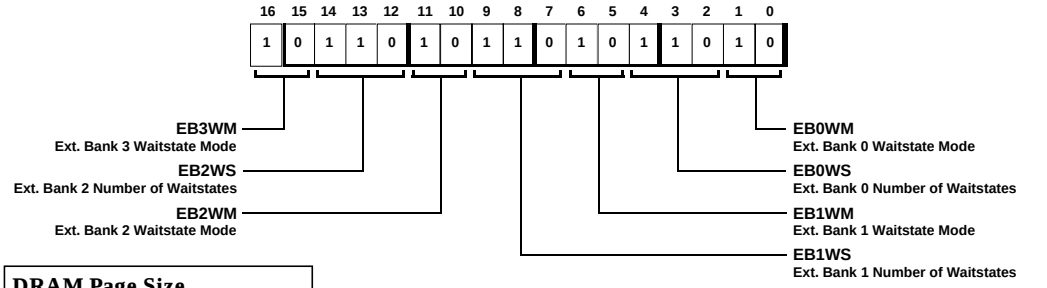
Hold Time Cycle – inactive bus cycle automatically generated at the end of a read or write to allow a longer hold time for address and data; the address and data will remain unchanged and driven for one cycle after the read or write strobes are deasserted

Note that the bus idle cycle or hold time cycles will occur if programmed, regardless of the waitstate mode. For example, the ACK-only waitstate mode may have a hold time cycle programmed for it.

WAIT

0x0002





DRAM Page Size	
PAGSZ	*DRAM Page Size*
000	256 words
001	512 words
010	1024 words (1K)
011	2048 words (2K)
100	4096 words (4K)
101	8192 words (8K)
110	16384 words (16K)
111	32768 words (32K)

All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

E.12 EXTERNAL PORT DMA CONTROL (DMAC6-DMAC9)

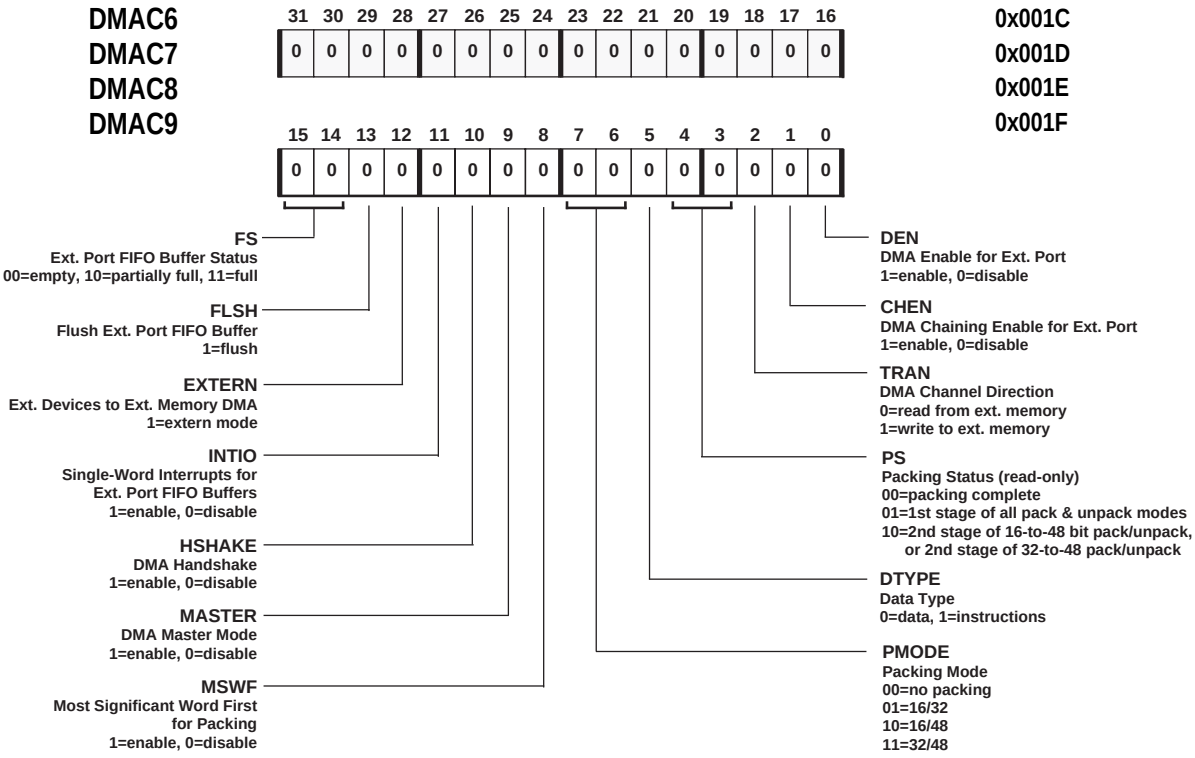
The DMAC6, DMAC7, DMAC8, and DMAC9 registers on the ADSP-21060 and ADSP-21062 are used to control external port DMA operations on DMA channels 6, 7, 8, and 9. (On the ADSP-21061, only DMAC6 and DMAC7 are available.) These registers are memory-mapped at internal memory addresses 0x001C, 0x001D, 0x001E, and 0x001F, respectively. After reset, the DMAC7, DMAC8, and DMAC9 registers are cleared (initialized to 0x0000 0000). DMAC6 is initialized during booting according to the booting mode used.

Bit	*Name*	*Definition*
0	DEN	DMA Enable for external port
1	CHEN	DMA Chaining Enable for external port
2	TRAN	DMA Transfer Direction (1=transmit, 0=receive)
3-4	PS	Pack Status (read-only)
5	DTYPE	Data Type (0=data, 1=instructions)
6-7	PMODE	Packing Mode (00=none, 01=16/32, 10=16/48, 11=32/48)
8	MSWF	Most-Significant-Word-First during packing
9	MASTER	DMA Master Mode Enable
10	HSHAKE	DMA Handshake (use DMARx pin to initiate DMA)
11	INTIO	Single-Word Interrupt Enable for external port buffers
12	EXTERN	Handshake DMA is ext. device to ext. memory
13	FLSH	Flush external port buffer (to empty status)
14-15	FS	External port buffer status (00=empty, 11=full, 10=partially full)
16-31		*reserved*

DEN **Ext. Port DMA Enable**—Enables DMA for the external port buffers. (Note that the DMA channels shared between the external port and link ports, channels 6 and 7, may also become enabled by the link buffers on the ADSP-21060 and ADSP-21062.)

CHEN **Ext. Port DMA Chaining Enable**—Enables chained DMA transfers. When CHEN=1 and DEN=0, the DMA channel is placed in *chain insertion* mode in which a new DMA chain can be inserted into the current chain without affecting the current DMA transfer. This mode of operation is identical to CHEN=1 and DEN=1 except that automatic chaining is disabled when the current DMA transfer ends. The complete list of modes selected by the CHEN and DEN bits are as follows:

CHEN	*DEN*	*Mode of Operation*
0	0	Chaining disabled, DMA disabled
0	1	Chaining disabled, DMA enabled
1	0	*Chain Insertion* mode (chaining enabled, DMA enabled, auto-chaining disabled)
1	1	Chaining enabled, DMA enabled, auto-chaining enabled



All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background; these bits should always be written with zeros.

The DMAC8 and DMAC9 registers are not available on the ADSP-21061.

TRAN **Transfer Direction**—Specifies the data transfer direction as internal-to-external when set to 1. (When EXTERN is selected, TRAN=1 specifies a read from external memory and TRAN=0 specifies a write to external memory.)

PS **Packing Status**—Indicates whether the packing buffer is on its first, second, or last packing stage:

PS	*Status*
00	packing complete
01	1st stage of all pack and unpack modes
10	2nd stage of 16-to-48 bit pack or unpack modes, or 2nd stage of 32-to-48 bit pack or unpack modes
11	*reserved*

DTYPE **Data Type**—Specifies the type of data being transferred; this information is used by internal memory to determine the word width. DTYPE=1 overrides the IMDW bits and forces a 48-bit (3-column) memory transfer. DTYPE=0 defers to the data word setting of the IMDW bits in the SYSCON register. The data word may be 32-bit or 40-bit, as determined by the setting of the IMDW bits in the SYSCON register.

PMODE **Packing Mode**—Specifies the EPBx buffer packing mode. For host processor accesses of the EPBx buffers, the HPM bits of the SYSCON register must be set to correspond to the external bus width specified by PMODE.

PMODE	*Packing Mode*
00	No packing / unpacking
01	16-bit external bus to / from 32-bit internal packing
10	16-bit external bus to / from 48-bit internal packing
11	32-bit external bus to / from 48-bit internal packing

MSWF **Most Significant Word First**—Specifies the order in which words are packed, for 16-to-32 bit packing and 16-to-48 bit packing. MSWF is ignored for 32-to-48 bit packing. When MSWF=1, packing is done MSW first (most significant 16-bit word first). When MSWF=0, packing is is done LSW first.

1=MSW first
0=LSW first

INTIO **Single-Word I/O Interrupts**—Used when DEN=0, to allow the external port DMA interrupts to occur for individual words received and transmitted. Generating DMA interrupts in this fashion is useful for implementing interrupt-driven single-word transfers under control of the ADSP-2106x core processor. Setting INTIO=1 causes the interrupts to occur when an EPBx input buffer is “not empty” (for TRAN=0) or when an output buffer is “not full” (for TRAN=1).

FLSH **Flush DMA Channel**—Reinitializes the state of the DMA channel, clearing the FS and PS status bits to zero. The EPBx buffer is flushed, and any internal DMA states are reset. This operation has a two-cycle latency. FLSH is a self-clearing control bit which is not latched and will always read as a 0.

The FLSH bit should only be used to clear the DMA channel when the channel is **not active**. *Use of the FLSH bit while the channel is active may cause unexpected results.* The DMASTAT register can be read to determine if the channel is active. (For a particular channel, the *channel active* status bit in DMASTAT will be set if DMA is enabled and the current DMA sequence has not completed.) Also, due to the

added latency of the FLSH bit, it should not be set in the same DMACx write in which the DEN bit is set. As a general rule, set FLSH at least one cycle before setting any other DMACx control bits.

FS **EPBx Buffer Status**—FS is a two-bit status field that indicates whether data is present in the EPBx buffer. When data is being transferred out from the ADSP-2106x, these status bits indicate whether there is room in the buffer for more data.
When data is being transferred into the ADSP-2106x, these status bits indicate whether new (unread) data is available in the buffer.

FS	*Status*
00	empty
01	*undefined*
10	partially full
11	full

MASTER **Master Mode DMA Enable**—The MASTER, HSHAKE, and EXTERN bits are used in combination, as described below.

HSHAKE **Handshake DMA Enable**—The MASTER, HSHAKE, and EXTERN bits are used in combination, as described below.

EXTERN **External Handshake Mode Enable**—Specifies an external memory to external device DMA transfer. HSHAKE must equal 1 and MASTER equal 0 in this mode.

The MASTER, HSHAKE, and EXTERN bits are used in combination to provide the following DMA transfer modes:

M	*H*	*E*	*DMA Mode of Operation*[1]
0	0	0	**Slave Mode.** The DMA request is generated whenever the receive buffer is not empty or the transmit buffer is not full.[2]
0	0	1	*Reserved*
0	1	0	**Handshake Mode.** (For the ADSP-21060 and ADSP-21062, applies to EPB1, EPB2 buffers, channels 7, 8 only. For the ADSP-21061, applies to EPB0, EPB1 buffers, channels 6, 7 only.) The DMA request is generated when the $\overline{\text{DMAR}}$x line is asserted. The transfer occurs when $\overline{\text{DMAG}}$x is asserted.[1]
0	1	1	**External Handshake Mode.** (For the ADSP-21060 and ADSP-21062, applies to EPB1, EPB2 buffers, channels 7, 8 only. For the ADSP-21061, applies to EPB0, EPB1 buffers, channels 6, 7 only.) Identical to Handshake Mode, but with data transferred between external memory and an external device.
1	0	0	**Master Mode.** The DMA controller will attempt a transfer whenever the receive buffer is not empty or the transmit buffer is not full and the DMA counter is non-zero.[1] $\overline{\text{DMAR}}$1 should be kept high (inactive) if channel 7 is in master mode, and $\overline{\text{DMAR}}$2 should be kept high if channel 8 is in master mode on the ADSP-21060 or ADSP-21062. $\overline{\text{DMAR}}$2 should be kept high if channel 6 is in master mode on the ADSP-21061.
1	0	1	*Reserved*
1	1	0	**Paced Master Mode.** (For the ADSP-21060 and ADSP-21062, applies to EPB1, EPB2 buffers, channels 7, 8 only. For the ADSP-21061, applies to EPB0, EPB1 buffers, channels 6, 7 only.) In this mode the transfers are paced by the $\overline{\text{DMAR}}$x signal—the DMA request is generated when $\overline{\text{DMAR}}$x is asserted. $\overline{\text{DMAR}}$x requests operate in the same way as in handshake mode. The bus transfer occurs when $\overline{\text{RD}}$ or $\overline{\text{WR}}$ is asserted. The address is driven as in normal master mode. No external gates are required to OR the $\overline{\text{RD}}$-$\overline{\text{DMAG}}$x and $\overline{\text{WR}}$-$\overline{\text{DMAG}}$x pairs, thus allowing the buffer access to be zero-waitstate with no idle states. Waitstates and acknowledge (ACK) apply to Paced Master Mode transfers; see Section 5.4.4, “Wait States & Acknowledge” in Chapter 5, *Memory*.
1	1	1	*Reserved*

1. When an external port DMA channel is configured for output (i.e., TRAN=1), the EPBx buffer will start to fill as soon as that DMA channel is enabled. The EPBx buffer will start to fill up even if no DMAR assertions or slave mode DMA buffer reads have been made yet.
2. If data is to be read from the ADSP-2106x (i.e. TRAN=1), the EPBx buffer will be filled as soon as the DEN enable bit is set to 1.

E.13 DMA CHANNEL STATUS (DMASTAT)

The DMASTAT register maintains status bits for each DMA channel. This register is memory-mapped at internal memory addresses 0x0037. For a particular channel, the *channel active* status bit will be set if DMA is enabled and the current DMA sequence has not completed. The *chaining* status bit will be set if the channel is currently performing chaining operations or if chaining is pending. There is a single cycle of latency between internal status changes and the update of the DMASTAT register.

Bit	*Definition*
0	DMA Channel 0 Status[1]
1	DMA Channel 1 Status[1]
2	DMA Channel 2 Status[1]
3	DMA Channel 3 Status[1]
4	DMA Channel 4 Status[1, 2]
5	DMA Channel 5 Status[1, 2]
6	DMA Channel 6 Status[1]
7	DMA Channel 7 Status[1]
8	DMA Channel 8 Status[1]
9	DMA Channel 9 Status[1]
10	DMA Channel 0 Chaining Status[3]
11	DMA Channel 1 Chaining Status[3]
12	DMA Channel 2 Chaining Status[3]
13	DMA Channel 3 Chaining Status[3]
14	DMA Channel 4 Chaining Status[2, 3]
15	DMA Channel 5 Chaining Status[2, 3]
16	DMA Channel 6 Chaining Status[3]
17	DMA Channel 7 Chaining Status[3]
18	DMA Channel 8 Chaining Status[3]
19	DMA Channel 9 Chaining Status[3]
20-31	*undefined/reserved*

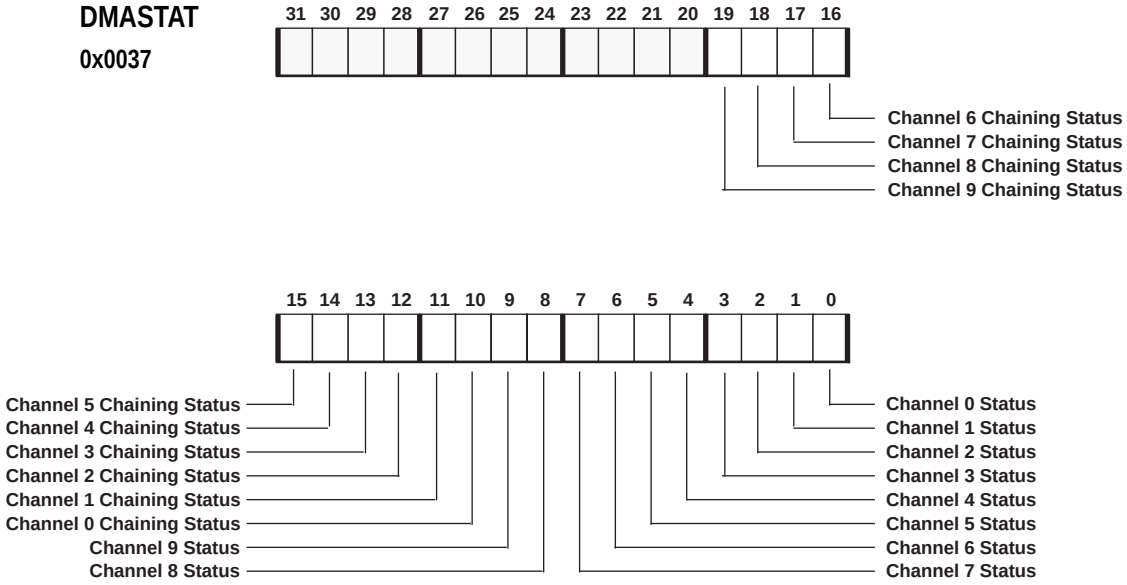
1. **Channel Status: 1 (active)**=transferring data or waiting to transfer the current block, and not transferring TCB. **0 (inactive)**=DMA disabled, transfer complete, or chaining.
2. **Not valid for the ADSP-21061.**
3. **Channel Chaining Status: 1=**transferring TCB or waiting to transfer TCB. **0**=chaining disabled.

Note 1: Status does not change on the master ADSP-2106x during external port DMA until the external portion is completed (i.e., the EPBx buffers are emptied).

Note 2: If in chain insertion mode (DEN=0, CHEN=1), then *channel chaining status* will never go to 1. Therefore, test *channel status* to see if it is ready so that your program can rewrite the chain pointer (CPx register).

DMASTAT

0x0037



1. **Channel Status:**
   1 (active) = transferring data or waiting to transfer the current block, and not transferring TCB.
   0 (inactive) = DMA disabled, transfer complete, or chaining.

2. **Channel Chaining Status:**
   1 (active) = transferring TCB or waiting to transfer TCB.
   0 (inactive) = chaining disabled.

3. Status does not change on the master ADSP-2106x during external port DMA until the external portion is completed (i.e., the EPBx buffers are emptied).

4. If in chain insertion mode (DEN=0, CHEN=1), then *channel chaining status* will never go to 1. Therefore, test *channel status* to see if it is ready so that your program can rewrite the chain pointer (CPx register).

5. On the ADSP-21061, the DMASTAT register bits for channels 4, 5, 8, and 9 are not valid. These bits include bits: 4, 5, 8, 9, 14, 15, 18, and 19.

All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

E.14 LINK BUFFER CONTROL (LCTL)

LCTL is the main control register for the six link port data buffers (LBUF0-5). [This register is not available on the ADSP-21061.] The LCTL register contains control bits unique to each link buffer. LCTL is memory-mapped at address 0x00C6. After reset, LCTL is cleared (initialized to 0x0000 0000).

Bit(s)	*Name*	*Definition*
0-3	*	Link Buffer 0 control bits
4-7	*	Link Buffer 1 control bits
8-11	*	Link Buffer 2 control bits
12-15	*	Link Buffer 3 control bits
16-19	*	Link Buffer 4 control bits
20-23	*	Link Buffer 5 control bits
24	LEXT0	Extended word size
25	LEXT1	Extended word size
26	LEXT2	Extended word size
27	LEXT3	Extended word size
28	LEXT4	Extended word size
29	LEXT5	Extended word size
30-31	*reserved*	

* Each four-bit group includes the following control bits for each link buffer (x=0,1,2,3,4,5):

Bit#	*Name*	*Definition*
0+4x	LxEN	LBUFx enable
1+4x	LxDEN	LBUFx DMA enable
2+4x	LxCHEN	LBUFx DMA chaining enable
3+4x	LxTRAN	LBUFx direction: 1=transmit, 0=receive

LxEN **Link Buffer Enable**—Enables / disables the link buffer. When the buffer is disabled, the assigned link port will deassert LxACK if receiving and LxCLK if transmitting. LxSTAT and LRERR are cleared when LxEN transitions from high to low.

LxDEN **Link Buffer DMA Enable**—Enables the associated DMA channel.

LxCHEN **Link Buffer DMA Chaining Enable**—DMA chaining enable for that channel.

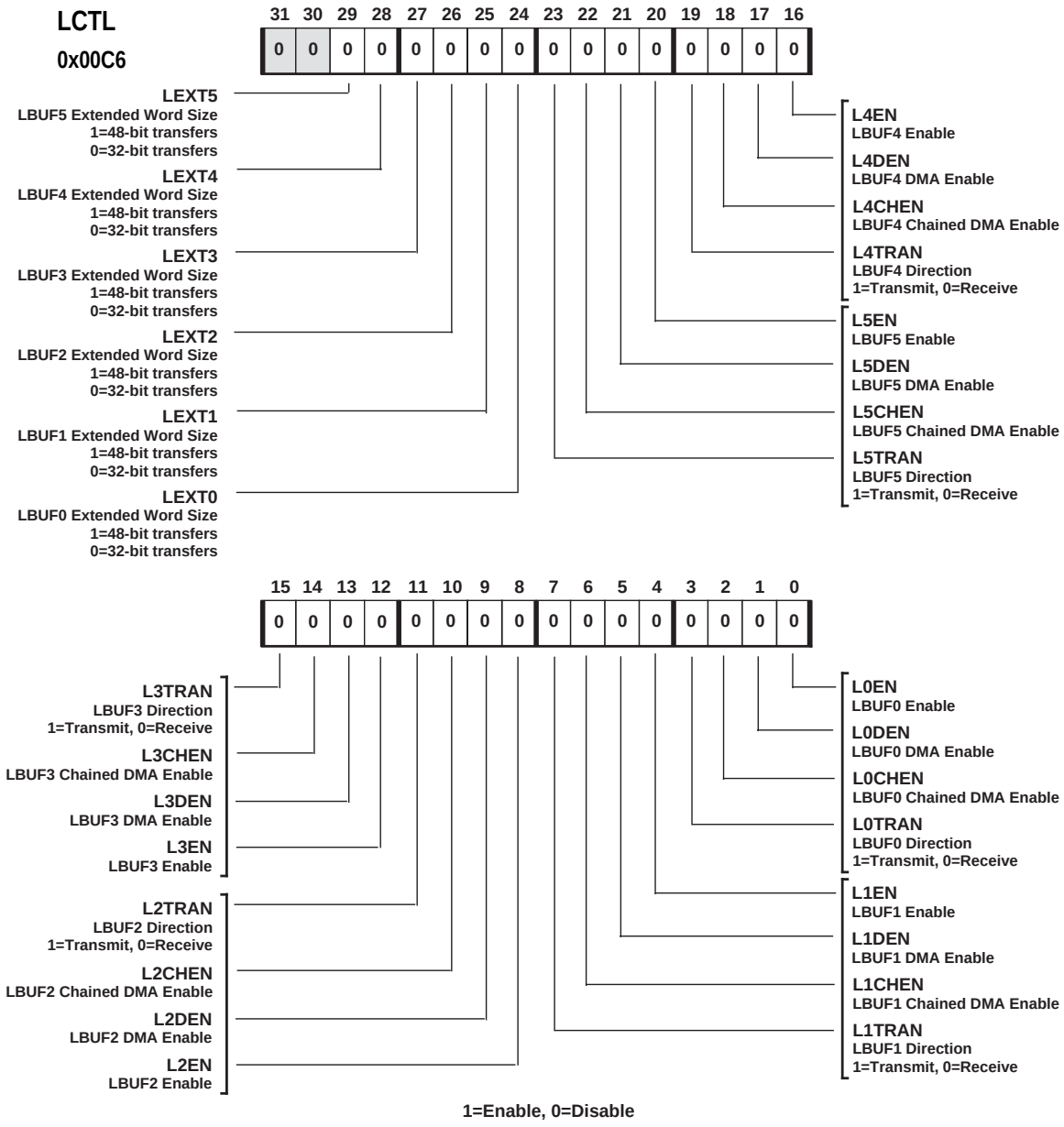
LxTRAN **Link Port Transmit/Receive Select**—Selects the direction of the link buffer, link port and DMA channel: 0 to receive data, 1 to transmit data.

LEXTx **Extended Word Size**—Specifies word size for link port transfers:

LEXTx=1 specifies 48-bit transfers in link buffer x
LEXTx=0 specifies 32-bit transfers in link buffer x

The LEXTx bits override the setting of the IMDW memory word width bits in SYSCON. If LEXTx=1, data to be transmitted will be read from 48-bit word space in memory, regardless of the setting of IMDW.

E Control/Status Registers



All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

E.15 LINK BUFFER COMMON CONTROL (LCOM)

The LCOM register contains status bits for each buffer, functions common to all links, and mesh multiprocessing functions. [This register is not available on the ADSP-21061.] LCOM is memory-mapped at address 0x00C7. After reset, LCOM is cleared (initialized to 0x0000 0000). All status bits are read-only.

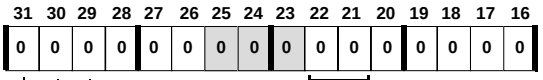
Bit(s)	*Name*	*Definition*
0-1	L0STAT	Link Buffer 0 status. 11=full, 00=empty,10=one word *
2-3	L1STAT	Link Buffer 1 status. 11=full, 00=empty,10=one word *
4-5	L2STAT	Link Buffer 2 status. 11=full, 00=empty,10=one word *
6-7	L3STAT	Link Buffer 3 status. 11=full, 00=empty,10=one word *
8-9	L4STAT	Link Buffer 4 status. 11=full, 00=empty,10=one word *
10-11	L5STAT	Link Buffer 5 status. 11=full, 00=empty,10=one word *
12	LCLKX20	Transfer data at 2x clock rate on Link Buffer 0
13	LCLKX21	Transfer data at 2x clock rate on Link Buffer 1
14	LCLKX22	Transfer data at 2x clock rate on Link Buffer 2
15	LCLKX23	Transfer data at 2x clock rate on Link Buffer 3
16	LCLKX24	Transfer data at 2x clock rate on Link Buffer 4
17	LCLKX25	Transfer data at 2x clock rate on Link Buffer 5
18	L2DDMA**	Enable 2-dimensional DMA
19	LPDRD**	Disable internal pulldown resistor for LxCLK and LxACK
20	LMSP**	Mesh multiprocessing coupled transmission
21-22	LPATHD**	Mesh multiprocessing LPATH change-over delay: 00=no additional delay, 01=1 additional delay, 10=2 additional delays, 11=3 additional delays
23-25	*reserved*	
26	LRERR0	Receive pack error status for Link Buffer 0: 1=incomplete, 0=complete
27	LRERR1	Receive pack error status for Link Buffer 1: 1=incomplete, 0=complete
28	LRERR2	Receive pack error status for Link Buffer 2: 1=incomplete, 0=complete
29	LRERR3	Receive pack error status for Link Buffer 3: 1=incomplete, 0=complete
30	LRERR4	Receive pack error status for Link Buffer 4: 1=incomplete, 0=complete
31	LRERR5	Receive pack error status for Link Buffer 5: 1=incomplete, 0=complete

* The code 01 does not appear as a valid status.
** Common to all link ports.

Status bits are read-only.

LCOM

0x00C7



LRERR5
Rcv. Pack Error Status for Link Buffer 5
1=incomplete, 0=complete

LRERR4
Rcv. Pack Error Status for Link Buffer 4
1=incomplete, 0=complete

LRERR3
Rcv. Pack Error Status for Link Buffer 3
1=incomplete, 0=complete

LRERR2
Rcv. Pack Error Status for Link Buffer 2
1=incomplete, 0=complete

LRERR1
Rcv. Pack Error Status for Link Buffer 1
1=incomplete, 0=complete

LRERR0
Rcv. Pack Error Status for Link Buffer 0
1=incomplete, 0=complete

LCLKX24
Transfer at 2x Clock Rate
on Link Buffer 4

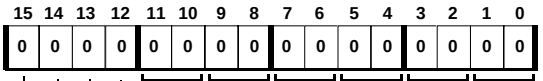
LCLKX25
Transfer at 2x Clock Rate
on Link Buffer 5

L2DDMA
Enable 2-D DMA

LPDRD
Link Pull-Down Resistor Disable
1=disable, 0=enable

LMSP
Mesh Multiproc.-Coupled Transmission

LPATHD
Mesh Multiproc. LPATH Changeover Delay
00=no additional delay, 01=one additional delay
10=two additional delays, 11=three additional delays



LCLKX23
Transfer at 2x Clock Rate
on Link Buffer 3

LCLKX22
Transfer at 2x Clock Rate
on Link Buffer 2

LCLKX21
Transfer at 2x Clock Rate
on Link Buffer 1

LCLKX20
Transfer at 2x Clock Rate
on Link Buffer 0

L0STAT
Link Buffer 0 Status (read-only)
11=full, 00=empty, 10=one word

L1STAT
Link Buffer 1 Status (read-only)
11=full, 00=empty, 10=one word

L2STAT
Link Buffer 2 Status (read-only)
11=full, 00=empty, 10=one word

L3STAT
Link Buffer 3 Status (read-only)
11=full, 00=empty, 10=one word

L4STAT
Link Buffer 4 Status (read-only)
11=full, 00=empty, 10=one word

L5STAT
Link Buffer 5 Status (read-only)
11=full, 00=empty, 10=one word

All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

LxSTAT(0:1) **Link Buffer Status**—When transmitting, these status bits indicate whether there is room in the buffer for more data. When receiving, these status bits indicate whether new (unread) data is available in the receive buffer. LxSTAT(1)=1 if there is data in the buffer. LxSTAT(0)=0 if there is room in the buffer. These bits are cleared when LxEN changes from 1 to 0. They may subsequently change state when the data buffer is read or written.

LCLKX2x **2x Clock Rate**—This specifies link transfers to operate at twice the ADSP-2106x clock rate. If LCLKX2x=0, transmit transfers occur at the ADSP-2106x clock frequency, and receive transfers occur at (up to) the ADSP-2106x clock frequency. Set LCLKX2x=1 for receive transfers occurring at greater than the ADSP-2106x clock frequency.

L2DDMA **2-D DMA Enable**—This directs the DMA controller to address memory as a two-dimensional array as specified in the DMA address registers. Only DMA channels 0-5 support 2D DMA. Link ports 4 and 5 on DMA channels 6 and 7 do not have 2D DMA support.

LPDRD **Disable Pulldown Resistors**—This disables the internal 50 kΩ pulldown resistors on the LxACK, LxCLK, and $LxDAT_{3-0}$ pins for disabled link ports. When this bit is clear, the pulldown resistors are enabled.

LMSP **Mesh Multiprocessing Enable**—Enables link port mesh multiprocessing. In non-mesh-multiprocessing operation (LMSP=0), all six link buffers operate independently and the LPATH registers are not part of the chaining.

LPATHD **Mesh Multiprocessing LPATH Changeover Delay**—In a mesh multiprocessing application, this selection allows 1, 2 or 3 additional clock delays to be inserted before changing to the next LPATH register.

LRERRx **Receive Pack Error Status**—These bits reflects the status of the receive nibble packer for each link buffer. LRERRx will equal 0 when the nibble packer is set to start receiving a new word. Otherwise it will be 1. If this bit is equal to 1 after a word is received, then an error has occurred (e.g. clock glitch). The LRERRx bits are cleared when LxEN changes from 1 to 0. They may subsequently change state when the link buffer is read or written or while a word is being received.

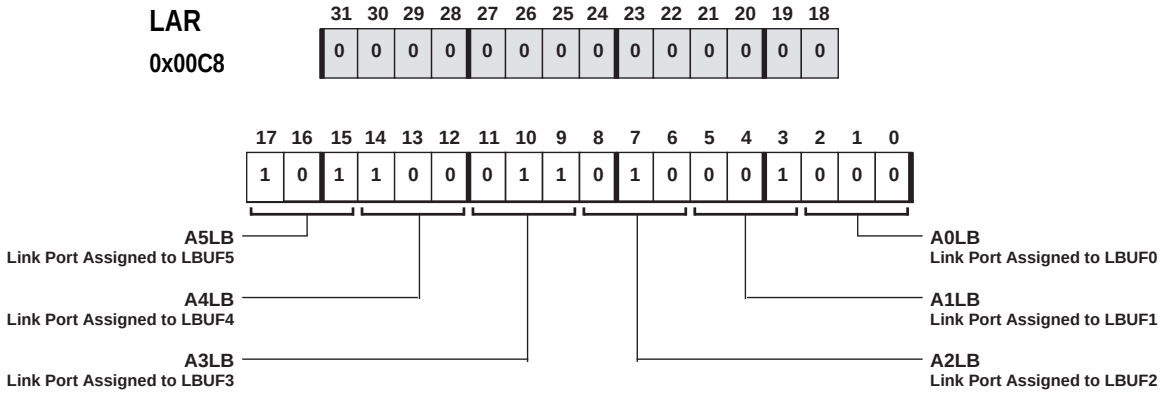
E.16 LINK ASSIGNMENT REGISTER (LAR)

The LAR register is used to select link port to link buffer connections. LAR is memory-mapped at address 0x00C8. [This register is not available on the ADSP-21061.] After reset LAR is initialized to 0x0002 C688, assigning Link Port 0 to Link Buffer 0, Link Port 1 to Link Buffer 1, Link Port 2 to Link Buffer 2, Link Port 3 to Link Buffer 3, Link Port 4 to Link Buffer 4, and Link Port 5 to Link Buffer 5.

Bits	*Name*	*Description*
0-2	A0LB*	Link port assignment for LBUF0
3-5	A1LB*	Link port assignment for LBUF1
6-8	A2LB*	Link port assignment for LBUF2
9-11	A3LB*	Link port assignment for LBUF3
12-14	A4LB*	Link port assignment for LBUF4
15-17	A5LB*	Link port assignment for LBUF5
18-31	*reserved*	

*

AxLB	*Link Port #*
000	Link Port 0
001	Link Port 1
010	Link Port 2
011	Link Port 3
100	Link Port 4
101	Link Port 5
110	*reserved*
111	inactive buffer



All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

E.17 LINK SERVICE REQUEST (LSRQ)

The LSRQ register indicates when a disabled link port is accessed from an external source. It also contains mask bits for these interrupts. [This register is not available on the ADSP-21061.] LSRQ is memory-mapped at address 0x00C9. After reset LSRQ is initialized to 0x0000 0000.

Bit	*Name*	*Description*
0-3	*reserved*	
4	L0TM	Link Port 0 transmit mask
5	L0RM	Link Port 0 receive mask
6	L1TM	Link Port 1 transmit mask
7	L1RM	Link Port 1 receive mask
8	L2TM	Link Port 2 transmit mask
9	L2RM	Link Port 2 receive mask
10	L3TM	Link Port 3 transmit mask
11	L3RM	Link Port 3 receive mask
12	L4TM	Link Port 4 transmit mask
13	L4RM	Link Port 4 receive mask
14	L5TM	Link Port 5 transmit mask
15	L5RM	Link Port 5 receive mask
16-19	*reserved*	
20	L0TRQ	Link Port 0 transmit request status *(read-only)*
21	L0RRQ	Link Port 0 receive request status *(read-only)*
22	L1TRQ	Link Port 1 transmit request status *(read-only)*
23	L1RRQ	Link Port 1 receive request status *(read-only)*
24	L2TRQ	Link Port 2 transmit request status *(read-only)*
25	L2RRQ	Link Port 2 receive request status *(read-only)*
26	L3TRQ	Link Port 3 transmit request status *(read-only)*
27	L3RRQ	Link Port 3 receive request status *(read-only)*
28	L4TRQ	Link Port 4 transmit request status *(read-only)*
29	L4RRQ	Link Port 4 receive request status *(read-only)*
30	L5TRQ	Link Port 5 transmit request status *(read-only)*
31	L5RRQ	Link Port 5 receive request status *(read-only)*

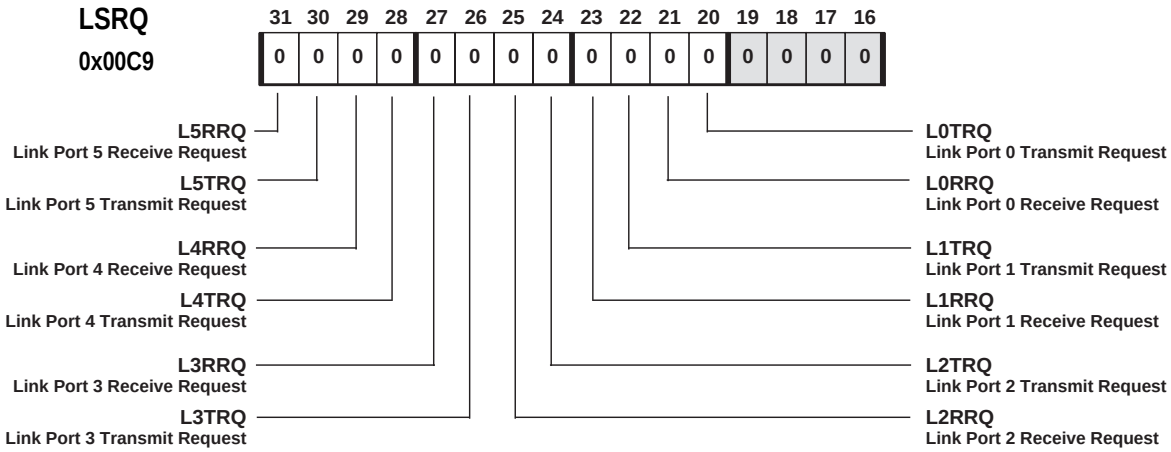
For *transmit request status* bits, LxTRQ=1 means LxACK=1.

For *receive request status* bits, LxRRQ=1 means LxCLK=1.

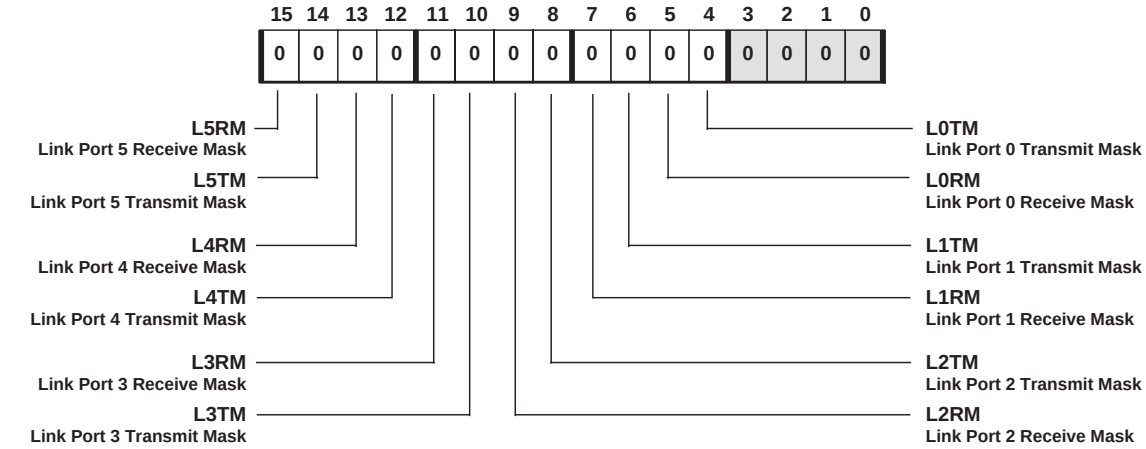
igned to LBUF0

igned to LBUF1

igned to LBUF2



Request Bits are Read-Only Status



All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

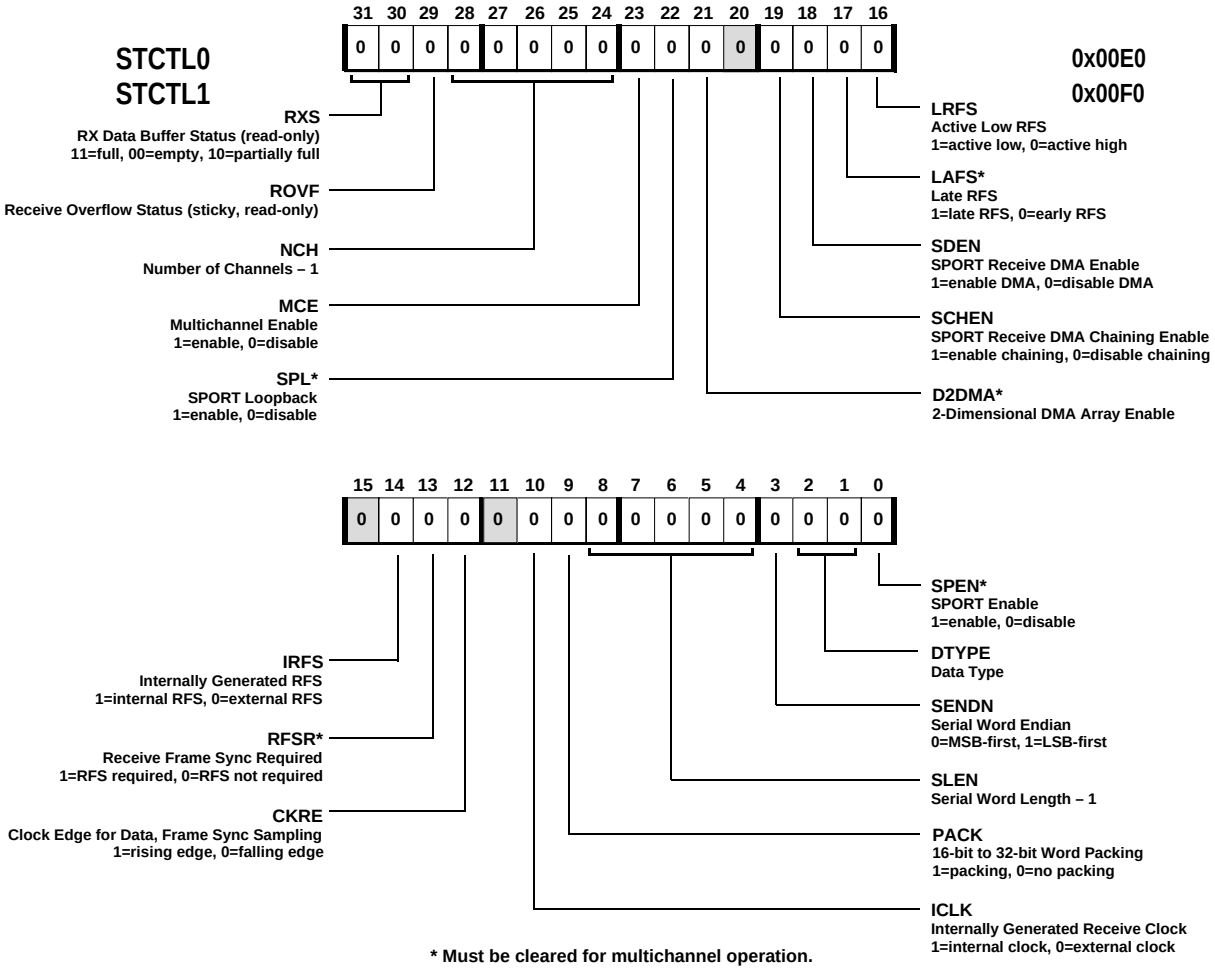
E.18 SPORT TRANSMIT CONTROL (STCTL0, STCTL1)

STCTL0 and STCTL1 are the transmit control registers for SPORT0 and SPORT1 respectively. STCTL0 is memory-mapped at address 0x00E0, and STCTL1 is memory-mapped at address 0x00F0. After reset, these registers are cleared (initialized to 0x0000 0000). When changing operating modes, a serial port control register should be cleared (i.e. written with all zeros) before the new mode is written to the register.

Bit(s)	*Name*	*Definition*
0	SPEN*	SPORT Enable (1=enable, 0=disable)
1-2	DTYPE	Data Type
3	SENDN	Serial Word Endian (0=MSB-first, 1=LSB-first)
4-8	SLEN	Serial Word Length – 1
9	PACK	16-bit to 32-bit Word Packing (1=packing, 0=no packing)
10	ICLK*	Internally Generated Transmit Clock (1=int. clock, 0=ext. clock)
11	–	*reserved*
12	CKRE	Clock Edge for Data, Frame Sync Sampling (1=rising edge, 0=falling edge)
13	TFSR*	Transmit Frame Sync Required (1=required, 0=not required)
14	ITFS*	Internally Generated TFS (1=internal TFS, 0=external TFS)
15	DITFS	Data-Independent TFS (1=continuous, data-independent TFS, 0=data-dependent TFS)
16	LTFS	Active Low TFS (1=active low, 0=active high)
17	LAFS*	Late TFS (1=late TFS, 0=early TFS)
18	SDEN	SPORT Transmit DMA Enable (1=enable, 0=disable)
19	SCHEN	SPORT Transmit DMA Chaining Enable (1=enable, 0=disable)
20-23	MFD	Multichannel Frame Delay
24-28	CHNL	Current Channel Selected (read-only)
29	TUVF	Transmit Underflow Status (sticky, read-only)
30-31	TXS	TX Data Buffer Status (read-only) (11=full, 00=empty, 10=partially full)

* Must be cleared for multichannel operation.

** Status bits are read-only. They are cleared by disabling the serial port (setting SPEN=0). TXS may subsequently change state if the data is read or written by the ADSP-2106x core while the SPORT is disabled.



*** Must be cleared for multichannel operation.**

DTYPE: Normal Operation (Non-Multichannel)

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

DTYPE: Multichannel Operation

DTYPE	*Data Formatting*
x0	Right-justify, zero-fill unused MSBs
x1	Right-justify, sign-extend into unused MSBs
0x	Compand using μ-law
1x	Compand using A-law

All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

E.19 SPORT RECEIVE CONTROL (SRCTL0, SRCTL1)

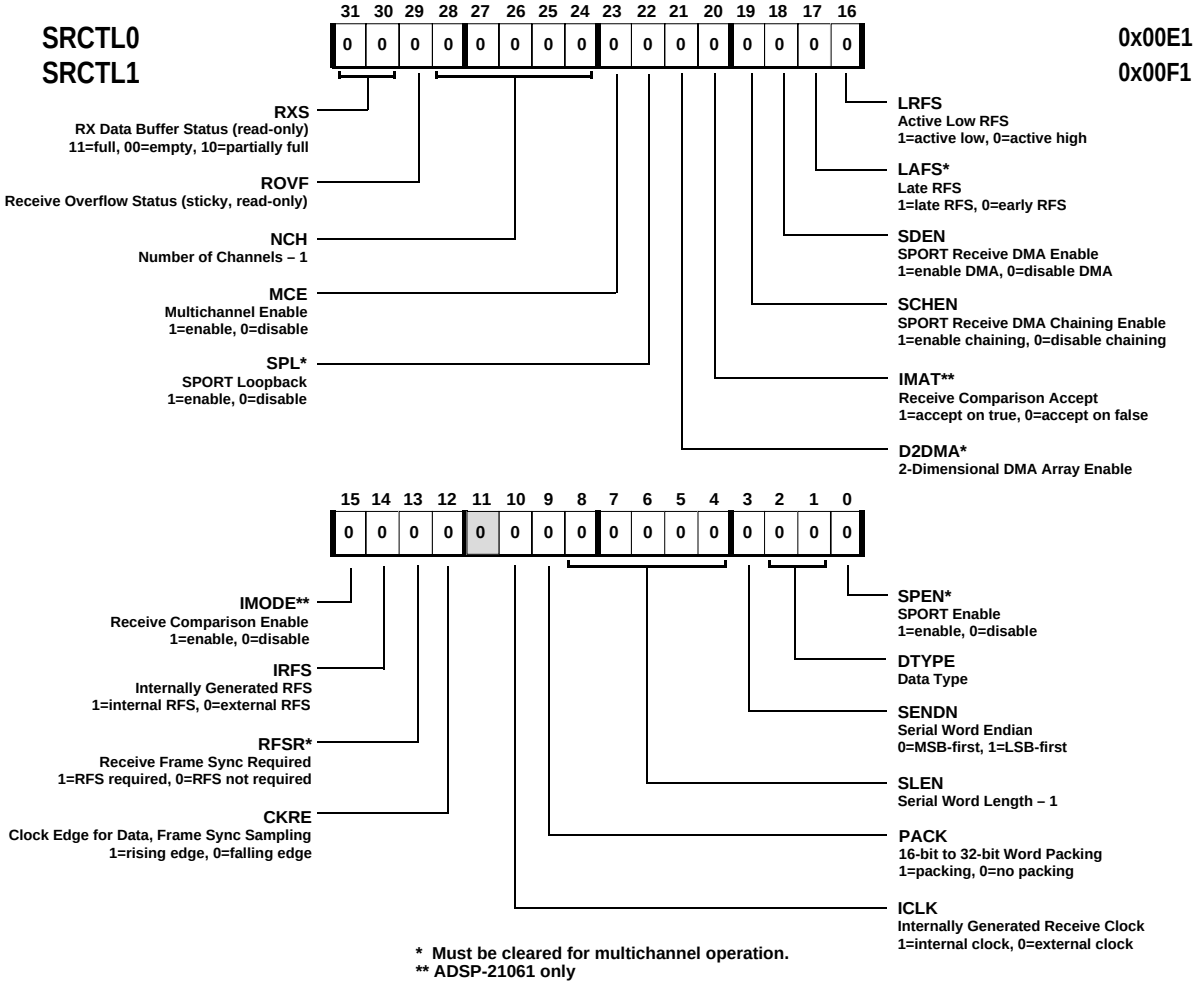
SRCTL0 and SRCTL1 are the transmit control registers for SPORT0 and SPORT1 respectively. SRCTL0 is memory-mapped at address 0x00E1 and SRCTL1 is memory-mapped at address 0x00F1. After reset, these registers are cleared (initialized to 0x0000 0000). When changing operating modes, a serial port control register should be cleared (i.e. written with all zeros) before the new mode is written to the register.

Bit(s)	*Name*	*Definition*
0	SPEN*	SPORT Enable (1=enable, 0=disable)
1-2	DTYPE	Data Type
3	SENDN	Serial Word Endian (0=MSB-first, 1=LSB-first)
4-8	SLEN	Serial Word Length – 1
9	PACK	16-bit to 32-bit Word Packing (1=packing, 0=no packing)
10	ICLK	Internally Generated Receive Clock (1=int. clock, 0=ext. clock)
11	–	*reserved*
12	CKRE	Clock Edge for Data, Frame Sync Sampling (1=rising edge, 0=falling edge)
13	RFSR*	Receive Frame Sync Required (1=required, 0=not required)
14	IRFS	Internally Generated RFS (1=internal RFS, 0=external RFS)
15	–	*reserved*
16	LRFS	Active Low RFS (1=active low, 0=active high)
17	LAFS*	Late RFS (1=late RFS, 0=early RFS)
18	SDEN	SPORT Receive DMA Enable (1=enable, 0=disable)
19	SCHEN	SPORT Receive DMA Chaining Enable (1=enable, 0=disable)
20	–	*reserved*
21	D2DMA*	2-Dimensional DMA Array Enable
22	SPL*	SPORT Loopback (1=enable, 0=disable)
23	MCE	Multichannel Enable (1=enable, 0=disable)
24-28	NCH	Number of Channels – 1
29	ROVF	Receive Overflow Status (sticky, read-only)
30-31	RXS	RX Data Buffer Status (read-only) (11=full, 00=empty, 10=partially full)

* Must be cleared for multichannel operation.

** Status bits are read-only. They are cleared by disabling the serial port (setting SPEN=0). RXS may subsequently change state if the data is read or written by the ADSP-2106x core while the SPORT is disabled.

E Control/Status Registers



*** Must be cleared for multichannel operation.**
**** ADSP-21061 only**

DTYPE: Normal Operation (Non-Multichannel)	
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

DTYPE: Multichannel Operation	
DTYPE	*Data Formatting*
x0	Right-justify, zero-fill unused MSBs
x1	Right-justify, sign-extend into unused MSBs
0x	Compand using μ-law
1x	Compand using A-law

All control and status bits are active high unless otherwise noted. Default bit values after reset are shown; if no value is shown, the bit is undefined at reset or depends upon processor inputs. Reserved bits are shown with a gray background. *Reserved bits should always be written with zeros.*

0x00E1
0x00F1

E.20 SPORT DIVISORS (TDIV, RDIV)

The TDIV0, TDIV1, RDIV0, and RDIV1 registers contain divisor values that determine the frequencies for internally generated serial port clocks and frame syncs. These four registers are memory-mapped at addresses 0x00E4, 0x00F4, 0x00E6, and 0x00F6 respectively. These registers are not initialized after reset.

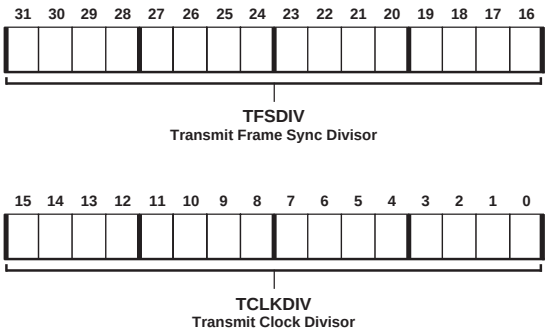
TDIVx

Bits	*Name*	*Definition*
15-0	TCLKDIV	Transmit Clock Divisor
31-16	TFSDIV	Transmit Frame Sync Divisor

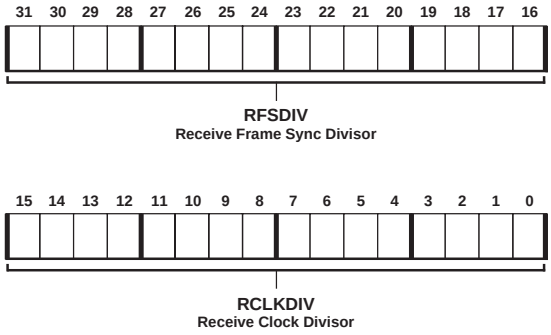
RDIVx

Bits	*Name*	*Definition*
15-0	RCLKDIV	Receive Clock Divisor
31-16	RFSDIV	Receive Frame Sync Divisor

TDIV0 **0x00E4**
TDIV1 **0x00F4**



RDIV0 **0x00E6**
RDIV1 **0x00F6**



$$\text{xCLKDIV} = \frac{f_{CLKIN}}{\text{serial clock frequency}} - 1$$

$$\text{xFSDIV} = \frac{\text{serial clock frequency}}{\text{frame sync frequency}} - 1$$

E.21 SYMBOL DEFINITIONS FILE (def21060.h)

The IOP registers are programmed by writing to the appropriate address in memory. The symbolic names of the registers and individual bits can be used in ADSP-2106x programs—the `#define` definitions for these symbols are contained in the file `def21060.h` which is provided in the INCLUDE directory of the ADSP-21000 Family Development Software. The `def21060.h` file is shown here for reference.

```
/* ————————————————————————————————————————————————————————————————
def21060.h - SYSTEM & IOP REGISTER BIT & ADDRESS DEFINITIONS FOR ADSP-2106x

Last Modification on: Feb-15-95

This include file contains a list of macro defines to enable the programmer
to use symbolic names for all of the system register bits for the ADSP-2106x.
It also contains macros for the IOP register addresses and some bit fields.
Here are some example uses:

   bit set mode1 BR0|IRPTEN|ALUSTAT;

   ustat1=BSO|HPM01|HMSWF;
   dm(SYSCON)=ustat1;

———————————————————————————————————————————————————————————————— */


/* MODE1 register */
#define BR8      0x00000001 /* Bit  0: Bit-reverse for I8                      */
#define BR0      0x00000002 /* Bit  1: Bit-reverse for I0 (uses DMS0- only )   */
#define SRCU     0x00000004 /* Bit  2: Alt. register select for comp. units    */
#define SRD1H    0x00000008 /* Bit  3: DAG1 alt. register select (7-4)         */
#define SRD1L    0x00000010 /* Bit  4: DAG1 alt. register select (3-0)         */
#define SRD2H    0x00000020 /* Bit  5: DAG2 alt. register select (15-12)       */
#define SRD2L    0x00000040 /* Bit  6: DAG2 alt. register select (11-8)        */
#define SRRFH    0x00000080 /* Bit  7: Register file alt. select for R(15-8)   */
#define SRRFL    0x00000400 /* Bit 10: Register file alt. select for R(7-0)    */
#define NESTM    0x00000800 /* Bit 11: Interrupt nesting enable                */
#define IRPTEN   0x00001000 /* Bit 12: Global interrupt enable                 */
#define ALUSAT   0x00002000 /* Bit 13: Enable ALU fixed-pt. saturation         */
#define SSE      0x00004000 /* Bit 14: Enable short word sign extension        */
#define TRUNCATE 0x00008000 /* Bit 15: 1=fltg-pt. truncation 0=Rnd to nearest  */
#define RND32    0x00010000 /* Bit 16: 1=32-bit fltg-pt.rounding 0=40-bit rnd  */
#define CSEL     0x00060000 /* Bit 17-18: CSelect: Bus Mastership             */
```

```
/* MODE2 register */
#define IRQ0E   0x00000001 /* Bit  0: IRQ0- 1=edge sens. 0=level sens. */
#define IRQ1E   0x00000002 /* Bit  1: IRQ1- 1=edge sens. 0=level sens. */
#define IRQ2E   0x00000004 /* Bit  2: IRQ2- 1=edge sens. 0=level sens. */
#define CADIS   0x00000010 /* Bit  4: Cache disable                    */
#define TIMEN   0x00000020 /* Bit  5: Timer enable                     */
#define BUSLK   0x00000040 /* Bit  6: External bus lock                */
#define FLG0O   0x00008000 /* Bit 15: FLAG0 1=output 0=input           */
#define FLG1O   0x00010000 /* Bit 16: FLAG1 1=output 0=input           */
#define FLG2O   0x00020000 /* Bit 17: FLAG2 1=output 0=input           */
#define FLG3O   0x00040000 /* Bit 18: FLAG3 1=output 0=input           */
#define CAFRZ   0x00080000 /* Bit 19: Cache freeze                     */


/* ASTAT register */
#define AZ      0x00000001 /* Bit  0: ALU result zero or fltg-pt underflow */
#define AV      0x00000002 /* Bit  1: ALU overflow                         */
#define AN      0x00000004 /* Bit  2: ALU result negative                  */
#define AC      0x00000008 /* Bit  3: ALU fixed-pt. carry                  */
#define AS      0x00000010 /* Bit  4: ALU X input sign (ABS and MANT ops)  */
#define AI      0x00000020 /* Bit  5: ALU fltg-pt. invalid operation       */
#define MN      0x00000040 /* Bit  6: Multiplier result negative           */
#define MV      0x00000080 /* Bit  7: Multiplier overflow                  */
#define MU      0x00000100 /* Bit  8: Multiplier fltg-pt. underflow        */
#define MI      0x00000200 /* Bit  9: Multiplier fltg-pt. invalid operation*/
#define AF      0x00000400 /* Bit 10: ALU fltg-pt. operation               */
#define SV      0x00000800 /* Bit 11: Shifter overflow                     */
#define SZ      0x00001000 /* Bit 12: Shifter result zero                  */
#define SS      0x00002000 /* Bit 13: Shifter input sign                   */
#define BTF     0x00040000 /* Bit 18: Bit test flag for system registers   */
#define FLG0    0x00080000 /* Bit 19: FLAG0 value                          */
#define FLG1    0x00100000 /* Bit 20: FLAG1 value                          */
#define FLG2    0x00200000 /* Bit 21: FLAG2 value                          */
#define FLG3    0x00400000 /* Bit 22: FLAG3 value                          */
#define CACC0   0x01000000 /* Bit 24: Compare Accumulation Bit 0           */
#define CACC1   0x02000000 /* Bit 25: Compare Accumulation Bit 1           */
#define CACC2   0x04000000 /* Bit 26: Compare Accumulation Bit 2           */
#define CACC3   0x08000000 /* Bit 27: Compare Accumulation Bit 3           */
#define CACC4   0x10000000 /* Bit 28: Compare Accumulation Bit 4           */
#define CACC5   0x20000000 /* Bit 29: Compare Accumulation Bit 5           */
#define CACC6   0x40000000 /* Bit 30: Compare Accumulation Bit 6           */
#define CACC7   0x80000000 /* Bit 31: Compare Accumulation Bit 7           */
```

```
/* STKY register */
#define AUS     0x00000001 /* Bit  0: ALU fltg-pt. underflow                  */
#define AVS     0x00000002 /* Bit  1: ALU fltg-pt. overflow                   */
#define AOS     0x00000004 /* Bit  2: ALU fixed-pt. overflow                  */
#define AIS     0x00000020 /* Bit  5: ALU fltg-pt. invalid operation          */
#define MOS     0x00000040 /* Bit  6: Multiplier fixed-pt. overflow           */
#define MVS     0x00000080 /* Bit  7: Multiplier fltg-pt. overflow            */
#define MUS     0x00000100 /* Bit  8: Multiplier fltg-pt. underflow           */
#define MIS     0x00000200 /* Bit  9: Multiplier fltg-pt. invalid operation   */
#define CB7S    0x00020000 /* Bit 17: DAG1 circular buffer 7 overflow         */
#define CB15S   0x00040000 /* Bit 18: DAG2 circular buffer 15 overflow        */
#define PCFL    0x00200000 /* Bit 21: PC stack full                           */
#define PCEM    0x00400000 /* Bit 22: PC stack empty                          */
#define SSOV    0x00800000 /* Bit 23: Status stack overflow (MODE1 and ASTAT) */
#define SSEM    0x01000000 /* Bit 24: Status stack empty                      */
#define LSOV    0x02000000 /* Bit 25: Loop stack overflow                     */
#define LSEM    0x04000000 /* Bit 26: Loop stack empty                        */

/* IRPTL and IMASK and IMASKP registers */
#define RSTI    0x00000002 /* Bit  1: Offset: 04: Reset                       */
#define SOVFI   0x00000008 /* Bit  3: Offset: 0c: Stack overflow              */
#define TMZHI   0x00000010 /* Bit  4: Offset: 10: Timer = 0 (high priority)   */
#define VIRPTI  0x00000020 /* Bit  5: Offset: 14: Vector interrupt            */
#define IRQ2I   0x00000040 /* Bit  6: Offset: 18: IRQ2- asserted              */
#define IRQ1I   0x00000080 /* Bit  7: Offset: 1c: IRQ1- asserted              */
#define IRQ0I   0x00000100 /* Bit  8: Offset: 20: IRQ0- asserted              */
#define SPR0I   0x00000400 /* Bit 10: Offset: 28: SPORT0 receive DMA channel  */
#define SPR1I   0x00000800 /* Bit 11: Offset: 2c: SPORT1 receive (or LBUF0)   */
#define SPT0I   0x00001000 /* Bit 12: Offset: 30: SPORT0 transmit DMA channel */
#define SPT1I   0x00002000 /* Bit 13: Offset: 34: SPORT1 transmit (or LBUF0)  */
#define LP2I    0x00004000 /* Bit 14: Offset: 38: Link buffer 2 DMA channel   */
#define LP3I    0x00008000 /* Bit 15: Offset: 3c: Link buffer 3 DMA channel   */
#define EP0I    0x00010000 /* Bit 16: Offset: 40: External port channel 0 DMA */
#define EP1I    0x00020000 /* Bit 17: Offset: 44: External port channel 1 DMA */
#define EP2I    0x00040000 /* Bit 18: Offset: 48: External port channel 2 DMA */
#define EP3I    0x00080000 /* Bit 19: Offset: 4c: External port channel 3 DMA */
#define LSRQI   0x00100000 /* Bit 20: Offset: 50: Link service request        */
#define CB7I    0x00200000 /* Bit 21: Offset: 54: Circ. buffer 7 overflow     */
#define CB15I   0x00400000 /* Bit 22: Offset: 58: Circ. buffer 15 overflow    */
#define TMZLI   0x00800000 /* Bit 23: Offset: 5c: Timer = 0 (low priority)    */
#define FIXI    0x01000000 /* Bit 24: Offset: 60: Fixed-pt. overflow          */
#define FLTOI   0x02000000 /* Bit 25: Offset: 64: fltg-pt. overflow           */
#define FLTUI   0x04000000 /* Bit 26: Offset: 68: fltg-pt. underflow          */
#define FLTII   0x08000000 /* Bit 27: Offset: 6c: fltg-pt. invalid            */
#define SFT0I   0x10000000 /* Bit 28: Offset: 70: user software int 0         */
#define SFT1I   0x20000000 /* Bit 29: Offset: 74: user software int 1         */
#define SFT2I   0x40000000 /* Bit 30: Offset: 78: user software int 2         */
#define SFT3I   0x80000000 /* Bit 31: Offset: 7c: user software int 3         */
```

```
/* I/O Processor Registers */
#define SYSCON 0x00     /* System configuration register           */
#define VIRPT  0x01     /* Vector interrupt register               */
#define WAIT   0x02     /* Wait state configuration for ext. memory */
#define SYSTAT 0x03     /* System status register                  */
#define EPB0   0x04     /* External port DMA buffer 0              */
#define EPB1   0x05     /* External port DMA buffer 1              */
#define EPB2   0x06     /* External port DMA buffer 2, not on 21061 */
#define EPB3   0x07     /* External port DMA buffer 3, not on 21061 */
#define MSGR0  0x08     /* Message register 0                      */
#define MSGR1  0x09     /* Message register 1                      */
#define MSGR2  0x0a     /* Message register 2                      */
#define MSGR3  0x0b     /* Message register 3                      */
#define MSGR4  0x0c     /* Message register 4                      */
#define MSGR5  0x0d     /* Message register 5                      */
#define MSGR6  0x0e     /* Message register 6                      */
#define MSGR7  0x0f     /* Message register 7                      */
#define BMAX   0x18     /* Bus time-out maximum                    */
#define BCNT   0x19     /* Bus time-out counter                    */
#define ELAST  0x1b     /* Address of last external access         */
#define DMAC6  0x1c     /* DMA6 control register                   */
#define DMAC7  0x1d     /* DMA7 control register                   */
#define DMAC8  0x1e     /* DMA8 control register, not on 21061     */
#define DMAC9  0x1f     /* DMA9 control register, not on 21061     */

/* II4, IM4, C4, CP4, Gp4, DB4, & DA4 reg's are not on the ADSP-21061  */
#define II4    0x30     /* Internal DMA4 memory address               */
#define IM4    0x31     /* Internal DMA4 memory access modifier       */
#define C4     0x32     /* Contains number of DMA4 transfers remaining */
#define CP4    0x33     /* Points to next DMA4 parameters             */
#define GP4    0x34     /* DMA4 General purpose / 2-D DMA             */
#define DB4    0x35     /* DMA4 General purpose / 2-D DMA             */
#define DA4    0x36     /* DMA4 General purpose / 2-D DMA             */

#define DMASTAT 0x37    /* DMA channel status register                */

/* II5, IM5, C5, CP5, Gp5, DB5, & DA5 reg's are not on the ADSP-21061  */
#define II5    0x38     /* Internal DMA5 memory address               */
#define IM5    0x39     /* Internal DMA5 memory access modifier       */
#define C5     0x3a     /* Contains number of DMA5 transfers remaining */
#define CP5    0x3b     /* Points to next DMA5 parameters             */
#define GP5    0x3c     /* DMA5 General purpose / 2-D DMA             */
#define DB5    0x3d     /* DMA5 General purpose / 2-D DMA             */
#define DA5    0x3e     /* DMA5 General purpose / 2-D DMA             */

#define II6    0x40     /* Internal DMA6 memory address               */
#define IM6    0x41     /* Internal DMA6 memory access modifier       */
#define C6     0x42     /* Contains number of DMA6 transfers remaining */
#define CP6    0x43     /* Points to next DMA6 parameters             */
#define GP6    0x44     /* DMA6 General purpose                       */
#define EI6    0x45     /* External DMA6 address                      */
#define EM6    0x46     /* External DMA6 address modifier             */
```

```
#define EC6    0x47     /* External DMA6 counter                        */

#define II7  0x48   /* Internal DMA7 memory address                */
#define IM7  0x49   /* Internal DMA7 memory access modifier        */
#define C7   0x4a   /* Contains number of DMA7 transfers remaining */
#define CP7  0x4b   /* Points to next DMA7 parameters              */
#define GP7  0x4c   /* DMA7 General purpose                        */
#define EI7  0x4d   /* External DMA7 address                       */
#define EM7  0x4e   /* External DMA7 address modifier              */
#define EC7  0x4f   /* External DMA7 counter                       */

/* II8, IM8, C8, CP8, Gp8, EI8, EM8, & EC8 reg's are not on the 21061  */
#define II8  0x50   /* Internal DMA8 memory address                */
#define IM8  0x51   /* Internal DMA8 memory access modifier        */
#define C8   0x52   /* Contains number of DMA8 transfers remaining */
#define CP8  0x53   /* Points to next DMA8 parameters              */
#define GP8  0x54   /* DMA8 General purpose                        */
#define EI8  0x55   /* External DMA8 address                       */
#define EM8  0x56   /* External DMA8 address modifier              */
#define EC8  0x57   /* External DMA8 counter                       */

/* II9, IM9, C9, CP9, Gp9, EI9, EM9, & EC9 reg's are not on the 21061  */
#define II9  0x58   /* Internal DMA9 memory address                */
#define IM9  0x59   /* Internal DMA9 memory access modifier        */
#define C9   0x5a   /* Contains number of DMA9 transfers remaining */
#define CP9  0x5b   /* Points to next DMA9 parameters              */
#define GP9  0x5c   /* DMA9 General purpose                        */
#define EI9  0x5d   /* External DMA9 address                       */
#define EM9  0x5e   /* External DMA9 address modifier              */
#define EC9  0x5f   /* External DMA9 counter                       */

#define II0  0x60   /* Internal DMA0 memory address                */
#define IM0  0x61   /* Internal DMA0 memory access modifier        */
#define C0   0x62   /* Contains number of DMA0 transfers remaining */
#define CP0  0x63   /* Points to next DMA0 parameters              */
#define GP0  0x64   /* DMA0 General purpose / 2-D DMA              */
#define DB0  0x65   /* DMA0 General purpose / 2-D DMA, not on 21061 */
#define DA0  0x66   /* DMA0 General purpose / 2-D DMA, not on 21061 */

#define II2  0x70   /* Internal DMA2 memory address                */
#define IM2  0x71   /* Internal DMA2 memory access modifier        */
#define C2   0x72   /* Contains number of DMA2 transfers remaining */
#define CP2  0x73   /* Points to next DMA2 parameters              */
#define GP2  0x74   /* DMA2 General purpose / 2-D DMA              */
#define DB2  0x75   /* DMA2 General purpose / 2-D DMA, not on 21061 */
#define DA2  0x76   /* DMA2 General purpose / 2-D DMA, not on 21061 */

#define II1  0x68   /* Internal DMA1 memory address                */
#define IM1  0x69   /* Internal DMA1 memory access modifier        */
#define C1   0x6a   /* Contains number of DMA1 transfers remaining */
#define CP1  0x6b   /* Points to next DMA1 parameters              */
```

```
#define GP1  0x6c   /* DMA1 General purpose / 2-D DMA              */
#define DB1  0x6d   /* DMA1 General purpose / 2-D DMA, not on 21061 */
#define DA1  0x6e   /* DMA1 General purpose / 2-D DMA, not on 21061 */

#define II3  0x78   /* Internal DMA3 memory address                */
#define IM3  0x79   /* Internal DMA3 memory access modifier        */
#define C3   0x7a   /* Contains number of DMA3 transfers remaining */
#define CP3  0x7b   /* Points to next DMA3 parameters              */
#define GP3  0x7c   /* DMA3 General purpose / 2-D DMA              */
#define DB3  0x7d   /* DMA3 General purpose / 2-D DMA, not on 21061 */
#define DA3  0x7e   /* DMA3 General purpose / 2-D DMA, not on 21061 */

/* LBUF0, LBUF1, LBUF2, LBUF4, LBUF5, LCTL, LCOM, LAR, LSRQ, LPATH1,
   LPATH2, LPATH3, LPCNT, CNST1, and CNST2 reg's are not on the 21061 */
#define LBUF0  0xc0     /* Link buffer 0                          */
#define LBUF1  0xc1     /* Link buffer 1                          */
#define LBUF2  0xc2     /* Link buffer 2                          */
#define LBUF3  0xc3     /* Link buffer 3                          */
#define LBUF4  0xc4     /* Link buffer 4                          */
#define LBUF5  0xc5     /* Link buffer 5                          */
#define LCTL   0xc6     /* Link buffer control                    */
#define LCOM   0xc7     /* Link common control                    */
#define LAR    0xc8     /* Link assignment register               */
#define LSRQ   0xc9     /* Link service request and mask register */
#define LPATH1 0xca     /* Link path register 1                   */
#define LPATH2 0xcb     /* Link path register 2                   */
#define LPATH3 0xcc     /* Link path register 3                   */
#define LPCNT  0xcd     /* Link path counter                      */
#define CNST1  0xce     /* Link port constant 1 register          */
#define CNST2  0xcf     /* Link port constant 2 register          */

#define STCTL0    0xe0 /*SPORT0 Transmit Control Register         */
#define SRCTL0    0xe1 /*SPORT0 Receive  Control Register         */
#define TX0       0xe2 /*SPORT0 Transmit Data Buffer              */
#define RX0       0xe3 /*SPORT0 Receive Data Buffer               */
#define TDIV0     0xe4 /*SPORT0 Transmit Divisor                  */
#define TCNT0     0xe5 /*SPORT0 Transmit Count Reg                */
#define RDIV0     0xe6 /*SPORT0 Receive Divisor             */
#define RCNT0     0xe7 /*SPORT0 Receive Count Reg                 */
#define MTCS0     0xe8 /*SPORT0 Multichannel Transmit Selector    */
#define MRCS0     0xe9 /*SPORT0 Multichannel Receive Selector     */
#define MTCCS0    0xea /*SPORT0 Multichannel Transmit Selector    */
#define MRCCS0    0xeb /*SPORT0 Multichannel Receive Selector     */
#define KEYWD0    0xec /*SPORT0 Receive Comparison, 21061 only    */
#define KEYMASK0  0xed /*SPORT0 Receive Comparison Mask, 21061 only */
#define SPATH0    0xee /*SPORT0 Path Length (MMP), not on 21061   */
#define SPCNT0    0xef /*SPORT0 Path Counter (MMP), not on 21061  */
```

```
#define STCTL1    0xf0 /*SPORT1 Transmit Control Register         */
#define SRCTL1    0xf1 /*SPORT1 Receive  Control Register         */
#define TX1       0xf2 /*SPORT1 Transmit Data Buffer              */
#define RX1       0xf3 /*SPORT1 Receive Data Buffer               */
#define TDIV1     0xf4 /*SPORT1 Transmit Divisor                  */
#define TCNT1     0xf5 /*SPORT1 Transmit Count Reg                */
#define RDIV1     0xf6 /*SPORT1 Receive Divisor             */
#define RCNT1     0xf7 /*SPORT1 Receive Count Reg                 */
#define MTCS1     0xf8 /*SPORT1 Multichannel Transmit Selector   */
#define MRCS1     0xf9 /*SPORT1 Multichannel Receive Selector    */
#define MTCCS1    0xfa /*SPORT1 Multichannel Transmit Selector   */
#define MRCCS1    0xfb /*SPORT1 Multichannel Receive Selector*/
#define KEYWD1    0xfc /*SPORT1 Receive Comparison, 21061 only    */
#define KEYMASK1  0xfd /*SPORT1 Receive Comparison Mask, 21061 only */
#define SPATH1    0xfe /*SPORT1 Path Length (MMP), not on 21061   */
#define SPCNT1    0xff /*SPORT1 Path Counter (MMP), not on 21061  */


/* SYSCON Register */
#define SYSCON 0x00        /* System Configuration Register */
#define SRST   0x00000001  /* Soft Reset                                        */
#define BSO    0x00000002  /* Boot Select Override                              */
#define IIVT   0x00000004  /* Internal Interrupt Vector Table                   */
#define IWT    0x00000008  /* Instruction word transfer (0=data,1=instr)        */
#define HPM00  0x00000000  /* Host packing mode: None                           */
#define HPM01  0x00000010  /* Host packing mode: 16/32                          */
#define HPM10  0x00000020  /* Host packing mode: 16/48                          */
#define HPM11  0x00000030  /* Host packing mode: 32/48                          */
#define HMSWF  0x00000040  /* Host packing order (0=LSW-first, 1=MSW-first)     */
#define HPFLSH 0x00000080  /* Host pack flush                                   */
#define IMDW0X 0x00000100  /* Internal memory block 0, extended data (40-bit)   */
#define IMDW1X 0x00000200  /* Internal memory block 1, extended data (40-bit)   */
#define EBPR00 0x00000000  /* External bus priority: Even                       */
#define EBPR01 0x00010000  /* External bus priority: Core has priority          */
#define EBPR10 0x00020000  /* External bus priority: IO has priority            */
#define DCPR   0x00040000  /* Select rotating access priority on DMA6 - DMA9    */
#define IMGR   0x10000000  /* Internal memory block grouping (mesh multiproc)   */


/* SYSTAT Register */
#define SYSTAT 0x03        /* System Status Register */
#define HSTM   0x00000001  /* Host is the Bus Master                            */
#define BSYN   0x00000002  /* Bus arbitration logic is synchronized             */
#define CRBM   0x00000070  /* Current ADSP2106x Bus Master                      */
#define IDC    0x00000700  /* ADSP2106x ID Code                                 */
#define DWPD   0x00001000  /* Direct write pending (0=none, 1=pending)          */
#define VIPD   0x00002000  /* Vector interrupt pending (1=pending)              */
#define HPS    0x0000c000  /* Host packing status                               */
```

Interrupt Vector Addresses F

IRPTL/ IMASK Bit #	*Vector Address**	*Interrupt Name***	*Function*	
0	0x00	–	*reserved*	
1	0x04	RSTI	Reset (read-only, non-maskable)	**HIGHEST PRIORITY**
2	0x08	–	*reserved*	
3	0x0C	SOVFI	Status stack or loop stack overflow or PC stack full	
4	0x10	TMZHI	Timer=0 (high priority option)	
5	0x14	VIRPTI	Vector Interrupt	
6	0x18	IRQ2I	IRQ2 asserted	
7	0x1C	IRQ1I	IRQ1 asserted	
8	0x20	IRQ0I	IRQ0 asserted	
9	0x24	–	*reserved*	
10	0x28	SPR0I	DMA Channel 0 – SPORT0 Receive	
11	0x2C	SPR1I	DMA Channel 1 – SPORT1 Receive (or Link Buffer 0)	
12	0x30	SPT0I	DMA Channel 2 – SPORT0 Transmit	
13	0x34	SPT1I	DMA Channel 3 – SPORT1 Transmit (or Link Buffer 1)	
14	0x38	LP2I	DMA Channel 4 – Link Buffer 2	
15	0x3C	LP3I	DMA Channel 5 – Link Buffer 3	
16	0x40	EP0I	DMA Channel 6 – Ext. Port Buffer 0 (or Link Buffer 4)	
17	0x44	EP1I	DMA Channel 7 – Ext. Port Buffer 1 (or Link Buffer 5)	
18	0x48	EP2I	DMA Channel 8 – Ext. Port Buffer 2	
19	0x4C	EP3I	DMA Channel 9 – Ext. Port Buffer 3	
20	0x50	LSRQ	Link Port Service Request	
21	0x54	CB7I	Circular Buffer 7 overflow	
22	0x58	CB15I	Circular Buffer 15 overflow	
23	0x5C	TMZLI	Timer=0 (low priority option)	
24	0x60	FIXI	Fixed-point overflow	
25	0x64	FLTOI	Floating-point overflow exception	
26	0x68	FLTUI	Floating-point underflow exception	
27	0x6C	FLTII	Floating-point invalid exception	
28	0x70	SFT0I	User software interrupt 0	
29	0x74	SFT1I	User software interrupt 1	
30	0x78	SFT2I	User software interrupt 2	
31	0x7C	SFT3I	User software interrupt 3	**LOWEST PRIORITY**

Table F.1 Interrupt Vectors & Priority

* Offset from base address: 0x0002 0000 for interrupt vector table in internal memory, 0x0040 0000 for interrupt vector table in external memory

** These IRPTL/IMASK bit names are defined in the `def21060.h` include file supplied with the ADSP-21000 Family Development Software.

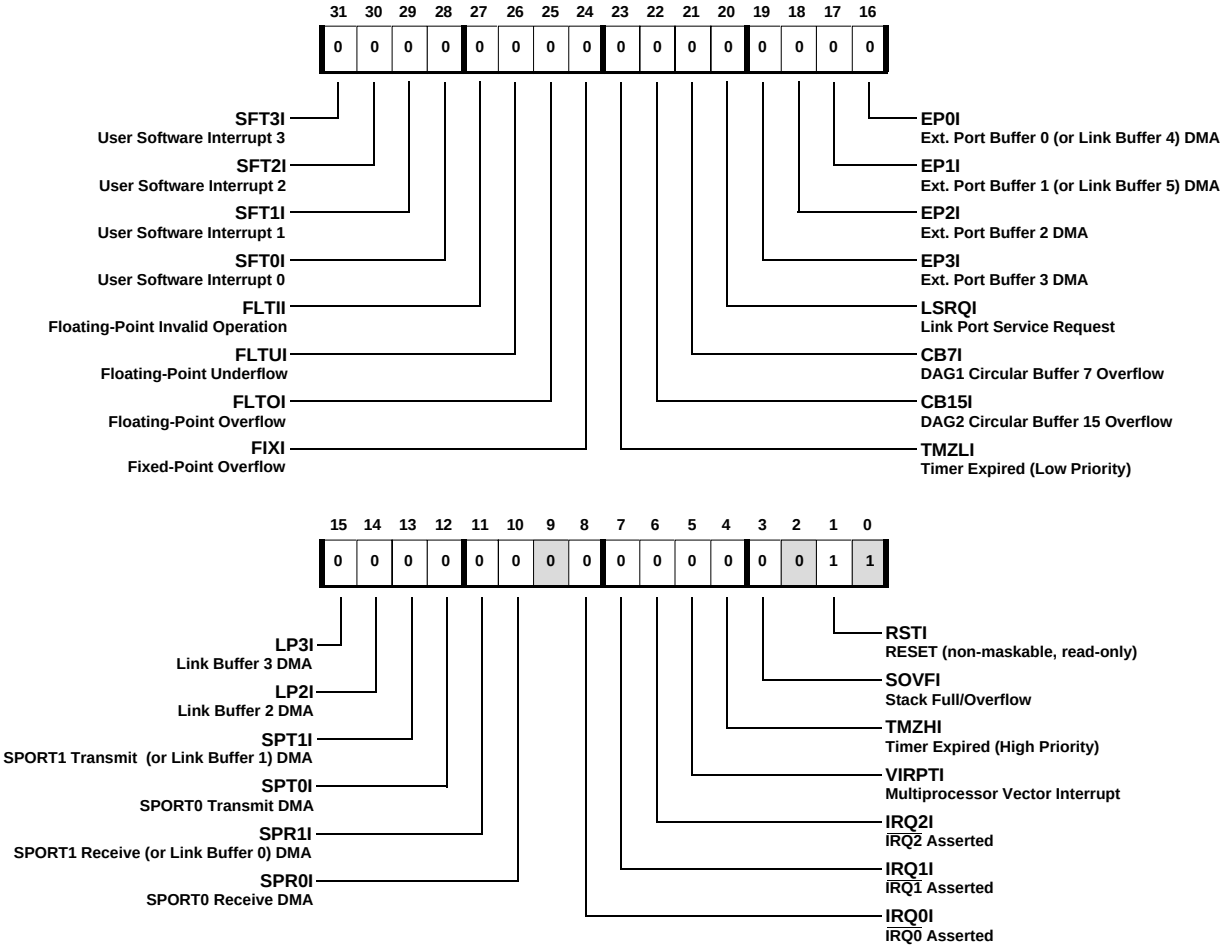
F Interrupt Vector Addresses

Table F.1 shows all ADSP-2106x interrupts, listed according their bit position in the IRPTL and IMASK registers. Also shown is the address of the interrupt vector; each vector is separated by eight memory locations. The addresses in the vector table represent offsets from a base address. For an interrupt vector table in internal memory, the base address is 0x0002 0000; for an interrupt vector table in external memory, the base address is 0x0040 0000. The third column in Table F.1 lists a mnemonic name for each interrupt. These names are provided for convenience, and are not required by the assembler.

The interrupt vector table may be located in internal memory, at address 0x0002 0000 (the beginning of Block 0), or in external memory at address 0x0040 0000. If the ADSP-2106x's on-chip memory is booted from an external source, the interrupt vector table will be located in internal memory. If, however, the ADSP-2106x is not booted (because it will execute from off-chip memory), the vector table must be located in the off-chip memory. See "Booting" in the *System Design* chapter for details on booting mode selection. Also, if booting is from an external EPROM or host processor, bit 16 of IMASK (the EP0I interrupt for DMA Channel 6) will automatically be set to 1 following reset—this enables the *DMA done* interrupt for Channel 6. IRPTL is initialized to all zeros following reset.

The IIVT bit in the SYSCON control register can be used to override the booting mode in determining where the interrupt vector table is located. If the ADSP-2106x is not booted (*no boot* mode), setting IIVT to 1 selects an internal vector table while IIVT=0 selects an external vector table. If the ADSP-2106x is booted from an external source (any mode other than *no boot* mode), then IIVT has no effect.

IRPTL & IMASK



Default values for IMASK only; IRPTL is cleared after reset.
For IMASK: 1=unmasked (enabled), 0=masked (disabled)

F Interrupt Vector Addresses

SHARC Glossary G

Term	*Definition*
core processor *or* processor core	ADSP-21000 core DSP processor—program sequencer, instruction cache, timer, DAG1, DAG2, register file (R15-0), computation units. Does not include ADSP-2106x's internal memory, external port, and I/O processor. An "action performed by the core processor" implies an action caused by the program executing on the ADSP-2106x. This is in contrast to an action performed by the on-chip DMA controller or by an external bus master, either host processor or another ADSP-2106x.
external bus	$DATA_{47-0}$, $ADDR_{31-0}$, $\overline{RD}$, $\overline{WR}$, $\overline{MS}_{3-0}$, $\overline{BMS}$, ADRCLK, PAGE, $\overline{SW}$, ACK, and $\overline{SBTS}$ signals
multiprocessor system	a system with multiple ADSP-2106xs, with or without a host processor; the ADSP-2106xs are connected by the external bus and/or link ports
multiprocessor memory space	portion of the ADSP-2106x's memory map that includes the internal memory and IOP registers of each ADSP-2106x in a multiprocessing system; this address space is mapped into the unified address space of the ADSP-2106x
IOP register	one of the control, status, or data buffer registers of the ADSP-2106x's on-chip I/O processor
bus slave *or* slave mode	an ADSP-2106x can be a bus slave to another ADSP-2106x or to a host processor (the ADSP-2106x becomes a host slave when the $\overline{HBG}$ signal is returned)

G SHARC Glossary

bus transition cycle (BTC)	a cycle in which control of the external bus is passed from one ADSP-2106x to another (in a multiprocessor system)
host transition cycle (HTC)	a cycle in which control of the external bus is passed from the ADSP-2106x to the host processor—during this cycle the ADSP-2106x stops driving the $\overline{RD}$, $\overline{WR}$, $ADDR_{31-0}$, $\overline{MS}_{3-0}$, ADRCLK, PAGE, $\overline{SW}$, and $\overline{DMAG}$x signals, which must then be driven by the host
asynchronous transfers	asynchronous host accesses of the ADSP-2106x; after acquiring control of the ADSP-2106x's external bus, the host must assert the $\overline{CS}$ pin of the ADSP-2106x it wants to access; the ADSP-2106x uses the REDY output to add wait states to an asynchronous access
synchronous transfers	synchronous host accesses of the ADSP-2106x; $\overline{CS}$ is not asserted and the host must act like another ADSP-2106x in a multiprocessor system, by generating an address in multiprocessor memory space, asserting $\overline{WR}$ or $\overline{RD}$, and driving out or latching in the data; the ADSP-2106x uses ACK to add wait states to a synchronous access
direct reads & writes	a direct access of the ADSP-2106x's internal memory or IOP registers by another ADSP-2106x or by a host processor
external port FIFO buffers	EPB0, EPB1, EPB2, and EPB3—the IOP registers used for external port DMA transfers and single-word data transfers (from other ADSP-2106xs or from a host processor); these buffers are 6-deep FIFOs
single-word data transfers	reads and writes to the EPBx external port buffers, performed externally by the ADSP-2106x bus master or internally by the ADSP-2106x slave's core; these occur when DMA is disabled in the DMACx control register

single-word data transfers	*(host processor)* reads and writes to the EPBx external port buffers, performed externally by the host or internally by the ADSP-2106x core; these occur when DMA is disabled in the DMACx control register
link port *vs.* link buffer	the link ports receive and transmit data on their $LxDAT_{3-0}$ data pins; the six independent link buffers may be connected to any of the six link ports
48-bit word	usually implies instruction word, but may also imply 48-bit instructions *and* 40-bit extended-precision data values that are transferred within 48-bit words; 48-bit words use three 16-bit memory columns
32-bit word	standard 32-bit data word; uses two 16-bit memory columns
16-bit word	16-bit short data word; uses one 16-bit memory column
data memory	region of memory in which 32-bit data words and 16-bit short words are stored; *implies that the DM bus is used for accesses* (see the following sections in the *Memory* chapter of this manual for details: "Overview," "Dual Data Accesses," and "On-Chip Memory Buses & Address Generation")
program memory	region of memory in which 48-bit instruction words and (optionally) 32-bit or 40-bit data words are stored; *implies that the PM bus is used for accesses* (see the following sections in the *Memory* chapter of this manual for details: "Overview," "Dual Data Accesses," "Instruction Cache & PM Bus Data Accesses," and "On-Chip Memory Buses & Address Generation")
program memory data access	when an ADSP-2106x instruction reads or writes data over the PM Data Bus; the address is generated by DAG2 on the PM Address Bus
DMACx control registers	the DMA control registers for the EPBx external port buffers: DMAC6, DMAC7, DMAC8, and DMAC9 (corresponding respectively to EPB0, EPB1, EPB2, and EPB3)
DMA control registers	*see "DMACx control registers"*

G SHARC Glossary

DMA parameter registers	the address (II), modifier (IM), count (C), chain pointer (CP), etc., registers used to set up a DMA transfer
transfer control block (TCB)	a set of DMA parameter register values stored in memory that are downloaded by the ADSP-2106x's DMA controller for chained DMA operations
TCB chain loading	the process in which the ADSP-2106x's DMA controller downloads a TCB from memory and autoinitializes the DMA parameter registers
cycle *or* processor cycle	one cycle of the ADSP-2106x's CLKIN input
extra cycle	a cycle generated by the ADSP-2106x when an instruction cannot be completed in a single CLKIN cycle (e.g. to allow an additional access of internal or external memory); see "Execution Stalls" in the *System Design* chapter of this manual
sticky status bit	(in STKY status register) a "sticky" status bit, once set, remains set until it is explicitly cleared (with the status bit manipulation instruction)

Equivalent Terms

multiprocessor system = multiprocessing system = multiprocessor cluster
core processor = processor core = ADSP-2106x core
DMA operation = DMA sequence
cycle = clock cycle = processor cycle = CLKIN cycle

Acronym	*Definition*
IOP	I/O Processor
DAG	Data Address Generator
SPORT	Serial Port
PMA	Program Memory Address
DMA	Data Memory Address
PMD	Program Memory Data
DMD	Data Memory Data
EPA	External Port Address
EPD	External Port Data
IOA	I/O Address
IOD	I/O Data

H DOCUMENTATION ERRATA

This revision of the ADSP-2106x SHARC Processor User's Manual contains corrections to errata in the previous, Second Edition, published May 1997. All of the pertinent corrections reported in the second edition's documentation errata on the Analog Devices Web site, `www.analog.com/dsp`, are reproduced below.

Errata and Corrections

Chapter: 3 **Page:** 12

Revision Needed:
Replace the five bulleted points at the top of the page with the following:

- Other Jumps, Calls, or Returns

  These instructions cannot be provided following a delayed branch instruction. This can be demonstrated with an example, using the JUMP instruction.

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

In this case, the delayed branch instruction `R0=R0+R1;` is executed but the instruction `R1=R1+R2` is not executed. Also, the control jumps to `my` instead of `foo`, with the delayed branch instruction being the execution of `foo`.

The exception is for JUMP, which can be done for mutually-exclusive conditions for both (EQ, NE). If the first EQ condition works, then the NE conditional jump has no meaning; it is as good as a NOP. Code for these exceptions is shown below:

```
if eq jump label1 (db);
if ne jump label2 (db);
nop;
nop;
```

- Pushes or Pops of the PC stack

  Push of the PC stack in the delayed branch is followed by a `pop`. If a value is pushed in the delayed branch of call, it is popped first in the called subroutine, which is followed by return to subroutine RTS.

  For example:

```
20119 call foo (db);
2011A push PCSTK;
2011B nop;
2011C foo;
```

  `PCSTK 2011B`: second push due to `PCSTK`
  `2011C`: first push due to call

  This demonstrates that when the user pushes the `PCSTK` during a delayed slot, the PC stack pointer is pushed onto the `PCSTK`.

Next, execute the following instruction before doing an RTS:

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

If pushing a PC stack, do a `pop` first and then an RTS. If a value is popped inside the delayed branch, whatever subroutine return address is pushed is popped back and this is restricted.

- Writes to the PC stack or PC stack pointer

  If writing to a PC stack inside the delayed branch, there can be two instances, as follows:

  1. Write to a PC stack inside a jump

     If the user writes onto the PC stack inside the delayed branch of a jump, two situations can occur:

     a. PC stack having a value already pushed onto the PC stack

        When the PC stack has a value and the user writes a value onto the PC stack, the value in the PC stack is overwritten by the value written onto the PC stack. Therefore, the value in the PC stack is corrupted, hence this is restricted.

     b. PC stack is empty

        When the PC stack is empty and a value is written onto the PC stack, the PC stack is empty, because there is no value in the PC stack, even after writing the value onto the PC stack.

2. Write to a PC stack inside a call

   If the user writes to the PC stack inside a call, the value pushed onto the PC stack due to a call is overwritten by the value written onto the PC stack. Hence, when the user does an RTS, the user returns to the address pushed onto the PC stack and not to the address pushed while branching to the subroutine.

   For example:

```
[20111] call foo3(db);
[20112] PCSTK=0x2011C;
[20113] nop;
[20114]
```

   The value `20114` is pushed onto the PC stack, while the value `2011C` is written to the PC stack. Accordingly, the value `20114` is overwritten by `2011C` in the PC stack. Thus, when the user comes back by doing an RTS, the return is to the address `2011C` and not to `20114`. Consequently, this is restricted.

- DO UNTIL instruction

  If a loop is present inside the delayed branch, it performs a sequential operation after executing the loop and does not jump to the label. This happens because the address of the destination of the jump is flushed out of the pipeline. Instead, in the pipeline (Fetch, Decode, and Execute) are the instructions of the loop, which causes sequential operation.

For example:

```
20118 LCNTR=10;
20119 jump my(db); 2012C my:
2011A do my until LCE;
2011B my: r0=r0+r1;

2011C r2=r2+r3;
2011D r1=r1+r2;
```

In the example, there is a loop inside a delayed branch. Because the loop executes the instructions inside the loop ten times, the address of the destination of the jump (`2012C`) is flushed. After that, instead of going to label `my` (`2012C`), the processor executes the next sequential instruction at address `2011C` and then continues the sequential execution. This is the reason why loop is restricted inside the delayed branch.

- IDLE instruction

  To come out of the idle instruction, an interrupt is needed. If the user puts an IDLE instruction inside the delayed branch, the processor is always in the idle state, unless there is an interrupt, hence this instruction is restricted.

Chapter: 3 **Page:** 21

Revision Needed:
In the third paragraph, third sentence, the segment "8-instruction intervals" should be replaced with "4-instruction intervals."

Chapter: 3 **Page:** 24

Revision Needed:
In paragraph 3.6.2 at the bottom of the page, second sentence, the segment "eight memory locations" shou ld be replaced with "four memory locations."

Chapter: 3 **Page:** 41

Revision Needed:
Add the following paragraph and note to the end of section 3.10.3 Cache Disable & Cache Freeze:

The CADIS bit directs the sequencer to disable the cache (if 1) or enable the cache (if 0). Disabling the cache does not mark the current content of the cache as invalid. When the cache is enabled again, the existing content is used again. To clear the cache use the FLUSH CACHE instruction.

Note: If self-modifying code (for example, software loader kernel) or software overlays are used, execute a FLUSH CACHE instruction followed by a NOP before executing the new code. Otherwise, old content from the cache could still be used, although the code has changed.

Chapter: 7 **Page:** 31

Revision Needed:
A NOP instruction should be inserted as follows:

```
BIT SET MODE2 BUSLK;
NOP;
/* NOP accommodates one cycle effect latency when writing to
Mode2 */
IF NOT BM JUMP (PC,0);
```

Chapter: 7 **Page:** 31

Revision Needed:
The instruction `IF NE JUMP (PC,-2);` should be replaced with `IF TF JUMP (PC,-2);`

Chapter: 8 **Page:** 8

Revision Needed:
The last paragraph on this page is confusing. It should be changed to:

Table 8.2 covers all cases including various multiprocessing systems. There is a special case for a single ADSP-2106x system where id=0. In this scenario, the host needs to drive only the $ADDR_{18-0}$ pins.

Chapter: 10 **Page:** 18

Revision Needed:
Add the following statement to the third paragraph of the Companding section:

Program the SPORT registers prior to loading data values into the SPORT buffers for companding to execute properly.

Chapter: A **Page:** 5

Revision Needed:
The condition labeled NBM is incorrect syntax. The correct syntax for the condition is NOT BM.

Chapter: A **Page:** 20

Revision Needed:
There is a typographical error in the first compute/immediate modify example. Change as follows:

From: `IF FLAG0_IN F1=F5*F12, F11=PM(I10,40);`

To: `IF FLAG0_IN F1=F5*F12, F11=PM(I10,4);`

Chapter: A **Page:** 24

Revision Needed:
There is a typographical error in the first line of the immediate shift example. Change as follows:

From: `IF GT R2=R6 LSHIFT BY 30, DM(I4,M4)=R0;`

To: `IF GT LSHIFT BY 30, DM(I4,M4)=R0;`

Chapter: B **Page:** 81

Revision Needed:
The fifth instruction from the bottom of the page should read:

Fm=F3-0 * F7-4, Ra=FIX F11-8 by R15-12

Chapter: E **Page:** 50

Revision Needed:
The bit fields shown are for the SPORT Receive Control register, not the Transmit Control register. See Figure 10.2 on page 10-10 for the correct bit fields of the Transmit Control register.

Chapter: E **Page:** 51

Revision Needed:
Bit 20 of the SRCTLx register is incorrectly marked as being reserved. See page E-52 for the correct bit description.

Chapter: F **Page:** 2

Revision Needed:
In the first paragraph, second sentence, the segment "eight memory locations" should be replaced with "four memory locations."

Index

Symbols

A

B

Index

C

Index

D

Index

E

Index

Index

J

K

L

Index

M

Index

N

O

P

Index

Q

no entries

R

S

Index

T

Index

Index