Texas Instruments

TMS320C5x

User's Guide

1991

Digital Signal Processing Products

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to or to discontinue any semiconductor product or service identified in this publication without notice. TI advises its customers to obtain the latest version of the relevant information to verify, before placing orders, that the information being relied upon is current.

TI warrants performance of its semiconductor products to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Texas Instruments products are not intended for use in life-support appliances, devices, or systems. Use of a TI product in such applications without the written consent of the appropriate TI officer is prohibited.

Appendix A Electrical Specifications
Provides design documentation for the TMS320C50 and TMS320C51 devices. This data is based upon design goals and modeling information.

Appendix B External Interface Timing
Provides functional timing of operation on the external interface bus.

Appendix C TMS320C5x System Migration
Provides information for upgrading a TMS320C25 system to a TMS320C5x system. Includes package dimensions and pinouts, timing similarities and differences, source-code compatibility, memory maps, on-chip peripheral interfacing, and development tool enhancements.

Appendix D TMS320C5x Development Tools
Lists and briefly describes the hardware and software development tools that support the TMS320C5x.

Appendix E XDS510 Design Considerations
Provides information to meet the design requirements of the XDS510 emulator and to support XDS510 Cable #2563988–001 Rev. B.

Appendix F Memories, Analog Converters, Sockets, and Crystals
Provides product information regarding memories, analog converters, and sockets that are manufactured by Texas Instruments and are compatible with the TMS320C5x. Information is also given regarding crystal frequencies, specifications, and vendors.

Appendix G ROM Codes
Provides information regarding the procedural flow for TMS320 masked parts.

Appendix H Device and Development Support Tool Nomenclature
Provides a description of the nomenclature used to designate the stages in the product development cycle.

Related Documentation

A wide variety of related documentation is available on digital signal processing. These references fall into one of the following application categories:

- digital control systems
- digital signal processing
- image processing
- speech processing

Within those areas, the references appear in alphabetical order according to author. The documents contain beneficial information regarding designs, operations, and applications for general and/or specific signal-processing systems as well as circuits; all of the documents provide additional references. There-

11) Oppenheim, Alan V. (Editor), *Applications of Digital Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

12) Oppenheim, Alan V., and R.W. Schafer, *Digital Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

13) Oppenheim, A.V., A.N. Willsky, and I.T. Young, *Signals and Systems*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

14) Parks, T.W., and C.S. Burrus, *Digital Filter Design*, New York, NY: John Wiley and Sons, Inc., 1987.

15) Rabiner, Lawrence R., and Bernard Gold, *Theory and Application of Digital Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

16) Texas Instruments (Engineers), *Digital Signal Processing Applications with the TMS320 Family*, 1986; Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

17) Treichler, J.R., C.R. Johnson, Jr., and M.G. Larimore, *A Practical Guide to Adaptive Filter Design*, New York, NY: John Wiley and Sons, Inc., 1987.

Image Processing:

1) Andrews, H.C., and B.R. Hunt, *Digital Image Restoration*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1977.

2) Gonzales, Rafeal C., and Paul Wintz, *Digital Image Processing*, Reading, MA: Addison-Wesley Publishing Company, Inc., 1977.

3) Pratt, Willaim K., *Digital Image Processing*, New York, NY: John Wiley and Sons, 1978.

Speech Processing:

1) Gray, A.H., and J.D. Markel, *Linear Prediction of Speech*, New York, NY: Springer-Verlag, 1976.

2) Jayant, N.S., and Peter Noll, *Digital Coding of Waveforms*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

3) Papamichalis, Panos, *Practical Approaches to Speech Coding*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

4) Rabiner, L.R., and R.W. Schafer, *Digital Processing of Speech Signals*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

```
{ * | *+ | *- }
```

This provides three choices: `*`, `*+`, or `*-`.

Unless the list is enclosed in square brackets, you must choose one item from the list.

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

  .byte *value$_1$ [, ... , value$_n$]*

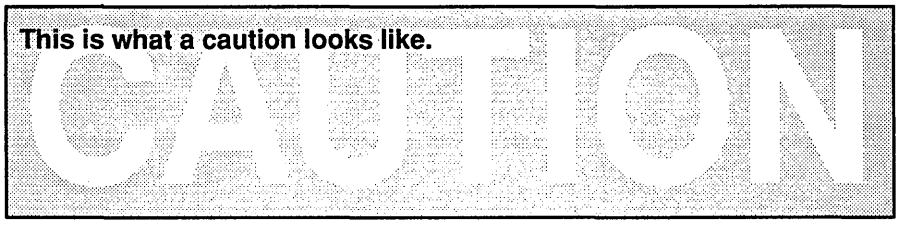
  This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas.

Information About Cautions and Warnings

This book may contain cautions and warnings.

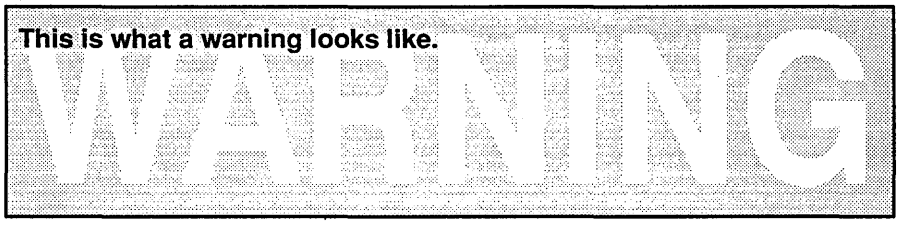
- A **caution** describes a situation that could potentially damage your software or equipment.

This is what a caution looks like.



- A **warning** describes a situation that could potentially cause harm to **you.**

This is what a warning looks like.

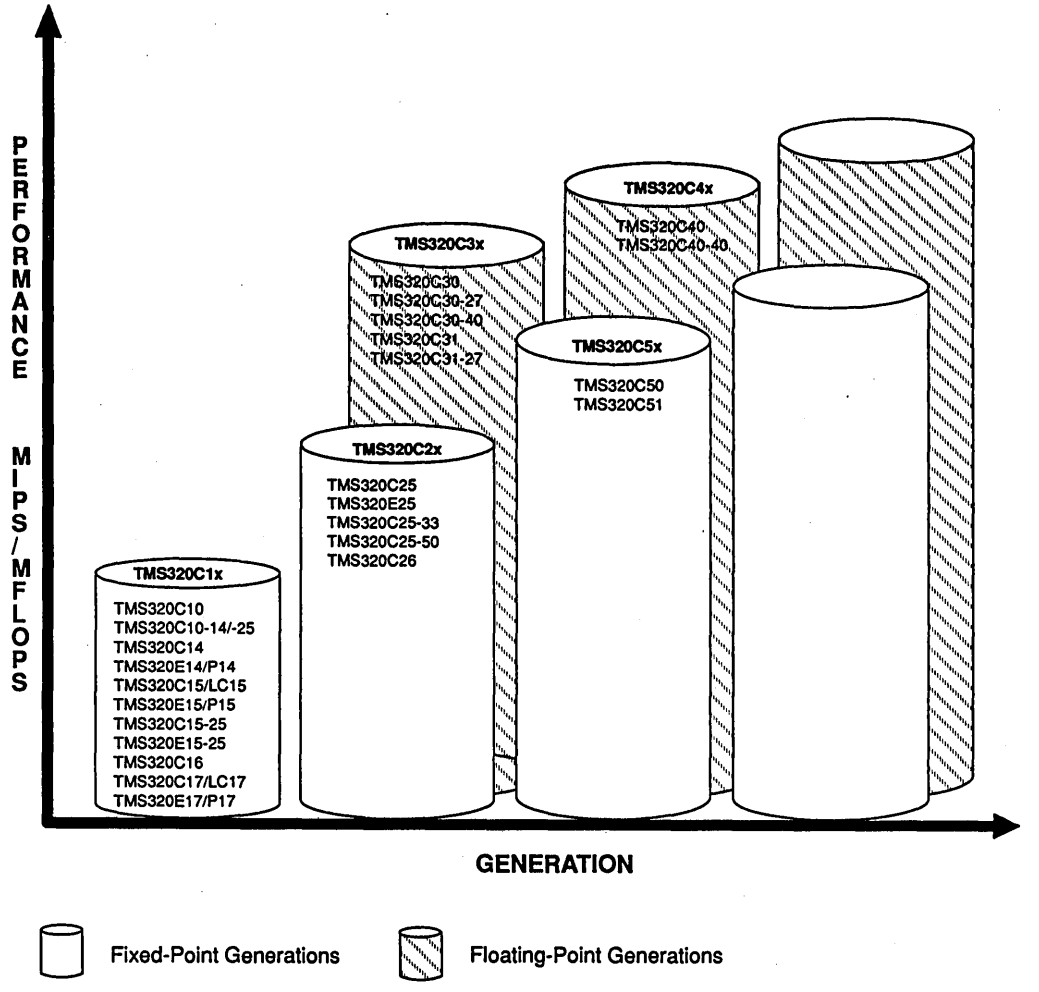


The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Figures

1 Introduction

Figure 1–1. Evolution of the TMS320 Family



1.2 General Description

The TMS320C5x generation consists of the TMS320C50 and TMS320C51 devices. These digital signal processors are fabricated in accordance with static CMOS integrated-circuit technology. Their architectural design is based upon that of the TMS320C25. The combination of an advanced Harvard architecture (separate buses for program memory and data memory), additional on-chip peripherals, more on-chip memory, and a highly specialized instruction set is the basis of the operational flexibility and speed of these DSP devices. TMS320C5x devices are designed to execute more than 28 MIPS (million instructions per second). Spin-off devices with the core CPU and customized on-chip memory and peripheral configurations can be developed for specialized areas of the electronics market.

The TMS320C5x generation offers these advantages:

- enhanced TMS320 architectural design for increased performance and versatility
- a modular architectural design for rapidly developing spin-off devices
- advanced IC processing technology for increased performance
- source-code compatibility with TMS320C1x and TMS320C2x DSPs for maintaining a roadmap between fixed-point processors and for protecting the TMS320 design investments
- enhanced TMS320 instruction set for faster algorithms and for optimized high-level language operation
- new static design techniques for minimizing power consumption and maximizing radiation hardness

Table 1–1 provides an overview of the TMS320C5x generation of digital signal processors. It shows the capacity of on-chip RAM and ROM memories, number of serial and parallel I/O ports, execution time of one machine cycle, and type of package with total pin count. The chart should help you choose the best processor for an application.

The following subsections summarize features of the TMS320C5x processors. The description of the CPU applies to all TMS320C5x-generation members (current and future). At this time, however, descriptions of the remaining features pertain only to the TMS320C50 and/or the TMS320C51. Detailed information on their CPU, memory, and on-chip peripherals is given in Chapters 3, 6, and 5, respectively.

1.3 Key Features

At this time, the TMS320C5x generation consists of the TMS320C50 and the TMS320C51 digital signal processors. Key features of these DSPs are listed below. Where a feature is exclusive to a particular member, the member's name is enclosed within a set of parentheses and noted after that feature.

- 35–50-ns single-cycle fixed-point instruction execution time (28.6 – 20 MIPS)
- Upward source-code compatible with all TMS320C1x and TMS320C2x devices
- RAM-based memory operation (TMS320C50)
- ROM-based memory operation (TMS320C51)
- 9K × 16-bit single-cycle on-chip program/data RAM (TMS320C50)
- 1K × 16-bit single-cycle on-chip program/data RAM (TMS320C51)
- 2K × 16-bit single-cycle on-chip boot ROM (TMS320C50)
- 8K × 16-bit single-cycle on-chip program ROM (TMS320C51)
- 1056 × 16-bit dual-access on-chip data RAM
- 224K × 16-bit maximum addressable external memory space (64K program, 64K data, 64K I/O, and 32K global)
- 32-bit arithmetic logic unit (ALU), 32-bit accumulator (ACC), and 32-bit accumulator buffer (ACCB)
- 16-bit parallel logic unit (PLU)
- 16 × 16-bit parallel multiplier with a 32-bit product capability
- Single-cycle multiply/accumulate instructions
- Eight auxiliary registers with a dedicated arithmetic unit for indirect addressing
- Eleven context-switch registers (shadow registers) for storing strategic CPU-controlled registers during an interrupt service routine
- Eight-level hardware stack
- 0- to 16-bit left and right data barrel-shifters and a 64-bit incremental data shifter
- Two indirectly addressed circular buffers for circular addressing

memory-mapped core-CPU registers and 16 memory-mapped I/O ports. See Chapter 3 for more details.

1.3.2 On-Chip ROM

The TMS320C50 features a 2K x 16-bit on-chip, maskable, programmable ROM. This memory is used for booting from slower external ROM or EPROM of program to fast on-chip or external SRAM. ROM can be selected during reset by driving the MP/$\overline{MC}$ pin low. Once your program has been booted into the RAM, this boot ROM can be operationally removed from the program memory space via the MP/$\overline{MC}$ bit in the PMST status register. If the ROM is not selected, the TMS320C50 starts its execution via an off-chip memory.

The TMS320C51 features an 8K x 16-bit on-chip maskable ROM. You can use this memory for your specified program. Once the development of the program has stabilized, submit a ROM code to Texas Instruments for implementation into your device. See Chapter 6 for more details.

1.3.3 On-Chip Data RAM

Both TMS320C5x devices carry a 1056 x 16-bit on-chip data RAM. This RAM can be accessed twice per machine cycle (dual-access RAM) as long as both accesses are not write operations. This block of memory is primarily intended to store data values but, when needed, can be used to store programs as well as data. It can be configured in one of two ways: either all 1056 x 16 bits as data memory or 544 x 16 bits as data memory with 512 x 16 bits as program memory. You can select the configuration with the CNF bit in status register ST1. See Chapter 6 for more details.

1.3.4 On-Chip Program/Data RAM

The TMS320C50 has a 9K x 16-bit on-chip RAM. The TMS320C51 has a 1K x 16-bit on-chip RAM. This memory is software configurable as program and/or data memory space. Code can be booted from an off-chip nonvolatile memory and then executed at full speed, once it is loaded into this RAM. See Chapter 6 for more details.

1.3.5 On-Chip Memory Security

The TMS320C5x generation has a maskable option to protect the contents of on-chip memories. When the related bit is set, no externally originating instruction can access the on-chip memory spaces. See Chapter 6 for more details.

1.3.6 Address-Mapped Software Wait-State Generators

Software wait-state logic is incorporated without any external hardware into TMS320C5x for interfacing with slower off-chip memory and I/O devices. This

ing devices. Also, it can be used to test pin-to-pin continuity as well as to perform operational tests on those peripheral devices that surround the TMS320C5x. It is interfaced to another internal scanning logic circuitry, which has access to all of the on-chip resources. Thus, the TMS320C5x can perform on-board emulation by means of the JTAG serial scan pins and the emulation-dedicated pins. See IEEE Standard P1149.1 for more details.

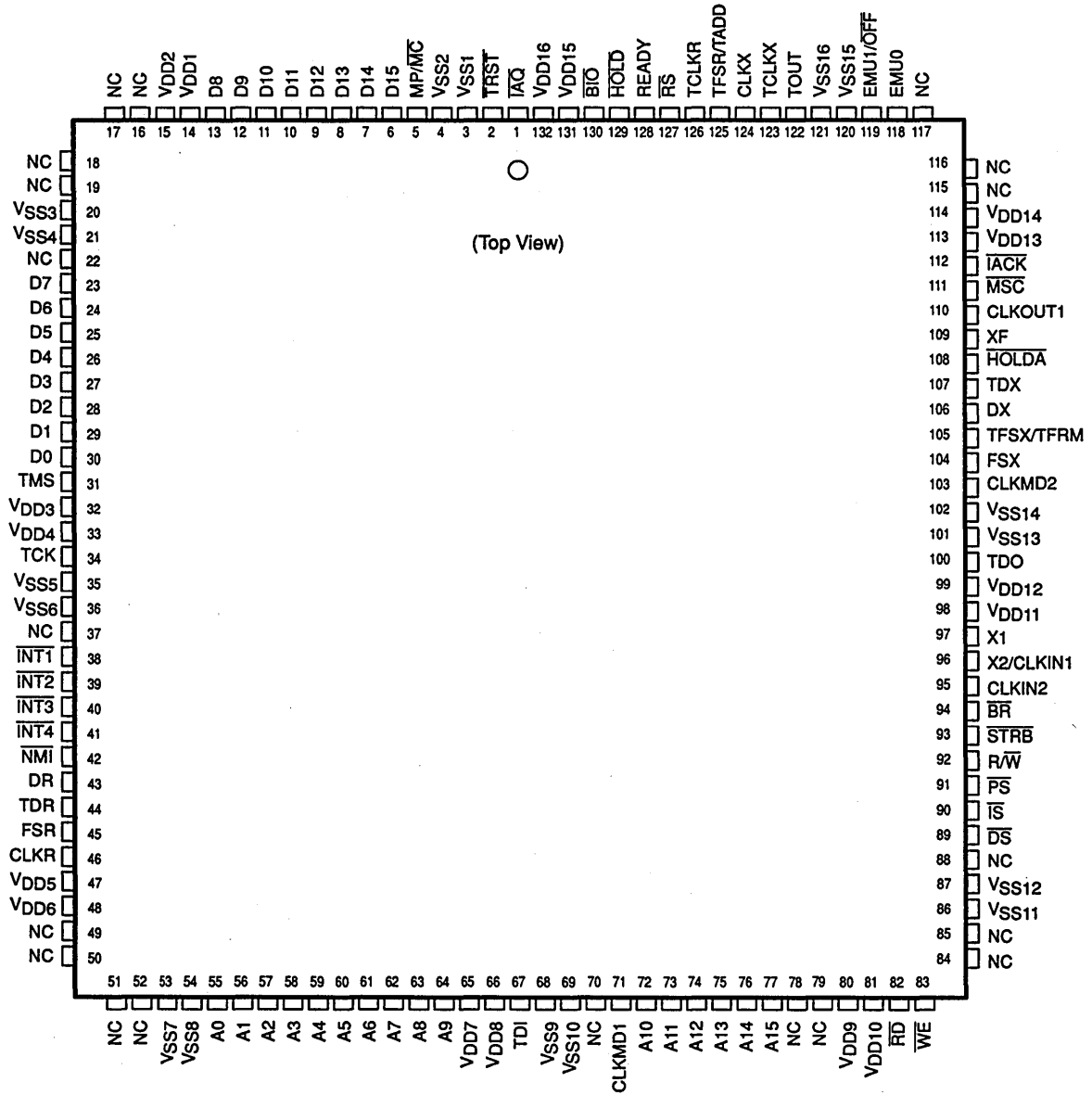
1.3.12 TMS320C5x Package

TMS320C5x devices are packaged in a 132-pin Quad Flat Pack package (QFP). With consideration for the pin layout of a TMS320C25 package, the TMS320C5x package is designed to minimize printed circuit board modifications when a TMS320C2x processing system is upgraded to a TMS320C5x processing system. Signal call-outs for the TMS320C5x appear on the same side and in the same order as those for the TMS320C25. See Chapter 2 for details.

2.1 Pin Layout

Both the *TMS320C50* and the *TMS320C51* devices are packaged in a 132-pin Quad Flat Pack package (QFP) and have the same pin-to-signal relationship. Figure 2–1 shows the pin/signal call-outs for this package.

Figure 2–1. Signal Assignments for TMS320C5x 132-Pin QFP



Note: NC = No connect. (These pins are reserved.)

Table 2–1. TMS320C5x Signal Descriptions (Continued)

Signal	Pin	State	Description
Memory Control Signals			
$\overline{DS}$ $\overline{PS}$ $\overline{IS}$	89 91 90	O/Z	Data, Program, and I/O space select signals. Always high unless low level asserted for communicating to a particular external space. Placed into a high-impedance state in hold mode. These signals also go into high-impedance when $\overline{OFF}$ is active low.
READY	128	I	Data ready input. Indicates that an external device is prepared for the bus transaction to be completed. If the device is not ready (READY is low), the processor waits one cycle and checks READY again. READY also indicates a bus grant to an external device after a $\overline{BR}$ (bus request) signal.
R/$\overline{W}$	92	I/O/Z	Read/Write signal. Indicates transfer direction during communication to an external device. Normally in read mode (high), unless low level asserted for performing a write operation. Placed in high-impedance state in hold mode. This signal also goes into high impedance when $\overline{OFF}$ is active low, and it is used in external DMA access of the 9K RAM cell. While $\overline{HOLDA}$ and $\overline{IAQ}$ are active low, this signal is used to indicate the direction of the data bus for DMA reads (high) and writes (low).
$\overline{STRB}$	93	I/O/Z	Strobe signal. Always high unless asserted low to indicate an external bus cycle. Placed in high-impedance state in the hold mode. This signal also goes into high impedance when $\overline{OFF}$ is active low, and it is used in external DMA access of the 9K RAM cell or the 1K RAM cell on C51. While $\overline{HOLDA}$ and $\overline{IAQ}$ are active low, this signal is used to select the memory access.
$\overline{RD}$	82	O/Z	Read select indicates an active, external read cycle and may connect directly to the output enable ($\overline{OE}$) of external devices. This signal is active on all external program, data, and I/O reads. Placed into high-impedance state in hold mode. This signal also goes into high impedance when $\overline{OFF}$ is active low.
$\overline{WE}$	83	O/Z	Write enable. The falling edge of this signal indicates that the device is driving the external data bus (D15–D0). Data may be latched by an external device on the rising edge of $\overline{WE}$. This signal is active on all external program, data, and I/O writes. Placed into high-impedance state in hold mode. This signal also goes into high impedance when $\overline{OFF}$ is active low.

Table 2–1. TMS320C5x Signal Descriptions (Continued)

Signal	Pin	State	Description
Initialization, Interrupt, and Reset Operations			
$\overline{INT4}$ $\overline{INT3}$ $\overline{INT2}$ $\overline{INT1}$	41 40 39 38	I	External user interrupt inputs. Prioritized and maskable by the interrupt mask register and interrupt mode bit. Can be polled and reset via the interrupt flag register.
$\overline{NMI}$	42	I	Nonmaskable interrupt. External interrupt that cannot be masked via the INTM or the IMR. When $\overline{NMI}$ is activated, the processor traps to the appropriate vector location.
$\overline{RS}$	127	I	Reset input. Causes the device to terminate execution and forces the program counter to zero. When $\overline{RS}$ is brought to a high level, execution begins at location zero of program memory. $\overline{RS}$ affects various registers and status bits.
MP/$\overline{MC}$	5	I	Microprocessor/Microcomputer mode select pin. If active low at reset (microcomputer mode), the pin causes the internal program ROM to be mapped into program memory space. In the microprocessor mode, all program memory is mapped externally. This pin is sampled only during reset, and the mode that is set at reset can be overridden via the software control bit MP/$\overline{MC}$ in the PMST register.
Oscillator/Timer Signals			
CLKOUT1	110	O/Z	Master clock output signal ($\overline{CLKIN}$/2 or $\overline{CLKIN2}$ frequency). This signal cycles at the machine-cycle rate of the CPU. The internal machine cycle is bounded by the rising edges of this signal. This signal also goes into high impedance when $\overline{OFF}$ is active low.
CLKMD1 CLKMD2	71 103	I	CLKMD1 / CLKMD2 / Clock Mode 0 / 0 / External clock with divide-by-two option. Input clock provided to X2/CLKIN1 pin. Internal oscillator and PLL disabled. 0 / 1 / Reserved for test purposes. 1 / 0 / External divide-by-one option. Input clock provided to CLKIN2. Internal oscillator disabled. Internal PLL enabled. 1 / 1 / Internal or external divide-by-two option. Input clock provided to X2/CLKIN1 pin. Internal oscillator enabled. Internal PLL disabled.
X2/CLKIN1	96	I	Input pin to internal oscillator from the crystal. If the internal oscillator is not being used, a clock may be input to the device on this pin. The internal machine cycle is half this clock rate.
X1	97	O	Output pin from the internal oscillator for the crystal. If the internal oscillator is not used, this pin should be left unconnected. This signal does not go into high impedance when $\overline{OFF}$ is active low.

Table 2–1. TMS320C5x Signal Descriptions (Continued)

Signal	Pin	State	Description
Supply Pins (Concluded)			
V_{SS14}	102	S	Ground for inputs and internal logic.
V_{SS15}	120	S	Ground for inputs and internal logic.
V_{SS16}	121	S	Ground for inputs and internal logic.
Serial Port Signals			
CLKR TCLKR	46 126	I I	Receive clock inputs. External clock signal for clocking data from the DR/TDR (data receive) pins into the RSR (serial port receive shift register). Must be present during serial port transfers. If the serial port is not being used, these pins can be sampled as an input via the IN0 bit of the SPC/TSPC registers.
CLKX TCLKX	124 123	I/O/Z I/O/Z	Transmit clock. Clock signal for clocking data from the DR/TDR (data receive register) to the DX/TDX (data transmit pin). The CLKX can be an input if the MCM bit in the serial port control register is set to 0. It may also be driven by the device at 1/4 the CLKOUT1 frequency when the MCM bit is set to 1. If the serial port is not being used, this pin can be sampled as an input via the IN1 bit of the SPC/TSPC register. This signal goes into high impedance when $\overline{OFF}$ is active low.
DR TDR	43 44	I I	Serial data receive inputs. Serial data is received in the RSR (serial port receive shift register) via the DR/TDR pin.
DX TDX	106 107	O/Z	Serial port transmit outputs. Serial data transmitted from the XSR (serial port transmit shift register) via the DX/TDX pin. Placed in high-impedance state when not transmitting and also when $\overline{OFF}$ is active low.
FSR TFSR/TADD	45 125	I I/O/Z	Frame synchronization pulse for receive input. The falling edge of the FSR/TFSR pulse initiates the data receive process, beginning the clocking of the RSR. TFSR becomes an input/output (TADD) pin when the serial port is operating in TDM mode (TDM bit = 1). In TDM mode, this pin is used to output/input the address of the port. This signal goes into high impedance when $\overline{OFF}$ is active low.

Table 2–1. TMS320C5x Signal Descriptions (Concluded)

Signal	Pin	State	Description
Test Signals (Concluded)			
EMU1/$\overline{OFF}$	119	I/O/Z	Emulator pin 1/disable all outputs. When $\overline{TRST}$ is driven low or not connected, this pin is configured as $\overline{OFF}$. The EMU1/$\overline{OFF}$ signal, when active low, puts all output drivers into the high-impedance state. Note that $\overline{OFF}$ is used exclusively for testing and emulation purposes (not for multiprocessing applications). When TRST is driven high, this pin is used as an interrupt to or from the emulator system and is defined as input/output via JTAG scan.
RESERVED	16 17 18 19 22 37 49 50 51 52 78 79 84 85 88 111 115 116 117	N/C	Reserved pin. These pins are reserved for future TMS320C5x devices. These pins should be left unconnected.

3 Architecture

3.1 Architectural Overview

The TMS320C5x high-performance digital signal processors are designed, like the TMS320C25, with an advanced Harvard-type architecture that maximizes the processing power by maintaining two separate memory bus structures, program and data, for full-speed execution. Instructions support data transfers between the two spaces.

The TMS320C5x performs twos-complement arithmetic, using the 32-bit ALU and accumulator. The ALU is a general-purpose arithmetic unit that operates by using 16-bit words taken from data memory or derived from immediate instructions, or by using the 32-bit result from the multiplier. In addition to arithmetic operations, the ALU can perform Boolean operations. The accumulator stores the output from the ALU and is also the second input to the ALU. The accumulator is 32 bits in length and is divided into a high-order word (bits 31 through 16) and a low-order word (bits 15 through 0). Instructions are provided for storing those high- and low-order accumulator words in memory. For fast, temporary storage of the accumulator, there is a 32-bit accumulator buffer.

In addition to the main ALU, there is a parallel logic unit (PLU) that executes logic operations on data without affecting the contents of the accumulator. The PLU provides the bit-manipulation ability required of a high-speed controller and simplifies the bit setting, clearing, and testing required with control and status register operations.

The multiplier performs 16 x 16-bit twos-complement multiplication with a 32-bit result in a single-instruction cycle. The multiplier consists of three elements: multiplier array, PREG (product register), and TREG0 (temporary register). The 16-bit TREG0 temporarily stores the multiplicand; the PREG stores the 32-bit product. The multiplier's values come from data memory, come from program memory when the MAC/MACD/MADS/MADD instructions are used, or are derived immediately from the multiply immediate instructions (MPY #). The fast on-chip multiplier allows the device to efficiently perform fundamental DSP operations such as convolution, correlation, and filtering.

The TMS320C5x scaling shifter has a 16-bit input connected to the data bus and a 32-bit output connected to the ALU. The scaling shifter produces a left shift of 0 to 16 bits on the input data, as programmed in the instruction or defined in the shift count register (TREG1). The LSBs of the output are filled with zeros, while the MSBs may be either zero-filled or sign-extended, depending upon the state of the sign-extension mode bit (SXM) of status register ST1. Additional shift capabilities enable the processor to perform numerical-scaling, bit-extraction, extended-arithmetic, and overflow-prevention operations.

Eight levels of hardware stack are provided for saving the contents of the program counter during interrupts and subroutine calls. On interrupts, the strategic registers (ACC, ACCB, ARCR, INDX, PMST, PREG, ST0, ST1, TREGs) are pushed onto a one-deep stack and popped upon interrupt return, thus providing a zero-overhead interrupt context switch.

Figure 3–1. Block Diagram of TMS320C5x Internal Hardware

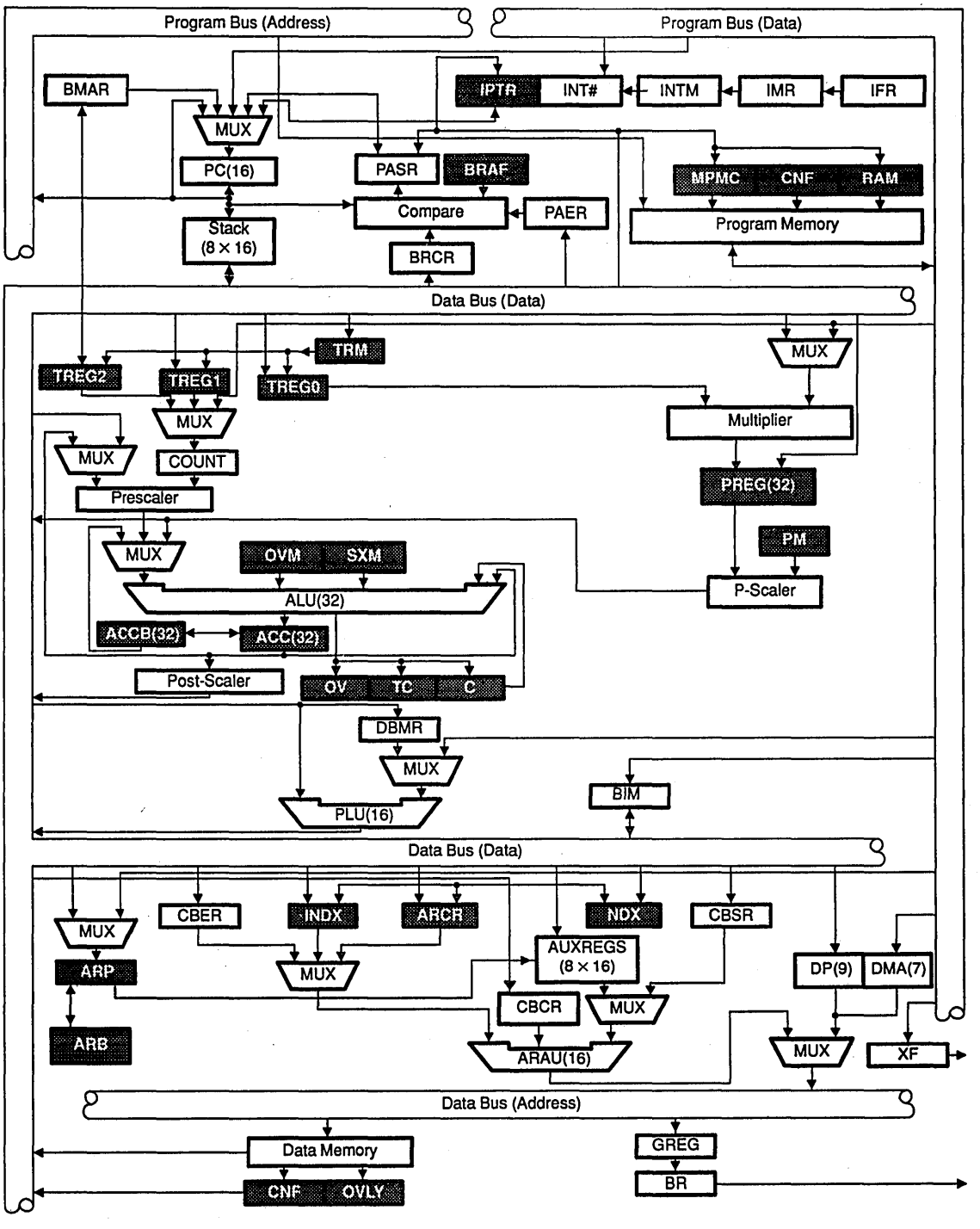


Table 3–1. TMS320C5x Internal Hardware (Continued)

Unit	Symbol	Function
Block Repeat Address Start Register	PASR(16)	A 16-bit memory-mapped register containing the start address of the segment of code being repeated. See subsection 3.6.5 for more details.
Block Repeat Counter Register	BRCR(16)	A 16-bit memory-mapped counter register used to limit the number of times the block is to be repeated. See subsection 3.6.5 for more details.
Bus Interface Module	BIM	A buffered interface used to pass data between the internal data and program buses.
Bus Request	$\overline{BR}$	This signal indicates that a data access is mapped to global memory space as defined by the GREG register. See Section 6.3 for more details.
Carry	C	This bit stores the carry output of the ALU. This bit resides in ST1. See subsection 3.5.2 for more information.
Central Arithmetic Logic Unit	CALU	The grouping of the ALU, multiplier, accumulator, and scaling shifters. See Section 3.5 for more information.
Circular Buffer Control Register	CBCR(8)	An 8-bit register used to enable/disable the circular buffers and define which auxiliary registers are mapped to the circular buffers. See subsection 3.4.3 for more information.
Circular Buffer End Address	CBER(16) CBER1(16) CBER2(16)	Two 16-bit registers indicating circular buffer end addresses. CBER1 and CBER2 are associated with circular buffers one and two, respectively. See subsection 3.4.3 for more information.
Circular Buffer Start Address	CBSR(16) CBSR1(16) CBSR2(16)	Two 16-bit registers indicating circular buffer start addresses. CBSR1 and CBSR2 are associated with circular buffers one and two, respectively. See subsection 3.4.3 for more information.
Compare of Program Address	COMPARE	This circuit compares the current value in the PC to the value in PAER if BRAF is active. If the compare shows equal, then the PASR is loaded into the PC. See subsection 3.4.3 for more information.
Configure Ram	CNF	This bit indicates whether on-chip dual-access RAM blocks are mapped to program or data space. The CNF bit resides in ST1. See subsection 3.6.3 for more information.
Data Bus	DATA	A 16-bit bus used to route data.
Data Memory	DATA MEMORY	This block refers to data memory used with the core and defined in specific device descriptions. It refers to both on- and off-chip memory blocks in data memory space.
Data Memory Address Bus	DATA ADDRESS	A 16-bit bus that carries the address for data memory accesses.
Data Memory Address Immediate Register	DMA(7)	A 7-bit register containing the immediate relative address within a 128-word data page. See subsection 3.4.2 for more information.
Data Memory Page Pointer	DP(9)	A 9-bit register containing the address of the current page. Data pages are 128 words each, resulting in 512 pages of addressable data memory space (some locations are reserved). See subsection 3.4.2 for more information.
Data RAM Map Bit	RAM(1)	This bit indicates if the single-access RAM is mapped into data space. See subsection 3.6.3 for more information.

Table 3–1. TMS320C5x Internal Hardware (Continued)

Unit	Symbol	Function
Multiplexer	MUX	A bus multiplexer used to select the source of operands for a bus or execution unit, depending on the nature of the current instruction.
Multiplier	MULTIPLIER	A 16 × 16-bit parallel multiplier. See subsection 3.5.3 for more information.
Overflow Flag	OV(1)	This bit resides in ST0 and indicates an overflow in an arithmetic operation in the ALU. See subsection 3.6.3 for more information.
Overflow Mode	OVM(1)	This bit resides in ST0 and determines whether an overflow in the ALU will wrap around or saturate. See subsection 3.6.3 for more information.
Overlay to Data Space	OVLY(1)	This bit resides in the PMST register and determines whether the on-chip single-access memory will be addressable in data address space. See subsection 3.6.3 for more information.
Parallel Logic Unit	PLU	A 16-bit logic unit that executes logic operations from either long immediate operands or the contents of the DBMR directly upon data locations without interfering with the contents of the CALU registers. See Section 3.7 for more information.
Prefetch Counter	PFC (15–0)	A 16-bit counter used to prefetch program instructions. The PFC contains the address of the instruction currently being prefetched. It is updated when a new prefetch is initiated. The PFC can also address program memory when the block move (BLPD), multiply-accumulate (MAC/MACD), and table read/write (TBLR/TBLW) instructions are used and can address data memory when the block move (BLDD) instruction is used.
Prescaler Count Register	COUNT(4)	A four-bit register that contains the value for the prescaling operation. When the register contents are used as prescaling data, this register is loaded from the dynamic shift count or from the instruction. In conjunction with the BIT and BITT instructions, this register is loaded from the dynamic bit pointer or the instruction word.
Product Register	PREG(32)	A 32-bit product register used to hold the multiplier's product. The high and low words of the PREG can be accessed individually. See subsection 3.5.3 for more information.
Program Bus	PROG DATA	A 16-bit bus used to route instructions (and data for the MAC and MACD instructions).
Program Counter	PC(16)	A 16-bit program counter used to address program memory sequentially. The PC always contains the address of the next instruction to be fetched. The PC contents are updated following each instruction decode operation.
Program Memory	PROGRAM MEMORY	This block refers to program memory used with the core and defined in specific device descriptions. It refers to both on- and off-chip memory blocks accessed in program memory space.
Program Memory Address Bus	PROG ADDRESS	A 16-bit bus that carries the program memory address.
Prescaling Shifter	PRESCALER	A 0- to 16-bit left barrel shifter used to prescale data coming into the ALU. Also used to align data for multiprecision operations. This shifter is also used as a 0- to 16-bit right barrel shifter of the ACC. See subsection 3.5.2 for more information.
Postscaling Shifter	POST-SCALER	A 0- to 7-bit left barrel shifter used to postscale data coming out of the CALU. See subsection 3.5.2 for more information.

3.4 Internal Memory Organization

This section describes the memory use of the TMS320C5x core and the addressing modes supported by the core.

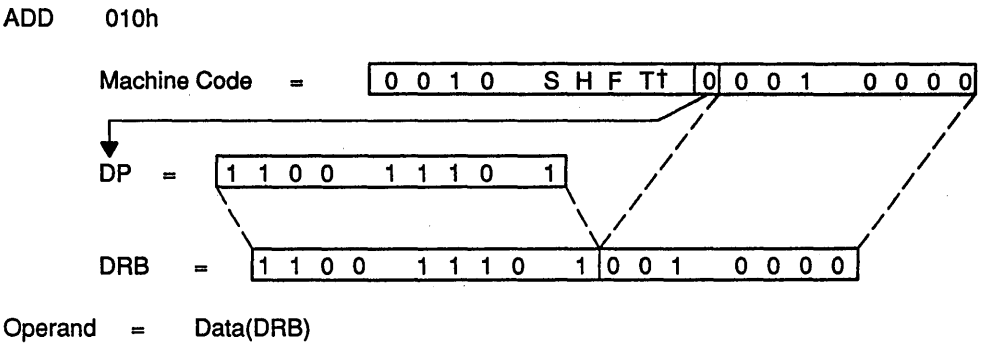
3.4.1 Memory-Mapped Registers

Twenty-eight core processor registers are mapped into the data memory space. These are listed in Table 3–2. An additional 64 memory-mapped registers are reserved in page 0 of data space. These data memory locations are reserved for peripheral control registers, which are described in Chapter 5.

Table 3–2. Memory-Mapped Registers

Name	Address		Description
	C5x Dec	C5x Hex	
—	0 – 3	0 – 3	Reserved
IMR	4	4	Interrupt mask register
GREG	5	5	Global memory allocation register
IFR	6	6	Interrupt flag register
PMST	7	7	Processor mode status register
RPTC	8	8	Repeat counter register
BRCR	9	9	Block repeat counter register
PASR	10	A	Block repeat program address start register
PAER	11	B	Block repeat program address end register
TREG0	12	C	Temporary register for multiplicand
TREG1	13	D	Temporary register for dynamic shift count
TREG2	14	E	Temporary register used as bit pointer in dynamic bit test
DBMR	15	F	Dynamic bit manipulation register
AR0	16	10	Auxiliary register zero
AR1	17	11	Auxiliary register one
AR2	18	12	Auxiliary register two
AR3	19	13	Auxiliary register three
AR4	20	14	Auxiliary register four
AR5	21	15	Auxiliary register five
AR6	22	16	Auxiliary register six
AR7	23	17	Auxiliary register seven
INDX	24	18	Index register
ARCR	25	19	Auxiliary register compare register
CBSR1	26	1A	Circular buffer 1 start address register
CBER1	27	1B	Circular buffer 1 end address register
CBSR2	28	1C	Circular buffer 2 start address register
CBER2	29	1D	Circular buffer 2 end address register
CBCR	30	1E	Circular buffer control register
BMAR	31	1F	Block move address register

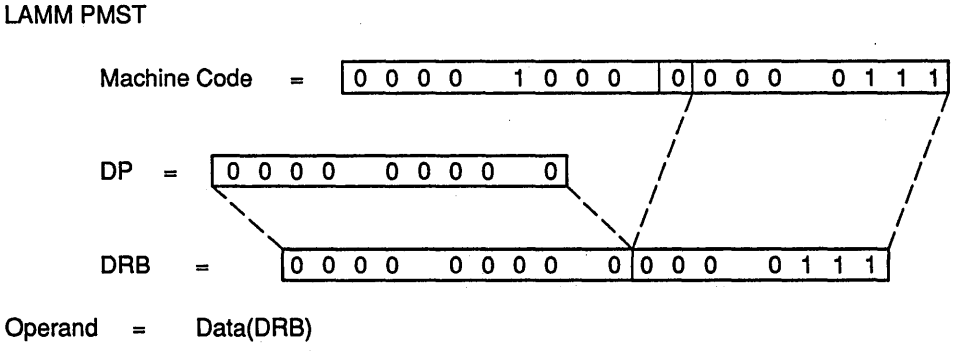
Figure 3–2. Direct Addressing Mode



† SHFT represents a 4-bit shift value.

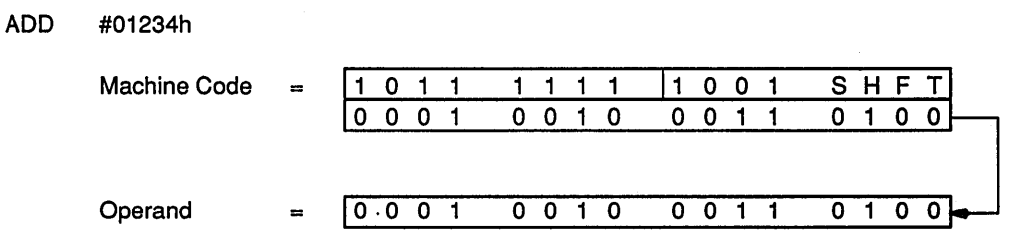
Memory-mapped addressing mode operates much like direct addressing mode except that the most significant 9 bits of the address are forced to zero instead of being loaded with the contents of the DP. This allows the user to directly address the memory-mapped registers of data page zero without the overhead of changing the DP or auxiliary register. Figure 3–3 illustrates memory-mapped addressing mode.

Figure 3–3. Memory-Mapped Addressing Mode



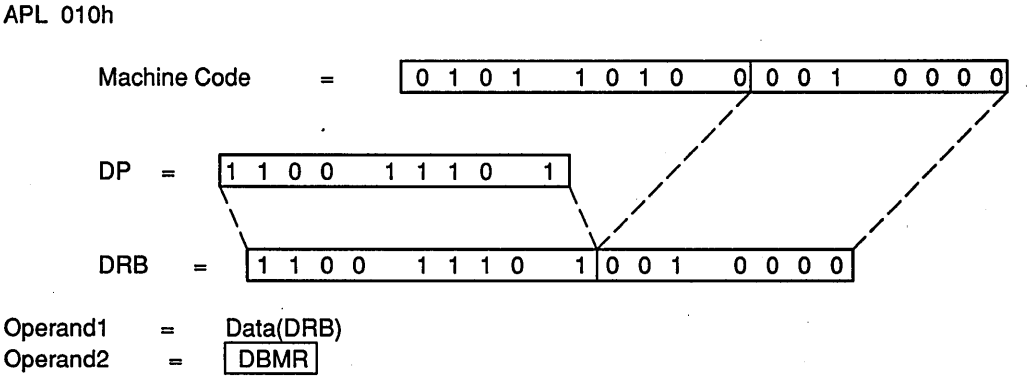
In the case of the long immediate operand, the operand immediately follows the opcode in the program sequence. The long immediate operand is 16 bits long. Figure 3–6 shows an example of long immediate mode. In this example, the second word of the two-word instruction is added to the ACC by the CALU.

Figure 3–6. Long Immediate Mode



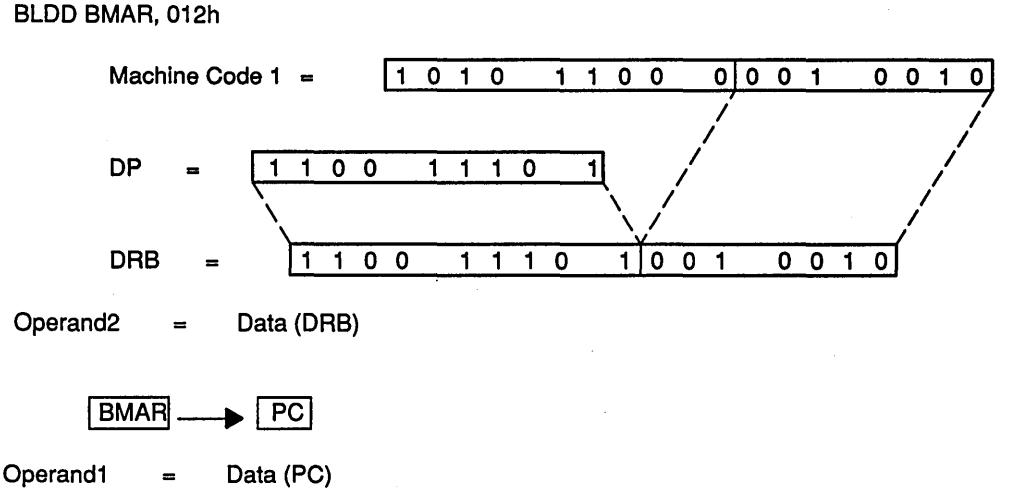
The operand may come from a CPU register. This type of operand is used in special cases. The CALU uses this in multiplying with TREG0, in shifting with TREG1 and PM, and in bit manipulation with TREG2. The ARAU uses this with INDX and ARCR. The PLU uses this with DBMR. Figure 3–7 illustrates the use of the DBMR register as an AND mask in the APL instruction.

Figure 3–7. Register Access Mode



In the long immediate addressing mode, an operand is addressed by the second word of a two-word instruction. In this case, the program address/data bus (PAB) is used for the operand fetch. The PC is stored in a temporary register, and the long immediate value is loaded into the PC. Then, the PAB is used for the operand fetch or write. At the completion of the instruction, the PC is restored from the temporary register, and execution continues. This technique is used when two memory addresses are required for the execution of the in-

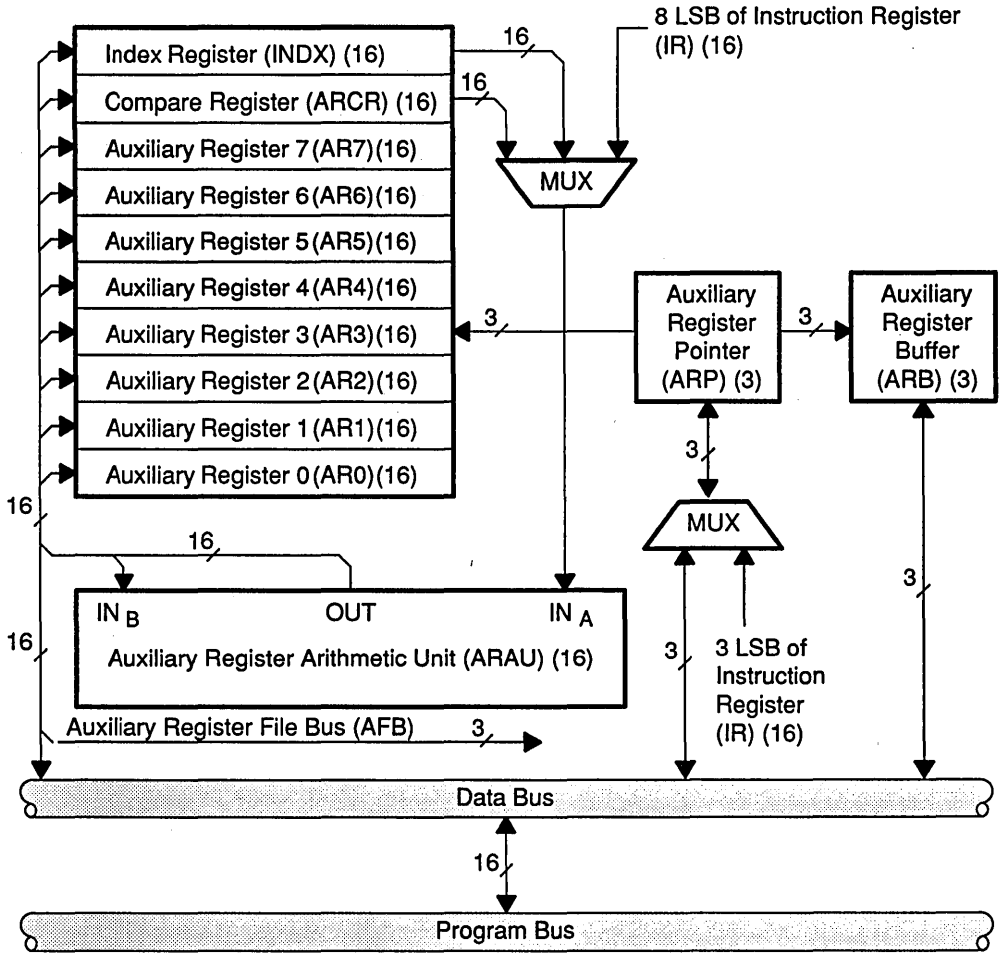
Figure 3–9. Registered Block Memory Addressing Mode



3.4.3 Auxiliary Registers

The TMS320C5x provides a register file containing eight auxiliary registers (AR0–AR7). The auxiliary registers may be used for indirect addressing of the data memory or for temporary data storage. Indirect auxiliary register addressing (see Figure 3–10) allows placement of the data memory address of an instruction operand into one of the auxiliary registers. These registers are pointed to by a three-bit auxiliary register pointer (ARP) that is loaded with a value from 0 through 7, designating AR0 through AR7, respectively. The auxiliary registers and the ARP may be loaded from data memory, the accumulator, the product register, or by an immediate operand defined in the instruction. The contents of these registers may also be stored in data memory or used as inputs to the CALU. These registers appear in the memory map as described in Table 3–2.

Figure 3–11. Auxiliary Register File



The index register (INDX) can be added to or subtracted from AR(ARP) on any AR update cycle. This 16-bit register is one of the memory-mapped registers and is used to increment or decrement the address in steps larger than one, which is useful for operations such as addressing down a column of a matrix. The auxiliary register compare register (ARCR) is used as a limit to blocks of data and, in conjunction with the CMPR instruction, supports logical comparisons between AR(ARP) and ARCR. The TMS320C25 uses AR0 for these two functions. After reset, a LAR load of AR0 also loads INDX and ARCR to maintain compatibility with the TMS320C25. The splitting of functions to the three registers is enabled by setting the NDX bit of PMST to one.

Because the auxiliary registers are memory-mapped, they can be acted upon directly by the CALU to provide for more advanced indirect addressing techniques. For example, the multiplier can be used to calculate the addresses of three-dimensional matrices. After a CALU load of the auxiliary register, there is, however, a two-instruction-cycle delay before auxiliary registers can be used for address generation. The INDX and ARCR registers are accessible via the CALU, regardless of the condition of the NDX bit (i.e., SAMM ARCR writes only to the ARCR).

In addition to its use for address manipulation in parallel with other operations, the ARAU may also serve as an additional general-purpose arithmetic unit because the auxiliary register file can directly communicate with data memory. The ARAU implements 16-bit unsigned arithmetic, whereas the CALU implements 32-bit twos-complement arithmetic. The BANZ and BANZD instructions permit the auxiliary registers to be used as loop counters, also.

The 3-bit auxiliary register pointer buffer (ARB), shown in Figure 3-11, provides storage for the ARP on subroutine calls when the automatic context switch compatibles of the device are not used.

Two circular buffers can operate at a given time and are controlled via the circular buffer control register (CBCR). The CBCR is defined as follows:

Bit	Name	Function
0–2	CAR1	Identifies which auxiliary register is mapped to circular buffer 1.
3	CENB1	Circular buffer 1 enable=1/disable=0. Set to 0 upon reset.
4–6	CAR2	Identifies which auxiliary register is mapped to circular buffer 2.
7	CENB2	Circular buffer 2 enable=1/disable=0. Set to 0 upon reset.

Upon reset ($\overline{RS}$ rising edge), both circular buffers are disabled. To define a circular buffer, load the CBSR1/2 with the start address of the buffer and CBER1/2 with the end address, and load the auxiliary register to be used with the buffer with an address between the start and end addresses. Finally, load CBCR with the appropriate auxiliary register number and set the enable bit. Note that the same auxiliary register can not be enabled for both circular buffers, or unexpected results will occur. As the address is stepping through the circular buffer, the auxiliary register value is compared against the value con-

3.5 Central Arithmetic Logic Unit (CALU)

The TMS320C5x central arithmetic logic unit (CALU) contains a 16-bit scaling shifter, a 16 x 16-bit parallel multiplier, a 32-bit arithmetic logic unit (ALU), a 32-bit accumulator (ACC), a 32-bit accumulator buffer (ACCB), and additional shifters at the outputs of both the accumulator and the multiplier. This section describes the CALU components and their functions. Figure 3–12 is a block diagram showing the components of the CALU. The following steps occur in the implementation of a typical ALU instruction:

1) Data is fetched from the RAM on the data bus,

2) Data is passed through the scaling shifter and the ALU where the arithmetic is performed, and

3) The result is moved into the accumulator.

One input to the ALU is always provided by the accumulator. The other input may be transferred from the product register (PREG) of the multiplier, the accumulator buffer (ACCB), or the scaling shifter that is loaded from data memory or the accumulator (ACC).

3.5.1 Prescaling Shifter

The TMS320C5x provides a scaling shifter that has a 16-bit input connected to the data bus and a 32-bit output connected to the ALU; see Figure 3–12. The scaling shifter produces a left shift of 0 to 16 bits on the input data. The shift count is specified by a constant embedded in the instruction word or by the value in TREG1. The LSBs of the output are filled with zeros; the MSBs may be either filled with zeroes or sign-extended, depending upon the value of the SXM bit (sign-extension mode) of status register ST1.

The TMS320C5x also contains several other shifters that allow it to perform numerical scaling, bit extraction, extended-precision arithmetic, and overflow prevention. These shifters are connected to the output of the product register and the accumulator.

3.5.2 ALU and Accumulator

The TMS320C5x 32-bit ALU and accumulator implement a wide range of arithmetic and logical functions, the majority of which execute in a single clock cycle. Once an operation is performed in the ALU, the result is transferred to the accumulator where additional operations, such as shifting, may occur. Data that is input to the ALU may be scaled by the prescaling shifter.

The ALU is a general-purpose arithmetic/logic unit that operates on 16-bit words taken from data RAM or derived from immediate instructions. In addition to the usual arithmetic instructions, the ALU can perform Boolean operations, facilitating the bit manipulation ability required of a high-speed controller. One input to the ALU is always supplied by the accumulator, and the other input may be furnished from the product register (PREG) of the multiplier, the accumulator buffer (ACCB), or the output of the scaling shifter (that has been read from data RAM or from the ACC). After the ALU has performed the arithmetic or logical operation, the result is stored in the accumulator. For the following example, assume ACC = 0, PREG = 000222200h, PM = 00, and ACCB = 000333300h:

```
LACC #01111h,8   ;ACC = 00111100. Load ACC from pre-
                 ;scaling shifter.
APAC             ;ACC = 00333300. Add to ACC the
                 ;product register.
ADDB             ;ACC = 00666600. Add to ACC the
                 ;accumulator buffer.
```

The 32-bit accumulator (ACC) can be split into two 16-bit segments for storage in data memory; see Figure 3–12. Shifters at the output of the accumulator provide a left shift of 0 to 7 places. This shift is performed while the data is being transferred to the data bus for storage. The contents of the accumulator remain unchanged. When the postscaling shifter is used on the high word of the accumulator (bits 16–31), the MSBs are lost and the LSBs are filled with bits shifted in from the low word (bits 0 – 15). When the postscaling shifter is used on the

more efficient computation of extended-precision products and additions or subtractions. It is quite useful in overflow management. The carry bit is affected by most arithmetic instructions as well as the single-bit shift and rotate instructions. It is not affected by loading the accumulator, logical operations, or other such non-arithmetic or control instructions. Examples of carry bit operations are shown in Figure 3–13.

Figure 3–13. Examples of Carry Bit Operations

```
C  MSB          LSB                C  MSB          LSB
X   F F F F F F F F ACC            X   0 0 0 0 0 0 0 0 ACC
   +             1                   -             1
1   0 0 0 0 0 0 0 0                0   F F F F F F F F

C  MSB          LSB                C  MSB          LSB
X   7 F F F F F F F ACC            X   8 0 0 0 0 0 0 1 ACC
   +             1 (OVM=0)           -             2 (OVM=0)
0   8 0 0 0 0 0 0 0                1   7 F F F F F F F

C  MSB          LSB                C  MSB          LSB
1   0 0 0 0 0 0 0 0 ACC            0   F F F F F F F F ACC
   +             0 (ADDC)            -             1 (SUBB)
0   0 0 0 0 0 0 0 1                1   F F F F F F F d
```

Shown in the examples of Figure 3–13, the value added to or subtracted from the accumulator may come from the input scaling shifter, ACCB, or PREG. The carry bit is set if the result of an addition or accumulation process generates a carry; it is reset to zero if the result of a subtraction generates a borrow. Otherwise, it is cleared after an addition or set after a subtraction.

The ADDC (add to accumulator with carry) and SUBB (subtract from accumulator with borrow) instructions use the previous value of carry in their addition/subtraction operation. The ADCB (add ACCB to accumulator with carry) and the SBBB (subtract ACCB from accumulator with borrow) also use the previous value of carry.

The one exception to operation of a carry bit, as shown in Figure 3–13, is in the use of ADD with a shift count of 16 (add to high accumulator) and SUB with a shift count of 16 (subtract from high accumulator). This case of the ADD instruction can set the carry bit only if a carry is generated, and this case of the SUB instruction can reset the carry bit only if a borrow is generated; otherwise, neither instruction affects it.

Two conditional operands, C and NC, are provided for branching, calling, returning, and conditionally executing according to the status of the carry bit. The CLRC, LST #1, and SETC instructions can also be used to load the carry bit. The carry bit is set to one on a hardware reset.

The SFL and SFR (in-place one-bit shift to the left/right) instructions and the ROL and ROR (rotate to the left/right) instructions shift or rotate the contents

fractional arithmetic, or justifying fractional products. The PM field of status register ST1 specifies the PM shift mode, as shown in Table 3–3.

Table 3–3. Product Shift Modes

PM	Resulting Shift
00	No shift
01	Left shift of 1 bit
10	Left shift of 4 bits
11	Right shift of 6 bits

The product is shifted one bit to compensate for the extra sign bit gained in multiplying two 16-bit twos-complement numbers (MPY). The four-bit shift is used in conjunction with the MPY instruction with a short immediate value (13 bits or less) to eliminate the four extra sign bits gained in multiplying a16-bit number times a 13-bit number. The output of PREG can, instead, be right-shifted 6 bits to enable the execution of up to 128 consecutive multiply/accumulates without the possibility of overflow. Note that, when the right shift is specified, the product is always sign-extended, regardless of the value of SXM.

The LT (load TREG0) instruction normally loads TREG0 to provide one operand (from the data bus), and the MPY (multiply) instruction provides the second operand (also from the data bus). A multiplication can also be performed with a short or long immediate operand by using the MPY instruction with an immediate operand. A product can be obtained every two cycles except when a long immediate operand is used.

Four multiply/accumulate instructions (MAC, MACD, MADD, and MADS) fully utilize the computational bandwidth of the multiplier, allowing both operands to be processed simultaneously.The data for these operations can be transferred to the multiplier each cycle via the program and data buses. This facilitates single-cycle multiply/accumulates when used with repeat (RPT and RPTZ) instructions. In these instructions, the coefficient addresses are generated by the PC, while the data addresses are generated by the ARAU. This allows the repeated instruction to sequentially access the values from the coefficient table and step through the data in any of the indirect addressing modes. The RPTZ instruction also clears the accumulator and the product register to initialize the multiply/accumulate operation. As an example, consider multiplying the row of one matrix times the column of a second matrix. For this example, consider 10 x 10 matrices, MTRX1 points to the beginning of the first matrix, INDX = 10, and AR(ARP) points to the beginning of the second matrix:

```
RPTZ #9          ;For i = 0, i < 10, i+=.
MAC  MTRX1,*0+ ;PREG = DATA(MTRX1 + i) x DATA[MTRX2 + (i x INDX)].
               ;ACC += PREG.
APAC           ;ACC += PREG.
```

The MAC and MACD instructions obtain their coefficient pointer from a long immediate address and are, therefore, two-word instructions. The MADS and

3.6 System Control

System control on the TMS320C5x is provided by the program counter, hardware stack, PC-related hardware, external reset signal, interrupts (see Section 3.8), status registers, and repeat counters. The following subsections describe the function of each of these components in system control and pipeline operation.

3.6.1 Program Address Generation and Control

The TMS320C5x has a 16-bit program counter (PC) and an eight-deep hardware stack for PC storage. The program counter addresses internal and external program memory in fetching instructions. The stack is used during interrupts and subroutines.

The program counter addresses program memory, either on-chip or off-chip, via the program address bus (PAB). Through the PAB, an instruction is addressed in program memory and loaded into the instruction register (IREG). When the IREG is loaded, the PC is ready to start the next instruction fetch cycle.

The PC can be loaded in a number of ways. When code is sequentially executed, the PC is loaded with PC + 1. When a branch is executed, the PC is loaded with the long immediate value directly following the branch instruction. In the case of a subroutine call, the PC+2 is pushed onto the stack and then loaded with the long immediate value directly following the call instruction. The return instructions pop the stack back into the PC to return to the calling or interrupting sequence of code. In the case of a software trap or interrupt trap, the PC is loaded with the address of the appropriate trap vector. The contents of the accumulator may be loaded into the PC in order to implement computed GOTO operations. This can be accomplished with the BACC (branch to address in accumulator) or CALA (call subroutine at location specified by ACC) instructions.

The PAB bus can also address data stored in either program or data space. This makes it possible, in repeated instructions, to fetch a second operand in parallel with the data bus for two-operand operations. When repeated, the array addressed by the PAB is sequentially accessed via the incrementing of the PC. The block transfer instructions (BLDD, BLDP, and BLPD) use both buses so that, when repeated, the pipeline structure can be reading the next operand while writing the current one. The BLPD instruction loads the PC with either the long immediate address following the BLPD or with the contents of the block move address register (BMAR). The PAB bus is then used to fetch the source data from program space in this block move operation. The BLDP executes much the same except that the PAB bus is used for the destination operation. The BLDD instruction uses the PAB bus to address data space.

The TBLR and TBLW instructions operate much like the BLPD and BLDP instructions, respectively, except that the PC is loaded with the low 16 bits of the

the PC is loaded with the second word and the core CPU starts refilling the pipeline with instructions at the branch address. Because the pipeline has been flushed, the branch instruction has an effective execution time of four cycles if the branch is taken. If, however, any of the conditions are not met, the pipeline controller allows the next instruction (already fetched) to be decoded. This means that if the branch is not taken, the effective execution time of the branch is two cycles.

The subroutine call can also be executed conditionally. The CC instruction operates like the BCND except that the PC pointing to the instruction following the CC is pushed onto the PC stack. This sets up the return (by RET) to pop the stack to return to the calling sequence. A subroutine or function can have multiple return paths based upon the data being processed. Using conditional returns (RETC) avoids the need for conditionally branching around the return. For example,

```
    CC OVER_FLOW,OV     ;If overflow,then execute the
       .                ;overflow-handling routine.
       .
       .
OVER_FLOW               ;Overflow-handling routine.
       .
       .
       .
    RETC GEQ            ;If ACC >= 0, then return.
       .
       .
       .
    RET                 ;Return.
```

In the example, an overflow-handling subroutine is called if the main algorithm causes an overflow condition. During the subroutine, the ACC is checked and, if it is positive, the subroutine returns to the calling sequence. If not, additional processing is necessary before the return. Note that RETC, like RET, is a single-word instruction. However, because of the potential PC discontinuity, it still operates with the same effective execution time as BCND and CC.

To avoid flushing the pipeline and causing extra cycles, the TMS320C5x has a full set of delayed branches, calls, and returns. In the delayed operation of branches, calls, or returns, the two-instruction words following the delayed instruction are executed while the instructions at and following the branch address are being fetched—therefore, giving an effective two-cycle branch instead of flushing the pipeline. If the instruction following the delayed branch is a two-word instruction, then only it will be executed. For example,

```
OPL    #030h,PMST
BCND   NEW_ADRS,EQ
```

or

```
BCNDD  NEW_ADRS,EQ
OPL    #030h,PMST.
```

to the XC and after the ADD so that the SPLK will not execute. In the second code segment, TEMP2 is not set to EEEE. The NEQ status, caused by the ADD instruction, is established one full cycle before the XC execution phase because the long immediate value (#01234h) used in the ADD caused it to be a two-cycle instruction. Since the condition is not met, a NOP is forced over both words of the two-word SPLK instruction, and, therefore, TEMP2 is not affected. Note that interrupts will have no effect on this instruction sequence.

The TMS320C5x also has a feature that allows the execution of a single instruction N + 1 times where N is the value loaded in a 16-bit repeat counter (RPTC). If the repeat feature is used, the instruction is executed and the RPTC is decremented until the RPTC goes to zero. This feature is useful with many instructions, such as NORM (normalize contents of accumulator), MACD (multiply and accumulate with data move), and SUBC (conditional subtract). As instructions repeat, the program address and data buses are freed to fetch a second operand in parallel with the data address and data buses. This allows instructions such as MACD and BLPD to effectively execute in a single cycle when they repeat. See Section 7.6, *Single Instruction Repeat Loops*, for details on these instructions.

The stack is 16 bits wide and eight levels deep. The PC stack is accessible through the use of the PUSH and POP instructions. Whenever the contents of the PC are pushed onto the top of the stack, the previous contents of each level are pushed down, and the bottom (eighth) location of the stack is lost. Therefore, data will be lost if more than eight successive pushes occur before a pop. The reverse happens on pop operations. Any pop after seven sequential pops yields the value at the bottom stack level, and all of the stack levels then contain the same value. Two additional instructions, PSHD and POPD, push a data memory value onto the stack or pop a value from the stack to data memory. These instructions allow a stack to be built in data memory for the nesting of subroutines/interrupts beyond eight levels. See Section 7.3, *Software Stack*, for details on software stack.

3.6.2 Pipeline Operation

Instruction pipelining consists of the sequence of bus operations that occur during instruction execution. In the operation of the pipeline, the instruction fetch, decode, operand fetch, and execute operations are independent, which allows overall instruction executions to overlap. Thus, during any given cycle, one to four different instructions can be active, each at a different stage of completion, resulting in a four-deep pipeline. Figure 3–14 shows the operation of the four-level pipeline for single-word single-cycle instructions executing with no wait states. The pipeline is essentially invisible to the user except in some cases, such as auxiliary register updates, memory-mapped accesses of the CPU registers, the NORM instruction, and memory configuration commands.

or

```
EXAM3  LAR    AR2,#067h ;AR2 = 67.
       LACC   #064h     ;ACC = 00000064.
       SAMM   AR2       ;AR2 = 64.
       NOP              ;Pipeline protection.
       NOP              ;Pipeline protection.
       LACC   *-        ;AR2 = 63.
       ADD    *-        ;AR2 = 62.
```

In EXAM1, the decode phase of the ADD instruction is on the same cycle as the execute (write) phase of the SAMM instruction. Both of these instructions are trying to load AR2. The ADD *– update does load AR2, while the SAMM execution is voided. In EXAM2, a NOP is strategically placed to avoid the conflict between the ADD *– update of the AR2 and the SAMM write to AR2. In this code's sequence,

$$AR2 = 67 \rightarrow 66 \rightarrow 64 \rightarrow 63$$

Note that the LACC address is based on the value in AR2 before the SAMM write to AR2. In EXAM3, the SAMM write to AR2 is completed before either the LACC or the ADD have updated AR2. Any two instruction words that do not update AR2 can be used in place of the two NOP instructions. This could be two one-word instructions or one two-word instruction. The results obtained by EXAM1 and EXAM2 code examples may be different if the code is interruptible. The user should avoid writing code similar to EXAM1 and EXAM2.

The pipeline effect described above requires writes to memory-mapped registers to allow for a latency between the write and an access of that register. These registers can be accessed by TMS320C5x instructions in the decode and operand fetch phases of the pipeline. Table 3–4 outlines the latency required between an instruction that writes the register and the access of that register.

3.6.3 Status and Control Registers

There are four key status and control registers for the TMS320C5x core. ST0 and ST1 contain the status of various conditions and modes compatible with the TMS320C25, while PMST and CBCR contain extra status and control information for control of the enhanced features of the TMS320C5x core. These registers can be stored into data memory and loaded from data memory, thus allowing the status of the machine to be saved and restored for subroutines. ST0, ST1, and PMST each have an associated one-deep stack for automatic context-saving when an interrupt trap is taken. The stack is automatically popped upon a return from interrupt. Note that the XF bit in ST1 is not saved on the one-deep stack or restored from that stack on an automatic context save. This feature allows the XF pin to be toggled in an interrupt service routine while still allowing automatic context saves.

The PMST and CBCR registers reside in the memory-mapped register space in page zero of data memory space. Therefore, they can be acted upon directly by the CALU and the PLU. They can be saved in the same way as any other data memory location. Note that the CALU and the PLU operations change the bits of these status registers during the execute phase of the pipeline. The next two instruction words, following an update of these status registers, may not be affected by the reconfiguration caused by the status update.

The LST instruction writes to ST0 and ST1, and the SST instruction reads from them, except that the INTM bit is not affected by the LST instruction. Unlike the PMST and CBCR registers, the ST0 and ST1 registers do not reside in the memory map and, therefore, cannot be handled by using the PLU instructions. The individual bits of these registers can be set or cleared with the SETC and CLRC instructions. For example, the sign-extension mode is set with SETC SXM or cleared with CLRC SXM.

Figure 3–15 shows the organization of the four status registers, indicating all status bits contained in each. Several bits in the status registers are reserved and read as logic ones. Table 3–5 defines all the status/control bits.

Table 3–5. Status Register Field Definitions (Continued)

Field	Function
C	Carry Bit. This bit is set to 1 if the result of an addition generates a carry, or is reset to 0 if the result of a subtraction generates a borrow. Otherwise, it is reset after an addition or is set after a subtraction, unless the instruction is ADD or SUB with a 16-bit shift. In these cases, the ADD can only set and the SUB only reset the carry bit, but they cannot affect it otherwise. The single-bit shift and rotate instructions, as well as the SETC, CLRC, and LST #1 instructions also affect this bit. C is set to 1 on a reset.
CAR1	Circular Buffer 1 Auxiliary Register. These three bits identify which auxiliary register is assigned to circular buffer 1.
CAR2	Circular Buffer 2 Auxiliary Register. These three bits identify which auxiliary register is assigned to circular buffer 2.
CENB1	Circular Buffer 1 Enable. This bit, when set to 1, enables circular buffer 1. When CENB1 is set to 0, circular buffer 1 is disabled. CENB1 is set to zero upon reset.
CENB2	Circular Buffer 2 Enable. This bit, when set to 1, enables circular buffer 2. When CENB2 is set to 0, circular buffer 2 is disabled. CENB2 is set to zero upon reset.
CNF	On-chip RAM Configuration Control Bit. If this bit is set to 0, the reconfigurable-data dual-access RAM blocks are mapped to data space; otherwise, they are mapped to program space. The CNF may be modified by the SETC CNF, CLRC CNF, and LST #1 instructions. $\overline{RS}$ sets the CNF to 0.
DP	Data Memory Page Pointer. The 9-bit DP register is concatenated with the 7 LSBs of an instruction word to form a direct memory address of 16 bits. DP may be modified by the LST and LDP instructions.
HM	Hold Mode Bit. When HM = 1, the processor halts internal execution when acknowledging an active $\overline{HOLD}$. When HM = 0, the processor may continue execution out of internal program memory but puts its external interface in a high-impedance state. This bit is set to 1 by reset.
INTM	Interrupt Mode Bit. When this bit is set to 0, all unmasked interrupts are enabled. When it is set to 1, all maskable interrupts are disabled. INTM is set and is reset by the SETC INTM and CLRC INTM instructions. $\overline{RS}$ and $\overline{IACK}$ also set INTM. INTM has no effect on the unmaskable $\overline{RS}$ and $\overline{NMI}$ interrupts. Note that INTM is unaffected by the LST instruction. This bit is set to 1 by reset. It is also set to 1 when a maskable interrupt trap is taken. It is reset to 0 when a RETE (return from interrupt with interrupt enable) is executed.
IPTR	Interrupt Vector Pointer. These five bits point to the 2K page where the interrupt vectors reside. This allows the user to remap the interrupt vectors to RAM for boot-loaded operations. At reset, these bits are all set to zero. Therefore, the reset vector always resides at zero in the program memory space.
MP/$\overline{MC}$	Microprocessor/Microcomputer Bit. When this bit is set to zero, the on-chip ROM is enabled. When it is set to one, the on-chip ROM is not addressable. This bit is set to the value corresponding to the logic level on the MP/$\overline{MC}$ pin at reset. The level on the MP/$\overline{MC}$ pin is sampled at device reset only and can have no effect until the next reset.
NDX	Enable Extra Index Register. This bit configures indexed indirect addressing and auxiliary address register compare to operate either in a TMS320C2x-compatible mode (NDX = 0) or in a TMS320C5x-enhanced mode (NDX = 1). When NDX = 0, the LAR AR0 instruction loads the INDX and ARCR registers in addition to AR0. This is because the TMS320C2x devices use AR0 for indexing and AR compare operations. When NDX = 1, INDX and ARCR are not affected by the LAR instruction. NDX = 0 at reset.
OV	Overflow Flag Bit. As a latched overflow signal, OV is set to 1 when overflow occurs in the ALU. Once an overflow occurs, the OV remains set until a reset, BCND(D) on OV/NOV, or LST instruction clears OV.

Table 3–6. On-Chip RAM† Configuration Control

Device	OVLY	RAM	Configuration
TMS320C50	0	0	On-chip 9K RAM is disabled
TMS320C50	0	1	On-chip 9K RAM is mapped into program space
TMS320C50	1	0	On-chip 9K RAM is mapped into data space
TMS320C50	1	1	On-chip 9K RAM is in both program and data spaces
TMS320C51	0	0	On-chip 1K RAM is disabled
TMS320C51	0	1	On-chip 1K RAM is mapped into program space
TMS320C51	1	0	On-chip 1K RAM is mapped into data space
TMS320C51	1	1	On-chip 1K RAM is in both program and data spaces

† Excluding on-chip dual-access RAM blocks.

3.6.4 Repeat Counter

RPTC is a 16-bit repeat counter, which, when loaded with a number N, causes the next single instruction to be executed N + 1 times. The RPTC register is loaded by either the RPT or the RPTZ instruction. This results in a maximum of 65,536 executions of a given instruction. RPTC is cleared by reset. The RPTZ instruction clears both ACC and PREG before the next instruction starts repeating. Once a repeat instruction (RPT or RPTZ) is decoded, all interrupts (except reset) are masked until the completion of the repeat loop. The RPTC register resides in the CPU's memory-mapped register space; however, you should avoid writing to this register.

The repeat function can be used with instructions such as multiply/accumulates (MAC and MACD), block moves (BLDD and BLPD), I/O transfers (IN/OUT), and table read/writes (TBLR/TBLW). These instructions, although normally multicycle, are pipelined when the repeat feature is used, and they effectively become single-cycle instructions. For example, the table read instruction may take three or more cycles to execute, but when the instruction is repeated, a table location can be read every cycle. Note that not all instructions can be repeated. Table 3–7 lists all of the TMS320C5x instructions, segregated according to their repeatability.

Table 3–7a. Repeatable Instructions (Concluded)

Repeatable Instructions		Description
PUSH		;Push low ACC to the PC stack
ROL		;Rotate ACC left once
ROLB		;Rotate combined ACC and ACCB left once
ROR		;Rotate ACC right once
RORB		;Rotate combined ACC and ACCB right once
SACH		;Store high ACC with shift
SACL		;Store low ACC with shift
SAMM		;Store low ACC direct/indirect to data page 0
SAR	AR,*	;Store AR indirect addressed
SATH		;Shift ACC right 0 or 16 bits as specified by TREG1(4)
SATL		;Shift ACC right 0 to 15 bits as specified by TREG1(0–3)
SBB		;Subtract ACCB from ACC
SBBB		;Subtract ACCB from ACC with borrow
SFL		;Shift ACC left once
SFLB		;Shift combined ACC and ACCB left once
SFR		;Shift ACC right once
SFRB		;Shift combined ACC and ACCB right once
SMMR		;Store memory-mapped register
SPAC		;Subtract PREG from ACC
SPH		;Store high PREG to direct/indirect addressed
SPL		;Store low PREG to direct/indirect addressed
SQRA		;Add PREG to ACC and square direct/indirect addressed
SQRS		;Subtract PREG from ACC and square direct/indirect addressed
SST		;Store status registers
SUB	dma,shft	;Subtract from ACC direct addressed with shift
SUB	*,shft	;Subtract from ACC indirect addressed with shift
SUBB		;Subtract from ACC direct/indirect with borrow
SUBC		;Conditional subtract from ACC direct/indirect
SUBS		;Subtract from low ACC direct/indirect with sign suppressed
SUBT		;Subtract from ACC direct/indirect with shift specified by TREG1
TBLR		;Read from program space to data space
TBLW		;Write from data space to program space
XPL		;XOR DBMR to direct/indirect addressed

Table 3–7b. Instructions Not Meaningful to Repeat (Concluded)

Instructions Not Meaningful to Repeat	Description
SPM	;Set PREG shift mode
XOR	;XOR to low ACC direct/indirect
XORB	;XOR ACCB to ACC
ZALR	;Zero low ACC, load high ACC with rounding
ZAP	;Zero ACC and PREG
ZPR	;Zero PREG

Table 3–7c. Nonrepeatable Instructions

Nonrepeatable Instructions		Description
ADD	*#k*	;Add to ACC short immediate
ADD	*#lk,shift*	;Add to ACC long immediate with shift
ADRK		;Add to AR short immediate
AND	*#lk,shft*	;AND to ACC long immediate with shift
APL	*#lk*	;AND long immediate to direct/indirect addressed
B[D]		;Branch [delayed] unconditionally
BACC[D]		;Branch [delayed] to address specified in low ACC
BANZ[D]		;Branch [delayed] on AR(ARP) not zero
BCND[D]		;Branch [delayed] conditionally
CALA[D]		;Call [delayed] to address specified in low ACC
CALL[D]		;Call [delayed] subroutine
CC[D]		;Call [delayed] subroutine conditionally
CPL	*#lk*	;Compare long immediate to direct/indirect addressed
IDLE		;Idle CPU
IDLE2		;Idle until interrupt — low power mode
INTR		;Soft interrupt
LACC	*#lk,shft*	;Load ACC long immediate
LACL	*#k*	;Load ACC short immediate
LAR	*#lk*	;Load AR with long immediate
LDP	*#k*	;Load DP short immediate
NMI		;Non-maskable interrupt
OPL	*#lk*	;OR long immediate to direct/indirect addressed
OR	*#lk,shft*	;OR to ACC long immediate with shift
RCND[D]		;Return [delayed] from subroutine conditionally
RET		;Return from subroutine
RETE		;Return from interrupt service routine with automatic global enable

```
    SPLK   #010h,BRCR    ;Set loop count to 16.
    RPTB   END_LOOP-1    ;For I = BRCR; I > =0; I- -.
*
    ZAP                  ;ACC = PREG = 0.
    SQRA   *,AR2         ;PREG = X2.
    SPL    SQRX          ;Save X2.
    MPY    *             ;PREG = b x X.
    LTA    SQRX          ;ACC = bX.  TREG = X2.
    MPY    *             ;PREG = aX2.
    APAC                 ;ACC = aX2 + bX.
    ADD    *,0,AR3       ;ACC = aX2 + bX + c = Y.
    SACL   *,0,AR1       ;Save Y.
    CRGT                 ;Save MAX.
END_LOOP
```

The example implements 16 executions of $Y = aX^2 + bX + c$ and saves the maximum value in ACCB. Note that the initialization of the auxiliary registers is not shown in the coded example. PAER is loaded with the address of the last word in the code segment. The label END_LOOP is placed after the last instruction, and the RPTB instruction long immediate is defined as END_LOOP–1 in case the last word in the loop is a two-word instruction.

There is only one set of block repeat registers, so multiple block repeats cannot be nested without saving the context of the outside block or using BANZD. The simplest method of executing nested loops is to use the RPTB for only the innermost loop and using BANZD for all the outer loops. This is still a valuable cycle-saving operation because the innermost loop is repeated significantly more times than the outer loops. Block repeats can be nested by storing the context of the outer loop before initiating the inner loop, then restoring the outer loop's context after completing the inner loop. The context save and restore are shown in the following example:

```
    SMMR BRCR,TEMP1      ;Save block repeat counter.
    SMMR PASR,TEMP2      ;Save block start address.
    SMMR PAER,TEMP3      ;Save block end address.

    SPLK #NUM_LOOP,BRCR ;Set inner loop count.
    RPTB END_INNER       ;For I = 0; I<=BRCR; I++.
      .
      .
      .
END_INNER
    OPL  #1,PMST         ;Set BRAF to continue outer loop.
    LMMR BRCR,TEMP1      ;Restore block repeat counter.
    LMMR PASR,TEMP2      ;Restore block start address.
    LMMR PAER,TEMP3      ;Restore block end address.
```

In this example, the context save and restore operations take 14 cycles. Note that repeated single and BANZ/BANZD loops can also be inside a block repeat. The repeated code can include subroutine calls. Upon returning, the block repeat resumes. Repeated blocks can be interrupted. When an enabled interrupt occurs during a repeated block of code, the CALU traps to the interrupt and, when the ISR returns, the block repeat resumes.

of the last word of the table. Notice that the label marking the end of the loop is placed after the last instruction, then the PAER is loaded with that label, minus 1. It is possible to place the label before the CALA instruction, then load the PAER with the label address because this is a one-word instruction. However, if the last instruction in this loop had been a two-word instruction, the second word of the instruction would not be read, and the long immediate operand would be substituted with the first instruction in the loop.

Inside the loop, the pointer to the task table is incremented and saved. Then, the task address is read from the table and loaded into the accumulator. Next, the task is called by the CALA instruction. Notice that, when the task returns to the task handler, it returns to the top of the loop. This is because the PC has already been loaded with the PASR before the CALA executes the PC discontinuity. Therefore, when the CALA is executed, the address of the top of the loop is pushed onto the PC stack.

The last two words of a repeat-block loop are not interruptible. In other words, the interrupt path will not be taken while the last two instruction words of a repeat block are being fetched.

Example 3–1. Interrupt Operation With a Single-Word Instruction at the End of an RPTB

```
            RPTB    END_LOOP-1
            SAR     AR0,*           ← interrupt path taken here
                                      if not the last loop iteration
            .
            .
            .
            LACC    *+
            SACL    *               ← interrupt occurs here
ENDLOOP:
            MAR     *,AR1           ← Interrupt path taken here if interrupt
                                      occurs during last two instruction words
                                      of the last loop iteration
```

Example 3–2. Interrupt Operation With a Two-Word Instruction at the End of an RPTB

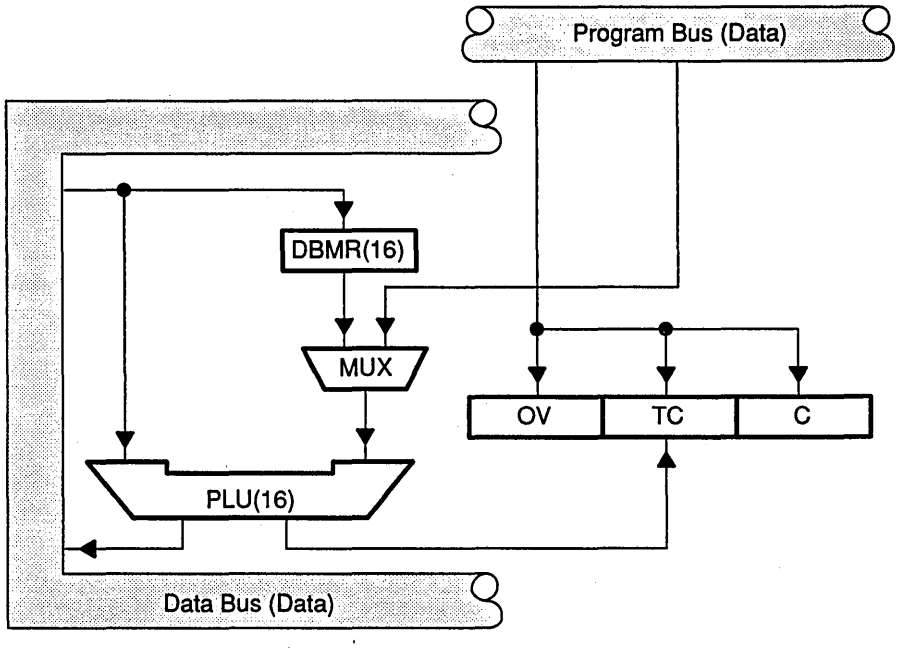
```
            RPTB    END_LOOP-1
            SAR     AR0,*           ← interrupt path taken here
                                      if not the last loop iteration
            .
            .
            .
            LACC    *+
            SPLK    #1234h,*        ← interrupt occurs here
ENDLOOP:
            MAR     *,AR1           ← Interrupt path taken here if interrupt
                                      occurs during last two instruction words
                                      of the last loop iteration
```

Note that any incoming interrupt will be latched by the TMS320C5x as soon as it meets the interrupt timing requirement. However, the PC will not branch

3.7 Parallel Logic Unit (PLU)

The parallel logic unit (PLU) can directly set, clear, test, or toggle multiple bits in a control/status register or any data memory location. The PLU, shown in the block diagram in Figure 3–16, provides a direct logic operation path to data memory values without affecting the contents of the accumulator or product register. It can be used to set or clear multiple bits in a control register or to test multiple bits in a flag register.

Figure 3–16. Parallel Logic Unit Block Diagram



The PLU executes a read-modify-write operation on data stored in data space. The PLU operation begins with the fetching of one operand from data memory space and the fetching of the second from either long immediate on the program bus or the dynamic bit manipulation register (DBMR). Then, the PLU executes a logical operation defined by the instruction on the two operands. The result is written to the same data memory location from which the first operand was fetched.

The PLU allows the direct manipulation of bits in any location in data memory space. This direct bit manipulation is done by ANDing, ORing, XORing, or loading a 16-bit long immediate value to a data location. For example, to use AR1 for circular buffer 1 and AR2 for circular buffer 2 but not enable the circular buffers, initialize the circular buffer control register (CBCR) by executing this:

```
SPLK #021h,CBCR  ;Store peripheral long immediate.;(DP = 0).
```

To later enable circular buffers 1 and 2, execute

3.8 Interrupts

The TMS320C5x core CPU supports sixteen user-maskable interrupts ($\overline{INT16}$–$\overline{INT1}$). However, each TMS320C5x DSP does not necessarily use all 16. For example, the TMS320C50 and TMS320C51 use only nine of these interrupts (the others are tied high inside the device). Interrupts can be generated by the serial ports (RINT and XINT), by the timer (TINT), and by the software interrupt (TRAP) instruction. The reset ($\overline{RS}$) interrupt has the highest priority, and the $\overline{INT16}$ interrupt has the lowest priority.

3.8.1 Reset

Reset ($\overline{RS}$) is a nonmaskable external interrupt that can be used at any time to put the TMS320C5x into a known state. Reset is typically applied after power-up when the machine is in an unknown state.

Driving the $\overline{RS}$ signal low causes the TMS320C5x to terminate execution and forces the program counter to zero. $\overline{RS}$ affects various registers and status bits. At power-up, the state of the processor is undefined. For correct system operation after power-up, a reset signal must be asserted low for one full clock cycle. The device will latch the reset pulse and generate an internal reset pulse of five cycles, long enough to guarantee a reset of the device. Processor execution begins at location 0, which normally contains a branch instruction to the system initialization routine.

When the $\overline{RS}$ signal is received, the following actions occur:

1) A logic 0 is loaded into the CNF (configuration control) bit in status register ST1, mapping dual-access RAM block 0 into data address space.

2) The program counter (PC) is set to 0. The address bus (lines A15 – A0) is unknown while $\overline{RS}$ is low, unless the $\overline{HOLD}$ input of the device is low. In this case, the address lines are placed into a high-impedance state until $\overline{HOLD}$ is brought back high.

3) All interrupts are disabled by setting the INTM bit (interrupt mode) to 1; note that $\overline{RS}$ is nonmaskable. The interrupt flag register (IFR) is cleared.

4) Status bits are set as follows:
   $0 \rightarrow$ OV, $1 \rightarrow$ XF, $1 \rightarrow$ SXM, $0 \rightarrow$ PM, $1 \rightarrow$ HM, $0 \rightarrow$ BRAF,
   $0 \rightarrow$ TRM, $0 \rightarrow$ NDX, $0 \rightarrow$ CENB1, $0 \rightarrow$ CENB2, $0 \rightarrow$ IPTR,
   $0 \rightarrow$ OVLY, $0 \rightarrow$ AVIS, $0 \rightarrow$ RAM, $0 \rightarrow$ BIG, $0 \rightarrow$ CNF,
   $1 \rightarrow$ INTM, MP/$\overline{MC}$ (Pin) $\rightarrow$ PMST (MP/$\overline{MC}$), and $1 \rightarrow$ C,

   Note that the remaining status bits remain undefined and should be initialized appropriately.

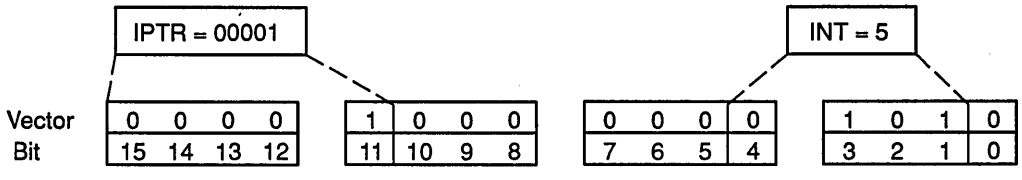
5) The global memory allocation register (GREG) is cleared to make all memory local.

Table 3–8. Interrupt Locations and Priorities

Name †	Location		Priority	Function
	Dec	Hex		
$\overline{RS}$	0	0	1 (highest)	reset signal
$\overline{INT1}$	2	2	3	user interrupt #1
$\overline{INT2}$	4	4	4	user interrupt #2
$\overline{INT3}$	6	6	5	user interrupt #3
$\overline{INT4}$	8	8	6	user interrupt #4
$\overline{INT5}$	10	A	7	user interrupt #5
$\overline{INT6}$	12	C	8	user interrupt #6
$\overline{INT7}$	14	E	9	user interrupt #7
$\overline{INT8}$	16	10	10	user interrupt #8
$\overline{INT9}$	18	12	11	user interrupt #9
$\overline{INT10}$	20	14	12	user interrupt #10
$\overline{INT11}$	22	16	13	user interrupt #11
$\overline{INT12}$	24	18	14	user interrupt #12
$\overline{INT13}$	26	1A	15	user interrupt #13
$\overline{INT14}$	28	1C	16	user interrupt #14
$\overline{INT15}$	30	1E	17	user interrupt #15
$\overline{INT16}$	32	20	18	user interrupt #16
TRAP	34	22	N/A	TRAP instruction vector
$\overline{NMI}$	36	24	2	nonmaskable interrupt

† The interrupt numbers here do not correspond to any specific TMS320C5x device. The definitions of the interrupts, specific to particular TMS320C5x devices, are covered in Chapter 5.

Figure 3–17. Interrupt Vector Address Generation



Upon reset, the IPTR bits are all set to zero, thus mapping the vectors to page zero in program memory space. This means the reset vector always resides at zero. The interrupt vectors can be moved to another location by loading a nonzero value into the IPTR bits. For example, the interrupt vectors can be moved to start at location 0800h by loading the IPTR with 1.

When an interrupt occurs, a flag is activated in the 16-bit interrupt flag register (IFR). Each interrupt is stored in the IFR until it is recognized by the CPU. Any of the following four events will clear the interrupt flag:

ecuting the soft vector. The following example, Example 3–3, illustrates the minimum latency from the time an interrupt occurs externally to the interrupt acknowledge ($\overline{IACK}$). The minimum interrupt acknowledge time is defined as 8 cycles:

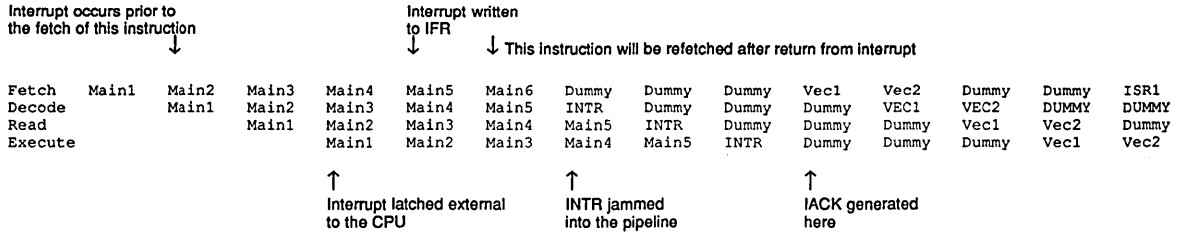
1) 3 cycles to externally synchronize the interrupt

2) 1 cycle to for the interrupt to be recognized by the CPU

3) 4 cycles to execute the INTR instruction and flush the pipeline

On the ninth cycle, the interrupt vector is fetched and the $\overline{IACK}$ is generated.

Example 3–3. Minimum Interrupt Latency



The maximum latency is a function of what is in the pipeline. Multicycle instructions add additional cycles to empty the pipeline. This applies to instructions that are extended via wait-state insertion on memory accesses. The wait states required for interrupt vector accesses also affect the latency. The repeat next instruction N times (RPT and RPTZ) also lock out interrupts, and the repeated instruction completes all executions before allowing the interrupt to execute. This is to protect the context of the repeated instructions because when repeated, the instructions run more parallel operations in the pipeline, and the context of these additional parallel operations cannot be saved in an ISR. The $\overline{HOLD}$ function takes precedence over interrupts and also can delay the interrupt trap. If an interrupt happens during an active-$\overline{HOLD}$ state, the interrupt is taken at the completion of the $\overline{HOLD}$ state.

Interrupts cannot be processed between CLRC INTM and the next instruction in a program sequence. For example, if an interrupt occurs during an CLRC INTM instruction execution, the device always completes CLRC INTM as well as the following instruction before the pending interrupt is processed. This ensures that a return (RET) can be executed before the next interrupt is processed—thus protecting against PC stack overflow. If the ISR is exited via a RETE (return from ISR with enable), the CLRC INTM is unnecessary.

3.8.3 Interrupt Context Save

When an interrupt trap is executed, certain strategic registers are saved automatically. When the return from interrupt instruction (RETE or RETI) is ex-

In the example, the address of the reentry point within the ISR is pushed onto the PC stack. The RETI instruction pops all the stacks, including the PC stack, and resumes execution. At the end of the ISR, a standard return is executed because the stack is already popped.

Not all of the 16 core CPU interrupts are necessarily used on any given TMS320C5x device. The vectors for the interrupts not tied to specific external pins or internal peripherals can be used as software interrupts. To use the corresponding interrupt vectors as software traps with full context save and restore, execute the INTR instruction with the appropriate interrupt number as an operand. These traps are protected from other interrupts in the same way the ISR is protected; all interrupts are globally masked via the INTM bit. To execute the context restore, these trap routines must be exited via the RETI or RETE instruction. For example,

```
INTR 15   ;Software trap to address 01Eh.
```

In this example, the processor will trap to the vector relatively located at 01Eh.

3.8.4 Nonmaskable Interrupt

The core of the TMS320C5x has two nonmaskable interrupts, reset and $\overline{\text{NMI}}$. Reset is discussed in subsection 3.8.1 $\overline{\text{NMI}}$ is used as a soft reset. It is different from a standard interrupt because it is not maskable, and it does not invoke the automatic context save. The context save is not invoked, because it is possible to take the $\overline{\text{NMI}}$ even during an interrupt service routine. In addition, interrupts are globally disabled during an NMI instruction. The $\overline{\text{NMI}}$ is different from reset in that it does not affect any of the modes of the device. Note that some TMS320C5x devices may not make the $\overline{\text{NMI}}$ available externally. The $\overline{\text{NMI}}$ is also delayed by multicycle instructions and $\overline{\text{HOLD}}$, as described in subsection 3.8.2. The $\overline{\text{NMI}}$ trap can also be initiated via software using the NMI instruction. This instruction forces the PC to the NMI trap location.

4 Assembly Language Instructions

4.1 Memory Addressing Modes

The TMS320C5x instruction set provides six basic memory addressing modes:

- Direct addressing mode
- Indirect addressing mode
- Immediate addressing mode
- Dedicated register addressing mode
- Memory-mapped register addressing mode
- Circular addressing mode

Both direct and indirect addressing can be used to access data memory. Direct addressing concatenates seven bits of the instruction word with the nine bits of the data memory page pointer to form the 16-bit data memory address. Indirect addressing accesses data memory through one of eight auxiliary registers. In immediate addressing, the data is based on a portion of the instruction word(s). Two types of immediate addressing modes are available: short and long. In short immediate addressing, an 8-/ 9-/13-bit operand is included in the instruction word. Long immediate addressing mode uses as its operand a 16-bit word following the instruction. Dedicated register addressing refers to the block move instructions in which the BMAR register addresses program or data memory and the parallel logic unit (PLU) instructions in which operands are obtained from the DBMR register. Memory-mapped register addressing mode is used to load and store memory-mapped registers. Circular addressing is an additional mode of indirect addressing that automatically wraps to the beginning of a block of data when the end of the block is reached. The following subsections describe each addressing mode and give the opcode formats and some examples for each mode.

4.1.1 Direct Addressing Mode

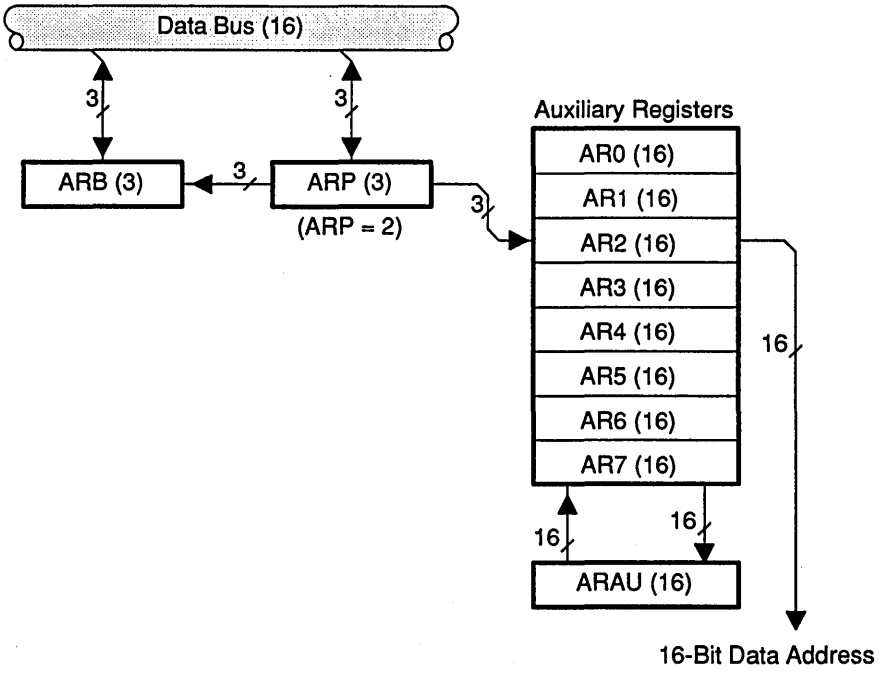
In the direct memory addressing mode, the instruction contains the lower seven bits of the data memory address (dma). This field is concatenated with the nine bits of the data memory page pointer (DP) register to form the full 16-bit data memory address. Thus, the DP register points to one of 512 possible 128-word data memory pages, and the 7-bit address in the instruction points to the specific location within that data memory page. The DP register is loaded by using the LDP (load data memory page pointer) or the LST #0 (load status register ST0) instructions.

The opcode of the ADD 9h,5 instruction is 25h and appears in bits 15 through 8. The shift count of 5 appears in bits 11 through 8 of the opcode. The data memory address 09h appears in bits 6 through 0.

4.1.2 Indirect Addressing Mode

Eight auxiliary registers (AR0–AR7) provide flexible and powerful indirect addressing on the TMS320C5x. To select a specific auxiliary register, load the auxiliary register pointer (ARP) with a value from 0 through 7, designating AR0 through AR7, respectively (see Figure 4–2).

Figure 4–2. Indirect Addressing Block Diagram



The contents of the auxiliary registers may be operated upon by the auxiliary register arithmetic unit (ARAU), which implements unsigned16-bit arithmetic. The ARAU performs auxiliary register arithmetic operations in the decode phase of the pipeline. This allows the address to be generated before the decode phase of the next instruction. The AR is incremented or decremented after it is used in the current instruction.

In indirect addressing, any location in the 64K data memory space can be accessed via a 16-bit address contained in an auxiliary register. The LAR instruction loads the address into the register. The auxiliary registers on the TMS320C5x may be modified by ADRK (add to auxiliary register short immedi-

forms the specified mathematical operation on the indicated auxiliary register. Additionally, the ARP may be loaded with a new value. All indexing operations are performed on the current auxiliary register in the same cycle as the original instruction decode phase of the pipeline.

Indirect auxiliary register addressing allows for post-access adjustments of the auxiliary register pointed to by the ARP. The adjustment may be an increment or decrement by one or may be based upon the contents of the INDX register. To maintain compatibility with the TMS320C2x devices, set the NDX bit in the PMST register to 0. In the TMS320C2x architecture, the current auxiliary register can be incremented or decremented by the value in the AR0 register. When the NDX bit is set to 0, every AR0 modification or LAR write also writes the ARCR and INDX registers with the same value. Subsequent modifications of the current auxiliary registers using indexed addressing will use the INDX register, therefore maintaining compatibility with existing TMS320C2x code. The NDX bit is set to 0 at reset.

Bit-reversed addressing modes on the TMS320C5x allow efficient I/O to be performed by the resequencing of data points in a radix-2 FFT program. The direction of carry propagation in the ARAU is reversed when this mode is selected, and INDX is added to/subtracted from the current auxiliary register. Typical use of this addressing mode requires that INDX first be set to a value corresponding to one-half of the array's size, and that AR(ARP) be set to the base address of the data (the first data point).

Indirect addressing can be used with all instructions except immediate operand instructions and instructions with no operands. The indirect addressing format is as follows:

15 14 13 12 11 10 9 8	7	6	5	4	3	2 1 0
opcode	1	IDV	INC	DEC	NAR	Y

Bits 15 through 8 contain the opcode, and bit 7 = 1 defines the addressing mode as indirect. Bits 6 through 0 contain the indirect addressing control bits.

Bit 6 contains the increment/decrement value (IDV). The IDV bit determines whether the INDX register will be used to increment or decrement the current auxiliary register. If bit 6 = 0, an increment or decrement (if any) by one occurs to the current auxiliary register. If bit 6 = 1, the INDX register is added to or subtracted from the current auxiliary register as defined by bits 5 and 4.

Bits 5 and 4 control the arithmetic operation to be performed with AR(ARP) and the INDX register. When set, bit 5 indicates that an increment is to be performed. If bit 4 is set, a decrement is to be performed. Table 4–1 shows the correspondence of bit pattern and arithmetic operation.

The CMPR (compare auxiliary register with ARCR) and TC/NTC conditions facilitate conditional branches, calls, returns, or conditional executes according to comparisons between the contents of ARCR and the contents of AR(ARP). To maintain compatibility with the TMS320C2x devices, set the NDX bit in the PMST register to 0. In the TMS320C2x architecture, the auxiliary register compare function is performed by comparing AR0 with the current auxiliary register. When the NDX bit is set to 0, every load to AR0 loads the ARCR register with the same value. Subsequent compares of the current auxiliary register will use the ARCR register, therefore maintaining compatibility with existing TMS320C2x code. The NDX bit is set to 0 at reset. The auxiliary registers may also be used for temporary storage via the load and store auxiliary register instructions, LAR and SAR, respectively, or via any instruction that can load and store the memory-mapped auxiliary registers.

The following examples illustrate the indirect addressing format:

Example 1 **ADD *+,8**

Add to the accumulator the contents of the data memory address defined by the contents of the current auxiliary register. This data is left-shifted 8 bits before being added. The current auxiliary register is autoincremented by one. The instruction word is 028A0h.

Example 2 **ADD *,8**

As in Example 1, but with no autoincrement; the instruction word is 02880h.

Example 3 **ADD *–,8**

As in Example 1, except that the current auxiliary register is decremented by one; the instruction word is 02890h.

Example 4 **ADD *0+,8**

As in Example 1, except that the contents of register INDX are added to the current auxiliary register; the instruction word is 028E0h.

Example 5 **ADD *0–,8**

As in Example 1, except that the contents of register INDX are subtracted from the current auxiliary register; the instruction word is 028D0h.

Example 6 **ADD *+,8,AR3**

As in Example 1, except that the auxiliary register pointer (ARP) is loaded with the value **3** for subsequent instructions; the instruction word is 028ABh.

Example 7 **ADD *BR0–,8**

The contents of register INDX are subtracted from the current auxiliary register, with reverse carry propagation; the instruction word is 028C0h.

The following is an example code and the instruction word format for the RPT instruction with long immediate addressing:

```
RPT #0FFFh ;Execute the instruction following the RPT instruction 1000h times.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	1	1	0	0	0	1	0	0
16-bit constant															

4.1.4 Dedicated Register Addressing

Nine instructions in the TMS320C5x instruction set can use one of two special-purpose memory-mapped registers in the core CPU. These two registers are the block move address register (BMAR) and the dynamic bit manipulation register (DBMR). The APL, OPL, CPL, and XPL parallel logic unit (PLU) instructions use the contents of the DBMR register when an immediate value is not specified as one of the operands. The BLDD, BLDP, and BLPD instructions can use the BMAR register to point at the source or destination space of a block move. The MADD and MADS also use the BMAR register to address an operand in program memory for a multiply-accumulate operation.

The syntax for dedicated register addressing can be stated in one of two ways:

1) specifying BMAR by its predefined symbol as shown below:

```
BLDD BMAR,DAT100  ;DP = 0. BMAR contains the value 200h.
```

The contents of data memory location 200h are copied to data memory location 100 on the current data page. The opcode for this instruction is 0AC64h.

2) excluding the immediate value from parallel logic unit instructions as shown below. The BMAR register is implied by the MADD and MADS instruction mnemonics.

```
OPL    DAT10  ;DP = 6.  DBMR contains the value 0FFF0h.
              ;Address  030Ah contains the value 01h
```

The contents of data memory location 030Ah are ORed with the contents of DBMR. The resulting 0FFF1h is stored back to memory location 030Ah. The opcode for this instruction is 590Ah.

4.1.5 Memory-Mapped Register Addressing

Memory-mapped register addressing is used for modifying the memory-mapped registers without affecting the current data page pointer value. In addition, any scratch pad RAM location or data page 0 can be modified by using this addressing mode. Figure 4–3 illustrates how this is done by forc-

implement a sliding window, which contains the most recent data to be processed. The TMS320C5x supports two concurrent circular buffers operating via the auxiliary registers. The following five memory-mapped registers control the circular buffer operation:

- CBSR1 – Circular Buffer One Start Register
- CBSR2 – Circular Buffer Two Start Register
- CBER1 – Circular Buffer One End Register
- CBER2 – Circular Buffer Two End Register
- CBCR – Circular Buffer Control Register

The 8-bit circular buffer control register enables and disables the circular buffer operation. The CBCR is defined as follows:

Bit	Name	Function
0–2	CAR1	Identifies which auxiliary register is mapped to circular buffer 1.
3	CENB1	Circular buffer 1, enable=1/disable=0. Set to 0 upon reset.
4–6	CAR2	Identifies which auxiliary register is mapped to circular buffer 2.
7	CENB2	Circular buffer 2, enable=1/disable=0. Set to 0 upon reset.

In order to define circular buffers, the start and end addresses should first be loaded into the corresponding buffer registers; next, a value between the start and end registers for the circular buffer is loaded into an auxiliary register. The proper auxiliary register value is loaded, and the corresponding circular buffer enable bit is set in the control register. Note that the same auxiliary register can not be enabled for both circular buffers, or unexpected results will occur.The algorithm for circular buffer addressing is as follows (note that the test of the auxiliary register value is performed before any modifications):

If (ARn = CBER) and (any AR modification),
Then: ARn = CBSR.
Else: ARn = ARn + step.

In addition, note that if ARn=CBER and no AR modification occurs, the current AR is not modified and is still equal to CBER.Note that when the current auxiliary register = CBER, any AR modification (increment or decrement) will set the current AR = CBSR. The following examples illustrate the operation:

```
splk   #200h,CBSR1   ; Circular buffer start register
splk   #203h,CBER1   ; Circular buffer end register
splk   #0eh,CBCR     ; Enable AR6 pointing to buffer 1

lar    ar6,#200h     ; Case 1
lacc   *             ; AR6 = 200h

lar    ar6,#203h     ; Case 2
lacc   *             ; AR6 = 203h

lar    ar6,#200h     ; Case 3
lacc   *+            ; AR6 = 201h

lar    ar6,#203h     ; Case 4
lacc   *+            ; AR6 = 200h
```

4.2 Instruction Set

The TMS320C5x assembly language instruction set supports both DSP-specific and general-purpose applications. This section lists and groups the TMS320C5x instruction set according to the following functional headings:

- Accumulator Memory Reference Instructions
- Auxiliary Registers and Data Page Pointer Instructions
- Parallel Logic Unit Instructions
- T Register, P Register, and Multiply Instructions
- Branch Instructions
- I/O and Data Memory Operations
- Control Instructions

Section 4.1 covers the addressing modes associated with the instruction set, and Section 4.3 describes individual instructions in more detail.

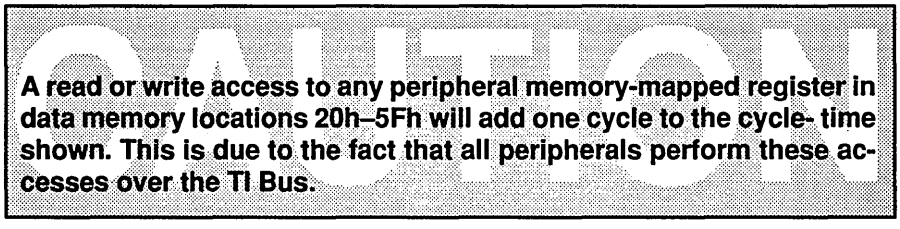
4.2.1 Symbols and Abbreviations

Table 4–3 lists symbols and abbreviations used in the instruction set summary (Table 4–4) and the individual instruction descriptions (Section 4.3).

4.2.2 Instruction Set Summary

Table 4–4 is a summary of the instruction set for the TMS320C5x digital signal processors. This instruction set is a superset of the TMS320C1x and TMS320C2x instruction sets.

The instruction set summary is arranged according to function and is alphabetized within each functional grouping. The number of words that an instruction occupies in program memory is specified in column four of the table. Several instructions specify two values, separated by a slash mark "/" for the number of words. Different forms of the instruction occupy a different number of words. For example, the ADD instruction occupies one word when the operand is a short immediate value or two words if the operand is a long immediate value. The number of cycles that an instruction requires to execute is in column four of the table. All instructions are assumed to be executed from internal program memory (RAM) and internal data dual-access memory. The cycle timings are for single-instruction execution, not for repeat mode. Additional information is presented in the Individual Instruction Descriptions in Section 4.3. The symbol # indicates those instructions that are new for the TMS320C5x instruction set.



Section 4.4 includes a table that maps TMS320C2x instructions to TMS320C5x instructions. Note that the Texas Instruments TMS320C5x assembler will accept TMS320C2x instructions as well as TMS320C5x instructions.

Table 4–4. Instruction Set Summary (Continued)

Accumulator Memory Reference Instructions (Concluded)			
Mnemonic	**Description**	**Words**	**Cycles**
SACB #	Store ACC in ACCB	1	1
SACH	Store high ACC with shift	1	1
SACL	Store low ACC with shift	1	1
SAMM #	Store ACC to memory-mapped register	1	1 (processor memory-mapped register) 2 (peripheral memory-mapped registers)
SATH #	Barrel-shift ACC right 0 or 16 bits as specified by TREG1	1	1
SATL #	Barrel-shift ACC right 0 to 15 bits as specified by TREG1	1	1
SBB #	Subtract ACCB from ACC	1	1
SBBB #	Subtract ACCB from ACC with borrow	1	1
SFL	Shift ACC left	1	1
SFLB #	Shift ACCB and ACC left	1	1
SFR	Shift ACC right	1	1
SFRB #	Shift ACCB and ACC right	1	1
SUB	Subtract from ACC	1/2	1 2 (long immediate value specified)
SUBB	Subtract from ACC with borrow	1	1
SUBC	Conditional subtract	1	1
SUBS	Subtract from low ACC with sign-extension suppressed	1	1
SUBT	Subtract from ACC with shift specified by TREG1	1	1
XOR	Exclusive-OR with ACC	1/2	1 2 (long immediate value specified)
XORB #	Exclusive-OR ACCB with ACC	1	1
ZALR	Zero low ACC and load high ACC with rounding	1	1
ZAP	Zero ACC and PREG	1	1

Table 4–4. Instruction Set Summary (Continued)

T Register, P Register, and Multiply Instructions			
Mnemonic	**Description**	**Words**	**Cycles**
APAC	Add PREG to ACC	1	1
LPH	Load high PREG	1	1
LT	Load TREG0	1	1
LTA	Load TREG0 & accumulate previous product	1	1
LTD	Load TREG0, accumulate previous product, and move data	1	1
LTP	Load TREG0 & store PREG in accumulator	1	1
LTS	Load TREG0 and subtract previous product	1	1
MAC	Multiply and accumulate	2	3
MACD	Multiply and accumulate with data move	2	3
MADD #	Multiply and accumulate with source pointed at by BMAR	1	3
MADS #	Multiply and accumulate both with source pointed at by BMAR and with data move	1	3
MPY	Multiply	1/2	1 2 (long immediate value specified)
MPYA	Multiply and accumulate previous product	1	1
MPYS	Multiply and subtract previous product	1	1
MPYU	Multiply unsigned	1	1
PAC	Load ACC with PREG	1	1
SPAC	Subtract PREG from ACC	1	1
SPH	Store high PREG	1	1
SPL	Store low PREG	1	1
SPM	Set PREG output shift mode	1	1
SQRA	Square and accumulate previous product	1	1
SQRS	Square and subtract previous product	1	1
ZPR #	Zero product register	1	1

Table 4–4. Instruction Set Summary (Continued)

I/O and Data Memory Operations			
Mnemonic	**Description**	**Words**	**Cycles**
BLDD	Block move from data memory to data memory	1/2	2 (operand specified by BMAR) 3 (operand specified by long immediate)
BLDP #	Block move from data memory to program memory	1	2
BLPD	Block move from program memory to data memory	1/2	2 (operand specified by BMAR) 3 (operand specified by long immediate)
DMOV	Data move in data memory	1	1
IN	Input data from port	2	2
LMMR #	Load memory-mapped register	2	2 (processor memory-mapped register) 3 (peripheral memory-mapped register)
OUT	Output data to port	2	3
SMMR #	Store memory-mapped register	2	2 (processor memory-mapped register) 3 (peripheral memory-mapped register)
TBLR	Table read	1	3
TBLW	Table write	1	3

4.3 Individual Instruction Descriptions

This section furnishes detailed information on the instruction set for the TMS320C5x family; see Table 4–4, *Instruction Set Summary*, for a complete list of available instructions. Each instruction presents the following information:

- assembler syntax
- operands
- opcode
- execution
- description
- words
- cycles
- examples

The **EXAMPLE** instruction is provided to familiarize the user with the instruction format and explain the contents of the instruction manual pages.

data RAM. The cycle timings are for single-instruction execution, not for repeat mode. Note that writing or reading any of the memory-mapped peripheral registers over the peripheral bus will add one additional cycle to the execution of that instruction.

Example Example code is included for each instruction. The effect of the code on memory and/or registers is summarized.

Syntax [*label*] **ADCB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	0	0	0	1

Execution (PC) + 1 → PC
(ACC) + (ACCB) + (C) → ACC

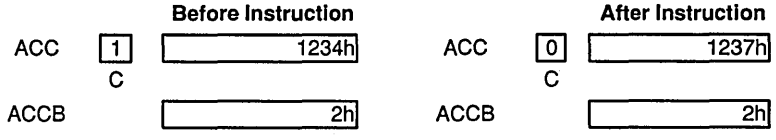
Affected by OVM; affects OV and C

Description The contents of the accumulator buffer (ACCB) and the value of the carry bit (C) are added to the accumulator. The carry bit is set to one if the result of the addition generates a carry from the MSB position of the accumulator.

Words 1

Cycles 1

Example ADCB



(PC) + 2 → PC
(ACC) + lk x 2^{shift2} → ACC
Affected by SXM and OVM; affects C and OV.

Description The contents of the addressed data memory location or an immediate constant are left-shifted and added to the accumulator. During shifting, low-order bits are zero-filled. High-order bits are sign-extended if SXM = 1 and zero-filled if SXM = 0. The result is stored in the accumulator. When short immediate addressing is used, the addition is unaffected by SXM and is not repeatable. Note that when the ARP is updated during indirect addressing, a shift operand must be specified. If no shift is desired, a 0 may be entered for this operand.

When adding with a shift of 16, the carry bit is set if the results of the addition generates a carry; otherwise, the carry bit is unaffected. This allows the accumulation to generate the proper single carry when adding a 32-bit number to the accumulator.

Words 1 (Direct, indirect, or short immediate addressing)
2 (Long immediate addressing)

Cycles 1 (Direct, indirect, or short immediate addressing)
2 (Long immediate addressing)

Example 1
```
ADD    DAT1,1  ;(DP = 6)
```

	Before Instruction		After Instruction	
Data Memory 301h		1h		1h
ACC	X	2h	0	04h
	C		C	

Example 2
```
ADD    *+,0,AR0
```

	Before Instruction		After Instruction	
ARP		4		0
AR4		0302h		0303h
Data Memory 302h		2h		2h
ACC	X	2h	0	04h
	C		C	

Example 3
```
ADD    #1h     ;Add short immediate
```

	Before Instruction		After Instruction	
ACC	X	2h	0	03h
	C		C	

Example 4
```
ADD    #1111h,1 ;Add long immediate with shift of 1
```

	Before Instruction		After Instruction	
ACC	X	2h	0	2224h
	C		C	

Syntax Direct: [*label*] **ADDC** *dma*
Indirect: [*label*] **ADDC** {*ind*} [,*next ARP*]

Operands 0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	0	0	0	0	Data Memory Address						
Indirect:	0	1	1	0	0	0	0	0	1	See Subsection 4.1.2						

Execution (PC) + 1 → PC
(ACC) + (dma) + (C) → ACC

Affected by OVM; affects OV and C. Not affected by SXM.

The contents of the addressed data memory location and the value of the carry bit are added to the accumulator with sign extension suppressed. The carry bit is then affected in the normal manner.

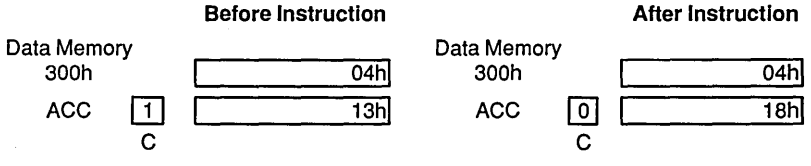
The ADDC instruction can be used in performing multiple-precision arithmetic.

Words 1

Cycles 1

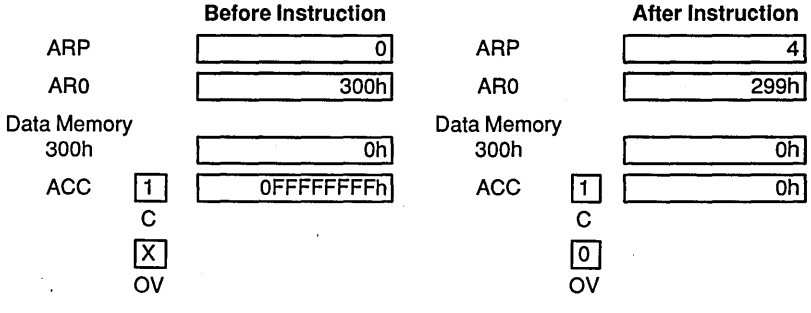
Example 1

```
ADDC   DAT0   ;(DP = 6)
```



Example 2

```
ADDC   *-,AR4 ;(OVM = 0)
```



Syntax Direct: [*label*] **ADDT** *dma*
Indirect: [*label*] **ADDT** {*ind*} [,*next ARP*]

Operands 0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	0	1	1	0	Data Memory Address						
Indirect:	0	1	1	0	0	0	1	1	1	See Subsection 4.1.2						

Execution (PC) + 1 → PC
(ACC) + [(dma) x $2^{TREG1(3–0)}$] → (ACC)
If SXM = 1:
Then (dma) is sign-extended.
If SXM = 0:
Then (dma) is not sign-extended.

Affected by SXM and OVM; affects OV and C.

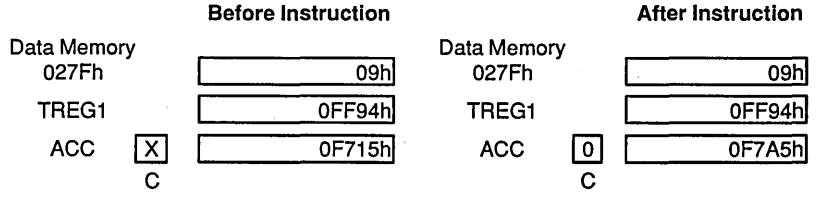
Description The data memory value is left-shifted and added to the accumulator, with the result replacing the accumulator contents. The left-shift is defined by the four LSBs of the TREG1, resulting in shift options from 0 to 15 bits. Sign extension on the data memory value is controlled by SXM. The carry bit is set when a carry is generated out of the MSB of the accumulator.

Software compatibility with the TMS320C25 can be maintained by setting the TRM bit of the PMST status register to zero. This causes any TMS320C25 instruction that loads TREG0 to write to all three TREGs. Subsequent calls to the ADDT instruction will shift the value by the TREG1 value (which is the same as TREG0), maintaining object-code compatibility.

Words 1

Cycles 1

Example 1
```
ADDT   DAT127  ;(DP = 4. SXM = 0)
```



Syntax [*label*] **ADRK** *#k*

Operands $0 \leq k \leq 255$

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Short:	0	1	1	1	1	0	0	0	8-Bit Constant							

Execution (PC) + 1 → PC
AR(ARP) + 8-bit positive constant → AR(ARP)

Description The 8-bit immediate value is added, right-justified, to the currently selected auxiliary register (as specified by the current ARP) with the result replacing the auxiliary register contents. The addition takes place in the ARAU, with the immediate value treated as an 8-bit positive integer. Note that all arithmetic operations on the auxiliary registers are unsigned.

Words 1

Cycles 1

Example

```
ADRK   #80h
```

	Before Instruction		After Instruction
ARP	5	ARP	5
AR5	4321h	AR5	43A1h

Example 1 `AND DAT16 ;(DP = 4)`

	Before Instruction		After Instruction
Data Memory 0210h	00FFh	Data Memory 0210h	00FFh
ACC	12345678h	ACC	00000078h

Example 2 `AND *`

	Before Instruction		After Instruction
ARP	0	ARP	0
AR0	0301h	AR0	0301h
Data Memory 0301h	0FF00h	Data Memory 0301h	0FF00h
ACC	12345678h	ACC	00005600h

Example 3 `AND #00FFh,4`

	Before Instruction		After Instruction
ACC	12345678h	ACC	00000670h

Syntax [*label*] **APAC**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	0	0	1	0	0

Execution (PC) + 1 → PC
(ACC) + (shifted P register) → ACC

Affected by PM and OVM; affects OV and C.
Not affected by SXM .

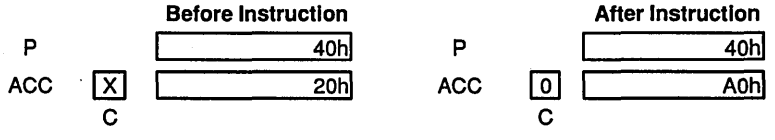
Description The contents of the P register are shifted as defined by the PM status bits and added to the contents of the accumulator. The result is placed in the accumulator. APAC is not affected by the SXM bit of the status register; the P register is always sign-extended.The APAC instruction is a subset of the LTA, LTD, MAC, MACD, MADS, MADD, MPYA, and SQRA instructions.

Words 1

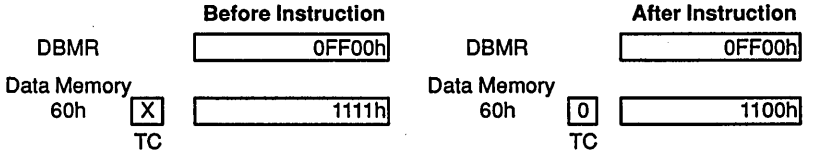
Cycles 1

Example

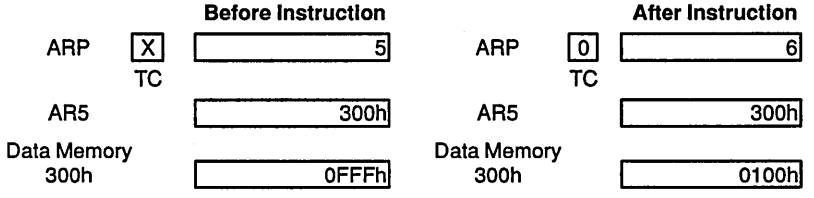
```
APAC   ;(PM = 01)
```



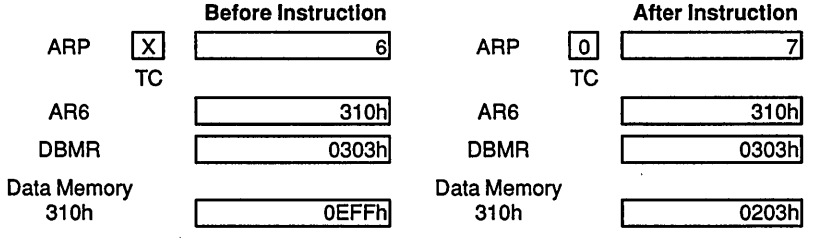
Example 2 `APL DAT96 ;(DP = 0)`



Example 3 `APL #0100h,*,AR6`



Example 4 `APL *,AR7`



Syntax [*label*] **BACC**[*D*]

Operands None

Opcode

BACC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	0	0	0	0	0

BACCD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	0	0	0	0	1

Execution ACC(15–0) → PC

Description Control is passed to the 16-bit address residing in the lower half of the accumulator. The one two-word instruction or two one-word instructions following the branch instruction are fetched from program memory and executed before the branch is taken, if the branch is a delayed branch (specified by the "D" suffix).

Words 1

Cycles 4 2 (If delayed)

Example 1
```
BACC    ;(ACC contains the value 191)
```

191 is loaded into the program counter, and the program continues executing from that location.

Example 2
```
BACCD           ;(ACC contains the value 191)
MAR     *+,AR1
LDP     #5
```

After the current AR, ARP, and DP are modified as specified, program execution continues from location 191.

The program counter (PC) is incremented by 2, and execution continues from that location.

Example 2

```
BANZD PGM0
LACC  #01h
LDP   #5
```

	Before Instruction		After Instruction
ARP	0	ARP	0
AR0	5h	AR0	4h
DP	4	DP	5
ACC	00h	ACC	01h

After the current DP and ACC are modified as specified, program execution continues from location 0.

Example 3

```
       MAR  *,AR0
       LAR  AR1,#3
       LAR  AR0,#60h
PGM191 ADD  *+,AR1
       BANZ PGM191,AR0
```

The contents of data memory locations 60h–63h are added to the accumulator.

Example 1 `BCND PGM191,LEQ,C`

If the accumulator contents are less than or equal to zero and the carry bit is set, program address 191 is loaded into the program counter, and the program continues executing from that location. If these conditions do not hold, execution continues from location PC + 2.

Example 2
```
BCNDD  PGM191,OV
MAR    *,AR1
LDP    #5
```

After the current AR, ARP, and DP are modified as specified, program execution continues at location 191 if the overflow flag (OV) in status register ST0 is set. If the flag is not set, execution continues at the instruction following the LDP instruction.

Example 1 `BIT  0h,15  ;(DP = 6).Test LSB at 300h`

	Before Instruction	After Instruction
Data Memory 300h	4DC8h	4DC8h
TC	0	0

Example 2 `BIT  *,0,AR1 ;Test MSB at 310h`

	Before Instruction	After Instruction
ARP	0	1
AR0	310h	310h
Data Memory 310h	8000h	8000h
TC	0	1

Words 1

Cycles 1

Example 1 `BITT 00h ;(DP = 6). Test bit 14 of data at 300h`

	Before Instruction	After Instruction
Data Memory 300h	4DC8h	4DC8h
TREG2	1h	1h
TC	0	1

Example 2 `BITT * ;Test bit 1 of data at 310h`

	Before Instruction	After Instruction
ARP	1	1
AR1	310h	310h
Data Memory 310h	8000h	8000h
TREG2	0Eh	0Eh
TC	0	0

Block move data to data with DEST in BMAR

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	1	0	1	1	0	1	0	Data Memory Address						
Indirect:	1	0	1	0	1	1	0	1	1	See Subsection 4.1.2						

Execution

(PFC) → MCS

If long immediate:
  (PC) + 2 → PC
  #lk → PFC
Else:
  (PC) + 1 → PC
  (BMAR) → PFC

While (repeat counter) ≠ 0:
  (src, addressed by PFC) → dst or src → (dst, addressed by PFC)
  Modify AR(ARP) and ARP as specified,
  (PFC) + 1 → PFC
  (repeat counter) –1 → repeat counter.
(src, addressed by PFC) →dst or src → (dst, addressed by PFC)
Modify AR(ARP) and ARP as specified.
(MCS) → PFC

Description

The word in data memory pointed at by *src* is copied to a data memory space pointed at by *dst*. The word of the source and/or destination space can be pointed at with a long immediate value, with the contents of the BMAR register, or by a data memory address. Note that not all src/dst combinations of pointer types are valid.

RPT can be used with the BLDD instruction in indirect addressing mode to move consecutive words in data memory. The number of words to be moved is one greater than the number contained in the repeat counter RPTC at the beginning of the instruction. The source or destination address for the BLDD instruction specified by the long immediate address or BMAR register contents are automatically incremented in repeat mode. If a direct memory address is specified, its address is not automatically incremented in repeat mode. Note that the source and destination blocks do not have to be entirely on-chip or off-chip. Interrupts are inhibited during a *BLDD* operation used with the RPT instruction. When used with RPT, BLDD becomes a single-cycle instruction once the RPT pipeline is started.

CAUTION

Neither the long immediate nor the BMAR can be used as the address to the on-chip memory-mapped registers. The direct or indirect addressing mode can be used to address the on-chip memory-mapped core processor and peripheral registers.

Example 5

```
RPTK   2
BLDD   #300h,*+
```

	Before Instruction		After Instruction
ARP	0	ARP	0
AR0	320h	AR0	323h
300h	7F98h	300h	7F98h
301h	0FFE6h	301h	0FFE6h
302h	9522h	302h	9522h
320h	8DEEh	320h	7F98h
321h	9315h	321h	0FFE6h
322h	2531h	322h	9522h

Example 2 `BLDP *,AR0`

	Before Instruction		After Instruction
ARP	7	ARP	0
AR7	310h	AR7	310h
Data Memory 310h	0F0F0h	Data Memory 310h	0F0F0h
BMAR	2800h	BMAR	2800h
Program Memory 2800h	1234h	Program Memory 2800h	0F0F0h

or the contents of the BMAR register. The data memory destination space is always pointed at by a data memory address or auxiliary register pointer. Note that not all src/dst combinations of pointer types are valid.

RPT can be used with the BLPD instruction if more than one word is to be moved. The number of words to be moved is one greater than the number contained in the repeat counter, RPTC, at the beginning of the instruction. The source address specified by the long immediate or BMAR value is automatically incemented in repeat mode. Note that the source and destination blocks do **not** have to be entirely on-chip or off-chip. Interrupts are inhibited during a repeated BLPD instruction. When used with RPT, BLPD becomes a single-cycle instruction once the RPT pipeline is started.

Words 1 (Source is specified by the BMAR register)
2 (Source is specified by a long immediate)

Cycles 2 (Source is specified by the BMAR register)
3 (Source is specified by a long immediate)

Example 1 `BLPD #800h,00h ;(DP=6)`

	Before Instruction		After Instruction
Program Memory 800h	0Fh	Program Memory 800h	0Fh
Data Memory 300h	0h	Data Memory 300h	0Fh

Example 2 `BLPD #800h,*,AR7`

	Before Instruction		After Instruction
ARP	0	ARP	7
AR0	310h	AR0	310h
Program Memory 800h	1111h	Program Memory 800h	1111h
Data Memory 310h	0100h	Data Memory 310h	1111h

Example 3 `BLPD BMAR,00h ;(DP=6)`

	Before Instruction		After Instruction
BMAR	800h	BMAR	800h
Program Memory 800h	0Fh	Program Memory 800h	0Fh
Data Memory 300h	0h	Data Memory 300h	0Fh

Syntax [*label*] **BSAR** *shift*

Operands 1 ≤ shift ≤ 16

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	SHFT †			

† See Section 4.5.

Execution (PC) + 1 → PC
(ACC) / 2^{shift} → ACC

Affected by SXM.

Description The BSAR instruction executes a 1- to 16-bit right-barrel arithmetic shift of the accumulator in a single cycle. The sign extension is determined by the sign-extension mode bit in status register 1 (ST1).

Words 1

Cycles 1

Example 1

```
BSAR 16        ;(SXM=0)
```

	Before Instruction		After Instruction
ACC	00010000h	ACC	00000001h

Example 2

```
BSAR 4         ;(SXM=1)
```

	Before Instruction		After Instruction
ACC	0FFF10000h	ACC	0FFFF1000h

Syntax [*label*] **CALL**[*D*] *pma* [,{*ind*} [,*next ARP*]]

Operands 0 ≤ pma ≤ 65535
0≤ next ARP ≤ 7

Opcode

CALL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	1	0	1	See Subsection 4.1.2						
16-Bit Constant															

CALLD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	See Subsection 4.1.2						
16-Bit Constant															

Execution Non-delayed: PC + 2 → TOS
Delayed: PC + 4 → TOS
pma → PC
Modify AR(ARP) and ARP as specified.

Description The current program counter (PC) is incremented and pushed onto the top of the stack (TOS). Then, the contents of the program memory address (pma), either a symbolic or numeric address, are loaded into the PC. Execution continues at this address. The current auxiliary register and ARP are modified as specified. If the call is a delayed call (specified by the "D" suffix), the one two-word instruction or two one-word instructions following the call instruction are fetched from program memory and executed before the call is executed.

Words 2

Cycles 4 2 (If delayed)

Example 1

```
CALL   PRG191,*+,AR0
```

	Before Instruction		After Instruction
ARP	1	ARP	0
AR1	05h	AR1	06h
PC	30h	PC	0BFh
TOS	100h	TOS	32h

0BFh is loaded into the program counter, and the program continues executing from that location.

Syntax [*label*] **CC**[*D*] *pma* [*cond1*] [,*cond2*] [,...

Operands $0 \leq pma \leq 65535$

Conditions:

ACC=0	EQ
ACC≠0	NEQ
ACC<0	LT
ACC≤0	LEQ
ACC>0	GT
ACC≥0	GEQ
C=0	NC
C=1	C
OV=0	NOV
OV=1	OV
TC=0	NTC
TC=1	TC
$\overline{BIO}$ low	BIO
Unconditionally	UNC

Opcode

CC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	TP †		ZLVC †				ZLVC †			
16-Bit Constant															

CCD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	TP †		ZLVC †				ZLVC †			
16-Bit Constant															

† See Section 4.5.

Execution

```
If(condition(s))
    Then
        Nondelayed: PC + 2 → TOS
        Delayed:  PC + 4 → TOS
        pma → PC
    Else
        PC + 2 → PC
```

Description Control is passed to the program memory address pma if the specified conditions are met. Note that not all combinations of conditions are meaningful. In addition, the NTC, TC, and BIO conditions are mutually exclusive. If the call is a delayed call (specified by the "D" suffix), the two one-word instructions or the one two-word instruction following the call are fetched from program memory and executed before the call is executed. The CC instruction operates like the CALL instruction if all conditions are true.

Words 2

Syntax [*label*] **CLRC** *control bit*

Operands *Control bit*: ST0, ST1 bit (from the following set):

{C, CNF, HM, INTM, OVM, TC, SXM, XF}

Opcode

Reset overflow mode (OVM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0

Reset sign extension mode (SXM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	0	1	1	0

Reset hold mode (HM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	1	0	0	0

Reset TC bit

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	1	0	1	0

Reset carry (C)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	1	1	1	0

Reset CNF bit

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	0	1	0	0

Reset INTM bit

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	0	0	0	0

Reset XF pin

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	1	1	0	0

Execution (PC) + 1 → PC
0 → control bit

Description The specified control bit is set to a logic zero. Note that the LST instruction may also be used to load ST0 and ST1. See subsection 3.6.3, *Status and Control Registers*, for more information on each of these control bits.

Words 1

Cycles 1

Syntax [*label*] **CMPL**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	1

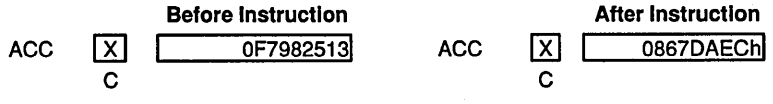
Execution $(PC) + 1 \rightarrow PC$
$(\overline{ACC}) \rightarrow ACC$

Description The contents of the accumulator are replaced with its logical inversion (ones complement). The carry bit is unaffected.

Words 1

Cycles 1

Example `CMPL`



Syntax

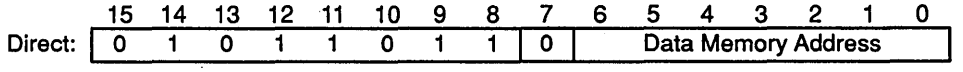
Direct: [*label*] **CPL** [,#*lk*] *dma*
Indirect: [*label*] **CPL** [,#*lk*] {*ind*} [,*next ARP*]

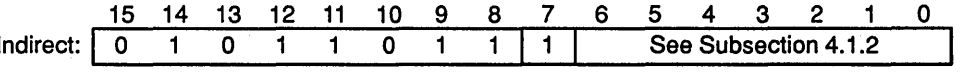
Operands

0 ≤ dma ≤ 127
lk: 16-bit constant
0 ≤ next ARP ≤ 7

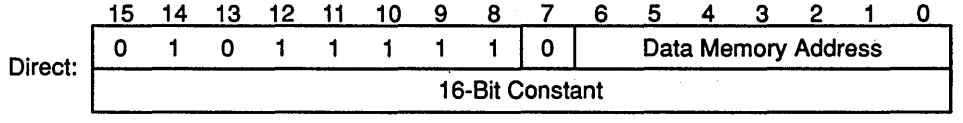
Opcode

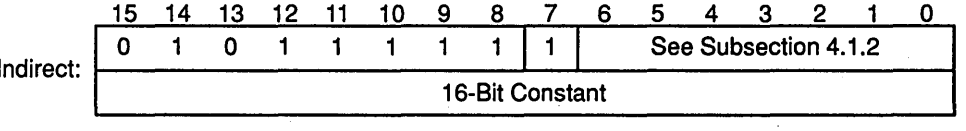
Compare DBMR to data value





Compare data with long immediate





Execution

lk unspecified:
(PC) + 1 → PC

Compare DBMR contents to (dma).
If (DBMR) = (dma),
    TC = 1;
Else,
    TC = 0.

lk specified:
(PC) + 2 → PC

Compare lk to (dma).
If lk = (dma),
    TC = 1;
Else
    TC = 0.

Affects TC.
Not affected by SXM.

Description

If the two quantities involved in the comparison are equal, the TC bit is set to one. TC is set to zero otherwise.

Words

1 (If long immediate value is not specified)
2 (If long immediate value is specified)

Syntax [*label*] **CRGT**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	1	0	1	1

Execution

(PC) + 1 → PC
If (ACC) > (ACCB)
  Then (ACC) → ACCB; 1 → C
If (ACC) < (ACCB)
  Then (ACCB) → ACC; 0 → C
If (ACC) = (ACCB)
  Then 1 → C

Affects C.

Description The contents of the accumulator (ACC) are compared to the contents of the accumulator buffer (ACCB). The larger value (signed) is loaded into both registers. If the contents of the accumulator are greater than or equal to the contents of the accumulator buffer, the carry bit is set to 1. Otherwise, it is set to 0.

Words 1

Cycles 1

Example 1 `CRGT`

	Before Instruction		After Instruction
ACCB	4h	ACCB	5h
ACC	5h	ACC	5h
C	0	C	1

Example 2 `CRGT`

	Before Instruction		After Instruction
ACCB	5h	ACCB	5h
ACC	5h	ACC	5h
C	0	C	1

Syntax Direct: [*label*] **DMOV** *dma*
Indirect: [*label*] **DMOV** {*ind*} [,*next ARP*]

Operands 0 ≤ dma ≤ 127
0 ≤ next ARP ≤7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	1	0	1	1	1	0	Data Memory Address						
Indirect:	0	1	1	1	0	1	1	1	1	See Subsection 4.1.2						

Execution (PC) + 1 → PC
(dma) → dma + 1

Affected by CNF and OVLY.

Description The contents of the specified data memory address are copied into the contents of the next higher address. DMOV works only within on-chip data RAM blocks. It works within any configurable RAM block if that block is configured as data memory. In addition, the data move function is continuous across block boundaries. The data move function cannot be used on external data memory or memory-mapped registers. If used on external memory or memory-mapped registers, DMOV will read the specified memory location but will perform no operations.

When data is copied from the addressed location to the next higher location, the contents of the addressed location remain unaltered.

The data move function is useful in implementing the z^{-1} delay encountered in digital signal-processing. The DMOV function is included in the LTD, MACD, and MADD instructions (see the LTD, MACD, and MADD instructions for more information).

Words 1

Cycles 1

Example 1
```
DMOV   DAT8   ;(DP = 6)
```

	Before Instruction		After Instruction
Data Memory 308h	43h	→	43h
Data Memory 309h	2h	→	43h

Example 2
```
DMOV   *,AR1
```

	Before Instruction		After Instruction
ARP	0	→	1
AR1	30Ah	→	30Ah
Data Memory 30Ah	40h	→	40h
Data Memory 30Bh	41h	→	40h

Syntax [*label*] **IDLE**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	0	0	0	1	0

Execution (PC) + 1 → PC

Affected by INTM.

Description The IDLE instruction forces the program being executed to wait until an unmasked interrupt (external or internal) or reset occurs. The PC is incremented only once, and the device remains in an idle state until interrupted.

The idle state is exited by an unmasked interrupt even if INTM is 1. In the case of INTM being 1, the program will continue executing at the instruction following the IDLE. If INTM is 0, then the program will branch to the corresponding interrupt service routine. Execution of the IDLE instruction causes the TMS320C5x to enter the power-down mode. During the idle mode, the timer and serial port peripherals are still active. Therefore, timer and peripheral interrupts, as well as reset or external interrupts, will remove the processor from the idle mode.

Words 1

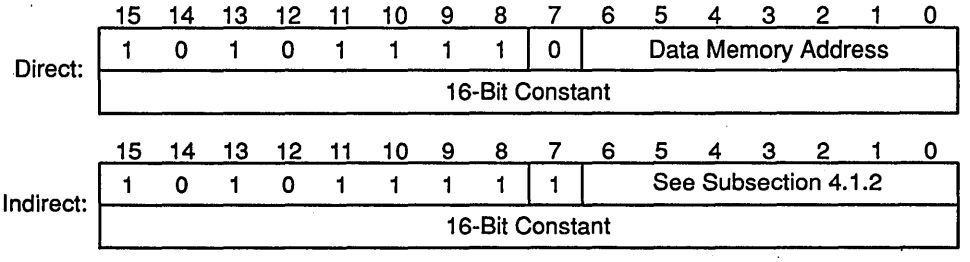
Cycles - - -

Example
```
IDLE    ;The processor idles until a reset or unmasked interrupt
        ;occurs.
```

Syntax Direct: [*label*] **IN** *dma* , *PA*
Indirect: [*label*] **IN** {*ind*} ,*PA* [,*next ARP*]

Operands 0 ≤ dma ≤ 127
0 ≤ next ARP ≤7
0 ≤ **PA** ≤ 65535

Opcode



Execution (PC) + 2 → PC
While (repeat counter) ≠ 0
Port address → address bus **A15–A0**
Data bus **D15–D0** → dma
Port address + 1 → Port address
(repeat counter – 1) → repeat counter

Description The IN instruction reads a 16-bit value from an external I/O port into the specified data memory location. The $\overline{IS}$ line goes low to indicate an I/O access, and the $\overline{STRB}$, $\overline{RD}$, and READY timings are the same as for an external data memory read. Note that port addresses 50h–5Fh are memory-mapped (see subsection 5.1.1), but the other port addresses are not.

RPT can be used with the IN instruction to read in consecutive words from I/O space to data space. In the repeat mode, the port address (PA) is incremented after each access.

Words 2

Cycles 2 (Each access cycle time increases by **i**, the number of I/O memory wait states. This is the number of cycles the device must wait for external I/O memory accesses.)

Example 1
```
IN    DAT7,PA5      ;Read in word from peripheral
                    ;on port address 5. Store in
                    ;data memory location 307h (DP=6).
```

Example 2
```
IN    *,PA0         ;Read in word from peripheral on
                    ;port address 0. Store in data memory
                    ;location specified by the current
                    ;auxiliary register.
```

k	Interrupt	Location	k	Interrupt	Location
0	$\overline{RS}$	0h	16	Reserved	20h
1	$\overline{INT1}$	2h	17	TRAP	22h
2	$\overline{INT2}$	4h	18	$\overline{NMI}$	24h
3	$\overline{INT3}$	6h	19	Reserved	26h
4	TINT	8h	20	user-defined	28h
5	RINT	Ah	21	user-defined	2Ah
6	XINT	Ch	22	user-defined	2Ch
7	TRNT	Eh	23	user-defined	2Eh
8	TXNT	10h	24	user-defined	30h
9	$\overline{INT4}$	12h	25	user-defined	32h
10	Reserved	14h	26	user-defined	34h
11	Reserved	16h	27	user-defined	36h
12	Reserved	18h	28	user-defined	38h
13	Reserved	1Ah	29	user-defined	3Ah
14	Reserved	1Ch	30	user-defined	3Ch
15	Reserved	1Eh	31	user-defined	3Eh

Words 1

Cycles 4

Example

```
INTR   3    ;Control is passed to program memory location 6h
            ;PC + 1 is pushed onto the stack.
```

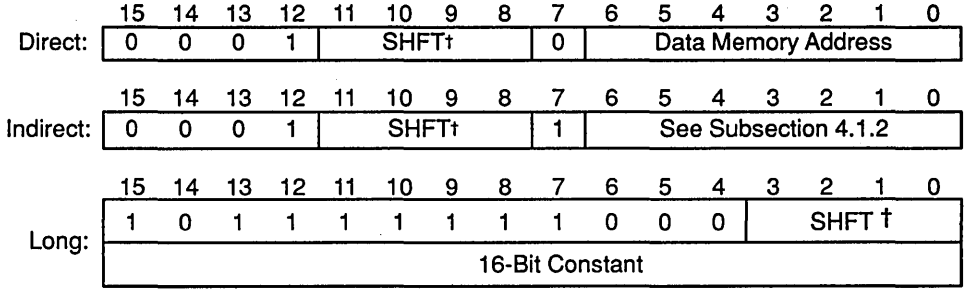
Syntax

Direct: [*label*] **LACC** *dma* [,*shift1*]
Indirect: [*label*] **LACC** {*ind*} [,*shift1* [,*next ARP*]]
Immediate: [*label*] **LACC** *#lk* [,*shift2*]

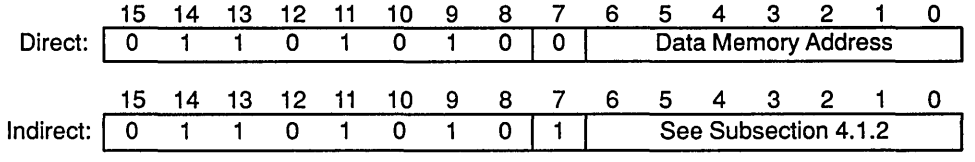
Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
0 ≤ shift1 ≤ 16 (defaults to 0)
–32768 ≤ lk ≤ 32767
0 ≤ shift2 ≤ 15 (defaults to 0)

Opcode



Load ACC with shift of 16



† See Section 4.5.

Execution

Direct or Indirect Addressing:

(PC) + 1 → PC
(dma) × 2^{shift1} → ACC

Long Immediate Addressing:
(PC) + 2 → PC
lk × 2^{shift2} → ACC

Affected by SXM.

Description

The contents of the specified data memory address or a 16-bit constant are left-shifted and loaded into the accumulator. During shifting, low-order bits are zero-filled. High-order bits are sign-extended if SXM = 1 and zeroed if SXM = 0.

Words

1 (Direct or indirect addressing)
2 (Long immediate addressing)

Cycles

1 (Direct or indirect addressing)
2 (Long immediate addressing)

Syntax

Direct: [*label*] **LACL** *dma*
Indirect: [*label*] **LACL** {*ind*} [,*next ARP*]
Immediate: [*label*] **LACL** *#k*

Operands

$0 \le dma \le 127$
$0 \le next\ ARP \le 7$
$0 \le k \le 255$

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	0	0	1	0	Data Memory Address						
Indirect:	0	1	1	0	1	0	0	1	1	See Subsection 4.1.2						
Short Immediate:	1	0	1	1	1	0	0	1	8-Bit Constant							

Execution

(PC) + 1 → PC

Direct or Indirect Addressing:

0 → ACC(31–16)
(dma) → ACC(15–0)

Short Immediate Addressing:

0 → ACC(31–8)
k → ACC(7–0)

Not affected by SXM.

Description The contents of the addressed data memory location or a zero-extended 8-bit constant are loaded into the 16 low-order bits of the accumulator. The upper half of the accumulator is zeroed. The data is treated as an unsigned 16-bit number rather than a twos-complement number. There is no sign-extension of the operand with this instruction, regardless of the state of SXM.

Words 1

Cycles 1

Syntax

Direct: [*label*] **LACT** *dma*
Indirect: [*label*] **LACT** {*ind*} [,*next ARP*]

Operands

$0 \leq dma \leq 127$
$0 \leq next\ ARP \leq 7$

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	0	1	1	0	Data Memory Address						
Indirect:	0	1	1	0	1	0	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(dma) X $2^{TREG1(3-0)}$ → ACC

If SXM = 1:
Then (dma) is sign-extended.
If SXM = 0:
Then (dma) is not sign-extended.

Affected by SXM.

Description

The LACT instruction loads the accumulator with a data memory value that has been left-shifted. The left-shift is specified by the four LSBs of TREG1, resulting in shift options from 0 to 15 bits. Using TREG1's contents as a shift code provides a dynamic shift mechanism. During shifting, the high-order bits are sign-extended if SXM = 1 and zeroed if SXM = 0.

LACT may be used to denormalize a floating-point number if the actual exponent is placed in the four LSBs of the T register and the mantissa is referenced by the data memory address. Note that this method of denormalization can be used only when the magnitude of the exponent is four bits or less.

Software compatibility with the TMS320C25 can be maintained by setting the TRM bit of the PMST status register to zero. This causes any TMS320C25 instruction that loads TREG0 to write to all three TREGs. Subsequent calls to LACT will contain the correct shift value in TREG1, maintaining object-code compatibility.

Words

1

Cycles

1

Syntax Direct: [*label*] **LAMM** *dma*
Indirect: [*label*] **LAMM** {*ind*} [,*next ARP*]

Operands 0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	0	0	0	1	0	0	0	0	Data Memory Address						
Indirect:	0	0	0	0	1	0	0	0	1	See Subsection 4.1.2						

Execution (PC) + 1 → PC
(dma) → ACC

Description The low word of the accumulator is loaded with the contents of the addressed memory-mapped register. The 9 MSBs of the data memory address are set to zero, regardless of the current value of DP or the upper 9 bits of AR(ARP). This instruction allows any location on data page zero to be loaded into the accumulator without modifying the DP field in status register ST0.

Words 1

Cycles 1 (For processor memory-mapped registers)
2 (For peripheral memory-mapped registers)

Example 1

```
LAMM BMAR     ;(DP = 6)
```

	Before Instruction		After Instruction
ACC	22221376h	ACC	5555h
BMAR	5555h	BMAR	5555h
Data Memory 31Fh	1000h	Data Memory 31Fh	1000h

Example 2

```
LAMM  *
```

	Before Instruction		After Instruction
ARP	1	ARP	1
AR1	325h	AR1	325h
ACC	22221376h	ACC	0Fh
PRD	0Fh	PRD	0Fh
Data Memory 325h	1000h	Data Memory 325h	1000h

Note that the value in data memory location 325h is not loaded into the accumulator. The value at data memory location 25h is loaded.

storage register, especially for swapping values between data memory locations without affecting the contents of the accumulator.

Words 1 (Direct, indirect, or short immediate addressing)
2 (Long immediate addressing)

Cycles 2

Example 1 `LAR AR0,DAT16 ;(DP = 6)`

	Before Instruction		After Instruction
Data Memory 310h	18h	Data Memory 310h	18h
AR0	6h	AR0	18h

Example 2 `LAR AR4,*-`

	Before Instruction		After Instruction
ARP	4	ARP	4
Data Memory 300h	32h	Data Memory 300h	32h
AR4	300h	AR4	32h

Note:

LAR in the indirect addressing mode ignores any AR modifications if the AR specified by the instruction is the same as that pointed to by the ARP. Therefore, in Example 2, AR4 is not decremented after the LAR instruction.

Example 3 `LAR AR4,#01h`

	Before Instruction		After Instruction
AR4	0FF09h	AR4	01h

Example 4 `LAR AR4,#3FFFh`

	Before Instruction		After Instruction
AR4	0h	AR4	3FFFh

Example 2 `LDP #0h`

	Before Instruction		After Instruction
DP	1FFh	DP	0h

Example 3 `LDP *,AR5`

	Before Instruction		After Instruction
ARP	4	ARP	5
AR4	300h	AR4	300h
Data Memory 300h	06h	Data Memory 300h	06h
DP	1FFh	DP	06h

Example 2 `LMMR *,#300h,AR4 ;CBCR = 1Eh`

	Before Instruction		After Instruction
ARP	0	ARO	4h
AR0	31Eh	AR0	31Eh
Data Memory 300h	20h	Data Memory 300h	20h
CBCR	0h	CBCR	20h

Syntax

Direct: [*label*] **LST** *#n, dma*
Indirect: [*label*] **LST** *#n, {ind} [,next ARP]*

Operands

$0 \le dma \le 127$
$n = 0,1$
$0 \le next\ ARP \le 7$

Opcode

LST #0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	0	0	0	1	1	1	0	0	Data Memory Address						
Indirect:	0	0	0	0	1	1	1	0	1	See Subsection 4.1.2						

LST #1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	0	0	0	1	1	1	1	0	Data Memory Address						
Indirect:	0	0	0	0	1	1	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(dma) → status register STn
dma (bits 13–15) → ARP (regardless of n)

Affects ARB, ARP, OV, OVM, DP, CNF, TC, SXM, C, HM, XF, and PM.
Does not affect INTM.

Description

Status register STn is loaded with the addressed data memory value. Note that the INTM bit is unaffected by LST #0. In addition, the LST #0 instruction does not affect the ARB field in the ST1 register even though a new ARP is loaded. If a next ARP value is specified via the indirect addressing mode, the specified value is ignored. Instead, ARP is loaded with the value contained within the addressed data memory word.

> **Note:**
>
> When ST1 is loaded, the value loaded into ARB is also loaded into ARP.

The LST instruction can be used for restoring the status registers after subroutine calls and interrupts.

Words

1

Cycles

2

Example 1

```
MAR   *,AR0
LST   #0,*,AR1  ;The data memory word addressed by the contents of
                ;auxiliary register AR0 is loaded into status register ST0,
                ;except for the INTM bit.  Note that even though a next
                ;ARP value is specified, that value is ignored, and the old
                ;ARP is not loaded into the ARB.
```

Syntax Direct: [*label*] **LT** *dma*
Indirect: [*label*] **LT** {*ind*} [,*next ARP*]

Operands 0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	1	0	0	1	1	0	Data Memory Address						

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	0	1	1	1	0	0	1	1	1	See Subsection 4.1.2						

Execution (PC) + 1 → PC
(dma) → TREG0

If TRM = 0:
(dma) → TREG1
(dma) → TREG2

Affected by TRM.

Description TREG0 is loaded with the contents of the specified data memory address (dma). The LT instruction may be used to load TREG0 in preparation for multiplication. See the LTA, LTD, LTP, LTS, MPY, MPYA, MPYS,and MPYU instructions. If the TRM bit of the PMST register is 0, then TREG1 and TREG2 are also loaded to maintain compatibility with the TMS320C25. The TREGs are memory-mapped registers and may be read and written with any instruction that accesses data memory. Note that TREG1 is only 5 bits and TREG2 is only 4 bits.

Words 1

Cycles 1

Example 1

```
LT      DAT24  ;(DP = 8. TRM = 1).
```

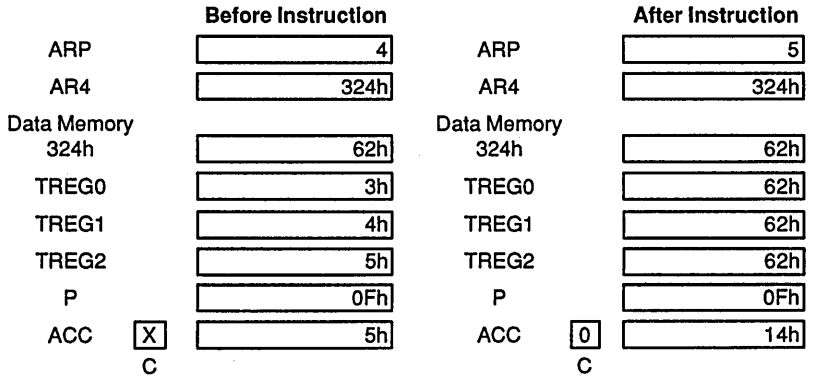
	Before Instruction		After Instruction
Data Memory 418h	62h	Data Memory 418h	62h
TREG0	3h	TREG0	62h

Example 2

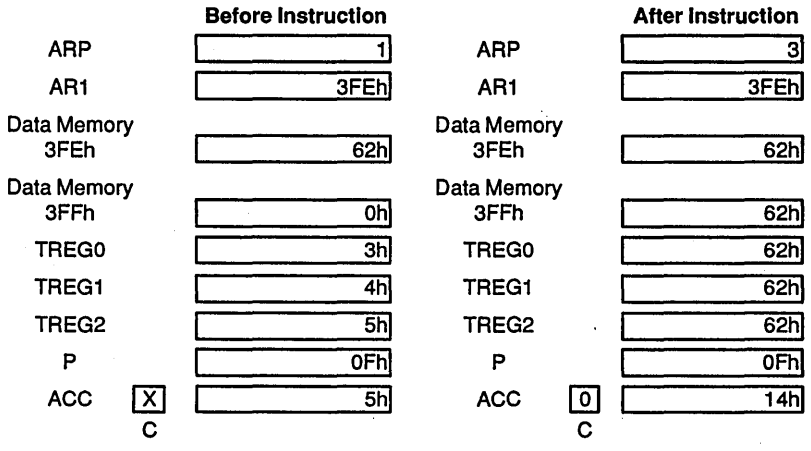
```
LT      *,AR3  ;(TRM = 0)
```

	Before Instruction		After Instruction
ARP	2	ARP	3
AR2	418h	AR2	418h
Data Memory 418h	62h	Data Memory 418h	62h
TREG0	3h	TREG0	62h
TREG1	4h	TREG1	62h
TREG2	5h	TREG2	62h

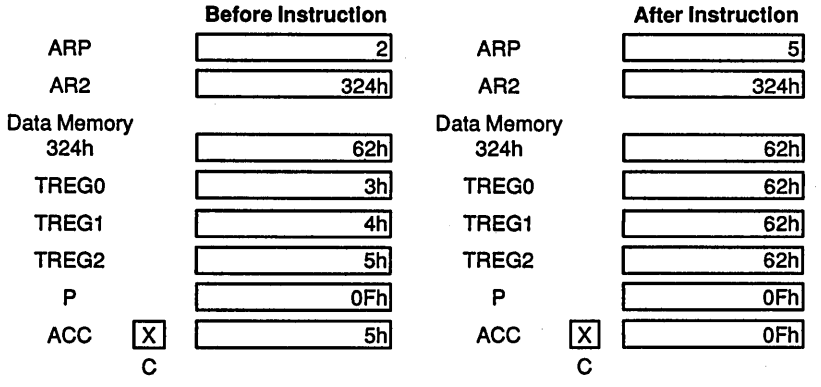
Example 2 `LTA *,5 ;(TRM = 0)`



Example 2 `LTD *,AR3 ;(TRM = 0)`



Example 2 `LTP *,AR5 ;(PM = 0, TRM = 0)`



Syntax

Direct: [*label*] **MAC** *pma, dma*
Indirect: [*label*] **MAC** *pma,* {*ind*} [*,next ARP*]

Operands

$0 \le pma \le 65535$
$0 \le dma \le 127$
$0 \le$ next ARP ≤ 7

Opcode

Direct:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	0	Data Memory Address						
16-Bit Constant															

Indirect:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	1	See Subsection 4.1.2						
16-Bit Constant															

Execution

(PC) + 2 → PC
(PFC) → MCS
(pma) → PFC

If (repeat counter) ≠ 0:
  Then (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) x (pma, addressed by PFC) → P register,
    Modify AR(ARP) and ARP as specified
    (PFC) + 1 → PFC
    (repeat counter) – 1 → repeat counter.
  Else (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) x (pma, addressed by PFC) → P register,
    Modify AR(ARP) and ARP as specified
(MCS) → PFC

Affected by OVM and PM; affects C and OV.

Description

The MAC instruction multiplies a data memory value (specified by dma) by a program memory value (specified by pma). It also adds the previous product, shifted as defined by the PM status bits, to the accumulator.

The data and program memory locations on the TMS320C5x may be any nonreserved, on-chip or off-chip memory locations. If the program memory is block B0 of on-chip RAM, then the CNF bit must be set to one. When the MAC instruction is used in the direct addressing mode, the dma cannot be modified during repetition of the instruction.

When the MAC instruction is repeated, the program memory address contained in the PFC is incremented by one during its operation. This makes it possible to access a series of operands in memory. MAC is useful for long sum-of-products operations because it becomes a single-cycle instruction, once the RPT pipeline is started.

Syntax

Direct: [*label*] **MACD** *pma, dma*
Indirect: [*label*] **MACD** *pma,* {*ind*} [*,next ARP*]

Operands

$0 \leq pma \leq 65535$
$0 \leq dma \leq 127$
$0 \leq$ next ARP ≤ 7

Opcode

Direct:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	1	0	Data Memory Address						
16-Bit Constant															

Indirect:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	1	1	See Subsection 4.1.2						
16-Bit Constant															

Execution

(PC) + 2 → PC
(PFC) → MCS
(pma) → PFC

If (repeat counter) ≠ 0:
  Then (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) x (pma, addressed by PFC) → P register
    Modify AR(ARP) and ARP as specified,
    (PFC) + 1 → PFC
    (dma) → (dma) + 1
    (repeat counter) – 1 → repeat counter.
  Else (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) x (pma, addressed by PFC) → P register
    (dma) → (dma) + 1
    Modify AR(ARP) and ARP as specified,
(MCS) → PFC

Affected by OVM and PM; affects C and OV.

Description

The MACD instruction multiplies a data memory value (specified by dma) by a program memory value (specified by pma). It also adds the previous product, shifted as defined by the PM status bits to the accumulator. The data and program memory locations on the TMS320C5x may be any nonreserved, on-chip or off-chip memory locations. If the program memory is block B0 of on-chip RAM, then the CNF bit must be set to one. When MACD is used in the direct addressing mode, the dma cannot be modified during repetition of the instruction. If MACD addresses one of the memory-mapped registers or external memory as a data memory location, the effect of the instruction will be that of a MAC instruction (see the DMOV instruction description).

Syntax

Direct: [*label*] **MADD** *dma*
Indirect: [*label*] **MADD** {*ind*} [,*next ARP*]

Operands

$0 \leq dma \leq 127$
$0 \leq next\ ARP \leq 7$

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	1	0	1	0	1	1	0	Data Memory Address						

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	1	0	1	0	1	0	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 2 → PC
(PFC) → MCS
(BMAR) → PFC

If (repeat counter) ≠ 0:
Then (ACC) + (shifted P register) → ACC,
(dma) → TREG0
(dma) x (pma, addressed by PFC) → P register,
Modify AR(ARP) and ARP as specified,
(PFC) + 1 → PFC
(dma) → (dma) + 1
(repeat counter) – 1 → repeat counter.
Else (ACC) + (shifted P register) → ACC,
(dma) → TREG0
(dma) x (pma, addressed by PFC) → P register
(dma) → (dma) + 1
Modify AR(ARP) and ARP as specified.
(MCS) → PFC

Affected by OVM and PM; affects C and OV.

Description

The MADD instruction multiplies a data memory value (specified by the dma) by a program memory value. The program memory address is contained in the BMAR register; it is not specified by a long immediate constant. This facilitates dynamic addressing of coefficient tables. In addition, the previous product, shifted as defined by the PM status bits, is added to the accumulator. The data and program memory locations on the TMS320C5x may be any nonreserved, on-chip or off-chip memory locations. If the program memory is block B0 of on-chip RAM, then the CNF bit must be set to one. When MADD instruction is used in the direct addressing mode, the dma cannot be modified during repetition of the instruction. If MADD addresses one of the memory-mapped registers or external memory as a data memory location, the effect of the instruction will be that of a MADS instruction (see the DMOV instruction description).

MADD functions in the same manner as MADS, with the addition of *data move* for on-chip RAM blocks. Otherwise, the effects are the same as for MADS. This feature makes MADD useful for applications such as convolution and transversal filtering.

Syntax

Direct: [*label*] **MADS** *dma*
Indirect: [*label*] **MADS** {*ind*} [,*next ARP*]

Operands

$0 \leq dma \leq 127$
$0 \leq next\ ARP \leq 7$

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	1	0	1	0	1	0	0	Data Memory Address						

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	1	0	1	0	1	0	1	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(PFC) → MCS
(BMAR) → PFC

If (repeat counter) ≠ 0:
  Then (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) x (pma, addressed by PFC) → P register,
    Modify AR(ARP) and ARP as specified,
    (PFC) + 1 → PFC
    (repeat counter) – 1 → repeat counter.
  Else (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) x (pma, addressed by PFC) → P register,
    Modify AR(ARP) and ARP as specified,
(MCS) → PFC

Affected by OVM and PM; affects C and OV.

Description

The MADS instruction multiplies a data memory value (specified by dma) by a program memory value (specified by pma). It also adds the previous product, shifted as defined by the PM status bits, to the accumulator. The pma is specified by the contents of the BMAR register, rather than by a long immediate constant. This allows for dynamic addressing of coefficient tables.

The data and program memory locations on the TMS320C5x may be any nonreserved, on-chip or off-chip memory locations. If the program memory is block B0 of on-chip RAM, then the CNF bit must be set to one. When MADS is used in the direct addressing mode, the dma cannot be modified during repetition of the instruction.

When the MADS instruction is repeated, the program memory address contained in the PFC is incremented by one during its operation. This makes it possible to access a series of operands in memory. MADS is useful for long sum-of-products operations because this instruction becomes a single-cycle instruction, once the RPT pipeline is started.

Syntax Direct: [*label*] **MAR** *dma*
Indirect: [*label*] **MAR** {*ind*} [,*next ARP*]

Operands 0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	0	1	0	1	1	0	Data Memory Address						
Indirect:	1	0	0	0	1	0	1	1	1	See Subsection 4.1.2						

Execution (PC) + 1 → PC

Modifies ARP, AR(ARP) as specified by the indirect addressing field. Acts as a NOP in direct addressing mode.

Description In the indirect addressing mode, the auxiliary registers and the ARP are modified; however, no use is made of the memory being referenced. MAR is used to modify the auxiliary registers or the ARP. The old ARP is copied to the ARB field of the status register ST1. Note that any operation that MAR performs can also be performed with any instruction that supports indirect addressing. ARP can also be loaded by an LST instruction. The instruction LARP from the TMS320C25 instruction set is a subset of MAR (i.e., MAR *,4 performs the same function as LARP 4).

Words 1

Cycles 1

Example 1

```
MAR   *,AR1  ;Load the ARP with 1.
```

	Before Instruction		After Instruction
ARP	0	ARP	1
ARB	7	ARB	0

Example 2

```
MAR   *+,AR5 ;Increment current auxiliary register
             ;(AR1) and load ARP with 5.
```

	Before Instruction		After Instruction
AR1	34h	AR1	35h
ARP	1	ARP	5
ARB	0	ARP	1

Cycles 1 (Direct, indirect, or short immediate addressing)
2 (Long immediate addressing)

Example 1 `MPY DAT13 ;(DP = 8)`

	Before Instruction		After Instruction
Data Memory 40Dh	7h	Data Memory 40Dh	7h
TREG0	6h	TREG0	6h
P	36h	P	2Ah

Example 2 `MPY *,AR2`

	Before Instruction		After Instruction
ARP	1	ARP	2
AR1	40Dh	AR1	40Dh
Data Memory 40Dh	7h	Data Memory 40Dh	7h
TREG0	6h	TREG0	6h
P	36h	P	2Ah

Example 3 `MPY #031h`

	Before Instruction		After Instruction
TREG0	2h	TREG0	2h
P	36h	P	62h

Example 4 `MPY #01234h`

	Before Instruction		After Instruction
TREG0	2h	TREG0	2h
P	36h	P	2468h

Syntax Direct: [*label*] **MPYS** *dma*
Indirect: [*label*] **MPYS** {*ind*} [,*next ARP*]

Operands 0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	1	0	0	0	1	0	Data Memory Address						
Indirect:	0	1	0	1	0	0	0	1	1	See Subsection 4.1.2						

Execution (PC) + 1 → PC
(ACC) – (shifted P register) → ACC
(TREG0) x (dma) → P register

Affected by OVM and PM; affects C and OV.

Description The contents of TREG0 are multiplied by the contents of the addressed data memory location. The result is placed in the P register. The previous product, shifted as defined by the PM status bits, is also subtracted from the accumulator, and the result is placed in the accumulator.

Words 1

Cycles 1

Example 1

```
MPYS   DAT13  ;(DP = 6, PM = 0)
```

	Before Instruction		After Instruction	
Data Memory 30Dh	7h		7h	
TREG0	6h		6h	
P	36h		2Ah	
ACC	54h	X	1Eh	1
		C		C

Example 2

```
MPYS   *,AR5  ;(PM = 0)
```

	Before Instruction		After Instruction	
ARP	4		5	
AR4	30Dh		30Dh	
Data Memory 30Dh	7h		7h	
TREG0	6h		6h	
P	36h		2Ah	
ACC	54h	X	1Eh	1
		C		C

Syntax [*label*] **NEG**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	0	0	0	1	0

Execution (PC) + 1 → PC
(ACC) × −1 → ACC

Affected by OVM; affects OV and C.

Description The contents of the accumulator are replaced with its arithmetic complement (twos complement). The OV bit is set when taking the NEG of 80000000h. If OVM = 1, the accumulator contents are replaced with 7FFFFFFFh. If OVM = 0, the result is 80000000h. The carry bit C on the TMS320C5x is reset to zero by this instruction for all nonzero values of the accumulator, and is set to one if the accumulator equals zero.

Words 1

Cycles 1

Example 1

```
NEG     ;(OVM = X)
```

Before Instruction			After Instruction		
ACC	X	0FFFFF228h	ACC	0	0DD8h
C	X		C	X	
OV			OV		

Example 2

```
NEG     ;(OVM = 0)
```

Before Instruction			After Instruction		
ACC	X	080000000h	ACC	0	080000000h
C	X		C	1	
OV			OV		

Example 3

```
NEG     ;(OVM = 1)
```

Before Instruction			After Instruction		
ACC	X	080000000h	ACC	0	7FFFFFFFh
C	X		C	1	
OV			OV		

Syntax [*label*] **NOP**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0

Execution (PC) + 1 → PC

Description No operation is performed. The NOP instruction affects only the PC. The NOP instruction is useful to create pipeline and execution delays.

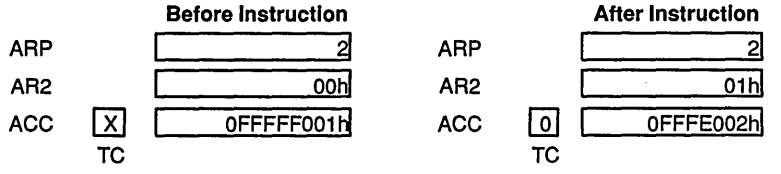
Words 1

Cycles 1

Example

```
NOP     ;No operation is performed.
```

Example 1 NORM *+



Example 2 31-Bit Normalization:

```
     MAR    *,AR1        ;Use AR1 to store the exponent.
     LAR    AR1,#0h      ;Clear out exponent counter.
LOOP NORM   *+           ;One bit is normalized.
     BCND   LOOP,NTC     ;If TC = 0, magnitude not found yet.
```

Example 3 15-Bit Normalization:

```
MAR  *,AR1        ;Use AR1 to store the exponent.
LAR  AR1,#0Fh     ;Initialize exponent counter.
RPT  #14          ;15-bit normalization is specified (yielding
                  ;a 4-bit exponent and 16-bit mantissa).
NORM *-           ;NORM automatically stops shifting when
                  ;the first significant magnitude bit is found,
                  ;performing NOPs for the remainder of the
                  ;repeat loops.
```

The method in Example 2 is used to normalize a 32-bit number and yields a 5-bit exponent magnitude. The method in Example 3 is used to normalize a 16-bit number and yields a 4-bit magnitude. If the number requires only a small amount of normalization, the Example 2 method may be preferable to the Example 3 method. This is because the loop in Example 2 runs only until normalization is complete. Example 3 always executes all 15 cycles of the repeat loop. Specifically, Example 2 is more efficient if the number requires three or less shifts. If the number requires six or more shifts, Example 3 is more efficient.

Note:

The NORM instruction may be used without a specified operand. In that case, any comments on the same line as the instruction will be interpreted as the operand. If the first character is an asterisk *, then the instruction will be assembled as NORM * with no auxiliary register modification taking place upon execution. Therefore, TI recommends that you replace the NORM instructions with NORM *+ when you want the default increment modification.

Example 1 `OPL DAT10 ;(DP=6)`

	Before Instruction		After Instruction
DBMR	0FFF0h	DBMR	0FFF0h
Data Memory 30Ah	0001h	Data Memory 30Ah	0FFF1h

Example 2 `OPL #0FFFh,DAT10 ;(DP=6)`

	Before Instruction		After Instruction
Data Memory 30Ah	0001h	Data Memory 30Ah	0FFFh

Example 3 `OPL *,AR6`

	Before Instruction		After Instruction
ARP	3	ARP	6
AR3	300h	AR3	300h
DBMR	0F0h	DBMR	0F0h
Data Memory 300h	0Fh	Data Memory 300h	0FFh

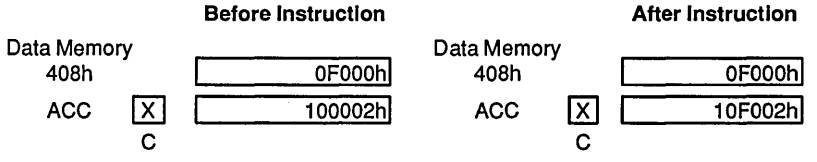
Example 4 `OPL #1111h,*,AR3`

	Before Instruction		After Instruction
ARP	6	ARP	3
AR6	306h	AR6	306h
Data Memory 306h	0Eh	Data Memory 306h	111Fh

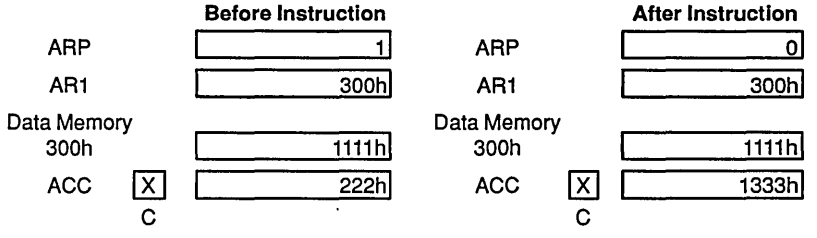
Words 1 (Direct or indirect addressing)
2 (Long immediate addressing)

Cycles 1 (Direct or indirect addressing)
2 (Long immediate addressing)

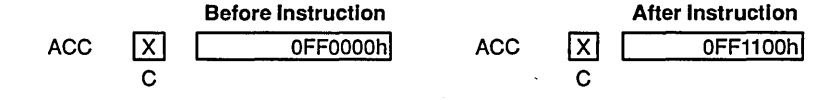
Example 1 `OR DAT8 ;(DP = 8)`



Example 2 `OR *,AR0`



Example 3 `OR #08111h,8`



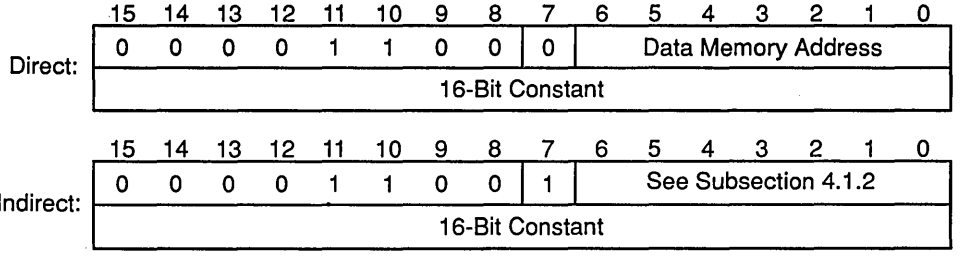
Syntax

Direct: [*label*] **OUT** *dma* , *PA*
Indirect: [*label*] **OUT** {ind}, *PA* [,*next ARP*]

Operands

$0 \le$ dma ≤ 127
$0 \le$ next ARP ≤ 7
$0 \le PA \le 65535$

Opcode



Execution

(PC) + 2 → PC
While (repeat counter) ≠ 0
  Port address → address bus A15–A0
  (dma) → Data bus D15–D0
  Port address + 1 → Port address
  (repeat counter – 1) → (repeat counter)

Description

The OUT instruction writes a 16-bit value from a data memory location to the specified I/O port. The $\overline{IS}$ line goes low to indicate an I/O access, and the $\overline{STRB}$, R/$\overline{W}$, and READY timings are the same as for an external data memory write. Note that port addresses 50h–5Fh are memory-mapped (see subsection 5.1.1); the other port addresses are not.

RPT can be used with the OUT instruction to write consecutive words from data memory to I/O space. In the repeat mode, the port address (PA) is incremented after each access.

Words

2

Cycles

3 (Each output will increase by **i**, I/O memory wait states. This is the number of cycles the device must wait for external I/O devices to access data.)

Example 1

```
OUT     DAT0,PA7  ;(DP = 4) Output data word stored in data memory
                  ;location 200h to peripheral on port address 7.
```

Example 2

```
OUT     *,PA15    ;Output data word referenced by current auxiliary
                  ;register to peripheral on port address 15.
```

Syntax [*label*] **POP**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0

Execution (PC) + 1 → PC
(TOS) → ACC(15–0)
0 → ACC(31–16)
Pop stack one level

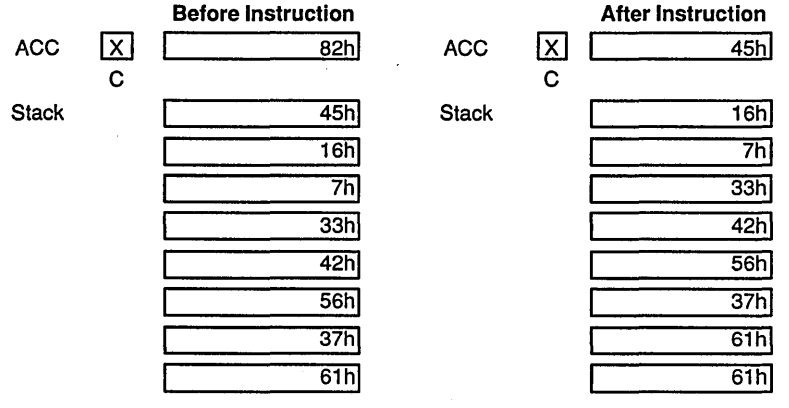
Description The contents of the top of the stack (TOS) are copied to the low accumulator, and the stack is popped after the contents are copied. The upper half of the accumulator is set to all zeroes.

The hardware stack is last-in, first-out with eight locations. Any time a pop occurs, every stack value is copied to the next higher stack location, and the top value is removed from the stack. After a pop, the bottom two stack words will have the same value. Because each stack value is copied, if more than seven stack pops (POP, POPD, RETC, RETE, RETI, or RET instructions) occur before any pushes occur, all levels of the stack contain the same value. No provision exists to check stack underflow.

Words 1

Cycles 1

Example `POP`



Example 2 `POPD *+,AR1`

	Before Instruction		After Instruction
ARP	0	ARP	1
AR0	300h	AR0	301h
Data Memory 300h	55h	Data Memory 300h	92h
Stack	92h	Stack	72h
	72h		8h
	8h		44h
	44h		81h
	81h		75h
	75h		32h
	32h		0AAh
	0AAh		0AAh

Example 2 `PSHD *,AR1`

	Before Instruction		After Instruction
ARP	0	ARP	1
AR0	1FFh	AR0	1FFh
Data Memory 1FFh	12h	Data Memory 1FFh	12h
Stack	2h	Stack	12h
	33h		2h
	78h		33h
	99h		78h
	42h		99h
	50h		42h
	0h		50h
	0h		0h

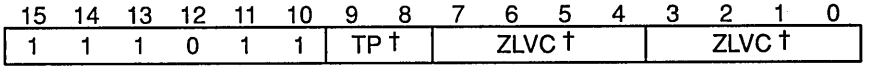
Syntax [*label*] **RETC** [*D*] [*cond1*] [, *cond2*] [,...

Operands Conditions:

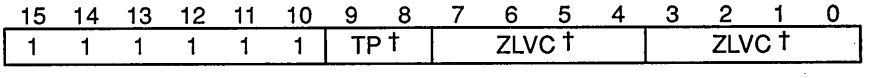
ACC=0	EQ
ACC≠0	NEQ
ACC<0	LT
ACC≤0	LEQ
ACC>0	GT
ACC≥0	GEQ
C=0	NC
C=1	C
OV=0	NOV
OV=1	OV
$\overline{\text{BIO}}$ low	BIO
TC=0	NTC
TC=1	TC
Unconditional	UNC

Opcode

RETC:



RETCD:



Execution

If (condition(s)) then
(TOS) → PC
Pop stack one level.
Else, continue

Description A standard return, RET, is executed if the specified conditions are met. Note that not all combinations of conditions are meaningful. The two one-word instructions or one two-word instruction following the RETC are fetched and executed before the execution of the return, if the delayed version is specified with the "D" suffix. If the delayed instruction is specified, the two instruction words following the RETCD instruction have no effect on the conditions being tested.

Words 1

Cycles

2	(Return not executed)	2	(If instruction delayed)
4	(Return executed)	2	(If instruction delayed)

Example 1

```
RETC  GEQ,NOV  ;A return, RET, is executed if the accumulator contents are
               ;positive and the OV bit is a zero.
```

Example 2

```
RETCD  C       ;A return, RET, is executed if the carry bit is set. The two
MAR *,4        ;instructions following the return instruction will be
LAR AR3,#1h    ;executed before the return is taken.
```

Example 2

```
RETD
MAR    *,4
LACC   #1h
```

	Before Instruction		After Instruction
PC	96h	PC	37h
ARP	0	ARP	4
ACC	0h	ACC	01h
Stack	37h	Stack	45h
	45h		75h
	75h		21h
	21h		3Fh
	3Fh		45h
	45h		6Eh
	6Eh		6Eh
	6Eh		6Eh

Syntax [*label*] **RETI**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	1	1	0	0	0

Execution (TOS) → PC
Pop stack one level.

Description The contents of the top stack register are copied into the program counter. The RETI instruction also pops the values in the shadow registers (stored when an interrupt was taken) back into their corresponding strategic registers. The following registers are shadowed: ACC, ACCB, PREG, ST0, ST1, PMST, ARCR, INDX, TREG0, TREG1, and TREG2. The XF bit in status register ST1 is not saved or restored to/from the shadow registers during interrupt service routines.

Words 1

Cycles 4

Example `RETI`

	Before Instruction		After Instruction
PC	96h	PC	37h
Stack	37h	Stack	45h
	45h		75h
	75h		21h
	21h		3Fh
	3Fh		45h
	45h		6Eh
	6Eh		6Eh
	6Eh		6Eh

Syntax [*label*] **ROLB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	0	1	0	0

Execution

(PC) + 1 → PC
C → ACCB(0)
(ACCB(30–0)) → ACCB(31–1)
(ACCB(31)) → ACC(0)
(ACC(30–0)) → ACC(31–1)
(ACC(31)) → C

Affects C.
Not affected by SXM.

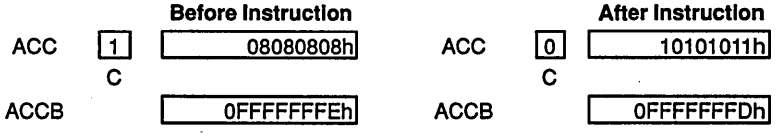
Description The ROLB instruction causes a 65-bit rotation. The contents of both the accumulator (ACC) and accumulator buffer (ACCB) are rotated to the left by one bit. The MSB of the original contents in the accumulator shifts into the carry position. The original value of the carry bit (C) shifts into the LSB position of the accumulator buffer, and the MSB of the original contents of the accumulator buffer shifts into the LSB position of the accumulator.

Words 1

Cycles 1

Example

```
ROLB
```



Syntax [*label*] **RORB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	0	1	0	1

Execution

(PC) + 1 → PC
C → ACC(31)
(ACC(31–1)) → ACC(30–0)
(ACC(0)) → ACCB(31)
(ACCB(31–1)) → ACCB(30–0)
(ACCB(0)) → C

Affects C.
Not affected by SXM.

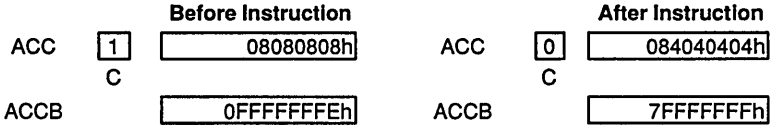
Description The RORB instruction causes a 65-bit rotation. The contents of both the accumulator (ACC) and accumulator buffer (ACCB) are rotated to the right by one bit. The LSB of the original contents in the accumulator buffer shifts into the carry position. The original value of the carry bit (C) shifts into the MSB position of the accumulator, and the LSB of the original contents of the accumulator shifts into the MSB position of the accumulator buffer.

Words 1

Cycles 1

Example

```
RORB
```



RPT is especially useful for block moves, multiply-accumulates, normalization, and other functions. The repeat instruction itself is not repeatable.

Words 1 (Direct, indirect, or short immediate addressing)
2 (Long immediate addressing)

Cycles 2

Example 1

```
RPT    DAT127 ;(DP = 31)
```

	Before Instruction		After Instruction
Data Memory 0FFFh	0Ch	Data Memory 0FFFh	0Ch
RPTC	0h	RPTC	0Ch

Example 2

```
RPT    *,AR1
```

	Before Instruction		After Instruction
ARP	0	ARP	1
AR0	300h	AR0	300h
Data Memory 300h	0FFFh	Data Memory 300h	0FFFh
RPTC	0h	RPTC	0FFFh

Example 3

```
RPT    #1 ;Repeat next instruction 2 times.
```

	Before Instruction		After Instruction
RPTC	0h	RPTC	1h

Example 4

```
RPT    #1111h ;Repeat next instruction 4370 times.
```

	Before Instruction		After Instruction
RPTC	0h	RPTC	1111h

Syntax Long Immediate: [*label*] **RPTZ** *#lk*

Operands 0 ≤ lk ≤ 65535

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	1	1	0	0	0	1	0	1
16-Bit Constant															

Execution 0 → ACC
0 → PREG
(PC) + 1 → PC
lk → RPTC

Description The RPTZ instruction clears the accumulator and product register and repeats the instruction following the RPTZ *n* times, where *n* = *lk+1*. RPTZ is equivalent to the following instruction sequence:

```
MPY #0
PAC
RPT #<lk>
```

Words 2

Cycles 2

Example
```
RPTZ   #7FFh     ;Zero product register and accumulator.
MACD   pma,*+    ;Repeat MACD 2048 times.
```

Syntax Direct: [*label*] **SACH** *dma* [,*shift*]
Indirect: [*label*] **SACH** {*ind*} [,*shift*[,*next ARP*]]

Operands 0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
0 ≤ shift ≤ 7 (defaults to 0)

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	1	1	SHF †			0	Data Memory Address						
Indirect:	1	0	0	1	1	SHF †			1	See Subsection 4.1.2						

† See Section 4.5.

Execution (PC) + 1 → PC
[(ACC) x 2^{shift}] → dma

Not affected by SXM

Description The SACH instruction copies the entire accumulator into a shifter, where it left-shifts the entire 32-bit number from 0 to 7 bits. It then copies the upper 16 bits of the shifted value into data memory. The accumulator itself remains unaffected.

Words 1

Cycles 1

Example 1

```
SACH   DAT10,1  ;(DP = 4)
```

Before Instruction			After Instruction		
ACC	X	4208001h	ACC	X	4208001h
	C			C	
Data Memory 20Ah		0h	Data Memory 20Ah		0841h

Example 2

```
SACH   *+,0,AR2
```

Before Instruction			After Instruction		
ARP		1	ARP		2
AR1		300h	AR1		301h
ACC	X	4208001h	ACC	X	4208001h
	C			C	
Data Memory 300h		0h	Data Memory 300h		0420h

Syntax

Direct: [*label*] **SAMM** *dma*
Indirect: [*label*] **SAMM** {*ind*} [, *next ARP*]

Operands

$0 \le dma \le 127$
$0 \le$ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	0	1	0	0	0	0	Data Memory Address						

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	1	0	0	0	1	0	0	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(ACC) → dma(0–7)

Description

The low word of the accumulator is copied to the addressed memory-mapped register. The upper 9 bits of the data address are set to zero, regardless of the current value of DP or the upper 9 bits of AR(ARP). This instruction allows the accumulator to be stored to any memory location on data page 0 without modifying the DP field in status register ST0.

Words

1

Cycles

1 (For processor memory-mapped registers)
2 (For peripheral memory-mapped registers)

Example 1

```
SAMM  PRD    ;(DP = 6)
```

	Before Instruction		After Instruction
ACC	80h	ACC	80h
PRD	05h	PRD	80h
Data Memory 325h	0Fh	Data Memory 325h	0Fh

Example 2

```
SAMM  *,AR2  ;(BMAR = 1Fh)
```

	Before Instruction		After Instruction
ARP	7	ARP	2
AR7	31Fh	AR7	31Fh
ACC	080h	ACC	080h
BMAR	0h	BMAR	080h
Data Memory 31Fh	11h	Data Memory 31Fh	11h

Syntax **SATH**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	1	1	0	1	0

Execution

(PC) + 1 → PC
16 x (TREG1(4)) → count
(ACC) right-shifted by count → ACC

Affected by SXM.

Description The accumulator is barrel-shifted right by 16 bits if bit 4 of TREG1 is a one. If bit 4 of TREG1 is a zero, the accumulator is unaffected. Zeroes are shifted in if SXM=0. Copies of ACC(31) are shifted in if SXM=1. The SATH instruction in conjunction with the SATL instruction allows a 2-cycle 0- to 31-bit right shift. The carry bit is unaffected.

Words 1

Cycles 1

Example 1

```
SATH   ;(SXM = 0)
```

	Before Instruction		After Instruction	
ACC	X	0FFFF0000h	ACC	X	0000FFFFh
	C			C	
TREG1		xx1xh	TREG1		xx1xh

Example 2

```
SATH   ;(SXM = 1)
```

	Before Instruction		After Instruction	
ACC	X	0FFFF0000h	ACC	X	0FFFFFFFFh
	C			C	
TREG1		xx1xh	TREG1		xx1xh

Syntax [*label*] **SBB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	1	0	0	0

Execution (PC) + 1 → PC
(ACC) – (ACCB) → ACC

Description The contents of the accumulator buffer (ACCB) are subtracted from the contents of the accumulator. The result is stored in the accumulator, and the accumulator buffer is not affected. The carry bit is reset to zero if the result of the subtraction generates a borrow.

Words 1

Cycles 1

Example SBB

	Before Instruction		After Instruction	
ACC	[X]	20000000h	[1]	10000000h
	C		C	
ACCB		10000000h		10000000h

Syntax [*label*] **SBRK** *#k*

Operands $0 \leq k \leq 255$

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Short:	0	1	1	1	1	1	0	0	8-Bit Constant							

Execution (PC) + 1 → PC
AR(ARP) – 8-bit positive constant → AR(ARP)

Description The 8-bit immediate value is subtracted, right-justified, from the currently selected auxiliary register with the result replacing the auxiliary register contents. The subtraction takes place in the ARAU, with the immediate value treated as a 8-bit positive integer.

Words 1

Cycles 1

Example
```
SBRK  #0FFh
```

	Before Instruction		After Instruction
ARP	7	ARP	7
AR7	0h	AR7	0FF01h

Example

```
SETC   TC  ;TC is bit 11 of ST1
```

	Before Instruction		After Instruction
ST1	x1xxh	ST1	x9xxh

Syntax [*label*] **SFLB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	0	1	1	0

Execution

(PC) + 1 → PC
0 → ACCB(0)
(ACCB(30–0)) → ACCB(31–1)
(ACCB(31)) → ACC(0)
(ACC(30–0)) → ACC(31–1)
(ACC(31) → C

Affects C.
Not affected by SXM bit.

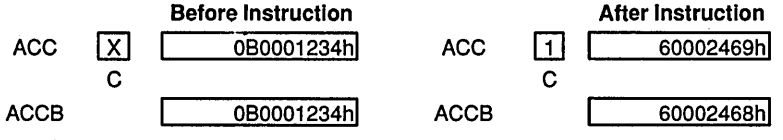
Description The SFLB instruction shifts the concatenation of the accumulator (ACC) and accumulator buffer (ACCB) left by one bit position. The least significant bit of the accumulator buffer is filled with a zero, and the most significant bit of the accumulator buffer is shifted into the least significant bit of the accumulator. The most significant bit of the accumulator is shifted into the carry bit (C). The SFLB instruction is unaffected by SXM.

Words 1

Cycles 1

Example

```
SFLB
```



Syntax [*label*] **SFRB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	0	1	1	1

Execution (PC) + 1 → PC

If SXM=0:
Then 0 → ACC(31)
If SXM=1:
Then (ACC(31)) → ACC(31)

(ACC(31–1)) → ACC(30–0)
(ACC(0)) → ACCB (31)
(ACCB(31–1)) → ACCB(30–0)
(ACCB(0)) → C

Affects C.
Affected by SXM.

Description The SFRB instruction shifts the concatenation of the accumulator (ACC) and accumulator buffer (ACCB) right by one bit position. The LSB of the ACCB is shifted into the carry bit.

If SXM=1, the instruction produces an arithmetic right shift. The sign bit (MSB) of the accumulator is unchanged and is also copied into bit 30. Bit 0 of the accumulator buffer is shifted into the carry bit (C).

If SXM=0, the instruction produces a logic right shift. All of the accumulator and accumulator buffer bits are shifted right by one bit. The least significant bit of the accumulator buffer is shifted into the carry bit, and the most significant bit of the accumulator is filled with a zero.

Words 1

Cycles 1

Example 1
```
SFRB   ;(SXM = 0)
```

	Before Instruction		After Instruction	
ACC	[X]	0B0001235h	ACC	[0] 5800091Ah
	C			C
ACCB		0B0001234h	ACCB	0D800091Ah

Example 2
```
SFRB   ;(SXM = 1)
```

	Before Instruction		After Instruction	
ACC	[X]	0B0001234h	ACC	[0] 0D800091Ah
	C			C
ACCB		0B0001234h	ACCB	05800091Ah

Example 2 `SMMR *,#307h,AR6 ;(CBCR = 1Eh)`

	Before Instruction		After Instruction
ARP	6	ARP	6
AR6	0F01Eh	AR6	0F01Eh
Data Memory 307h	1376h	Data Memory 307h	5555h
CBCR	5555h	CBCR	5555h

Syntax Direct: [*label*] **SPH** *dma*
Indirect: [*label*] **SPH** {*ind*} [,*next ARP*]

Operands 0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	0	1	1	0	1	0	Data Memory Address						
Indirect:	1	0	0	0	1	1	0	1	1	See Subsection 4.1.2						

Execution (PC) + 1 → PC
(P register shifter output (31–16)) → dma

Affected by PM.

Description The high-order bits of the P register, shifted as specified by the PM bits, are stored in data memory. Neither the P register nor the accumulator is affected by this instruction. High-order bits are sign-extended when the right-shift-by-6 mode is selected. Low-order bits are taken from the low P register when left shifts are selected.

Words 1

Cycles 1

Example 1

```
SPH    DAT3   ;(DP = 4, PM = 0).
```

	Before Instruction		After Instruction
P	0FE079844h	P	0FE079844h
203h	4567h	203h	0FE07h

Example 2

```
SPH    *,AR7  ;(PM = 2)
```

	Before Instruction		After Instruction
ARP	6	ARP	7
AR6	203h	AR6	203h
P	0FE079844h	P	0FE079844h
Data Memory 203h	4567h	Data Memory 203h	0E079h

Syntax Direct: [*label*] **SPLK** #*lk,dma*
Indirect: [*label*] **SPLK** #*lk*, {*ind*} [,*next ARP*]

Operands 0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
lk: 16-bit constant

Opcode

Direct:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	0	0	Data Memory Address						
16-Bit Constant															

Indirect:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	0	1	See Subsection 4.1.2						
16-Bit Constant															

Execution (PC) + 2 → PC
lk → dma

Description The SPLK instruction allows a full 16-bit pattern to be written into any memory location. The parallel logic unit (PLU) supports this bit manipulation independently of the ALU so that the ACC is unaffected.

Words 2

Cycles 2

Example 1

```
SPLK   #7FFFh,DAT3   ;(DP = 6)
```

	Before Instruction		After Instruction
Data Memory 303h	0FE07h	Data Memory 303h	7FFFh

Example 2

```
SPLK   #1111h,*+,AR4
```

	Before Instruction		After Instruction
ARP	0	ARP	4
AR4	300h	AR4	301h
Data Memory 300h	07h	Data Memory 300h	1111h

Syntax Direct: [*label*] **SQRA** *dma*
Indirect: [*label*] **SQRA** {*ind*} [,*next ARP*]

Operands 0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	1	0	0	1	0	0	Data Memory Address						
Indirect:	0	1	0	1	0	0	1	0	1	See Subsection 4.1.2						

Execution (PC) + 1 → PC
(ACC) + (shifted P register) → ACC
(dma) → TREG0
(dma) x (dma) → P register

Affects OV and C.
Affected by PM and OVM.

Description The contents of the P register, shifted as defined by the PM status bits, are added to the accumulator. The addressed data memory value is then loaded into TREG0, squared, and stored in the P register.

Words 1

Cycles 1

Example 1

```
SQRA   DAT30  ;(DP = 6, PM = 0).
```

	Before Instruction	After Instruction
Data Memory 31Eh	0Fh	0Fh
TREG0	3h	0Fh
P	12Ch	0E1h
ACC	1F4h	320h
C	X	0

Example 2

```
SQRA   *,AR4  ;(PM = 0).
```

	Before Instruction	After Instruction
ARP	3	4
AR3	31Eh	31Eh
Data Memory 31Eh	0Fh	0Fh
TREG0	3h	0Fh
P	12Ch	0E1h
ACC	1F4h	320h
C	X	0

Syntax Direct: [*label*] **SST** *#n, dma*
Indirect: [*label*] **SST** *#n, {ind} [,next ARP]*

Operands $0 \le dma \le 127$
$n = 0,1$
$0 \le$ next ARP ≤ 7

Opcode

Store Status Register 0 SST#0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	0	1	1	1	0	0	Data Memory Address						
Indirect:	1	0	0	0	1	1	1	0	1	See Subsection 4.1.2						

Store Status Register 1 SST#1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	0	1	1	1	1	0	Data Memory Address						
Indirect:	1	0	0	0	1	1	1	1	1	See Subsection 4.1.2						

Execution (PC) + 1 → PC
(status register STn) → dma

Description Status register STn is stored in data memory. In the direct addressing mode, status register STn is always stored in page 0, regardless of the value of the DP register. The processor automatically forces the page to be 0, and the specified location within that page is defined in the instruction. Note that the DP register is not physically modified. This allows storage of the DP register in the data memory on interrupts, etc., in the direct addressing mode without having to change the DP. In the indirect addressing mode, the data memory address is obtained from the auxiliary register selected (see the LST instruction for more information). In the indirect addressing mode, any page in data memory may be accessed.

Status registers ST0 and ST1 are defined in subsection 3.6.3, *Status and Control Registers.*

Words 1

Cycles 1

Example 1

```
SST    #0,DAT96 ;(DP = 6)
```

	Before Instruction		After Instruction
ST0	0A408h	ST0	0A408h
Data Memory 60h	0Ah	Data Memory 60h	0A408h

Syntax

Direct:	[*label*] **SUB** *dma* [,*shift1*]
Indirect:	[*label*] **SUB** {*ind*} [,*shift1* [,*next ARP*]]
Short Immediate:	[*label*] **SUB** #*k*
Long Immediate:	[*label*] **SUB** #*lk* [,*shift2*]

Operands

$0 \le dma \le 127$
$0 \le shift1 \le 16$ (defaults to 0)
$0 \le next\ ARP \le 7$
$0 \le k \le 255$
$-32768 \le lk \le 32767$
$0 \le shift2 \le 15$ (defaults to 0)

Opcode

Subtract from accumulator with shift

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	0	1	1	SHFT †				0	Data Memory Address						
Indirect:	0	0	1	1	SHFT †				1	See Subsection 4.1.2						

Subtract from accumulator with shift of 16

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	1	0	1	0	Data Memory Address						
Indirect:	0	1	1	0	0	1	0	1	1	See Subsection 4.1.2						

Subtract from ACC short immediate

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Short:	1	0	1	1	1	0	1	0	8-Bit Constant							

Subtract from ACC long immediate with shift

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Long:	1	0	1	1	1	1	1	1	1	0	1	0	SHFT †			
	16-Bit Constant															

† See Section 4.5.

Execution

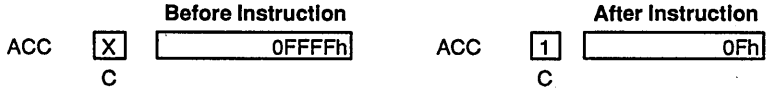
Direct or Indirect Addressing:

(PC) + 1 → PC
(ACC) – [(dma) x 2^{shift1}] → ACC
Affects C and OV.
Affected by SXM and OVM.

Short Immediate Addressing:

(PC) + 1 → PC
(ACC) – k → ACC

Example 4 `SUB #0FFFh,4 ;(SXM = 0)`



Syntax

Direct: [*label*] **SUBC** *dma*
Indirect: [*label*] **SUBC** {*ind*} [,*next ARP*]

Operands

$0 \leq$ dma ≤ 127
$0 \leq$ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	0	0	0	1	0	1	0	0	Data Memory Address						

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	0	0	0	0	1	0	1	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(ACC) – [(dma) x 2^{15}] → ALU output

If ALU output ≥ 0:
Then (ALU output) x 2 + 1 → ACC;
Else (ACC) x 2 → ACC.

Affects OV and C.
Affected by SXM.
Not affected by OVM (no saturation) .

Description

The SUBC instruction performs conditional subtraction, which may be used for division. The 16-bit dividend is placed in the low accumulator, and the high accumulator is zeroed. The divisor is in data memory. SUBC is executed **16** times for 16-bit division. After completion of the last SUBC, the quotient of the division is in the lower-order 16-bit field of the accumulator, and the remainder is in the higher-order 16-bits of the accumulator. SUBC assumes that the divisor and the dividend are both positive. The SXM bit will affect this operation. If SXM=1, then the divisor must have a **0** value in the MSB. If SXM=0, then any 16-bit divisor value will produce the expected results. The dividend, which is in the accumulator, must initially be positive (i.e., bit 31 must be **0**) and must remain positive following the accumulator shift, which occurs in the first portion of the SUBC execution.

If the 16-bit dividend contains fewer than 16 significant bits, the dividend may be placed in the accumulator and left-shifted by the number of leading nonsignificant zeroes. The number of executions of SUBC is reduced from 16 by that number. One leading zero is always significant.

Note that SUBC affects OV but is not affected by OVM, and therefore the accumulator does not saturate upon positive or negative overflows when executing this instruction. The carry bit is affected in the normal manner during this instruction.

Words

1

Cycles

1

Syntax Direct: [*label*] **SUBS** *dma*
Indirect: [*label*] **SUBS** {*ind*} [,*next ARP*]

Operands 0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	1	1	0	0	Data Memory Address						

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	0	1	1	0	0	1	1	0	1	See Subsection 4.1.2						

Execution (PC) + 1 → PC
(ACC) – (dma) → ACC

Affects OV and C; affected by OVM.
Not affected by SXM.

Description The contents of the specified data memory location are subtracted from the accumulator with sign extension suppressed. The data is treated as a 16-bit unsigned number, regardless of SXM. The accumulator behaves as a signed number. SUBS produces the same results as a SUB instruction with SXM = 0 and a shift count of 0.

Words 1

Cycles 1

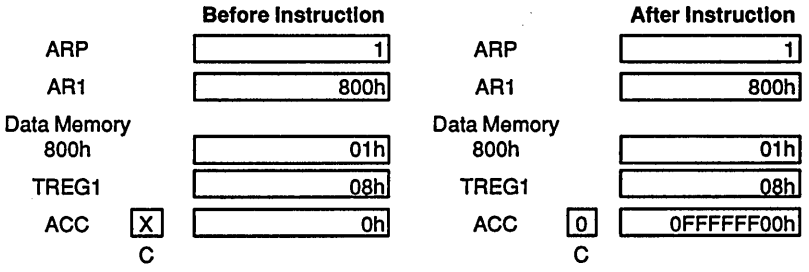
Example 1
```
SUBS   DAT2   ;(DP = 16, SXM = 1).
```

	Before Instruction		After Instruction	
Data Memory 802h	0F003h		0F003h	
ACC	X	0F105h	1	102h
	C		C	

Example 2
```
SUBS   *   ;(SXM = 1)
```

	Before Instruction		After Instruction	
ARP	0		0	
AR0	310h		310h	
Data Memory 310h	0F003h		0F003h	
ACC	X	0FFFF105h	1	0FFF0102h
	C		C	

Example 2 `SUBT *`



Example 2 `TBLR *,AR7`

	Before Instruction		After Instruction
ARP	0	ARP	7
AR0	300h	AR0	300h
ACC	24h	ACC	24h
Program Memory 24h	307h	Program Memory 24h	307h
Data Memory 300h	75h	Data Memory 300h	307h

Example 2 `TBLW *`

	Before Instruction		After Instruction
ARP	6	ARP	6
AR6	1006h	AR6	1006h
ACC	258h	ACC	258h
Data Memory 1006h	4340h	Data Memory 1006h	4340h
Program Memory 258h	307h	Program Memory 258h	4340h

Syntax [*label*] **XC** *k* [,*cond1*] [,*cond2*] [,...

Operands k = 1 or 2

Conditions:

ACC=0	EQ
ACC≠0	NEQ
ACC<0	LT
ACC≤0	LEQ
ACC>0	GT
ACC≥0	GEQ
C=0	NC
C=1	C
OV=0	NOV
OV=1	OV
$\overline{\text{BIO}}$ low	BIO
TC=0	NTC
TC=1	TC
Unconditional	UNC

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	N †	0	1	TP †		ZLVC †				ZLVC †			

† See Section 4.5.

Execution If (condition(s))
Then next **k** instructions executed
Else execute NOP's for next **k** instructions

Description If k = 2 and conditions are met, the one two-word instruction or two one-word instructions following the XC instruction execute. If k = 1 and conditions are met, the one-word instruction following the XC instruction executes. If the conditions are not met, one or two NOPs are executed. Note that not all combinations of conditions are meaningful. The XC instruction and two-instruction words following the XC are uninterruptible.

Conditions tested are sampled one full cycle before the XC is executed. Therefore, if the instruction before the XC is a single-cycle instruction, its execution will not affect the condition of the XC. If the instruction prior to the XC does affect the condition being tested, interrupt operation with the XC can cause undesired results.

Words 1

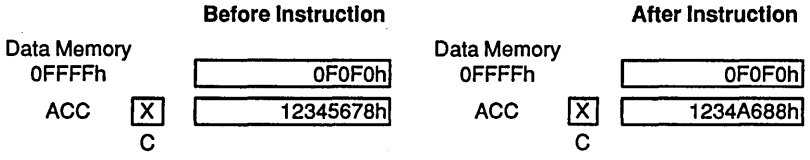
Cycles 1

Example

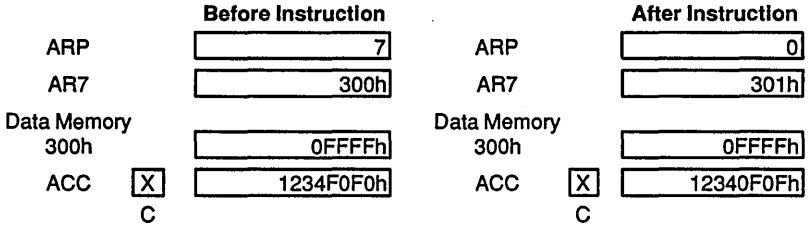
```
XC  1,LEQ,C
MAR  *+
ADD DAT100
```

If the accumulator contents are less than or equal to zero and the carry bit is set, the ARP is modified prior to the execution of the ADD instruction.

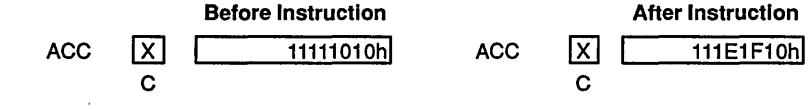
Example 1 `XOR DAT127 ;(DP = 511)`



Example 2 `XOR *+,AR0`



Example 3 `XOR #0F0F0h,4`

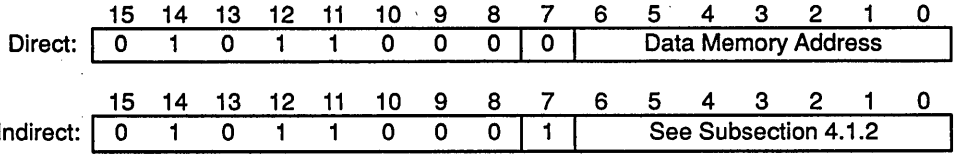


Syntax Direct: [*label*] **XPL** [#*lk*,] *dma*
Indirect: [*label*] **XPL** [#*lk*,] {*ind*} [,*next ARP*]

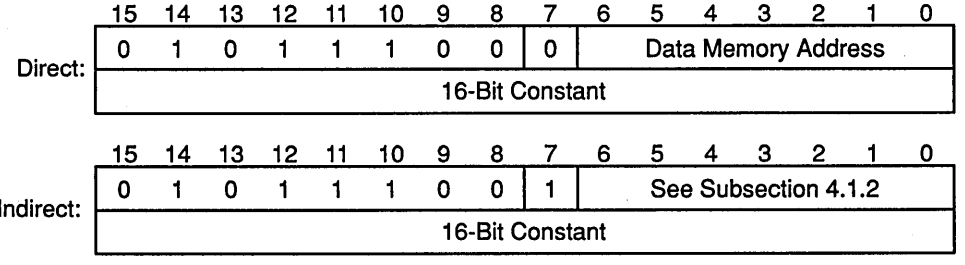
Operands 0 ≤ dma ≤ 127
lk: 16-bit constant
0 ≤ next ARP ≤ 7

Opcode

XOR DBMR with data value



XOR long immediate with data value



Execution lk unspecified:

(PC) + 1 → PC
(dma) **XOR** (DBMR) → dma

lk specified:

(PC) + 2 → PC
(dma) **XOR** lk → dma
Affects TC.

Description If a long immediate constant is specified, it is XORed with the addressed data memory value. If it is not specified, the addressed data memory value is XORed with the contents of the dynamic bit manipulation register (DBMR). In either case, the result is written back into the specified data memory location, and the accumulator contents are not disturbed. If the result of the XOR operation is 0, then the TC bit is set to 1. Otherwise, the TC bit is set to 0.

Words 1 (Long immediate value not specified)
2 (Long immediate value specified)

Cycles 1 (Long immediate value not specified)
2 (Long immediate value specified)

Syntax

Direct: [*label*] **ZALR** *dma*
Indirect: [*label*] **ZALR** {*ind*} [,*next ARP*]

Operands

$0 \le dma \le 127$
$0 \le$ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	0	0	0	0	Data Memory Address						
Indirect:	0	1	1	0	1	0	0	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
8000h → ACC(15–0)
(dma) → ACC(31–16)

Description

In order to load a data memory value into the high-order half of the accumulator, the ZALR instruction rounds the value by adding 1/2 LSB; that is, the 15 low bits (bits 0–14) of the accumulator are set to zero, and bit 15 of the accumulator is set to one.

Words

1

Cycles

1

Example 1

```
ZALR   DAT3   ;(DP = 32)
```

	Before Instruction		After Instruction	
Data Memory 1003h	3F01h		Data Memory 1003h	3F01h	
ACC	77FFFFh	X C	ACC	3F018000h	X C

Example 2

```
ZALR   *-,AR4
```

	Before Instruction		After Instruction	
ARP	7		ARP	4	
AR7	0FF00h		AR7	0FEFFh	
Data Memory 0FF00h	0E0E0h		Data Memory 0FF00h	0E0E0h	
ACC	107777h	X C	ACC	0E0E08000h	X C

Syntax [*label*] **ZPR**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	1	1	0	0	0

Execution (PC) + 1 → PC
0 → PREG

Description The product register is set to zero.

Words 1

Cycles 1

Example `ZPR`

	Before Instruction		After Instruction
PREG	3F011111h	PREG	00000000h

Table 4–5. Mapping Summary (Continued)

Accumulator Memory Reference Instructions (Concluded)	
TMS320C2x Mnemonic	**TMS320C5x Mnemonic**
SUBC	SUBC
SUBH	SUB
SUBK	SUB
SUBS	SUBS
SUBT	SUBT
XOR	XOR
XORK	XOR
ZAC	LACL
ZALH	LACC
ZALR	ZALR
ZALS	LACL
Auxiliary Registers and Data Page Pointer Instructions	
TMS320C2x Mnemonic	**TMS320C5x Mnemonic**
ADRK	ADRK
CMPR	CMPR
LAR	LAR
LARK	LAR
LARP	MAR
LDP	LDP
LDPK	LDP
LRLK	LAR
MAR	MAR
SAR	SAR
SBRK	SBRK

Table 4–5. Mapping Summary (Continued)

Branch/Call Instructions (Concluded)	
TMS320C2x Mnemonic	**TMS320C5x Mnemonic**
BGZ	BCND
BIOZ	BCND
BLEZ	BCND
BLZ	BCND
BNC	BCND
BNV	BCND
BNZ	BCND
BV	BCND
BZ	BCND
CALA	CALA
CALL	CALL
RET	RET
TRAP	TRAP
I/O and Data Memory Operations	
TMS320C2x Mnemonic	**TMS320C5x Mnemonic**
BLKD	BLDD
BLKP	BLPD
DMOV	DMOV
FORT†	OPL APL
IN	IN
OUT	OUT
RFSM†	APL
RTXM†	APL
RXF	CLRC
SFSM†	OPL
STXM	OPL
SXF	SETC
TBLR	TBLR
TBLW	TBLW

† The suggested mapping requires that the data page pointer be set to **0**.

4.5 Instruction Set Opcode Table

This section contains a table that summarizes the opcodes of the instruction set for the TMS320C5x digital signal processors. This instruction set is a superset of the TMS320C1x and TMS320C2x instruction sets. The instructions are arranged according to function and are alphabetized within each category.

The following symbols are used in the opcode table:

Symbol	Meaning
A	Data memory address bit.
A R X	Three-bit field containing the auxiliary register value (0 – 7).
B I T X	Four-bit field specifies which bit to test for the BIT instruction.
CM	See CMPR instruction.
I	Addressing mode bit. 0 = direct addressing mode 1 = indirect addressing mode
I I I I I I I I	Short Immediate value.
I N T R #	Interrupt vector number.
PM	Constant copied into PM bits in status register ST1. See SPM instruction.
SHF	Three-bit shift value.
S H F T	Four-bit shift value.
N	Field for the XC instruction indicating the number of instructions (one or two) to conditionally execute.
T P	Two bits used by the conditional execution instructions to represent the conditions TC, NTC, and BIO.
Z L V C	Four-bit field representing the following conditions: Z: ACC = 0 L: ACC < 0 V: Overflow C: Carry A conditional instruction contains two of these four-bit fields. The four-LSB field of the instruction is a four-bit mask field. A one in the corresponding mask bit indicates that condition is being tested. The second four-bit field (bits 4 – 7) indicates the state of the conditions designated by the mask bits as being tested. For example, to test for ACC ≥ 0, the Z and L fields will be set, while the V and C fields are not set. The next four-bit field contains the state of the conditions to test. The Z field will be appropriately set to indicate to test the condition ACC = 0, and the L field will be reset to indicate to test the condition ACC greater than 0. The conditions that can be formed from these 8 bits are shown in the BCND, CC, and XC instruction set pages. In order to determine if the conditions are met, the four LSB bit mask is ANDed with the conditions. If any bits are set, the conditions are met.
+ 1 word	Indicates the instruction is a two-word instruction. The second word is a 16-bit long immediate value or a 16-bit program memory address for immediate addressing.

Table 4–6. Opcode Summary (Continued)

Accumulator Memory Reference Instructions (Concluded)			
Instruction	**Mnemonic**	**Opcode**	
Subtract from accumulator with shift	SUB	0011 SHFT IAAA AAAA	
Subtract from accumulator with shift of 16	SUB	0110 0101 IAAA AAAA	
Subtract from ACC short immediate	SUB	1011 1010 IIII IIII	
Subtract from ACC long immediate with shift	SUB	1011 1111 1010 SHFT	+ 1 word
Subtract from accumulator with borrow	SUBB	0110 0100 IAAA AAAA	
Conditional subtract	SUBC	0000 1010 IAAA AAAA	
Subtract from ACC with sign suppressed	SUBS	0110 0110 IAAA AAAA	
Subtract from ACC, shift specified by TREG1	SUBT	0110 0111 IAAA AAAA	
XOR accumulator with data value	XOR	0110 1100 IAAA AAAA	
XOR with ACC long immediate with shift	XOR	1011 1111 1101 SHFT	+ 1 word
XOR with ACC long immediate with shift of 16	XOR	1011 1110 1000 0011	+ 1 word
XOR ACCB with accumulator	XORB	1011 1110 0001 1010	
Zero ACC, load high ACC with rounding	ZALR	0110 1000 IAAA AAAA	
Zero accumulator and product register	ZAP	1011 1110 0101 1001	
Auxiliary Registers and Data Page Pointer Instructions			
Instruction	**Mnemonic**	**Opcode**	
Add to AR short immediate	ADRK	0111 1000 IIII IIII	
Compare AR with CMPR	CMPR	1011 1111 0100 01CM	
Load AR from addressed data	LAR	0000 0ARX IAAA AAAA	
Load AR short immediate	LAR	1011 0ARX IIII IIII	
Load AR long immediate	LAR	1011 1111 0000 1ARX	+ 1 word
Load data page pointer with addressed data	LDP	0000 1101 IAAA AAAA	
Load data page immediate	LDP	1011 110I IIII IIII	
Modify auxiliary register	MAR	1000 1011 IAAA AAAA	
Store AR to addressed data	SAR	1000 0ARX IAAA AAAA	
Subtract from AR short immediate	SBRK	0111 1100 IIII IIII	
Parallel Logic Unit Instructions			
Instruction	**Mnemonic**	**Opcode**	
AND DBMR with data value	APL	0101 1010 IAAA AAAA	
AND long immediate with data value	APL	0101 1110 IAAA AAAA	+ 1 word
Compare DBMR to data value	CPL	0101 1011 IAAA AAAA	
Compare data with long immediate	CPL	0101 1111 IAAA AAAA	+ 1 word
OR DBMR to data value	OPL	0101 1001 IAAA AAAA	
OR long immediate with data value	OPL	0101 1101 IAAA AAAA	+ 1 word
Store long immediate to data	SPLK	1010 1110 IAAA AAAA	+ 1 word
XOR DBMR to data value	XPL	0101 1000 IAAA AAAA	
XOR long immediate with data value	XPL	0101 1100 IAAA AAAA	+ 1 word

Table 4–6. Opcode Summary (Continued)

Branch Instructions			
Instruction	**Mnemonic**	**Opcode**	
Branch unconditional with AR update	B	0111 1001 1AAA AAAA	+ 1 word
Branch unconditional with AR update delayed	BD	0111 1101 1AAA AAAA	+ 1 word
Branch addressed by ACC	BACC	1011 1110 0010 0000	
Branch addressed by ACC delayed	BACCD	1011 1110 0010 0001	
Branch AR = 0 with AR update	BANZ	0111 1011 1AAA AAAA	+ 1 word
Branch AR = 0 with AR update delayed	BANZD	0111 1111 1AAA AAAA	+ 1 word
Branch conditional	BCND	1110 00TP ZLVC ZLVC	+ 1 word
Branch conditional delayed	BCNDD	1111 00TP ZLVC ZLVC	+ 1 word
Call subroutine addressed by ACC	CALA	1011 1110 0011 0000	
Call subroutine addressed by ACC delayed	CALAD	1011 1110 0011 1101	
Call unconditional with AR update	CALL	0111 1010 1AAA AAAA	+ 1 word
Call unconditional with AR update delayed	CALLD	0111 1110 1AAA AAAA	+ 1 word
Call conditional	CC	1110 10TP ZLVC ZLVC	+ 1 word
Call conditional delayed	CCD	1111 10TP ZLVC ZLVC	+ 1 word
Software interrupt	INTR	1011 1110 011 I NTR#	
Nonmaskable interrupt	NMI	1011 1110 0101 0010	
Return	RET	1110 1111 0000 0000	
Return conditional	RETC	1110 11TP ZLVC ZLVC	
Return conditionally, delayed	RETCD	1111 11TP ZLVC ZLVC	
Return, delayed	RETD	1111 1111 0000 0000	
Return from interrupt with enable	RETE	1011 1110 0011 1010	
Return from interrupt	RETI	1011 1110 0011 1000	
Trap	TRAP	1011 1110 0101 0001	
Execute next one or two INST on condition	XC	111N 01TP ZLVC ZLVC	
I/O and Data Memory Operations			
Instruction	**Mnemonic**	**Opcode**	
Block move from data to data memory	BLDD	1010 1000 IAAA AAAA	+ 1 word
Block move data to data DEST long immediate	BLDD	1010 1001 IAAA AAAA	+ 1 word
Block move data to data with source in BMAR	BLDD	1010 1100 IAAA AAAA	
Block move data to data with DEST in BMAR	BLDD	1010 1101 IAAA AAAA	
Block move data to PROG with DEST in BMAR	BLDP	0101 0111 IAAA AAAA	
Block move from program to data memory	BLPD	1010 0101 IAAA AAAA	+ 1 word
Block move Prog to data with source in BMAR	BLPD	1010 0100 IAAA AAAA	
Data move in data memory	DMOV	0111 0111 IAAA AAAA	
Input external access	IN	1010 1111 IAAA AAAA	+ 1 word
Load memory mapped register	LMMR	1000 1001 IAAA AAAA	+ 1 word
Out external access	OUT	0000 1100 IAAA AAAA	+ 1 word
Store memory mapped register	SMMR	0000 1001 IAAA AAAA	+ 1 word
Table read	TBLR	1010 0110 IAAA AAAA	
Table write	TBLW	1010 0111 IAAA AAAA	

5 Peripherals

5.1 Peripheral Control

Peripheral circuits are operated and controlled through access of memory-mapped control and data registers. The operation of the serial ports and timer is synchronized to the processor via interrupts or through interrupt polling. Setting and clearing bits can enable, disable, initialize, and dynamically reconfigure the peripherals. Data is transferred to and from the peripherals through memory-mapped data registers. When a peripheral is not in use, the internal clocks are shut off from that peripheral, allowing for lower power consumption when the device is in normal run mode or idle mode.

5.1.1 Memory-Mapped Registers and I/O Ports

Twenty-eight core processor registers are mapped into the data memory space. These are listed in subsection 3.4.1 of this user's guide. In addition to these core registers, 15 peripheral registers and 16 I/O ports are mapped into the data memory space. Table 5–1 lists the memory-mapped registers and I/O ports of the TMS320C5x. Note that all writes to memory-mapped peripheral registers require one additional machine cycle.

Table 5–1. Memory-Mapped Registers and I/O Ports

Memory-Mapped Core Processor Registers			
Name	**Address**		**Description**
	Dec	**Hex**	
—	0–3	0–3	Reserved
IMR	4	4	Interrupt Mask Register
GREG	5	5	Global Memory Allocation Register
IFR	6	6	Interrupt Flag Register
PMST	7	7	Processor Mode Status Register
RPTC	8	8	Repeat Counter Register
BRCR	9	9	Block Repeat Counter Register
PASR	10	A	Block Repeat Program Address Start Register
PAER	11	B	Block Repeat Program Address End Register
TREG0	12	C	Temporary Register Used for Multiplicand
TREG1	13	D	Temporary Register Used for Dynamic Shift Count (5 bits only)
TREG2	14	E	Temporary Register Used as Bit Pointer in Dynamic Bit Test (4 bits only)
DBMR	15	F	Dynamic Bit Manipulation Register
AR0	16	10	Auxiliary Register Zero
AR1	17	11	Auxiliary Register One
AR2	18	12	Auxiliary Register Two

Table 5–1. Memory-Mapped Registers and I/O Ports (Concluded)

Name	Address		Description
	Dec	Hex	
Memory-Mapped I/O Ports			
—	54–79	36–4F	Reserved
PA0	80	50	I/O Port 50h
PA1	81	51	I/O Port 51h
PA2	82	52	I/O Port 52h
PA3	83	53	I/O Port 53h
PA4	84	54	I/O Port 54h
PA5	85	55	I/O Port 55h
PA6	86	56	I/O Port 56h
PA7	87	57	I/O Port 57h
PA8	88	58	I/O Port 58h
PA9	89	59	I/O Port 59h
PA10	90	5A	I/O Port 5Ah
PA11	91	5B	I/O Port 5Bh
PA12	92	5C	I/O Port 5Ch
PA13	93	5D	I/O Port 5Dh
PA14	94	5E	I/O Port 5Eh
PA15	95	5F	I/O Port 5Fh

5.1.2 Interrupts

The TMS320C5x devices have four external, maskable user interrupts ($\overline{INT4}$–$\overline{INT1}$) that external devices can use to interrupt the processor; there is one nonmaskable interrupt ($\overline{NMI}$). Internal interrupts are generated by the serial port (RINT and XINT), by the timer (TINT), by the TDM port (TRNT and TXNT), and by the software interrupt instructions (TRAP, NMI, and INTR). Interrupt priorities are set so that reset ($\overline{RS}$) has the highest priority and the TDM port transmit interrupt (TXNT) has the lowest priority. The $\overline{NMI}$ effectively has the same priority as $\overline{RS}$.

This subsection explains interrupt organization and management. Vector relative locations and priorities for all internal and external interrupts are shown in Table 5–2. No priority is set for the TRAP instruction (used for software interrupts), but it is included here because it has its own vector location. Each interrupt address has been spaced apart by two locations so that branch instructions can be accommodated in those locations.

The interrupt vectors reside at locations determined by the five-bit IPTR field of the PMST and the address values shown in Table 5–2. The IPTR field is set

15–9	8	7	6	5	4	3	2	1	0
RESERVED	$\overline{INT4}$	TXNT	TRNT	XINT	RINT	TINT	$\overline{INT3}$	$\overline{INT2}$	$\overline{INT1}$

Note that the TMS320C50 and TMS320C51 make use of only ten of the sixteen generic interrupt lines to the core CPU shown in Section 3.8.

A one in a specific bit, when read, indicates an active interrupt. For example, if the IFR is read to be 0005h, then $\overline{INT3}$ and $\overline{INT1}$ are active. A one can be written to a specific bit to clear the corresponding interrupt. In the example, if a one is written to bit zero (0001h to IFR), then the $\overline{INT1}$ interrupt would be cleared. In the above example, the value 0005h could be written back into the IFR to clear both pending interrupts.

A corresponding interrupt flag is automatically cleared when the interrupt trap is taken. When the CPU accepts the interrupt and fetches the instruction at the interrupt vector location, it generates an interrupt acknowledge ($\overline{IACK}$) signal that clears the appropriate interrupt flag bit. A hardware reset ($\overline{RS}$ active low) clears all pending interrupt flags.

The TMS320C5x devices have a memory-mapped interrupt mask register (IMR) for masking external and internal interrupts. The layout of the register is as follows:

15–9	8	7	6	5	4	3	2	1	0
RESERVED	$\overline{INT4}$	TXNT	TRNT	XINT	RINT	TINT	$\overline{INT3}$	$\overline{INT2}$	$\overline{INT1}$

A 1 in bit positions 8 through 0 of the IMR enables the corresponding interrupt, provided that INTM = 0. The IMR is accessible with both read and write operations. Note that $\overline{RS}$ and $\overline{NMI}$ are not included in the IMR; the IMR has no effect on reset or a nonmaskable interrupt.

Interrupts may be asynchronously triggered. In the functional logic organization for $\overline{INT4}$–$\overline{INT1}$, shown in Figure 5–1, the external interrupt $\overline{INTn}$ is synchronized to the core via a five flip-flop synchronizer. The actual implementation of the interrupt circuits is similar to this logic implementation. A one is loaded into the IFR if a 1-1-0-0-0 sequence on five consecutive CLKOUT1 cycles is detected.

The TMS320C5x devices sample the external interrupt pins multiple times to avoid noise-generated interrupts. To detect an active interrupt, these devices must sample the signal low on at least three consecutive machine cycles. Once an interrupt is detected, the devices must sample the signal high on at least two consecutive machine cycles to be able to detect another interrupt. The external interrupt pins are sampled on the rising edge of CLKOUT1. If the external interrupts are running asynchronously, the pulses should be stretched to guarantee three consecutive low samples.

1) The two software wait-state registers are set to 0FFFFh, causing all external accesses to occur with 7 wait states. The CWSR is loaded with 0Fh.

2) The FO bits of the SPC and TSPC registers are set to zero, selecting a word length of 16 bits for each serial port.

3) The FSM bits of the SPC and TSPC registers are set to zero. FSM must be set to one for operation with frame sync pulses.

4) The TXM bits of the SPC and TSPC are set to zero, configuring the FSX and TFSX pins as inputs.

5) The SPC and TSPC registers are loaded with 0y00h, where the 2 MSBs of y are 10 (binary) and the 2 LSBs of y reflect the current levels on the transmit and receive clock pins of the respective port.

6) The TIM and PRD registers are loaded with 0FFFFh. The TDDR field of the TCR is set to zero. The timer is started.

5.3 Software-Programmable Wait-State Generators

Software-programmable wait-state generators can be used to extend external bus cycles by up to 7 machine cycles. This provides a convenient means for interfacing external devices that do not satisfy the full-speed access-time requirements of the TMS320C5x. Devices requiring more than 7 wait states can be interfaced with the hardware READY line. When all external accesses are configured for zero wait states, the internal clocks to the wait-state generator are shut off, allowing the device to run in a lower power mode of operation.

The software-programmable wait-state generators are controlled by two 16-bit wait-state registers (PDWSR and IOWSR) and a 5-bit control register (CWSR). Each of the three external spaces (program, data, and I/O spaces) has an assigned field in a software wait-state register. Wait states for the program and data spaces are specified in the lower and upper halves of PDWSR, respectively. Wait states for I/O space are specified in IOWSR. The bits of CWSR control the mapping between wait-state register contents and the number of wait states.

The program and data spaces each consist of 64K addresses. Each 64K space can be viewed as being composed of four 16K-word blocks. Each 16K address segment in program and data space is associated with 2 bits in PDWSR, as shown in Table 5–3. The value of a 2-bit field in PDWSR specifies the number of wait states to be inserted for each access in the given space and address range.

Table 5–4. Table 5–5 shows the layout of the CWSR register in PDWSR and IOWSR registers. You should always program the CWSR register prior to configuring the PDWSR and IOWSR registers to avoid configuring memory with too few wait states during the set-up of wait-state registers.

Table 5–4. Mapping Between Wait-State Field Values and # of Wait States as a Function of CWSR Bit n

Wait-State Field of PDWSR or IOWSR (Binary Value)	No. of Wait States (CWSR Bit n = 0)	No. of Wait States (CWSR Bit n = 1)
00	0	0
01	1	1
10	2	3
11	3	7

Table 5–5. Space Controlled by CSWR Bit n

n (Bit Position in CWSR)	Space
0	Program
1	Data
2	I/O (lower-half: PORT0–PORT7 if BIG=0, 0000h–7FFFh if BIG=1)
3	I/O (upper-half: PORT8–PORTF if BIG=0, 8000h–0FFFFh if BIG=1)
4	BIG mode bit

Figure 5–3 shows a block diagram of the wait-state generator logic for external program space. When an external program access is decoded, the appropriate field of the PDWSR wait-state register is loaded into the counter. If the field is not 000, a not-ready signal is sent to the CPU. The not-ready condition is maintained until the counter decrements to zero and the external READY line is high. The external READY and the wait state register READY are OR'd together to generate the CPU $\overline{\text{WAIT}}$ signal.

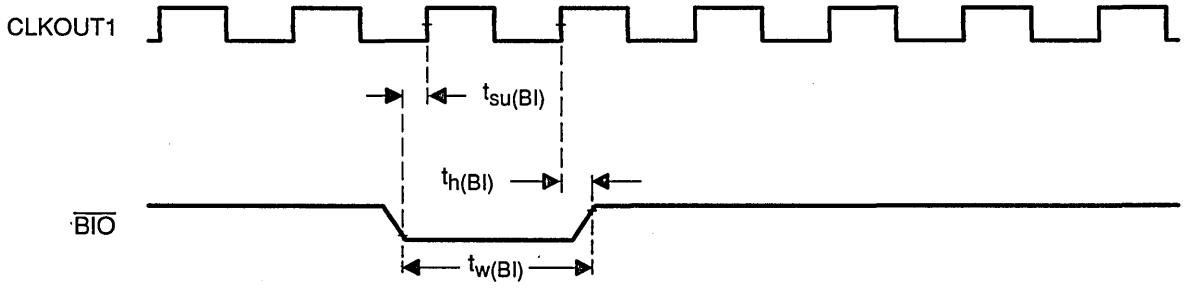
Upon reset, all the software wait-state control register fields are set to 7. CWSR is set to 0Fh. Device reset also sets the BIG bit of the CWSR register to zero.

5.4 General-Purpose I/O Pins

The TMS320C5x devices have two general-purpose pins that are software controlled. The $\overline{BIO}$ pin is a branch control input pin, and the XF pin is an external flag output pin.

The $\overline{BIO}$ pin is useful for monitoring peripheral device status—especially as an alternative to an interrupt when time-critical loops must not be disturbed. A branch can be conditionally executed when the $\overline{BIO}$ input is active (low). The timing diagram, shown in Figure 5–4, is an example of the $\overline{BIO}$ operation. This timing diagram is for a sequence of single-cycle, signal-word instructions located in external memory. The $\overline{BIO}$ condition is sampled during the decode phase of the pipeline for the XC instruction. All other instructions sample the $\overline{BIO}$ pin during the execute phase of the pipeline.

Figure 5–4. $\overline{BIO}$ Timing Diagram



XF (external flag) is useful for signalling to external devices via software. The XF output pin is set to a high level by the SETC XF (set external flag) instruction and reset to a low level by the CLRC XF (reset external flag) instruction. XF is set high upon device reset. The relationship between the time SETC/CLRC instruction is fetched and the time the XF pin is set or reset is shown in Figure 5–5. As with $\overline{BIO}$, the timing shown for XF is for a sequence of single-cycle, single-word instructions located in external memory. Actual timing may vary with different instruction sequences.

Figure 5–5. External Flag Timing Diagram

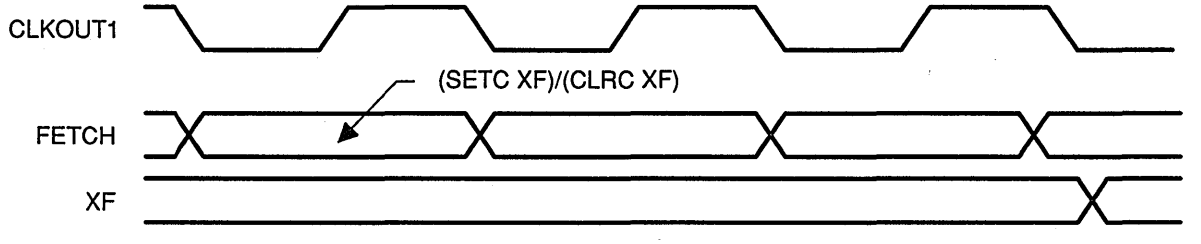


Table 5–6. Serial Port Bits, Pins, and Registers (Continued)

Name	Description
Registers	
DXR	Data Transmit Register
DRR	Data Receive Register
XSR	Transmit Shift Register
RSR	Receive Shift Register
SPC	Serial Port Control Register

The serial port uses two memory-mapped registers: the data transmit register (DXR) that holds the data to be transmitted by the serial port, and the data receive register (DRR) that holds the received data. Both registers operate in either the 8-bit byte mode or the 16-bit word mode and may be accessed in the same manner as any other memory-mapped data memory location. Each register has an external clock, a framing synchronization signal, and an associated shift register. Any instruction accessing data memory or memory-mapped registers can be used to read from or write to the DXR and DRR. The DXR and DRR registers are mapped into data address space. The XSR and RSR registers are not directly accessible through software.

If the serial port is not being used, the DXR and DRR registers can be used as general-purpose registers. In this case, FSR should be connected to a logic low to prevent a possible receive operation from being initiated.

The control bits (DLB, FO, TXM, FSM, MCM, XRST, RRST) for the serial port reside in the serial control register (SPC). Figure 5–6 shows the serial control register bit positions. These bits can be set, cleared, toggled, or loaded via the PLU instructions.

Figure 5–6. Serial Port Control Register (SPC)

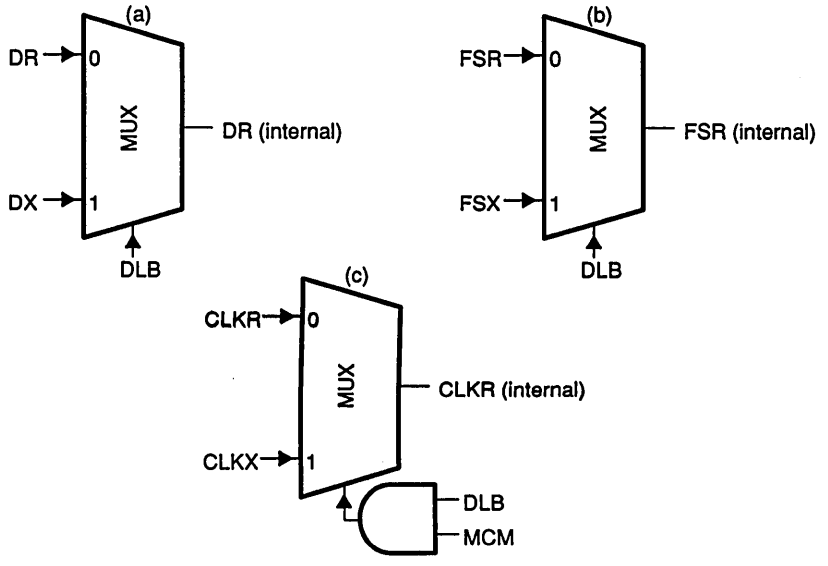
15–14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES	RSRFULL	XSREMPTY	XRDY	RRDY	IN1	IN0	RRST	XRST	TXM	MCM	FSM	FO	DLB	RES
R	R	R	R	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R

Note: R = Read
W = Write

Table 5–7. Serial Port Control Register Bits Summary

Bit	Name	Function
0	Reserved	Always read as zero.
1	DLB	The Digital Loopback Mode Bit can be used to put the serial port in digital loopback mode. When DLB=1, DR and FSR are connected to DX and FSX, respectively, through multiplexers, as shown in Figure 5–7(a) and Figure 5–7(b). Additionally, CLKR is driven by CLKX if MCM=1. If DLB=1 and MCM=0, CLKR is taken from the CLKR pin of the device. This configuration allows CLKX and CLKR to be tied together externally and supplied by a common external clock source. The logic diagram for CLKR is shown in Figure 5–7(c). If DLB=0, DR, FSR, and CLKR are taken from the respective device pins. Note that TXM must be set to one for proper operation in DLB mode. Note also that the FSX and DX signals appear on the device pins when DLB=1, but FSR and DR do not.

Figure 5–7. Receiver Signal MUXes



The value of the SPC, upon device reset, is 0y00h where the 2 MSBs of y are 10 (binary) and the two LSBs of y reflect the current levels on the CLKX and CLKR pins.

5.5.1 Transmit and Receive Operations

The transmit and receive sections of the serial port are implemented separately to allow independent transmit and receive operations. Externally, the serial port interface is implemented via the six serial port pins. Figure 5–8 shows the registers and pins used in transmit and receive operations.

Figure 5–9. Serial Port Transmit Timing Diagram (FSM=1, first byte = 62h)

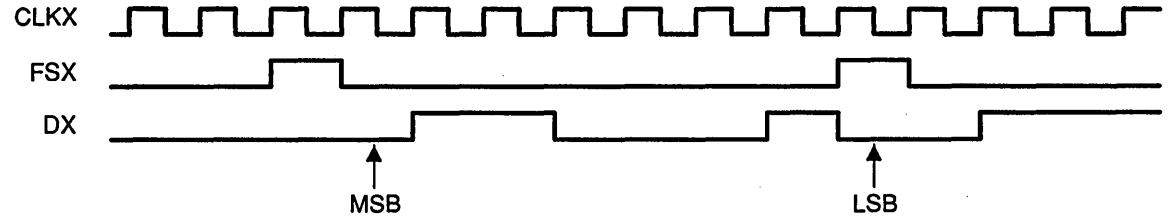
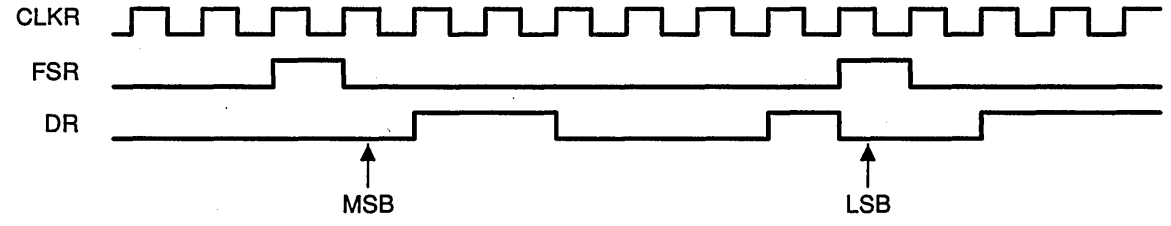


Figure 5–10. Serial Port Receive Timing Diagram (FSM=1, first byte=62h)



If DXR is reloaded before the old DXR contents have been transferred to XSR, the old DXR contents will be overwritten. The DXR is copied to the XSR only if the XSR is empty and the DXR has been loaded since the last DXR-to-XSR transfer. The DXR should be written only when XRDY=1. This condition is guaranteed if the DXR write is made in response to a transmit interrupt.

If TXM=1 and FSM=1, FSX pulses are generated internally and the FSX pin is configured as an output. To sustain a continuous bit stream on the DX transmitter output, DXR must be loaded every 8 or 16 bits, depending on the value of FO. Furthermore, the next word to be transmitted must be loaded in DXR at least 2 CLKX cycles prior to completion of transmission of the current word. If this condition is not satisfied, the transmitter will send the previous data from the register.

If TXM=0, the FSX pin is configured as an input. The transmitter behaves in the same way as when TXM=1, except that FSX pulses are supplied externally. A consequence of this is that the timing requirement on loading DXR for continuous-mode transmission is relaxed, because the processor does not impose a latency between DXR write and FSX active in this case.

The transmitter's operation with frame synchronization pulses has been described above. Both continuous operation and burst-mode operation (operation with periods of transmitter inactivity) are possible when FSM=1. When FSM=0, only continuous-mode transmission is possible. Timing diagrams for transmit and receive operations in this mode are shown in Figure 5–11 and Figure 5–12.

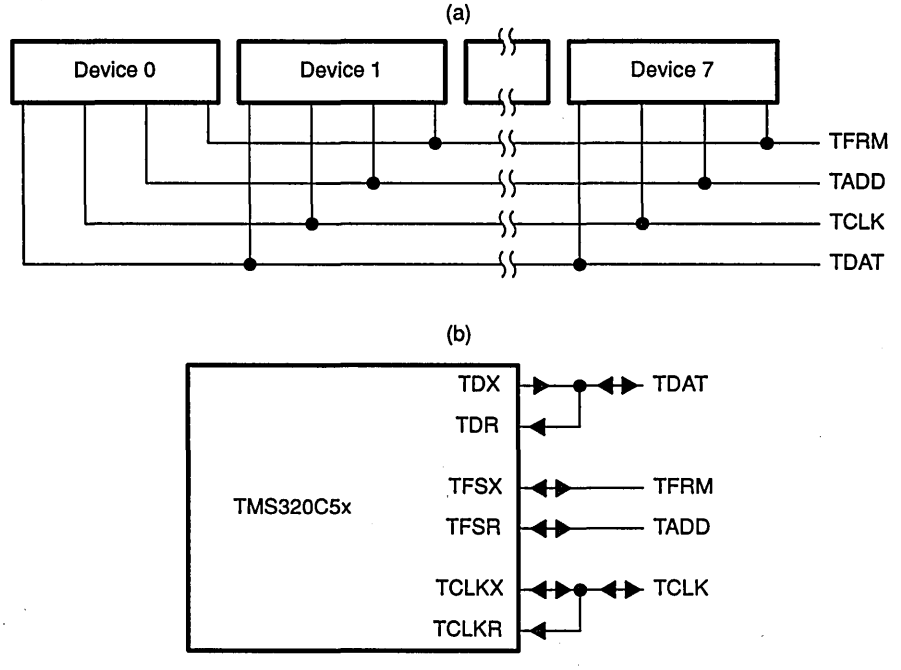
contents of DXR when FSX goes high. If TXM=1 and DXR are written more than once during transmission of a given word, only the last word written to DXR will be transmitted; any previous values will be overwritten. Therefore, too many writes to the DXR during a given interval will not disturb the XSR contents, but an external FSX pulse will.

The receive operation is similar to the transmit operation. The receive timing diagram with FSM=1 is shown in Figure 5–10. Reception is initiated by a frame synchronization pulse on the FSR pin. After FSR goes low, data on the DR pin is clocked into the RSR register on every negative-going edge of CLKR. The first data bit is considered the MSB, and RSR is filled accordingly. After all the bits have been received (as specified by FO), an internal receive interrupt (RINT) is generated on the falling edge of CLKR, while the contents of RSR are transferred to DRR. If, during a receive operation, a new FSR pulse comes in, the bit counter is reset and the RSR starts over. The bits already received are lost.

5.6.2 TDM Port Operation

Figure 5–14(a) shows the TMS320C5x TDM port architecture. Up to eight devices can be placed on the four-wire serial bus. The four-wire bus consists of a conventional three-wire bus (TDAT, TFRM, and TCLK) and an additional line (TADD) to carry device-addressing information. Data is transmitted and received on the bidirectional TDAT line. Note that the device TDX and TDR pins are tied together externally to form the TDAT line. A framing pulse is supplied by one of the devices on the bus on the TFRM line.

Figure 5–14. TDM Four-Wire Bus

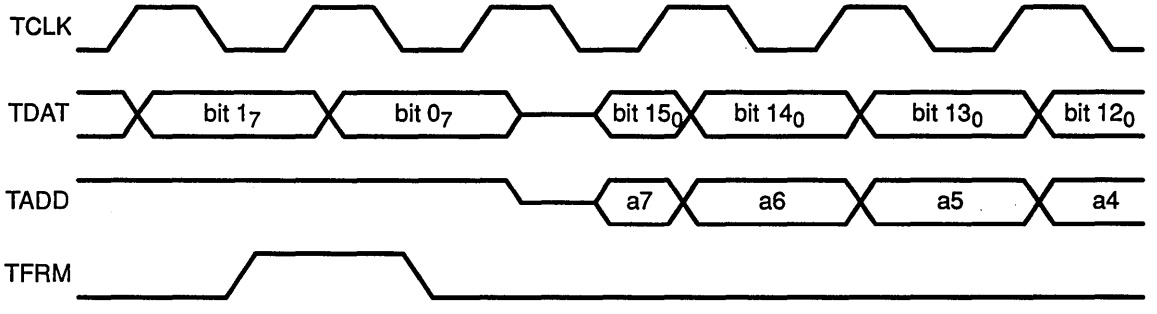


This device is identified by setting the TXM bit of its TSPC register to one. Only one device should have TXM=1 at any given time. Typically, this processor is the same one that supplies the TDM port clock signal on TCLK. The TCLKX and TCLKR pins are tied together externally to form the TCLK line. TCLKR is always an input. TCLKX is an input if MCM=0 and an output if MCM=1. In the latter case, one device (the one whose MCM bit=1) can supply the clock (frequency=one-fourth of CLKOUT1 frequency) for all devices on the bus. The clock can be supplied by an external source if MCM=0 for all devices. No more than one device should have MCM=1 at any given time. The specification of which processor is to supply clock and framing signals is typically made only once, during system initialization. The TADD line carries the transmit address byte sent by the transmitting device. Figure 5–14(b) shows how the four-wire bus is formed from the six serial port pins.

The TDM received address (TRAD) register holds the last value received from the TADD line. This register can be used to verify the integrity of the serial interface and/or to extract partial or complete information as to which device in the system transmitted the last data word. For example, if there is a unique transmit address for each channel, the transmitter can be uniquely identified. Bits 0–7 hold the received transmit address. Bits 10–8 hold the last time slot number (i.e., channel ID number). Bits 13–11 hold the current time slot number. This number is simply the last slot number plus one, modulo 8.

Figure 5–16 shows the timing for TDM port transfers. Near the end of a frame (8 time slots), the single device having TXM=1 outputs a pulse one TCLK cycle wide on the TFRM line. TFRM pulses occur only once every 128 TCLK cycles. TFRM is driven low during the remainder of the frame.

Figure 5–16. Serial Port Timing in TDM Mode



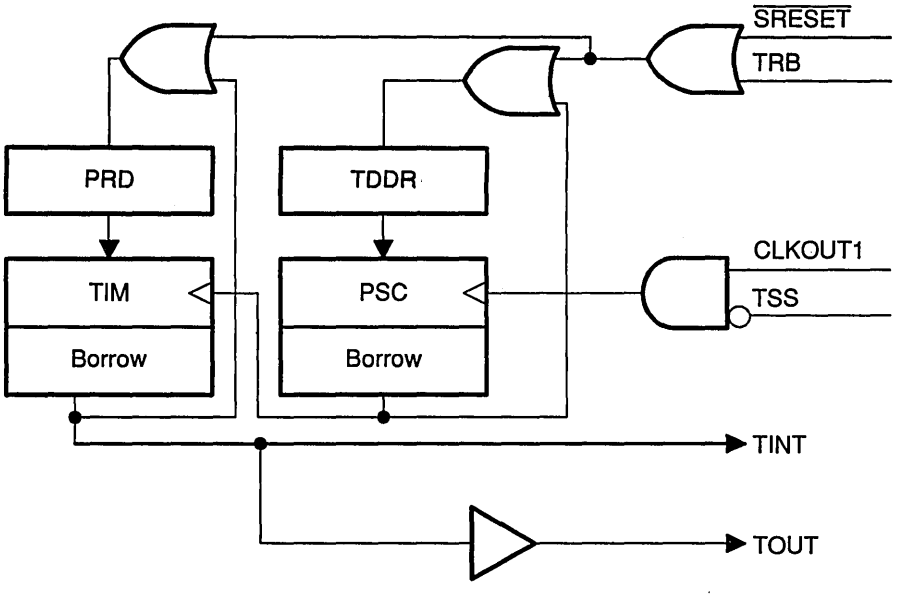
After the LSB of a given data word is transmitted, the TDAT line goes into the high-impedance state. TDAT comes out of high impedance shortly after the next falling edge of TCLK. The next 15 data bits are transmitted on rising edges of TCLK. In Figure 5–16, the data bits are shown with subscripts that indicate the channel (i.e., time slot) number.

The timing for TADD signal for channel 0 is shown in Figure 5–16. After the LSB of the channel 7 data is transmitted, the TADD line goes into the high-impedance state. TDAT comes out of high impedance shortly after the next falling edge of TCLK. The 8 address bits associated with channel 0 are then transmitted on TADD. After these have been transmitted, TADD goes high and remains high until the channel 1 transfer begins. For channels 1–7, TADD carries the address during the first 8 clocks and is high during the remaining 8 clocks. Note that the short interval between completion of transmission of the channel 7 LSB and initiation of transmission of the channel 0 MSB is the only time during which TADD is in the high-impedance state. Note that the address line TADD must be pulled down to V_{SS} if there are any channels available with no processor transmitting data. This is due to the fact that the address line could float high. This indicates that when no one is transmitting, all devices will receive

5.7 Timer

The timer is an on-chip down counter that can be used to periodically generate CPU interrupts. The timer is decremented by one at every CLKOUT1 cycle. A timer interrupt (TINT) is generated each time the counter decrements to zero. The timer thus provides a convenient means of performing periodic I/O or other functions. Figure 5–17 shows a logical block diagram of the timer. When the timer is stopped, (TSS = 1), the internal clocks to the timer are shut off, allowing the device to run in a lower power mode of operation.

Figure 5–17. Timer Block Diagram



The timer interrupt rate is given by

$$\text{TINT rate} = \frac{1}{t_{c(C)} \times u \times v} = \frac{1}{t_{c(C)} \times (<\text{TDDR}> + 1) \times (<\text{PRD}> + 1)}$$

where $t_{c(C)}$ is the period of CLKOUT1, u is the sum of the TDDR contents (see Table 5–10) plus 1, and v is the sum of the PRD contents (see Figure 5–17) plus 1.

Thus, the timer interrupt rate is equal to the CLKOUT1 frequency divided by two independent factors. Referring to Figure 5–17, each of the two divisors is implemented with a down counter and period register. The counter and period registers for the first stage are the PSC and TDDR fields of the TCR, respectively, and each is 4 bits wide. The counter and period registers for the second stage are the memory-mapped, 16-bit wide TIM and PRD registers. Each time

being made, it may be more accurate to stop the timer to read these two values. The timer can be stopped by setting the TSS bit to one and restarted by resetting this bit to zero.

The timer provides a convenient and efficient way to generate a sample clock for an analog interface. Consider the following example of using the timer to generate a sample rate of 50 kHz. The initialization for this example is as follows:

```
*   Clkin frequency = 20 MHz; timer is running at 10 MHz.
*
    LDP     #0
    SPLK    #199,PRD    ;Load timer period for 20 usec period.
    OPL     #8,IMR      ;Set timer interrupt mask bit
    SPLK    #20h,TCR    ;reload and start timer.
    OPL     #1000h,IFR  ;Clear any pending timer interrupts.
    CLRC    INTM        ;global interrupt enable.
*
```

Consider an A/D that is operating at this sample rate. A typical interrupt service routine (ISR) would be as follows:

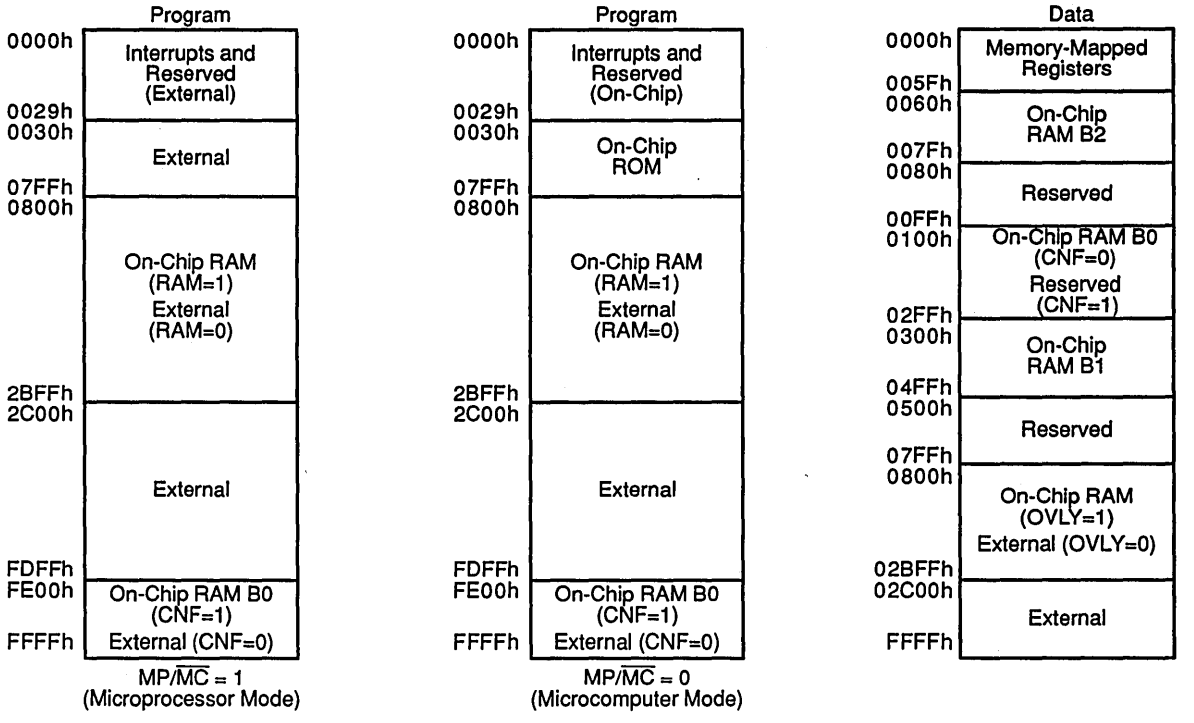
```
*    50 kHz sample rate A/D interrupt service routine
*
TIMER_ISR  MAR  *, AR3 ;Use auxiliary register reserved for Timer ISR.
           IN   *, 14  ;Read A/D.
           RETE        ;Re-enable interrupts and return.
*
```

machine cycle to perform a read or a write. The dual-access RAM can be read from and written to in the same cycle. The 1056 words of dual-access RAM are configured in three blocks: block 0 (B0) is 512 words at address 0100h–02FFh in local data memory or 0FE00h–0FFFFh in program space; block 1 (B1) is 512 words at address 0300h–04FFh in local data memory; and block 2 (B2) is 32 words at address 060h in local data memory.

The TMS320C51 removes the 2K boot ROM from the device's program memory space along with 8K words of single-access program/data RAM. Instead, the device replaces the 8K words of RAM with an 8K-word block of maskable ROM. The ROM is located in the address range 0h–1FFFh in program space. The additional 1K words of single-access RAM are mapped to data space (800h–0BFFh), program space (2000h–23FFh), or both spaces. The dual-access blocks of RAM on the TMS320C51 are mapped at the same addresses as the TMS320C50. The TMS320C50 and TMS320C51 memory maps are shown in Figures 6–1(a) and 6–1(b).

The major topics in this section are listed below:

Figure 6–1. TMS320C51 and TMS320C50 Memory Maps (Concluded)



(b) TMS320C50 Memory Map

```
OPL    #010h,PMST    ;Map TMS320C50 9K RAM or TMS320C51 1K RAM
                     ;in program space.
SETC   CNF           ;Map B0 to program space.
```

Table 6–1 shows the possible program memory configurations available on the TMS320C50 device. Table 6–2 shows the possible program memory configurations for the TMS320C51 device. Note that all addresses are specified in hexadecimal.

Table 6–1. TMS320C50 Program Memory Configuration Control

CNF	RAM	MP/$\overline{MC}$	ROM	RAM	B0	Off-Chip
0	0	0	0000–07FF			0800–FFFF
0	0	1				0000–FFFF
0	1	0	0000–07FF	0800–2BFF		2C00–FFFF
0	1	1		0800–2BFF		0000–07FF
						2C00–FFFF
1	0	0	0000–07FF		FE00–FFFF	0800–FDFF
1	0	1			FE00–FFFF	0000–FDFF
1	1	0	0000–07FF	0800–2BFF	FE00–FFFF	2C00–FDFF
1	1	1		0800–2BFF	FE00–FFFF	0000–07FF
						2C00–FDFF

Table 6–2. TMS320C51 Program Memory Configuration Control

CNF	RAM	MP/$\overline{MC}$	ROM	RAM	B0	Off-chip
0	0	0	0000–1FFF			2000–FFFF
0	0	1				0000–FFFF
0	1	0	0000–1FFF	2000–23FF		2400–FFFF
0	1	1		2000–23FF		0000–1FFF
						2400–FFFF
1	0	0	0000–1FFF		FE00–FFFF	2000–FDFF
1	0	1			FE00–FFFF	0000–FDFF
1	1	0	0000–1FFF	2000–23FF	FE00–FFFF	2400–FDFF
1	1	1		2000–23FF	FE00–FFFF	0000–1FFF
						2400–FDFF

6.1.2 Program Memory Address Map

The reset, interrupt, and trap vectors are addressed in program space. These vectors are soft—meaning that the processor, when taking the trap, will load the PC with the trap address and execute code at the vector location. Two words are reserved at each vector location for a branch instruction to the appro-

The reset vector can not be remapped, because reset loads the IPTR with 0. Therefore, the reset vector will always be fetched at location 0 in program memory. In addition, for the TMS320C51, 100 words are reserved in the on-chip ROM for device-testing purposes. Application code written to be implemented in on-chip ROM must reserve these 100 words at the top of the ROM addresses.

6.1.3 Program Memory Addressing

The program memory space contains the code for applications. It can also hold table information and immediate operands. The program memory is accessed only by the PAB. The address for this bus is generated by the program counter (PC) when instructions and long immediate operands are addressed. It can also be loaded with a long immediate, low accumulator, or registered addresses for block transfers, multiply/accumulates, and table read/writes.

The TMS320C5x devices address code by putting the PC out on the PAB bus and reading the appropriate location in memory. While the read is executing, the PC is incremented for the next fetch. If there is a program address discontinuity (for example, branch, call, return, interrupt, or block repeat), the appropriate address is loaded into the PC. The PC is also loaded when operands are fetched from program memory. Operands are fetched from program memory when the device reads or writes to tables (TBLR and TBLW), when it transfers data to/from data space (BLPD and BLDP), or when it uses the program bus to fetch a second multiplicand (MAC, MACD, MADS, and MADD). The PC is loaded with a value other than PC + 1 in the following ways:

- Long immediate address with branch or call instructions.
- Long immediate address with MAC, MACD, BLDP or BLPD instructions.
- Low accumulator with BACC or CALA instructions.
- Low accumulator with TBLR or TBLW instruction.
- BMAR with MADS, MADD, BLDP or BLPD instructions.
- CALU with an interrupt vector address (INTR, TRAP, or NMI) instruction.
- CALU with PASR when at the end of a block repeat loop.
- Pop top of stack with a return instruction.

The address flow of a program can be traced externally through the address visibility feature. This feature can debug during program development; it is enabled after reset and disabled/re-enabled by setting/clearing the AVIS bit in the PMST register. The address visibility mode puts the program address out to the address pins of the device even when on-chip program memory is addressed. Note that the memory control signals ($\overline{PS}$, $\overline{RD}$, etc.) are not active in address visibility mode. Instruction addresses can be externally clocked with the falling edge of the instruction acquisition ($\overline{IAQ}$) pin. Instruction addresses include both words of a two-word instruction but do not include block transfers,

- $\overline{WE}$ Write Enable
- $\overline{IACK}$ Interrupt Acknowledge
- READY Memory Ready to Complete Cycle
- $\overline{HOLD}$ Request for Control of Memory Interface
- $\overline{HOLDA}$ Acknowledge $\overline{HOLD}$ Request
- $\overline{BR}$ Bus Request
- $\overline{IAQ}$ Acknowledge Bus Request (when $\overline{HOLDA}$ is low)

An example of a minimal external program memory interface is shown in Figure 6–2. In this figure, the TMS320C5x device interfaces to an 8K x 8 EPROM. This is a useful interface when boot-loading code. The boot loader can concatenate the bytes to form the 16-bit word instructions. The use of 8-bit-wide memories saves power, board space, and cost over 16-bit wide memory banks. The 16-bit wide memory banks can be used with the same basic interface as the 8-bit wide memories.

low and a half cycle after $\overline{WE}$ goes high; this prevents buffer conflicts on the external buses. Additional write cycles can be obtained by modifying the software wait-state generator registers. Subsection 6.2.4 includes an example of interfacing to external RAM.

Table 6–4. TMS320C50 Local Data Memory Configuration Control

CNF	OVLY	B0	B1	B2	Single-Port RAM	Off-Chip
0	0	100h	300h	60h		800h–FFFFh
0	1	100h	300h	60h	800h–2BFFh	2C00h–FFFFh
1	0		300h	60h		800h–FFFFh
1	1		300h	60h	800h–2BFFh	2C00h–FFFFh

Table 6–5. TMS320C51 Local Data Memory Configuration Control

CNF	OVLY	B0	B1	B2	Single-Port RAM	Off-Chip
0	0	100h	300h	60h		800h–FFFFh
0	1	100h	300h	60h	800h–BFFh	C00h–FFFFh
1	0		300h	60h		800h–FFFFh
1	1		300h	60h	800h–BFFh	C00h–FFFF

6.2.2 Local Data Memory Address Map

The 64K words of local data memory space include the memory-mapped registers for the device. The memory-mapped registers reside in data page 0. Data page 0 has five sections of register banks: core CPU registers, peripheral registers, test/emulation reserved area, I/O space port hole, and scratch-pad RAM.

- There are 28 core CPU registers. These registers can be accessed with zero wait states. Some of these registers can be accessed through paths other than the data bus (i.e., auxiliary registers can be loaded by the ARAU).
- The peripheral registers are the control and data registers used in the peripheral circuits. These registers reside on a dedicated peripheral bus structure called the TIBUS. They require one wait state when accessed.
- The test/emulation reserved area is used by the test and emulation systems for special information transfers. **Writing to this area can cause the device to change its operational mode and, therefore, affect the operation of the application.**
- The I/O space port hole provides addressability to 16 words of I/O space within the data address space. This allows access to I/O space (other than IN and OUT instructions) via the more extensive addressing modes available within the data space. For example, the SACL instruction can write to an I/O memory-mapped port like an OUT instruction does. The external interface looks like an OUT instruction occurs ($\overline{IS}$ active). Port addresses reside off-chip and are subject to external wait states.
- The scratch-pad RAM block (B2) includes 32 words of dual-access RAM for variable storage without fragmenting the larger RAM blocks, both on

Table 6–6. Data Page 0 Address Map (Concluded)

Name	Address		Description
	Dec	Hex	
TIM	36	24	Timer Register
PRD	37	25	Period Register
TCR	38	26	Timer Control Register
—	39	27	Reserved
PDWSR	40	28	Program/Data S/W Wait-State Register
IOWSR	41	29	I/O Port S/W Wait-State Register
CWSR	42	2A	Control S/W Wait-State Register
—	43–47	2B–2F	Reserved for Test/Emulation
TRCV	48	30	TDM Data Receive Register
TDXR	49	31	TDM Data Transmit Register
TSPC	50	32	TDM Serial Port Control Register
TCSR	51	33	TDM Channel Select Register
TRTA	52	34	Receive/Transmit Address Register
TRAD	53	35	Received Address Register
—	54–79	36–4F	Reserved
PA0	80	50	I/O Port 80
PA1	81	51	I/O Port 81
PA2	82	52	I/O Port 82
PA3	83	53	I/O Port 83
PA4	84	54	I/O Port 84
PA5	85	55	I/O Port 85
PA6	86	56	I/O Port 86
PA7	87	57	I/O Port 87
PA8	88	58	I/O Port 88
PA9	89	59	I/O Port 89
PA10	90	5A	I/O Port 90
PA11	91	5B	I/O Port 91
PA12	92	5C	I/O Port 92
PA13	93	5D	I/O Port 93
PA14	94	5E	I/O Port 94
PA15	95	5F	I/O Port 95
B2	96–127	60–7F	Scratch Pad RAM

6.2.2.1 Auxiliary Register (AR0–AR7)

The eight 16-bit auxiliary registers (AR0–AR7) can be accessed by the CALU and modified by the ARAU or the PLU. The primary function of the auxiliary registers is generating 16-bit addresses to data space. However, these registers

The block repeat counter register (BRCR) holds the count value for the block repeat feature. This value is loaded before a block repeat operation is initiated. It can be changed while a block repeat is in progress; however, take caution in this case to avoid infinite loops. The program address start register (PASR) holds the start address of the block of code to be repeated. The program address end register (PAER) holds the end address of the block of code to be repeated. Both these registers are loaded by the RPTB instruction. Block repeats are described in more detail in subsection 3.6.5.

6.2.2.7 Interrupt Registers (IMR,IFR)

The interrupt mask register (IMR) is used to individually mask off specific interrupts at required times. The interrupt flag register (IFR) indicates the current status of the interrupts. Interrupts are described in detail in Section 3.8.

6.2.2.8 Global Memory Allocation Register (GREG)

The global memory allocation register (GREG) is used to allocate parts of the data address space as global memory. This register defines what amount of the local data space will be overlayed by global data space. The operation of GREG is further discussed in Section 6.3.

6.2.2.9 Dynamic Bit Manipulation Register (DBMR)

The dynamic bit manipulation register (DBMR) is used in conjunction with the PLU to provide a dynamic (execution time programmable) mask register. The use of this register is described in Section 3.7.

6.2.2.10 Temporary Registers (TREG0, TREG1, TREG2)

TREG0 holds one of the multiplicands of the multiplier. It can also be loaded via the CALU with the following instructions: LT, LTA, LTD, LTP, LTS, SQRA, SQRS, MAC, MACD, MADS, and MADD. TREG1 holds a dynamic (execution-time programmable) shift count for the prescaling shifter. TREG2 holds a dynamic bit address for the BITT instruction.

6.2.2.11 Processor Mode Status Register (PMST)

The processor mode status register (PMST) controls memory configurations of the TMS320C5x devices (with exception of the CNF bit in ST1). The PMST register is described in more detail in subsection 3.6.3 and in the configurability sections of Chapter 6.

6.2.2.12 Serial Port Registers (DRR, DXR, SPC)

Three registers are used to control and operate the serial port. The serial port control register (SPC) contains the mode control and status bits of the serial port. The data receive register (DRR) holds the incoming serial data, and the

instructions with only one data memory operand and program address bus (PAB) on instructions with a second data memory operand. An instruction operand is provided to the CALU in eight ways, as described in subsection 3.4.2. However, data memory addresses are generated in one of the following five ways:

1) By the direct address bus (DAB) using the direct addressing mode (e.g., ADD 010h) relative to the data page pointer (DP),
2) By the direct address bus (DAB) using the memory-mapped addressing mode (e.g., LAMM PMST) within data page zero,
3) By the auxiliary register file bus (AFB) using the indirect addressing mode (e.g., ADD *),
4) By the value pointed at by the PC in long immediate address mode (e.g., BLDD TBL1,*+),
5) By the block memory address register (BMAR) in registered block memory addressing mode (e.g., BLDD *+).

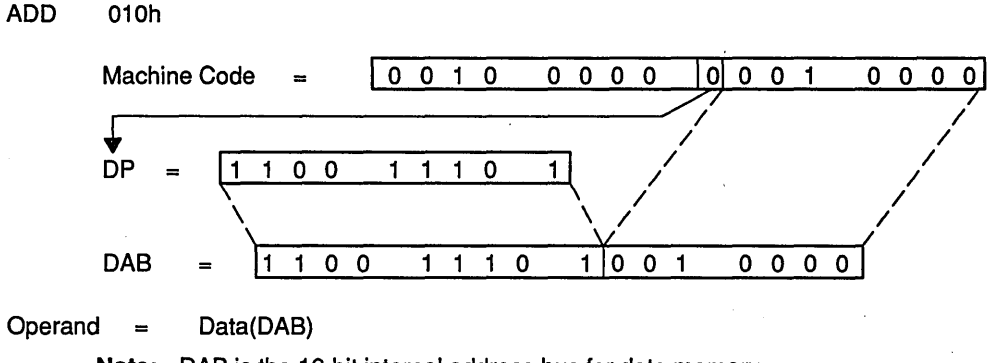
In the direct addressing mode, the 9-bit data memory page pointer (DP) points to one of 512 pages (1 page=128 words). The data memory address (dma), specified by the seven LSBs of the instruction, points to the desired word within the page. The address on the DAB is formed by concatenating the 9-bit DP with the 7-bit dma.

Figure 6–3 illustrates the direct addressing mode. In the illustration, the operand is fetched from data memory space via the data bus, and the address is the concatenated value of the DP and the seven LSBs of the instruction. For the following example, consider DP = 0184h and TEMP1 = 060h:

```
LACC  TEMP1    ;ACC = TEMP1.
```

In the example, the accumulator is loaded with DATA(C260).

Figure 6–3. Direct Addressing Mode



Note: DAB is the 16-bit internal address bus for data memory.

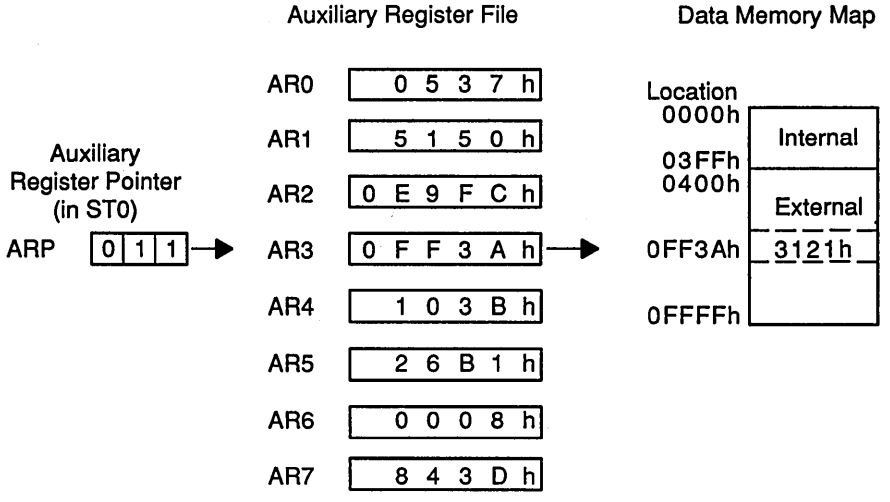
The memory-mapped addressing mode operates much like the direct addressing mode except that the most significant 9 bits of the address are forced to

```
* This routine uses indirect addressing to calculate the following equation:
*
*                  10
*                 -----
*                 \     X(I) x Y(I)
*                 /
*                 -----
*                 I = 1
*
* The routine assumes that the X values are located in on-chip RAM block B0,
* and the Y values in block B1. The efficiency of the routine is due to the use
* of indirect addressing and the repeat instruction.
*
SERIES      MAR  *,4            ;ARP POINTS TO ADDRESS REGISTER 4.
            SETC CNF            ;CONFIGURE BLOCK B0 AS PROGRAM MEMORY.
            LAR  AR4,#0300h     ;POINT AT BEGINNING OF DATA MEMORY.
            RPTZ #9             ;CLEAR ACC AND P; REPEAT NEXT INST. 10 TIMES
            MAC  0FF00h,*+      ;MULTIPLY AND ACCUMULATE; INCREMENT AR4.
            APAC                ;ACCUMULATE LAST PRODUCT.
            RET
```

In the long immediate addressing mode, an operand is addressed by the second word of a two-word instruction. In this case, the program address/data bus (PAB) is used for the operand fetch. The prefetch counter (PFC) is pushed onto the microcall stack (MCS), and the long immediate value is loaded into the PFC. The PAB is then used for the operand fetch or write. At the completion of the instruction, the MCS is popped back to the PFC. The PC is incremented by two, and execution continues. This technique is used when two memory addresses are required for the execution of the instruction. The PFC is used so that when the instruction is repeated, the address generated can be autoincremented. Figure 6–6 illustrates this mode. In this illustration, the source address (OPERAND1) is fetched via PAB, and the destination address (OPERAND2) uses the direct addressing mode.

TMS320C5x devices provide a register file containing eight auxiliary registers (AR0–AR7). The auxiliary registers may be used for indirect addressing of the data memory or for temporary data storage. Indirect auxiliary register addressing (see Figure 6–8) allows placement of the data memory address of an instruction operand into one of the auxiliary registers. These registers are pointed to by a three-bit auxiliary register pointer (ARP) that is loaded with a value from 0 through 7, designating AR0 through AR7, respectively.

Figure 6–8. Indirect Auxiliary Register Addressing Example



The auxiliary registers and the ARP may be loaded from data memory, from the accumulator, from the product register, or by an immediate operand defined in the instruction. The contents of these registers may also be stored in data memory or used as inputs to the CALU. These registers appear in the memory map as described in Table 6–6 on page 6-15.

The auxiliary register file (AR0–AR7) is connected to the auxiliary register arithmetic unit (ARAU), shown in Figure 6–9. The ARAU may autoindex the current auxiliary register while the data memory location is being addressed. Indexing either by ± 1 or by the contents of the INDX register may be performed. As a result, accessing tables of information does not require the central arithmetic logic unit (CALU) for address manipulation. The CALU is now free to perform other operations.

If more advanced address manipulation is required, such as multidimensional array addressing, the CALU can directly read from or write to the auxiliary registers. Take care, however, when writing from the CALU to the auxiliary register because the ARAU update of the ARs is done during the decode phase (second cycle) of the pipeline, whereas the CALU write is done during the execution phase (fourth cycle) of the pipeline. Therefore, the two instructions directly following the CALU write should not use the auxiliary register written by the CALU.

As shown in Figure 6–9, the index register, the compare register, or the eight LSBs of the instruction register can be connected to one of the inputs of the ARAU. The other input is fed by the current AR (being pointed to by ARP). AR(ARP) refers to the contents of the current AR pointed to by ARP. The ARAU performs the following functions:

AR(ARP) + INDX → AR(ARP)	Index the current AR by adding a 16-bit unsigned integer contained in INDX. Example: ADD *0+.
AR(ARP) – INDX → AR(ARP)	Index the current AR by subtracting a 16-bit unsigned integer contained in INDX. Example: ADD *0–.
AR(ARP) + 1 → AR(ARP)	Increment the current AR by one. Example: ADD *+.
AR(ARP) – 1 → AR(ARP)	Decrement the current AR by one. Example: ADD *–.
AR(ARP) → AR(ARP)	Do not modify the current AR. Example: ADD *.
AR(ARP) + IR(7–0) → AR(ARP)	Add an 8-bit immediate value to current AR. Example: ADRK #055h.
AR(ARP) – IR(7–0) → AR(ARP)	Subtract an 8-bit immediate value from current AR. Example: SBRK #055h.
AR(ARP) + rc(INDX) → AR(ARP)	Bit-reversed indexing, add INDX with reverse-carry (rc) propagation. Example: ADD *BR0+.
AR(ARP) – rc(INDX) → AR(ARP)	Bit-reversed indexing, subtract INDX with reverse-carry (rc) propagation. Example: ADD *BR0–.

If (AR(ARP) == ARCR), then TC = 1
If (AR(ARP) < ARCR), then TC = 1
If(AR(ARP) > ARCR), then TC = 1
If(AR(ARP) ≠ ARCR), then TC = 1
	Compare current AR with ARCR and if condition is true, then set TC bit of the status register (ST1) to one. If false, then clear TC. Example: CMPR 3.

If (AR(ARP) = CBER), then AR(ARP) =CBSR	If at end of circular buffer, reload start address.

The index register (INDX) can be added to or subtracted from AR(ARP) on any AR update cycle. This 16-bit register is one of the memory-mapped registers and is used to increment or decrement the address in steps larger than one for

auxiliary register modification occurring. The ARAU will not detect an AR update that steps over the value contained in CBER. Note that the test in the ARAU is performed before the auxiliary register update.

6.2.4 External Interfacing to Local Data Memory

The TMS320C5x devices can address up to 64K words of off-chip local data memory. These are the key signals for this interface:

A0–A15	16-Bit Bidirectional Address Bus
D0–D15	16-Bit Bidirectional Data Bus
$\overline{DS}$	Data Memory Select
$\overline{STRB}$	External Memory Access Active Strobe
$\overline{RD}$	Read Select (External Device Output Enable)
$\overline{WE}$	Write Enable
READY	Memory Ready to Complete Cycle
$\overline{HOLD}$	Request for Control of Memory Interface
$\overline{HOLDA}$	Acknowledge $\overline{HOLD}$ Request
$\overline{BR}$	Bus Request
$\overline{IAQ}$	Acknowledge Bus Request (when $\overline{HOLDA}$ is low)

An example of an external RAM interface is shown in Figure 6–10. In this figure, the TMS320C5x device interfaces to four 16K × 4-bit RAM devices. The data memory select ($\overline{DS}$) is directly connected to the chip select ($\overline{CS}$) of the devices. This means the external RAM block will be addressed in any of the four 16K banks of local data space. If there are additional banks of off-chip data memory, a decode circuit that gates $\overline{DS}$ with the appropriate address bits can be used to drive the memory block chip select.

device. If the RAM device does not have an $\overline{OE}$ pin, then $\overline{DS}$ should be gated with $\overline{STRB}$ and connected to the $\overline{CS}$ pin of the RAM to implement the same function. The $\overline{WE}$ signal of the TMS320C5x is tied to the $\overline{WE}$ signal of the RAM. The TMS320C5x takes at least two cycles on all external writes, including a half cycle before the $\overline{WE}$ goes low and a half cycle after $\overline{WE}$ goes high; this prevents buffer conflicts on the external buses. Additional wait states may be generated with the software wait-state generators.

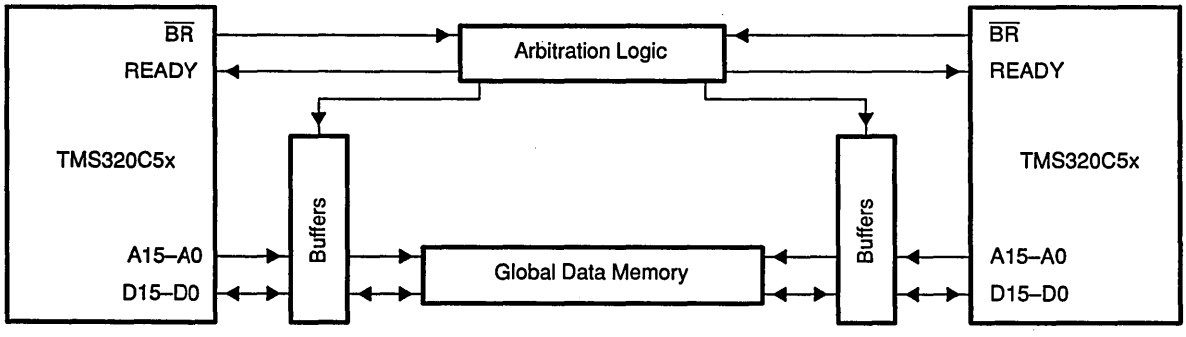
6.3.2 Global Memory Addressing

When a data memory address, either direct or indirect, corresponds to a global data memory address (as defined by GREG), $\overline{BR}$ is asserted low with $\overline{DS}$ to indicate that the processor wishes to make a global memory access. External logic then arbitrates for control of the global memory, asserting READY when the TMS320C5x device has control. The length of the memory cycle is controlled by the READY signal. In addition, the software wait-state generators can be used to extend the access times for slower, external memories. The wait-state generators corresponding to the overlapped memory address space in local data space will generate the wait states for the corresponding addresses in global data memory space.

6.3.3 External Interfacing of Global Memory

Global memory can be used in various digital signal processing tasks, such as filters or modems, where the algorithm being implemented may be divided into sections with a distinct processor dedicated to each section. With multiple processors dedicated to distinct sections of the algorithm, throughput may be increased via pipelined execution. Figure 6–11 illustrates an example of a global memory interface. Since the processors can be synchronized by using the $\overline{RS}$ pin, the arbitration logic may be simplified and the address and data bus transfers made more efficient.

Figure 6–11. Global Memory Interface



6.4 Input/Output Space

The TMS320C5x devices support an I/O address space of 64K 16-bit parallel input and output ports. I/O ports allow access to peripherals typically used in DSP applications such as codecs, digital-to-analog (D/A) converters, and analog-to-digital (A/D) converters. This section discusses addressing I/O ports and interfacing I/O ports to external devices.

6.4.1 Addressing Input/Output Ports

Access to external parallel I/O ports is multiplexed over the same address and data bus for program/data memory accesses. I/O space access is distinguished from program/data memory accesses by the $\overline{IS}$ signal going active low. All 65,536 ports can be accessed via the IN and OUT instructions, as shown in the following example:

```
IN  0FFFEh,DAT7   ;Read data to data memory from external
                  ;device on port 65534.
OUT 0FFFFh,DAT7   ;Write data from data memory to external
                  ;device on port 65535.
```

Sixteen of the 64K I/O ports are mapped in data memory space as shown in Table 6–4. The I/O ports may be accessed with the IN and OUT instructions along with any instruction that reads or writes a location in data space. In this way, I/O is treated the same way as memory. The following example illustrates the use of direct addressing to access an I/O device on port 51h:

```
SACL   51h     ;(DP = 0). Store accumulator to external device
               ;on port 81.
```

Accesses to memory-mapped I/O space are distinguished from program/data accesses by the $\overline{IS}$ signal. $\overline{DS}$ is not active, even though the user is writing to data space.

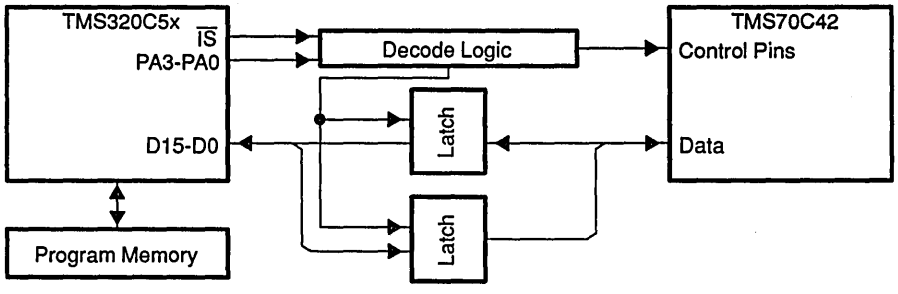
6.4.2 Interfacing to I/O Ports

The $\overline{RD}$ and $\overline{WE}$ signals can be used along with chip-select logic to output data to an external device. The port address can be decoded and used as a chip select for the input or output device. The access times to I/O ports can be modified through the CWSR and IOWSR software wait-state registers. The BIG bit in the CWSR register determines how the I/O space is mapped to the software control registers. If the BIG bit is set to 0 in the CWSR register, the first sixteen ports are assigned in pairs to a software wait-state generator. Each following set of 16 registers maps accordingly to the first 16 ports when BIG = 0. For example, the 16 ports that correspond to the addresses in the data space port hole (ports 50h–5Fh) have the same wait states as ports 0–Fh. If the BIG bit is set to 1, the wait states are mapped to program space in eight 8K blocks of memory. The following table shows how the software wait states are assigned to I/O ports according to the BIG bit:

6.5 Direct Memory Access (DMA)

The TMS320C5x supports multiprocessing designs using direct memory access (DMA) of external memory or the TMS320C5x on-chip single access RAM. The DMA features can be used for multiprocessing by temporarily halting the execution of one or more processors to allow another processor to read from or write to the TMS320C5x's local off-chip memory or on-chip single-access RAM. The external memory access may be controlled by using the $\overline{\text{HOLD}}$/$\overline{\text{HOLDA}}$ signals. The DMA access of internal RAM on the TMS320C5x is controlled by the $\overline{\text{HOLD}}$, $\overline{\text{HOLDA}}$, R/$\overline{\text{W}}$, $\overline{\text{STRB}}$, $\overline{\text{BR}}$, and $\overline{\text{IAQ}}$ lines.

Figure 6–12. I/O Port Interfacing Logic



The multiprocessing is typically a master-slave configuration. The master may initialize a slave by downloading a program into its program memory space and/or may provide the slave with the necessary data by using external memory to complete a task. In a typical TMS320C5x direct memory access scheme, the master may be a general-purpose CPU, another TMS320C5x, or even an analog-to-digital converter. A simple TMS320C5x master-slave configuration is shown in Figure 6–13.

Figure 6–13. Direct Memory Access Using a Master-Slave Configuration

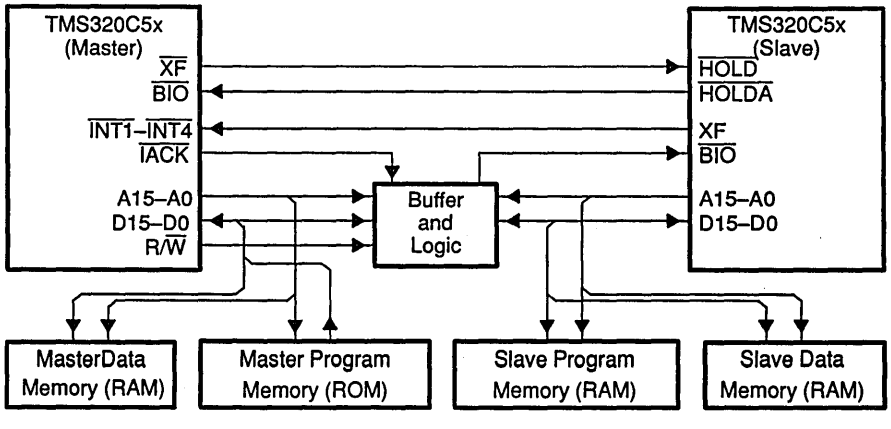
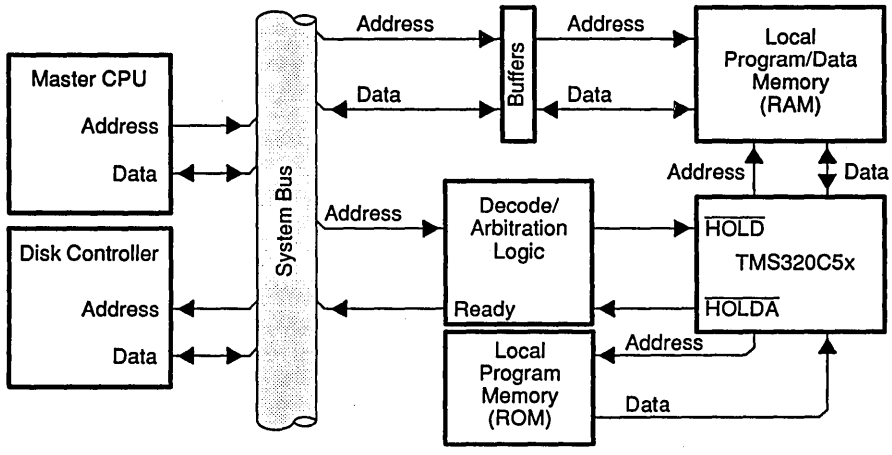


Figure 6–14. Direct Memory Access in a PC Environment



The TMS320C5x device also provides direct access of the on-chip RAM for external devices. DMA of the on-chip single-access RAM requires the following signals:

$\overline{\text{HOLD}}$	External request for control of address, data, and control lines.
$\overline{\text{HOLDA}}$	Indicates to external circuitry that the memory address, data, and control lines are in high impedance, allowing external access of on-chip single-access RAM.
$\overline{\text{BR}}$	Bus request signal. Externally driven low in hold mode to indicate a request for access to on-chip single-access RAM.
$\overline{\text{IAQ}}$	Acknowledge $\overline{\text{BR}}$ request for access to on-chip single-access RAM while $\overline{\text{HOLDA}}$ is low.
R/$\overline{\text{W}}$	Read/write signal indicates the data bus direction for DMA reads (high) and DMA writes (low).
$\overline{\text{STRB}}$	When active low and $\overline{\text{IAQ}}$ and $\overline{\text{HOLDA}}$ are low, this input signal is used to select the memory access. $\overline{\text{STRB}}$ determines the duration of the memory access.
A15–A0	Address inputs during $\overline{\text{HOLDA}}$ and $\overline{\text{BR}}$ active low.
D15–D0	DMA data.

In order to access the TMS320C5x device's on-chip single-access RAM, a master device must control the device. The master processor initiates a DMA transfer by placing the TMS320C5x device in $\overline{\text{HOLD}}$. Once the device responds with a $\overline{\text{HOLDA}}$, the master can select access to the internal on-chip single-access RAM by lowering the $\overline{\text{BR}}$ input. The device will respond with an $\overline{\text{IAQ}}$ to acknowledge access to the on-chip memory. At this time, the processor

6.6 Memory Management

The TMS320C5x devices have a programmable memory map, which can vary for each application. Instructions are provided for integrating the device memory into the system memory map. The TMS320C50 device includes 2K words of boot ROM, 9K words of single-access RAM, and 1056 words of dual-access RAM. The TMS320C51 device includes an 8K program ROM, 1K words of single-access RAM, and 1056 words of dual-access RAM. Examples of moving and configuring memory are provided in this section.

6.6.1 Block Moves

The TMS320C5x devices address a large amount of memory but are limited in the amount of on-chip memory. Several instructions are available for moving blocks of data from off-chip slower memories to on-chip memory for faster program execution. In addition, data can be transferred from on-chip to off-chip for storage or multiprocessor applications.

The BLDD instruction facilitates the transfer of data from external or internal data memory to internal or external data memory. Example 6–1 illustrates the use of the BLDD command to move data (for example, a table of coefficients) from external memory to internal data RAM.

Example 6–1. Moving External Data to Internal Data Memory With BLDD

```
*
* This routine uses the BLDD instruction to move external data memory to
* internal data memory.
*
MOVED   LMMR    BMAR,#2800h;BMAR contains source address in data memory.
        LAR     AR7,#300h  ;AR7 contains destination address in data memory.
        MAR     *,AR7      ;LARP = AR7.
        RPT     #511       ;Move 512 values to data memory block B1.
        BLDD    BMAR,*+
        RET
```

For systems with external data memory but no external program memory, the BLDP instruction can be used to move additional blocks of code into internal program memory. Example 6–2 illustrates the use of the BLDP instruction.

Example 6–5. Moving Data Memory to Program Memory With TBLW

```
*
* This routine uses the TBLW instruction to move data memory to
* program memory. The calling routine must contain the destination program
* memory address in the accumulator.
*
TABLEW  MAR    *,AR4          ;LARP = AR4.
        LAR    AR4,#300h      ;AR4 contains source address in data memory.
        RPT    #511           ;Move 512 items from data memory to program
        TBLW   *+             ;memory.
        RET                   ;Accumulator contains address of program RAM.
```

The IN and OUT instructions move data from data memory to an external port. The use of these instructions is illustrated in the following examples.

Example 6–6. Moving Data From I/O Space to Data Memory With IN

```
*
* This routine uses the IN instruction to move data from I/O space into
* data memory.
*
INPUT   MAR    *,AR2          ;LARP = AR2.
        LAR    AR2,#300h      ;AR2 = 300h.
                              ;Input value to data memory at 300h
        IN     *+,1           ;from port 1.
        RET
```

Example 6–7. Moving Data From Data Memory to I/O Space With OUT

```
*
* This routine uses the OUT instruction to move data from data space to
* I/O space.
*
OUTP    MAR    *,AR1          ;LARP = AR1
        LAR    AR1,#200h      ;AR1 = 200h
                              ;Output value to port 1.
        OUT    *+,1
        RET
```

6.6.2 On-Chip Boot ROM (TMS320C50)

The fifth generation of the Texas Instruments digital signal processors provides two different options regarding the chip count and the system flexibility. One member of the family, TMS320C51, has 8K words of mask-programmable on-chip ROM that allows the customer to use a code-customized processor for specific applications while taking advantage of the following:

- Greater memory expansion
- Lower system cost
- Less hardware and wiring
- Smaller PCB

User routines may be submitted customers to Texas Instruments to be masked to the on-chip ROM of TMS320C51.

low-order one. Data is read from the lower eight data lines, ignoring the upper byte on the data bus. The destination address and the length of the code are specified by the first two 16-bit words read from the source. The length is defined as:

length = number of 16-bit words to be transferred – 1

The code is transferred from the global data memory to the program memory. Note that there is at least a four-instruction cycle delay between a read from EPROM and write to destination address. This ensures that if the destination is external memory, there is enough time to turn off the source memory (EPROM) before the write operation is done.

7 Software Applications

7.1 Processor Initialization

Prior to the execution of a digital signal processing algorithm, it is necessary to initialize the processor. Generally, initialization takes place anytime the processor is reset.

The processor is reset by applying a low level to $\overline{RS}$ input for at least five machine cycles; IPTR bits of PMST register are all set to zero, thus mapping the vectors to page zero in program memory space. This means that the reset vector always resides at program memory location 0. This location normally contains a branch instruction in order to direct program execution to the system initialization routine. A hardware reset clears all pending interrupt flags and sets the INTM (global enable interrupts) bit to 1, thereby disabling all interrupts. It also initializes various status bits and peripheral registers. Refer to subsection 3.8.1 of this book for details.

To configure the processor after the reset, the following internal functions should be initialized.

- Memory-mapped core processor and peripheral control registers
- Interrupt structure (INTM)
- Mode control (OVM, SXM, PM, AVIS, NDX, TRM)
- Memory control (RAM, OVLY, CNF)
- Auxiliary registers and the auxiliary register pointer (ARP)
- Data memory page pointer (DP)

The OVM (overflow mode), TC (test/control flag), IMR (interrupt mask register), auxiliary register pointer (ARP), auxiliary register pointer buffer (ARB), and data memory page pointer (DP) are not initialized by reset.

Example 7–1 shows coding for initializing the TMS320C5x to the following machine state, and for the initialization performed during hardware reset:

- Internal single-access RAM configured as program memory
- Interrupt vector table loaded in internal program memory
- Interrupt vector table pointer (IPTR)
- Internal dual-access RAM blocks filled with zero
- Interrupts enabled

7.2 Interrupts

The TMS320C5x devices have four external maskable user interrupts ($\overline{INT1}$–$\overline{INT4}$) and one nonmaskable interrupt ($\overline{NMI}$) available for external devices. Internal interrupts are generated by the serial ports, the timer, and by the software interrupt instructions (INTR, TRAP, and NMI). The interrupt structure is described in subsection 5.1.2, *Interrupts*.

The TMS320C5x devices are capable of generating software interrupts using INTR instruction. This allows any of the 32 interrupt service routines to be executed from the user's software. The first 20 ISRs are reserved for external interrupts, peripheral interrupts, and future implementations. The other 12 locations in the interrupt vector table are user-definable. The INTR instruction can invoke any of the 32 interrupts available on the TMS320C5x devices.

The context saving and restoring function is done in hardware when an interrupt trap is executed. An 8-deep hardware stack is available for saving return addresses of the subroutines and the interrupt service routines. Also, there is a one-deep stack (or shadow registers) on the following registers:

ACC	accumulator
ACCB	accumulator buffer
PREG	product register
ST0	status register 0
ST1	status register 1
PMST	processor mode status register
TREG0	temporary register for multiplier
TREG1	temporary register for shift count
TREG2	temporary register for bit test
INDX	indirect address index register
ARCR	auxiliary register compare register

When the interrupt trap is taken, all these registers are pushed onto the one-deep stack. These shadow registers are popped when the return-from-interrupt (RETI or RETE) is executed. Detailed discussion of interrupts are given in Section 3.8, *Interrupts*.

The following example illustrates the use of INTR instruction. The foreground program sets up auxiliary registers and invokes user-defined interrupt number 20. Since the context is saved automatically, the interrupt service routine is free to use any of the saved registers without destroying the calling program's variables. The routine shown here uses the CRGT instruction to find the maximum value of 16 executions of the equation Y=aX^2+bX+c. The X values are pointed at by AR1. AR2 and AR3 point to the coefficients and Y results, respectively. In order to return the result to the calling routine, all the registers are restored by executing an RETI instruction. The computed value is placed in the accumulator, and a standard return is executed because the stack is already popped.

7.3 Software Stack

The TMS320C5x has an internal 8-deep hardware stack that is used to save and restore return addresses for subroutines and interrupts. See subsection 3.6.1 for further details. Provisions have been made on the TMS320C5x to extend the hardware stack into the data memory.

The PUSH and POP instructions can access the hardware stack via the accumulator. Two additional instructions, PSHD and POPD, are included in the instruction set so that the stack may be directly stored to and recovered from the data memory.

A software stack can be implemented by using POPD instruction at the beginning of each subroutine in order to save the PC in data memory. Then, before returning, a PSHD is used to put the proper value back onto the top of the stack.

When the stack has seven values stored on it, and two or more values are to be put on the stack before any other values are popped off, a subroutine that expands the stack is needed, such as the one shown in Example 7–3. In this example, the main program stores the stack, starting location in memory in AR2 and indicates to the subroutine whether to push the data from memory onto the stack or pop data from the stack to memory. If a zero is loaded into the accumulator before calling the subroutine, the subroutine pushes data from memory to the stack. If the accumulator contains a nonzero value, the subroutine pops data from the stack to memory.

Since the CALL instruction uses the stack to save the program counter, the subroutine pops this value into the accumulator and utilizes the BACC instruction to return to the main program. This prevents the program counter from being stored into a memory location. The subroutine in Example 7–3 uses the BCNDD (delayed conditional branch) instruction to determine whether a save or restore operation is to be performed.

Example 7–3. Software Stack Operation

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine expands the stack while letting the
; main program determine where to store the stack
; contents, or from where to restore them.
; Entry Conditions:
; ACC = 0 (restore stack); 1 (save stack)
; AR2 -> Top of software stack in data memory
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

STACK:  BCNDD   POP,NEQ    ;Delayed branch if POPD required
        MAR     *,AR2      ;Use AR2 as stack pointer
        POP                ;Get return address
        RPT     #6         ;repeat 7 times
        PSHD    *+         ;Put memory in stack
        BACC               ;Return to main program
POP:    MAR     *-         ;Align AR2
        RPT     #6         ;Repeat 7 times
        POPD    *-         ;Put stack in memory
        MAR     *+         ;Realign stack pointer
        BACC               ;Return to main program
```

```
            APL     *-                      ;Keep the LSB only
            SFR                             ;Shift right to throw away unpacked bit
    LOOP    RET                             ;Return back
```

Example 7–5. Using PLU to Do Packing

```
        .title 'Routine to pack input bits in a single word'
*;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
*   PCKD
*   ----------------
*   |Bn  ------  B0|
*   ----------------
*
*   UNPCKD
*   ----------------
*   |0    --    0 |Bn|
*   ----------------
*   |0    --   0|Bn-1|
*   ----------------
*           . .
*   ----------------
*   |0    --     0|B0|
*   ----------------
*;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .mmregs

        .data
NO_BITS .set    16              ;Number of bits to be packed

PCKD    .set    60h             ;Packed word
UNPCKD  .set    61h             ;Array of unpacked bits

        .text
PACK    LAR     AR0,#UNPCKD;AR0 points to start of UNPACKED array
        MAR     *,AR0           ;ARP <- AR0
        LDP     #0              ;DP=0

        SPLK    #NO_BITS-2,BRCR     ;Loop NO_BITS-1 times
        LACC    *+              ;Get the MSB

        RPTB    LOOP-1          ;Begin looping
        SFL                     ;Make space for next bit
        ADD     *+              ;Put next bit
        NOP
LOOP
        SACL    PCKD            ;Store the result
        RET                     ;Return back
```

7.4.2 Multiconditional Branch Instruction

The TMS320C5x allows multiple conditions to be tested before passing control to another section of program. Any of the following 13 conditions may be tested individually or in combination with others by CC, RETC, XC, and BCND instructions:

ACC=0	EQ
ACC≠0	NEQ
ACC<0	LT
ACC≤0	LEQ
ACC>0	GT
ACC≥0	GEQ

Example 7–7. Using CRGT and CRLT

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine searches through a block of data in the data memory
; to store the maximum value and the address of that value in memory
; locations MAXVAL and MAXADR, respectively. The data block could be
; of any size defined by the Block Repeat Counter Register (BRCR).
;
; KEY C5X instructions:
;
; RPTB   repeat a block of code as defined by repeat counter BRCR
; CRGT   compare ACC to ACCB. Store larger value in both ACC, ACCB.
;        Set CARRY bit if a value larger than the previously larger one is found
; XC     execute conditionally (1 or 2 words) if flag (Carry) is set.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
MAXADR  .set    60h
MAXVAL  .set    61h
        .mmregs
        .text
        LDP     #0              ; point to data page 0
        LAR     AR0, #0300h     ; AR= data memory addr
        SETC    SXM             ; set sign extension mode
        LACC    #08000h         ; load minimum value
;;;; Use #07FFFh (largest possible) to check for minimum value
        SACB                    ; into ACCB
        SPLK    #9,BRCR         ; rpt cont = 9 for 10 data values
        RPTB    endb -1         ; repeat block from here to endb-1
startb
        LACC    *               ; load data from <(AR0)> into ACC
        CRGT                    ; set carry if ACC > previous largest
;  Use CRLT to find minimum value
        SACL    MAXVAL          ; save new largest which is in ACC & ACCB
        XC      #1,C            ; save addr if current value > previous largest
        SAR     AR0,MAXADR      ;
        MAR     *+
endb    RET

; At the end of routine, following
; registers contain:
;   ACC         = 32050
;   ACCB        = 32050
;   (MAXVAL)    = 32050
;   (MAXADR)    = 0307h

    .data                       ; data is expected to be in data ram
    .word   5000                ; start address = 0300h
    .word   10000
    .word   320
    .word   3200
    .word   -5600
    .word   -2105
    .word   2100
    .word   32050
    .word   1000
    .word   -1
    .end
```

7.4.4 Matrix Multiplication Using Nested Loops

The TMS320C5x provides three different types of instructions to implement code loops. The RPT (single-instruction repeat) instruction allows the following instruction to be executed N times. The RPTB (repeat block) instruction repeatedly executes a block of instructions with the loop count determined by the BRCR count register. The BANZ (branch if AR not zero) instruction is another

7.5 Circular Buffers

Circular addressing is an important feature of the TMS320C5x instruction set. Algorithms like convolution, correlation, and FIR filters can make use of circular buffers in memory. The TMS320C5x supports two concurrent buffers operating via the auxiliary registers. These five memory-mapped registers control the circular buffer operation: CBSR1, CBSR2, CBER1, CBER2, CBCR. See subsection 4.1.6 of this book for details.

The start and end addresses must be loaded in the corresponding buffer registers before the circular buffer is enabled. Also, the auxiliary register that acts as a pointer to the buffer must be initialized with the proper value.

Example 7–9 illustrates the use of a circular buffer to generate a digital sine wave. A 256-word sine-wave table is loaded in the B1 block of dual-access internal data memory from external program memory. Accessing the internal dual-access memory requires only one machine cycle. The block move address register (BMAR) is loaded with the ROM address of the table. The block-move instruction moves 256 samples of sine wave to internal data memory, which is then set up as a circular buffer.

The start and end addresses of this circular buffer are loaded into the corresponding registers. The auxiliary register AR7 is also initialized to the beginning of the sine-wave table. Note the use of SAMM instruction to update AR7. This is possible because all auxiliary registers are memory-mapped at page 0. Finally, the circular buffer #1 is enabled, and AR7 is mapped to that buffer. The other circular buffer is disabled.

Whenever the next sample is to be pulled off from the table, postincrement indirect addressing may be used with AR7 as the pointer. This ensures that the pointer will wrap around to the beginning of the table if the previous sample was the last one on the table.

The following code does modulo-256 addressing:

```
START   .set    04000h      ; start address of the buffer

        LDP     #0
        LACL    #0FFh
        SAMM    DBMR        ; max value = 255
         .
         .
         .
        MAR     *0+         ; increment AR7 by some amount
        APL     AR7         ; extract lower 4 bits
        OPL     #START,AR7  ; add the start address
         .
         .
```

Example 7–10. Memory-to-Memory Block Moves Using RPT

```
        .mmregs
        .text

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the BLDD instruction to move external
; data memory to internal data memory.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

MOVEDD:
        SPLK #4000h,BMAR ; BMAR -> source in data memory.
        LAR  AR7,#100h   ; AR7 -> destination in data memory
        MAR  *,AR7       ; LARP = AR7.
        RPT  #1023       ; Move 1024 value to blocks B0 and B1
        BLDD BMAR,*+
        RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the BLDP instruction to move external
; data memory to internal program memory.  This
; instruction could be used to boot load a program to
; the 8K on chip program memory from external data memory.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

MOVEDP:
        SPLK #800H,BMAR  ; BMAR -> destination in program memory
        LAR  AR7,#0E000h ; AR7 -> source in data memory.
        RPT  #8191       ; Move 8k to program memory space.
        BLDP *+
        RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the BLPD instruction to move external
; program memory to internal data memory.  This routine
; is useful for loading a coefficient table stored in
; external program memory to data memory when no external
; data memory is available.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

MOVEPD:
        LAR  AR7,#100h   ; AR7 -> destination in data memory.
        RPT  #127        ; Move 128 values from external program
        BLPD #3800h,*+   ; to internal data memory B0.
        RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the TBLR instruction to move program
; memory to data memory space.  This differs from the BLPD
; instruction in that the accumulator contains the address
; in program memory from which to transfer.  This allows
; for a calculated, rather than pre-determined, location in
; program memory to be specified.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

TABLER:
        MAR  *,AR3       ; AR3 -> destination in data memory.
        LAR  AR3,#300h
        RPT  #127        ; Move 128 items to data memory block B1
        TBLR *+
        RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the TBLW instruction to move data
; memory to program memory.  The calling routine must
; contain the destination program memory address in the
; accumulator.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

TABLEW:
        MAR  *,AR4       ; ARP = AR4.
        LAR  AR4,#380h   ; AR4 -> source address in data memory.
```

7.7 Subroutines

Example 7–11 illustrates the use of a subroutine to determine the square root of a 16-bit number. The main routine executes to the point where the square root of a number should be taken. At this point, a delayed call (CALLD) is made to the subroutine, transferring control to that section of the program memory for execution and then returning to the calling routine via the delayed return (RETD) instruction when execution has completed.

This example shows several features of TMS320C5x instruction set. In particular, note the use of delayed-call (CALLD), delayed-return (RETD), and conditional-execute (XC) instructions. Due to the four-level-deep pipeline on TMS320C5x devices, normal branch instructions require 4 cycles to execute. Using delayed branches, only two cycles are required for execution. The XC instruction is useful where only one or two instructions are to be executed conditionally. In this example, notice how XC is used to avoid extra cycles due to branch instruction. Use of the XC instruction also helps in keeping the execution time of a routine constant, regardless of input conditions. This is because XC executes NOPs in place of instructions if conditions are not met.

Example 7–11. Square Root Computation Using XC

```
;  Autocorrelation

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  This routine performs a correlation of two vectors and then
;  calls a Square Root subroutine that will determine the RMS
;  amplitude of the wave form.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

AUTOC
        .
        .
        .
       CALLD     SQRT        ;Call square root subroutine after
       MAR       *,AR0       ; executing next two instructions
       LACC      *           ;Get the value to be passed to SQRT
        .                    ; subroutine
        .
        .

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Square Root Computation
;
;  This routine computes the square root of a number that is located
;  in the lower half of accumulator. The number is in Q15 format.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

BRCR    .set      09h         ;DP=0
ST0     .set      60h         ;Internal RAM block B2
ST1     .set      61h
NUMBER  .set      62h
TEMPR   .set      63h
GUESS   .set      64h

   .text
SQRT    SST       #0,ST0
        SST       #1,ST1      ;Save context
        LDP       #0
        SETC      SXM         ;Set SXM=1
        SPM       1           ;Set PM mode for fractional arithmetic
```

7.8 Extended-Precision Arithmetic

Numerical analysis, floating-point computations, or other operations may require arithmetic to be executed with more than 32 bits of precision. Since the TMS320C5x devices are 16/32-bit fixed-point processors, software is required for the extended precision of arithmetic operations. Subroutines that perform the extended-arithmetic functions for TMS320C5x are provided in the examples of this section. The technique consists of performing the arithmetic by parts, similar to the way in which longhand arithmetic is done.

The TMS320C5x has several features that help make extended-precision calculations more efficient. One of the features is the carry bit. This bit is affected by all arithmetic operations of the accumulator, including addition and subtraction with the accumulator buffer. This allows 32-bit-long arithmetic operations using the accumulator buffer as the second operand.

The carry bit is also affected by the rotate and shift accumulator instructions. It may also be explicitly modified by the load status register ST1 and the set/reset control bit instructions. For proper operation, the overflow mode bit should be reset (OVM = 0) so that the accumulator results will not be loaded with the saturation value.

7.8.1 Addition and Subtraction

The carry bit is set whenever the addition of a value from the input scaling shifter, the P register, or the accumulator buffer to the accumulator contents generates a carry out of bit 31. Otherwise, the carry bit is reset because the carry out of bit 31 is a zero. One exception to this case is the addition to the accumulator with a shift of 16 instruction (ADD mem,16), which can only set the carry bit. This allows the ALU to generate a proper single carry when the addition either to the lower or the upper half of the accumulator actually causes the carry. The following examples help to demonstrate the significance of the carry bit of the TMS320C5x for additions:

In a similar way to addition, the carry bit on the TMS320C5x is reset whenever the input scaling shifter, the P register, or the accumulator buffer value subtracted from the accumulator contents generates a borrow into bit 31. Otherwise, the carry bit is set because no borrow into bit 31 is required. One exception to this case is the SUB mem,16 instruction, which can only reset the carry bit. This allows the generation of the proper single carry when the subtraction from either the lower or the upper half of the accumulator actually causes the borrow. The examples in Figure 7–2 demonstrate the significance of the carry bit for subtraction.

Figure 7–2. 32-Bit Subtraction

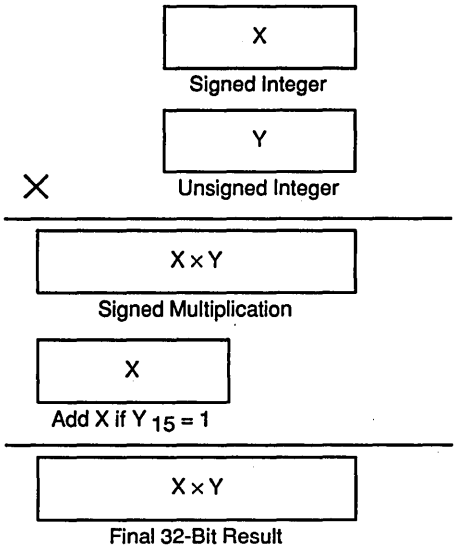
```
C  MSB           LSB                C  MSB           LSB
X   0 0 0 0 0 0 0 0 ACC             X   0 0 0 0 0 0 0 0 ACC
    -             1                     -F F F F F F F F
0   F F F F F F F F                 0   0 0 0 0 0 0 0 1

C  MSB           LSB                C  MSB           LSB
X   7 F F F F F F F ACC             X   7 F F F F F F F ACC
    -             1                     -F F F F F F F F
1   7 F F F F F F E                 C   8 0 0 0 0 0 0 0

C  MSB           LSB                C  MSB           LSB
X   8 0 0 0 0 0 0 0 ACC             X   8 0 0 0 0 0 0 0 ACC
    -             1                     -F F F F F F F F
1   7 F F F F F F F                 0   8 0 0 0 0 0 0 1

C  MSB           LSB                C  MSB           LSB
0   0 0 0 0 0 0 0 0 ACC             0   F F F F F F F F ACC
    -             0 (SUBB)              -             0 (SUBB)
0   F F F F F F F F                 1   F F F F F F F E

C  MSB           LSB                C  MSB           LSB
0   8 0 0 0 F F F F ACC             0   8 0 0 0 F F F F ACC
   -0 0 0 1 0 0 0 0 (SUB mem,16)       -F F F F 0 0 0 0 (SUB mem,16)
0   7 F F F F F F F                 0   8 0 0 1 F F F F
```

Example 7–13 implements the subtraction of two 64-bit numbers on the TMS320C5x. A borrow is generated within the accumulator for each of the 16-bit parts of the subtraction operation.

Example 7–13. 64-Bit Subtraction

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Two 64-bit numbers are subtracted, producing a 64-bit
; result. The number Y (Y3,Y2,Y1,Y0) is subtracted from
; X (X3,X2,X1,X0) resulting in W (W3,W2,W1,W0).
; If the result is required in 64-bit ACC/ACCB pair,
; replace the instructions as indicated in the comments
; below.
;
;   X3 X2 X1 X0
; - Y3 Y2 Y1 Y0
```

Figure 7–3. 16-Bit Integer Multiplication



Steps Required:

1) Multiply two operands X and Y as if they are signed integers,

2) If MSB of the unsigned integer Y is 1, add X to the upper half of the 32-bit signed product.

The correction factor must be added to the signed multiplication result because the bit weight of the MSB of any 16-bit unsigned integer is 2^{15}.

Consider following representation of a signed integer X and an unsigned integer Y:

$$X = -2^{15}x_{15} + 2^{14}x_{14} + 2^{13}x_{13} + \ldots + 2^{1}x_{1} + 2^{0}x_{0}$$

$$Y = 2^{15}y_{15} + 2^{14}y_{14} + 2^{13}y_{13} + \ldots + 2^{1}y_{1} + 2^{0}y_{0}$$

Multiplication of X and Y would yield:

$$X \times Y = X \times (2^{15}y_{15} + 2^{14}y_{14} + 2^{13}y_{13} + \ldots + 2^{1}y_{1} + 2^{0}y_{0})$$

$$= 2^{15}y_{15}X + 2^{14}y_{14}X + 2^{13}y_{13}X + \ldots + 2^{1}y_{1}X + 2^{0}y_{0}X \qquad (1)$$

However, if X and Y are considered signed integers, their multiplication would yield:

$$X \times Y = X \times (-2^{15}y_{15} + 2^{14}y_{14} + 2^{13}y_{13} + \ldots + 2^{1}y_{1} + 2^{0}y_{0})$$

The following example implements this algorithm. The product is a 64-bit integer number. Note in particular, the use of BSAR and XC instructions.

Example 7–14. 32-Bit Integer Multiplication

```
   .title "32-bit Optimized Integer Multiplication"
   .def    MPY32

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine multiplies two 32-bit signed integers result-
; ing in a 64-bit product. The operands are fetched from
; data memory and the result is written back to data memory.
; Data Storage:
;    X1,X0          32-bit operand
;    Y1,Y0          32-bit operand
;    W3,W2,W1,W0    64-bit product
; Entry Conditions:
;    DP  = 6, SXM = 1
;    OVM = 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

X1      .set    300h    ;DP=6
X0      .set    301h    ;DP=6
Y1      .set    302h    ;DP=6
Y0      .set    303h    ;DP=6
W3      .set    304h    ;DP=6
W2      .set    305h    ;DP=6
W1      .set    306h    ;DP=6
W0      .set    307h    ;DP=6

        .text

MPY32:
     BIT    X0,0  ;TC = X0 bit#15
     LT     X0    ;T = X0
     MPYU   Y0    ;P = X0Y0
     SPL    W0    ;Save W0
     SPH    W1    ;Save partial W1
     MPY    Y1    ;P = X0Y1
     LTP    X1    ;Acc = X0Y1, T = X1
     MPY    Y0    ;P = X1Y0
     MPYA   Y1    ;Acc = X0Y1+X1Y0, P=X1Y1
     ADDS   W1    ;Acc = X0Y1+X1Y0+X0Y02^-16
     SACL   W1    ;Save final W1
     BSAR   16    ;Shift Acc right by 16
     XC     1,TC  ;If MSB of X0 is 1
     ADD    Y1    ;Add Y1
     BIT    Y0,0  ;TC = Y0 bit#15
     APAC         ;ACC = X1Y1 + (X0Y1+X1Y0)2^-16
     XC     1,TC  ;IF MSB of Y0 is 1
     ADD    X1    ;Add X1
     SACL   W2    ;Save W2
     SACH   W3    ;Save W3
```

Integer and fractional division can be implemented with the SUBC instruction as shown in Example 7–16 and Example 7–17, respectively. When implementing a divide algorithm, it is important to know if the quotient can be represented as a fraction and the degree of accuracy to which the quotient is to be computed. For integer division, the absolute value of the numerator must be greater than the absolute value of the denominator. For fractional division, the absolute value of the numerator must be less than the absolute value of the denominator.

Long Division:

```
                    000000000000110    Quotient
0000000000000101  )000000000010001
                        -101
                         110
                        -101
                          11           Remainder
```

SUBC Method:

```
32   HIGH ACC          LOW ACC   0      Comment
|                   |  |              |
0000000000000000      0000000000100001  (1) Dividend is loaded into ACC. The di-
                -10   1000000000000000      visor is left-shifted 15 and subtracted
               -10    0111111111011111      from ACC. The subtraction is nega-
                                            tive, so discard the result and shift
                                            left the ACC one bit.
|                   |  |              |
0000000000000000      0000000001000001  (2) 2nd subtract produces negative an-
                -10   1000000000000000      swer, so discard result and shift ACC
               -10    0111111110111110      (dividend) left.
                      •                                 •
                      •                                 •
                      •                                 •
|                   |  |              |
0000000000000100      0010000000000000  (14) 14th SUBC command. The result is
                -10   1000000000000000      positive. Shift result left and replace
0000000000000001      1010000000000000      LSB with 1.
|                   |  |              |
0000000000000011      0100000000000001  (15) Result is again positive. Shift result
                -10   1000000000000000      left and replace LSB with 1.
0000000000000000      1100000000000001
|                   |  |              |
0000000000000001      1000000000000011  (16) Last subtract. Negative answer, so
                -10   1000000000000000      discard result and shift ACC left.
                      -1111111111111101

0000000000000011      0000000000000110      Answer reached after 16 SUBC in-
                                            structions.
|   Remainder       |  |   Quotient    |
```

Example 7–17. Fractional Division Using SUBC

```
*;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
* This routine implements fractional division with the SUBC instruction.  For
* this division routine, the absolute value of the denominator must be
* greater than the absolute value of the numerator.  In addition, the
* calling routine must check to verify that the divisor does not equal 0.
*
* The 16-bit dividend is placed in the high accumulator, and the low accumulator
* is zeroed.  The divisor is in data memory.
*;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

DENOM       .set    60h
NUMERA      .set    61h
QUOT        .set    62h
REM         .set    63h
TEMSGN      .set    64h
*
FRACDIV     LDP     #0
            LT      NUMERA       ; Determine sign of quotient.
*
            MPY     DENOM
            SPH     TEMSGN
            LACL    DENOM
            ABS                  ; Make denominator and numerator positive.
            SACL    DENOM
            LACC    NUMERA,16    ; Load high accumulator, zero low accumulator.
            ABS
*
* If divisor and dividend are aligned, division can start here.
*
            RPT     #15          ; 16-cycle division. Low accumulator contains
            SUBC    DENOM        ; the quotient and high accumulator contains the
                                 ; remainder at the end of the loop.
*
            BIT     TEMSGN,0     ; Test sign of quotient.
            RETCD   NTC              ; Return if sign positive, else continue.
            SACL    QUOT         ; Store quotient and remainder during delayed
            SACH    REM          ; return.
*

            LACL    #0           ; If sign negative, negate quotient
            RETD                 ; and return
            SUB     QUOT
            SACL    QUOT
```

```
*
*   SAMM    save the accumulator contents in a memory-mapped
*       register
*   LACB    accumulator is loaded with contents of accumulator
*       buffer
*   SACB    contents of accumulator are copied in accumulator
*       buffer
*   SATL    accumulator is barrel-shifted right by the value
*       specified in the 4 LSBs of TREG1
*   SATH    accumulator is barrel-shifted right by 16 bits
*       if bit 4 of TREG1 is a one.
*   SPLK    store immediate long constant in data memory
*   CPL compare long immediate value (or DBMR) with data
*       memory
*       TC=1 if two values are same
*       TC=0 otherwise
*;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
TREG1   .set    0dh

ASIGN   .set    60h         ;Sign, exponent, high and low part of mantissa
AEXP    .set    61h         ;of input number A
AHI     .set    62h
ALO     .set    63h

BSIGN   .set    64h         ;Sign, exponent, high and low part of mantissa
BEXP    .set    65h         ;of input number B
BHI     .set    66h
BLO     .set    67h

CSIGN   .set    68h         ;Sign, exponent, high and low part of mantissa
CEXP    .set    69h         ;of the resulting floating point number C
CHI     .set    6Ah
CLO     .set    6Bh
DIFFEXP .set    6Ch

    .text

FL_ADD  LDP     #0          ;Initialization
        SETC    SXM         ;Set sign extension mode
        MAR     *,AR0       ;ARP <- AR0
        LAR     AR0,#0      ;AR0 is used by NORM instruction

CMPEXP  LACL    BLO         ;Load low Acc with BLO
        ADD     BHI,16      ;Add BHI to high Acc
        SACB                ;AccB = BHIBLO
        LACC    AEXP
        SUB     BEXP        ;Acc = AEXP=BEXP
        SACL    DIFFEXP     ;Save the difference
        BCND    AEQB,EQ     ;If |A| == |B|
        BCND    ALTB,LT     ;If |A| < |B|

AGTB    LACC    DIFFEXP     ;If |A| > |B|
        SAMM    TREG1       ;Load TREG1 with # of right shifts reqd.
        SUB     #32
        BCND    AGRT32,GEQ  ;If difference > 32
        LACB                ;Acc = BHIBLO
        SATL
        SATH                ;Right justify BHIBLO
        SACB                ;Store the result back in AccB

AEQB    LACC    ASIGN       ;Copy sign and exponent values of
        SACL    CSIGN       ;A in C (i.e. the result)
        LACC    AEXP
        SACL    CEXP

CHKSGN  LACC    ASIGN       ;Acc=ASIGN-BSIGN
        SUB     BSIGN
        CLRC    TC          ;Clear TC flag
        XC      1,LT        ;If A<0 and B>0
        SETC    TC          ;Set TC flag
        BCNDD   ADNOW,EQ    ;If both A and B have same sign
```

```
        ADD     AHI,16          ;Acc=AHIALO
        SATL
        SATH                    ;Right-justify ALOAHI
        BD      CHKSGN          ;Jump back after next two instructions
        SACL    ALO             ;Save normalized value
        SACH    AHI             ;in ALO and AHI

BGRT32  LACC    BHI             ;If exponent of B > 32
        SACL    CHI             ;then C <- B.
        RETD                    ;Return after
        LACC    BLO             ;saving CHI and CLO
        SACL    CLO

AGRT32  LACC    AHI             ;If exponent of A > 32
        SACL    CHI             ;then C <- A.
        LACC    ALO
        SACL    CLO             ;Copy ALO to CLO
        LACC    ASIGN
        SACL    CSIGN           ;Copy ASIGN to CSIGN
        RETD                    ;Return after
        LACC    AEXP            ;copying AEXP to CEXP
        SACL    CEXP
```

Example 7–19. Floating-Point Multiplication Using BSAR

```
    .title 'Floating Point Multiplication Routine'

*;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
*   THIS SUBROUTINE MULTIPLIES TWO FLOATING-POINT NUMBERS PRODUCING
*   A NORMALIZED FLOATING-POINT PRODUCT. THE FORMAT OF FLOATING-
*   POINT NUMBERS IS SPECIFIED BELOW.
*
*   INPUT / OUTPUT FORMAT
*   =====================
*   ----------------
*   |  ALL 0 OR 1  |   SIGN WORD
*   ----------------
*
*   ----------------
*   |   16 BITS    |   EXPONENT
*   ----------------
*
*   ----------------
*   |0|   15 BITS  |   HIGH PART OF MANTISSA
*   ----------------
*
*   ----------------
*   |   16 BITS    |   LOW PART OF MANTISSA
*   ----------------
*
*   NOTE THAT EVEN IF THE PRODUCT IS ZERO, SIGN OF THE PRODUCT MAY
*   EITHER BE POSITIVE OR NEGATIVE DEPENDING ON THE INPUTS.
*
*   Key C5x Instructions:
*   BSAR    1-16 bit right barrel arithmetic shift in one cycle
*   CLRC    reset control bit
*   SETC    set control bit
*   BD  branch after executing next two one-word instructions
*       or one two-word instruction
*;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

ASIGN   .set    60h         ;Sign, exponent, high and low parts of mantissa
AEXP    .set    61h         ;of input number A
AHI     .set    62h
ALO     .set    63h

BSIGN   .set    64h         ;Sign, exponent, high and low parts of mantissa
BEXP    .set    65h         ;of input number B
BHI     .set    66h
BLO     .set    67h
```

7.10 Application-Oriented Operations

7.10.1 Modem Application

Digital signal processors are especially appropriate for modem applications. The TMS320C5x devices with their enhanced instruction set and reduced instruction cycle time are particularly effective in implementing encoding and decoding algorithms. Features like circular addressing, repeat block, and single-cycle barrel shift reduce the execution time of such routines.

Example 7–20 implements a differential and convolutional encoder for a 9600-bit/s V.32 modem. This encoder uses trellis coding with 32 carrier states. The data stream to be transmitted is divided into groups of four consecutive data bits. The first two bits in time $Q1_n$ and $Q2_n$ in each group are differentially encoded into $Y1_n$ and $Y2_n$ according to the following equations:

$$Y1_n = Q1_n \oplus Y1_{n-1}$$

$$Y2_n = (Q1_n \bullet Y1_{n-1}) \oplus Y2_{n-1} \oplus Q2_n$$

This is done by a subroutine called DIFF. The two differentially encoded bits Y1n and Y2n are used as inputs to a convolutional encoder subroutine ENCODE, which generates a redundant bit Y0n. These five bits are packed into a single word by the PACK subroutine.

Example 7–20. V.32 Encoder Using Accumulator Buffer

```
        .title  'Convolutional Encoding for a V.32 Modem'

        .mmregs

STATMEM .set    60h             ;(60h - 62h) Delay States S1,S2,S3
INPUT   .set    64h             ;(64h - 67h) Four input bits
YPAST   .set    68h             ;(68h - 69h) Past values of Y1 and Y2
OUTPUT  .set    63h             ;Y0, the redundant bit
LOCATE  .set    6ah             ;Temporary storage for current input word
PCKD_IP .set    1000h           ;Input buffer (4 bits packed per word)
PCKD_OP .set    2000h           ;Output buffer (5 bits packed per word)
COUNT   .set    50              ;# of input data words

        .text

INIT    LAR     AR1,#PCKD_IP
        LAR     AR2,#PCKD_OP
        LAR     AR3,#COUNT-1    ;COUNT contains # of input words
        LDP     #0

START   MAR     *,AR1
        LACC    *+,0,AR0
        SACL    LOCATE          ;Temporary storage for current input word

        LAR     AR0,#INPUT+3
        LACL    #3              ;Loop 4 times
        SAMM    BRCR
        LACL    #1
        SAMM    DBMR            ;Load DBMR with the mask for LSB

UNPACK  LACC    LOCATE          ;Acc = packed input bits
        RPTB    LOOP1-1         ;for I=0,I<=3,I++
```

7.10.2 Adaptive Filtering

There are many practical applications of adaptive FIR/IIR filtering; one example is in the adapting or updating of coefficients. This can become computationally expensive and time-consuming. The MPYA, ZALR, and RPTB instructions on TMS320C5x can reduce execution time.

A means of adapting the coefficients on the TMS320C5x is the least-mean-square algorithm given by the following equation:

$$b_k\ (i+1) = b_k\ (i) + 2Be(i)x(i-k)$$

where $e\ (i) = x\ (i) - y\ (i)$
and

$$y(i) = \sum_{k=0}^{N-1} b_k\ x(i-k)$$

Quantization errors in the updated coefficients can be minimized if the result is obtained by rounding rather than truncating. For each coefficient in the filter at a given point in time, the factor 2*B*e(i) is a constant. This factor can then be computed once and stored in the T register for each of the updates.

MPYA and ZALR instructions help in reducing the number of instructions in the main adaptation loop. Furthermore, the RPTB (repeat block) instruction allows the block of instructions to be repeated without any penalty for looping.

Example 7–21 shows a routine that implements a 128-tap FIR filter and an LMS adaptation of its coefficients. The single-access internal RAM of TMS320C50/C51 can be mapped in both the program and data spaces at the same time by setting OVLY and RAM control flags to 1. This feature can be used to advantage by locating the coefficients table in single-access internal RAM so that it can be accessed by MACD and MPY instructions without modifying RAM configuration. Note that the MACD instruction requires one of its operands to be in program space.

If the address of the coefficient table is to be determined in runtime, load the BMAR (block move address register) with the address computed dynamically and replace the instruction

```
MACD    COEFFP,*–
by
MADD    *–
```

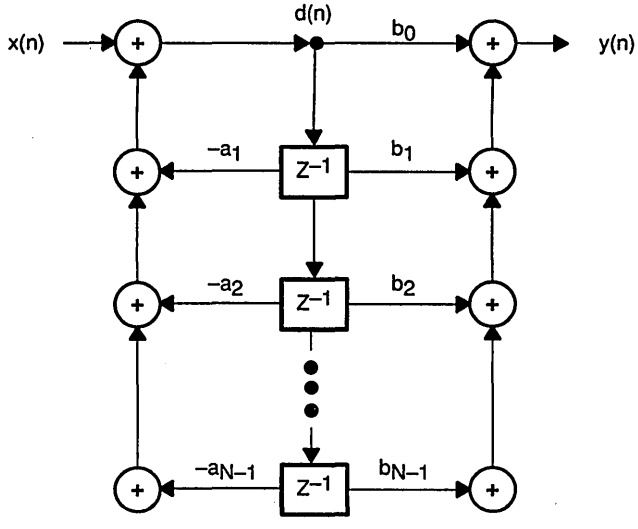
7.10.3 IIR Filters

Infinite impulse response (IIR) filters are widely used in digital signal processing applications. The transfer function of an IIR filter is given by:

$$H(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}} = \frac{Y(z)}{X(z)}$$

An Nth order direct-form II IIR filter can be represented by the following block diagram:

Figure 7–5. Nth Order Direct-Form Type II IIR Filter



In the time domain, an Nth order IIR filter is represented by the following two difference equations:

at time interval n:

x(n) is the current input sample

y(n) is the output of the IIR filter

$d(n) = x(n) - d(n-1)a_1 - \ldots - d(n-N+1)a_{N-1}$

$y(n) = d(n)b_0 + d(n-1)b_1 + \ldots + d(n-N+1)b_{N-1}$

The above two equations can easily be implemented on the TMS320C5x by using multiply-accumulate instructions (MAC, MACD, MADS, MADD). Note

Example 7–23. Using LTD and MPYA

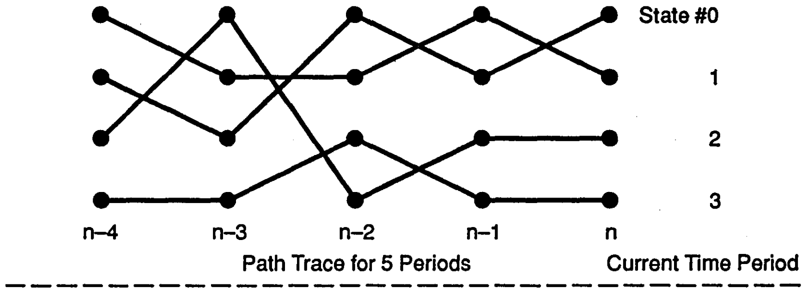
```
     .title "N Cascaded BiQuad IIR Filters"
     .mmregs

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine implements N cascaded blocks of biquad IIR
; canonic type II filters. Each biquad requires 3 data
; memory locations d(n),d(n-1),d(n-2), and 5 coefficients
; -a1,-a2,b0,b1,b2.
; For each block:  d(n)  = x(n)-d(n-1)a1-d(n-2)a2
;                  y(n)  = d(n)b0+d(n-1)b1+d(n-2)b2
;
; Coefficients Storage: (low to high data memory)
;    -a2,-a1,b2,b1,b0, ... ,-a2,-a1,b2,b1,b0
;       1st biquad             Nth biquad
;
; State Variables: (low to high data memory)
;    d(n),d(n-1),d(n-2), ... ,d(n),d(n-1),d(n-2)
;        Nth biquad             1st biquad
;
; Entry Conditions:
;    AR1 -> d(n-2) of 1st biquad
;    AR2 -> -a2 of 1st biquad
;    AR3 -> input sample (Q15 number)
;    AR4 -> output sample (Q15 number)
;    DP = 0, PM = 0, ARP = 3
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
BIQUAD: ;Setup variables;
     ZPR              ; Clear P register
     LACC *,15,AR1    ; Get Q15 input
     SPLK #2,INDX     ; Setup index register
     SPLK #N-1,BRCR   ; Setup count

     ;Begin computation;
     RPTB ELOOP-1     ; repeat for N biquads
LOOP:
     LT   *-,AR2      ; T = d(n-2)
     MPYA *+,AR1      ; Acc = x(n), P = -d(n-2)a2
     LTA  *-,AR2      ; Acc += -d(n-2)a2, T = d(n-1)
     MPY  *+          ; P = -d(n-1)a1
     LTA  *+,AR1      ; Acc += -d(n-1)a1, T = b2
     SACH *0+,1       ; Save d(n)
     MPY  *-          ; P = d(n-2)b2
     LACL #0          ; Acc = 0
     LTD  *-,AR2      ; T = d(n-1), d(n-2) = d(n-1)
     MPY  *+,AR1      ; Acc += d(n-2)b2, P = d(n-1)b1
     LTD  *-,AR2      ; T = d(n), d(n-1) = d(n)
     MPY  *+,AR1      ; Acc += d(n-1)b1, P = d(n)b0
ELOOP:
     LTA  *,AR4       ; Final accumulation
     SACH *,1         ; Save output in Q15 format
```

7.10.4 Dynamic Programming

Dynamic programming techniques are widely used in optimal search algorithms. Applications such as speech recognition, telecommunications, and robotics use dynamic programming algorithms. The TMS320C5x digital signal processors have an enhanced instruction set for efficient implementation of dynamic programming methods.

Most real-time search algorithms use the basic dynamic programming principle that the final optimal path from the start state to the goal state always pass through an optimal path from the start state to an intermediate state. This helps

Figure 7–6. Backtracking With Path History



Buffer

1	1	2	2	
0	0	0	1	
3	2	1	3	
2	3	3	0	Buffer + 15
n–1	n	n–3	n–2	

Current Time Period

Path History Circular Buffer (N = 4)

Example 7–24. Backtracking Algorithm Using Circular Addressing

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Backtracking Example
; This program back-tracks the optimal path expanded by
; a dynamic programming algorithm. The path history
; consists of four paths expanded N times. It is set up
; as a circular buffer of length N*4.
; Note that decrement type circular buffer is used.
; The start and end address of the circular buffer are
; initialized this way because of two reasons:
; 1- to avoid skipping the end-address of circ buffer
; 2- to ensure that wrap-around is complete before next
; iteration.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    LAR     AR0,#BUFFER ; get buffer address
    LMMR    INDX,PATH   ; get the selected path [0..3]
    SPLK    #N-1,BRCR   ; trace back N time periods
* init. AR0 as pointer to circular buffer#1; length=N*4 words
    SPLK    #BUFFER+(N-1)*4,CBSR1
    SPLK    #BUFFER-3,CBER1
    SPLK    #08h,CBCR
*
    RPTB    TLOOP-1     ; for i=0,i<N,i++
    MAR     *0+         ; offset by state#
    LACC    *0-         ; get next pointer & reset to state#0
    SAMM    INDX        ; save next state#
    SBRK    3           ; decrement AR0 to avoid skipping CBER1
    SBRK    1           ; now AR0 is correctly positioned 1 time
TLOOP:                  ; period back (circular addressing)
```

Figure 7–8. An In-Place DIT FFT With In-Order Inputs but Bit-Reversed Outputs

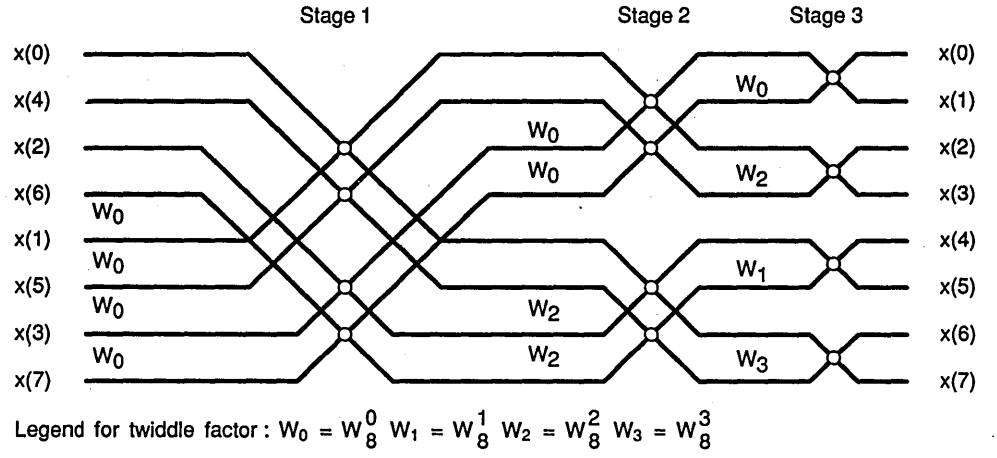


Table 7–1. Bit-Reversal Algorithm for an 8-Point Radix-2 DIT FFT

Index	Bit Pattern	Bit-Reversed Pattern	Bit-Reversed Index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

The bit-reversed addressing mode is part of the indirect addressing implemented with the auxiliary registers and the associated arithmetic unit. In this mode, a value (index) contained in INDX is either added to or subtracted from the auxiliary register being pointed to by the ARP. However, the carry bit is not propogated in the forward direction; instead, it is propagated in the reverse direction. The result is a scrambling in the address access.

The procedure for generating the bit-reversed address sequence is to load INDX with a value corresponding to one-half the length of the FFT and to load another auxiliary register—for example, AR1—with the base address of the data array. However, implementations of FFTs involve complex arithmetic; as a result, two data memory locations (one real and one imaginary) are associated with each data sample. For ease of addressing, the samples are stored in workspace memory in pairs with the real part in the even address locations and the imaginary part in the odd address locations. This means that the offset from the base address for any given sample is twice the sample index. If the incoming data is in the following form:

Example 7–25. Macros for 16-Point DIT FFT

```
*************************************************************************
* FILE: c5cxrad2.mac   --> macro file for radix 2 fft's based on 320c5x  *
*                                                                        *
* COPYRIGHT TEXAS INSTRUMENTS INC. 1990                                  *
*************************************************************************
*                                                                        *
* MACRO 'COMBO2X' FOR THE COMPLEX, RADIX-2 DIT FFT                       *
*                                                                        *
* ORGANIZATION OF THE INPUT DATA MEMORY: R1,I1,R2,I2,R3,I3,R4,I4          *
*                                                                        *
*************************************************************************
*                                                                        *
* THE MACRO 'COMBO2x' PERFORMS FOLLOWING CALCULATIONS:                   *
*                                                                        *
* R1  := [(R1+R2)+(R3+R4)]/4       INPUT              OUTPUT             *
* R2  := [(R1-R2)+(I3-I4)]/4       ----------------   ----------------   *
* R3  := [(R1+R2)-(R3+R4)]/4       AR0 =  7                              *
* R4  := [(R1-R2)-(I3-I4)]/4       AR1 -> R1,I1       AR1 - > R5,I5      *
* I1  := [(I1+I2)+(I3+I4)]/4       AR2 -> R2,I2       AR2 - > R6,I6      *
* I2  := [(I1-I2)-(R3-R4)]/4       ARP-> AR3 -> R3,I3 ARP - > AR3 - > R7,I7 *
* I3  := [(I1+I2)-(I3+I4)]/4       AR4 -> R4,I4       AR4 - > R8,I8      *
* I4  := [(I1-I2)+(R3-R4)]/4                                             *
*                                                                        *
* For a 16-point Radix 2 complex FFT the Macro 'COMBO2x' has to be       *
* repeated N/4 times (e.g. 4  times for a  16 point FFT).                *
*                                                                        *
*************************************************************************
;
COMBO5x $MACRO  num              ; REPEAT MACRO 'COMBO5x': N/4 times
        SPLK    #:num:-1,BRCR    ; execute 'num' times 'COMBO5x'
;
        RPTB    comboend  ;                                ARP AR1 AR2 AR3 AR4 AR5
                                                           --- --- --- --- --- ---
;
        LACC    *,14,AR4  ; ACC := (R3)/4                   4  R1  R2  R3  R4  T1
        SUB     *,14,AR5  ; ACC := (R3-R4)/4                5  R1  R2  R3  R4  T1
        SACH    *+,1,AR4  ; T1   = (R3-R4)/2                4  R1  R2  I3  R4  T2
;
        ADD     *+,15,AR5 ; ACC := (R3+R4)/4                5  R1  R2  R3  I4  T2
        SACH    *,1,AR2   ; T2   = (R3+R4)/2                2  R1  R2  R3  I4  T2
;
        ADD     *,14,AR1  ; ACC := (R2+R3+R4)/4             1  R1  R2  R3  I4  T2
        ADD     *,14      ; ACC := (R1+R2+R3+R4)/4          1  R1  R2  R3  I4  T2
        SACH    *+,0,AR5  ; R1  := (R1+R2+R3+R4)/4          5  I1  R2  R3  I4  T2
        SUB     *,16,AR3  ; ACC := (R1+R2-(R3+R4))/4        3  I1  R2  R3  I4  T2
        SACH    *+,0,AR5  ; R3  := (R1+R2-(R3+R4))/4        5  I1  R2  I3  I4  T2
;
        ADD     *,15,AR2  ; ACC := (R1+R2)/4                2  I1  R2  I3  I4  T2
        SUB     *,15,AR3  ; ACC := (R1-R2)/4                3  I1  R2  I3  I4  T2
        ADD     *,14,AR4  ; ACC := ((R1-R2)+(I3))/4         4  I1  R2  I3  I4  T2
        SUB     *,14,AR2  ; ACC := ((R1-R2)+(I3-I4))/4      2  I1  R2  I3  I4  T2
        SACH    *+,0,AR4  ; R2  := ((R1-R2)+(I3-I4))/4      4  I1  I2  I3  I4  T2
        ADD     *-,15,AR3 ; ACC := ((R1-R2)+ I3+I4 )/4      3  I1  I2  I3  R4  T2
        SUB     *,15,AR4  ; ACC := ((R1-R2)-(I3-I4))/4      4  I1  I2  I3  R4  T2
        SACH    *+,0,AR1  ; R4  := ((R1-R2)-(I3-I4))/4      1  I1  I2  I3  I4  T2
;
        LACC    *,14,AR2  ; ACC := (I1)/4                   2  I1  I2  I3  I4  T2
        SUB     *,14,AR5  ; ACC := (I1-I2)/4                5  I1  I2  I3  I4  T2
        SACH    *,1,AR2   ; T2  := (I1-I2)/2                2  I1  I2  I3  I4  T2
        ADD     *,15,AR3  ; ACC := ((I1+I2))/4              4  I1  I2  I3  I4  T2
        ADD     *,14,AR4  ; ACC := ((I1+I2)+(I3))/4         4  I1  I2  I3  I4  T2
        ADD     *,14,AR1  ; ACC := ((I1+I2)+(I3+I4))/4      1  I1  I2  I3  I4  T2
        SACH    *0+,0,AR3 ; I1  := ((I1+I2)+(I3+I4))/4      3  R5  I2  I3  I4  T2
        SUB     *,15,AR4  ; ACC := ((I1+I2)-(I3+I4))/4      4  R5  I2  I3  I4  T2
        SUB     *,15,AR3  ; ACC := ((I1+I2)-(I3+I4))/4      3  R5  I2  I3  I4  T2
        SACH    *0+,0,AR5 ; I3  := ((I1+I2)-(I3+I4))/4      5  R5  I2  R7  I4  T2
;
        LACC    *-,15     ; ACC := (I1-I2)/4                5  R5  I2  R7  I4  T1
```

```
*       QR'= PR - (W*QI + W*QR) = PR - W * QI - W * QR    (<- AR2)          *
*       PI'= PI + (W*QI - W*QR) = PI + W * QI - W * QR    (<- AR1+1)        *
*       QI'= PI - (W*QI - W*QR) = PI - W * QI + W * QR    (<- AR1+2)        *
*                                                                           *
*****************************************************************************
;
PBY4J   $MACRO              ; TREG= W                    AR5   PREG   AR1  AR2 ARP
        MPY     *+,AR5      ; PREG= W*QR/2                -    W*QR/2 PR   QI   5
        SPH     *,AR1       ; TMP = W*QR/2              W*QR/2 W*QR/2 PR   QI   1
        LACC    *,15,AR2    ; ACC = PR/2                W*QR/2 W*QR/2 PR   QI   2
        MPYS    *-          ; ACC = (PR-W*QR)/2         W*QR/2 W*QI/2 PR   QR   2
        SPAC                ; ACC = (PR-W*QI-W*QR)/2    W*QR/2 W*QI/2 PR   QR   2
        SACH    *+,0,AR1    ; QR  = (PR-W*QI-W*QR)/2    W*QR/2 W*QI/2 PR   QI   1
        SUB     *,16        ; ACC = (-PR-W*QI-W*QR)/2   W*QR/2 W*QI/2 PR   QI   1
        NEG                 ; ACC =  (PR+W*QI+W*QR)/2   W*QR/2 W*QI/2 PR   QI   1
        SACH    *+          ; QR  =  (PR+W*QI+W*QR)/2   W*QR/2 W*QI/2 PI   QI   1

        LACC    *,15,AR5    ; ACC = (PI)/2              W*QR/2 W*QI/2 PI   QI   5
        SPAC                ; ACC = (PI-W*QI)/2         W*QR/2   -    PI   QI   5
        ADD     *,16,AR2    ; ACC = (PI-W*QI+W*QR)/2      -      -    PI   QI   2
        SACH    *+,0,AR1    ; QI  = (PI-W*QI+W*QR)/2      -      -    PI   QR1  1
        SUB     *,16        ; ACCU= (-PI-W*QI+W*QR)/2     -      -    PI   QR1  1
        NEG                 ; ACCU=  (PI+W*QI-W*QR)/2     -      -    PI   QR1  1
        SACH    *+,0,AR2    ; PI  =  (PI+W*QI-W*QR)/2     -      -    PR1  QR1  2
        $ENDM
*
*****************************************************************************
*                                                                           *
*     MACRO 'P3BY4J'   number of words: 16                                  *
*                                                                           *
*   ENTRANCE IN THE MACRO: ARP=AR2                                          *
*                          AR1->PR,PI                                       *
*                          AR2->QR,QI                                       *
*                          TREG=W=COS(45)=SIN(45)                           *
*                                                                           *
*       PR'= PR + (W*QI - W*QR) = PR + W * QI - W * QR    (<- AR1)          *
*       QR'= PR - (W*QI - W*QR) = PR - W * QI + W * QR    (<- AR2)          *
*       PI'= PI - (W*QI + W*QR) = PI - W * QI - W * QR    (<- AR1+1)        *
*       QI'= PI + (W*QI + W*QR) = PI + W * QI + W * QR    (<- AR1+2)        *
*                                                                           *
*      EXIT OF THE MACRO: ARP=AR2                                           *
*                         AR1->PR+1,PI+1                                    *
*                         AR2->QR+1,QI+1                                    *
*                                                                           *
*****************************************************************************
P3BY4J  $MACRO              ; TREG= W                    AR5   PREG   AR1  AR2 ARP
        MPY     *+,AR5      ; PREG= W*QR/2                -    W*QR/2 PR   QI   5
        SPH     *,AR1       ; TMP = W*QR/2              W*QR/2 W*QR/2 PR   QI   1
        LACC    *,15,AR2    ; ACC = PR/2                W*QR/2 W*QR/2 PR   QI   2
        MPYA    *-          ; ACC = (PR+W*QR)/2         W*QR/2 W*QI/2 PR   QR   2
        SPAC                ; ACC = (PR-W*QI+W*QR)/2    W*QR/2 W*QI/2 PR   QR   2
        SACH    *+,0,AR1    ; QR' = (PR-W*QI+W*QR)/2    W*QR/2 W*QI/2 PR   QI   1
        SUB     *,16        ; ACC = (-PR-W*QI+W*QR)/2   W*QR/2 W*QI/2 PR   QI   1
        NEG                 ; ACC =  (PR+W*QI-W*QR)/2   W*QR/2 W*QI/2 PR   QI   1
        SACH    *+          ; PR' =  (PR+W*QI-W*QR)/2   W*QR/2 W*QI/2 PI   QI   1

        LACC    *,15,AR5    ; ACC = (PI)/2              W*QR/2 W*QI/2 PI   QI   5
        APAC                ; ACC = (PI+W*QI)/2         W*QR/2   -    PI   QI   5
        ADD     *,16,AR2    ; ACC = (PI+W*QI+W*QR)/2      -      -    PI   QI   2
        SACH    *0+,0,AR1   ; QI' = (PI+W*QI+W*QR)/2      -      -    PI   QR5  1
        SUB     *,16        ; ACCU= (-PI+W*QI+W*QR)/2     -      -    PI   QR5  1
        NEG                 ; ACCU=  (PI-W*QI-W*QR)/2     -      -    PI   QR5  1
        SACH    *0+,0,AR2   ; PI' =  (PI-W*QI-W*QR)/2     -      -    PR5  QR5  2
        $ENDM
;
*****************************************************************************
*                                                                           *
*     MACRO 'stage3'   number of words: 54                                  *
*                                                                           *
```

Example 7–26. Initialization Routine

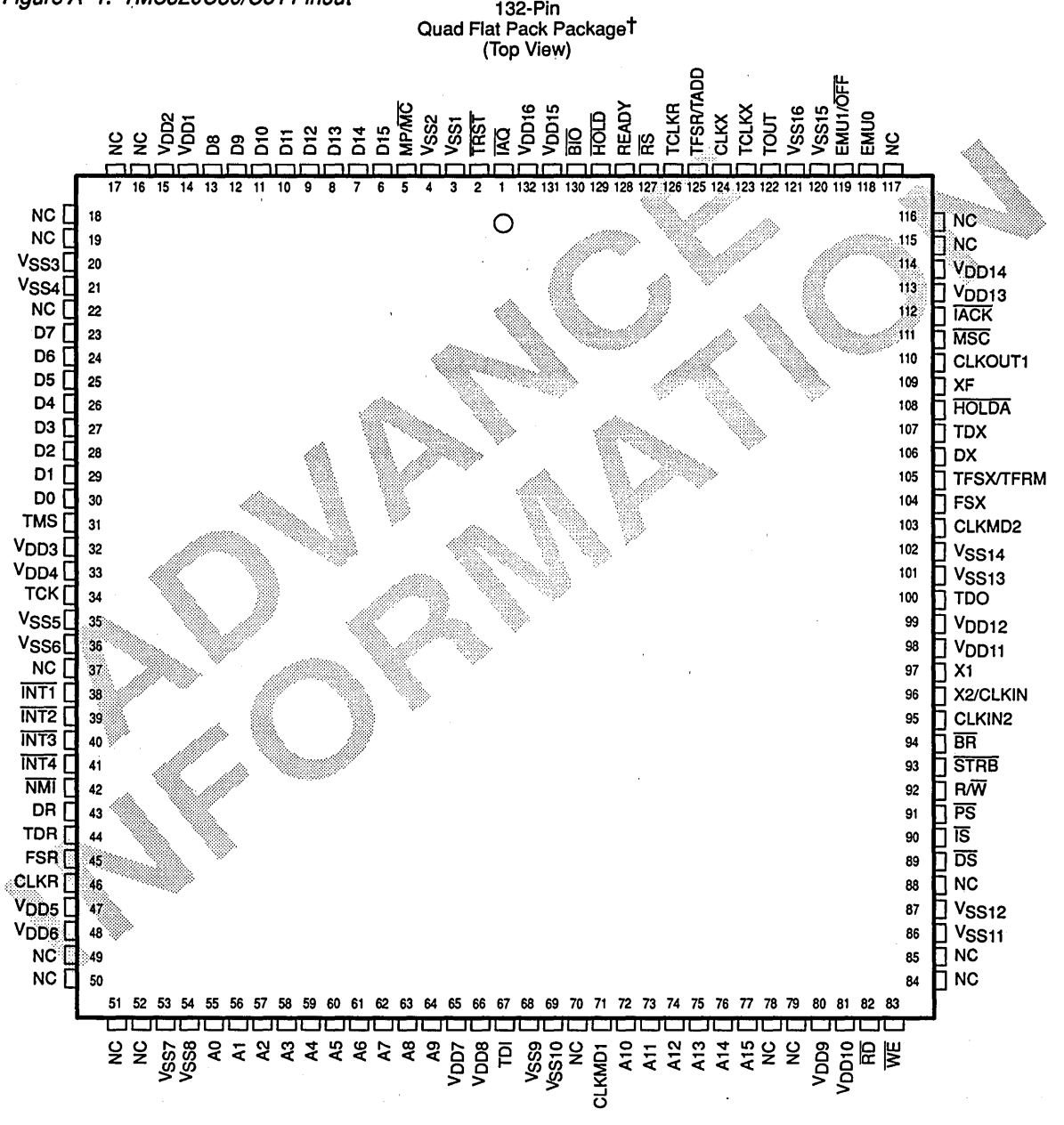
```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  file: INIT-FFT.ASM
;
;  Initialized  variables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
           .bss    NN,1                ; number of fft-points
           .bss    NN2,1               ; 2*N-1
           .bss    DATAADD,1           ; START ADDRESS OF DATA
           .bss    cos45,1
           .bss    sin4,1              ; start of sine in stage    4
           .bss    cos4,1              ; start of cosine in stage  4

;  Temp variables
;
           .bss    TEMP,2              ; used for temporary numbers
;
           .sect   "vectors"
           B       INIT,*,AR0

           .sect   "init"
TABINIT:   .word   N,N-1,2*N-1,DATA
           .word   5A82h               ; cos(45)=sin(45)
           .word   TWID,TWID+4
TABEND:    .set    $
;
INIT:      LDP     #0      ; use only B2 and mmregs for direct addressing
           SPM     0                   ; no shift from PREG to ALU
           CLRC    OVM                 ; disable overflowmode
           SETC    SXM                 ; enable sign extension mode
           SPLK    #pmstmask,PMST      ; ndx=trm=1
;
; INIT Block B2
;
           LAR     AR0,#NN             ; arp is already pointing to ar0
           LACC    #TABINIT
           RPT     #TABEND-TABINIT
           TBLR    *+
;
; INIT TWIDDLE FACTORS
;
           LAR     AR0,#TWID           ; arp is already pointing to ar0
           LACC    #TWIDSTRT
           RPT     #TWIDLEN
           TBLR    *+
;
; EXECUTE THE FFT
;
           LAR     AR5,#TEMP           ; pointer to 2 temp register
           CALL    FFT,*,AR3           ; ARP=AR3 FOR MACRO COMBO
;
WAIT       RET                         ; Return
;
```

```
          .sect     "fftprogram"
;
;       FFT CODE WITH BIT-REVERSED INPUT SAMPLES / ARP=AR3
;
FFT:      LAR       AR3,DATAADD   ; TRANSFER  32 WORDS FROM 'input' to 'data'
          LACC      NN
          SAMM      INDX          ; indexregister = N
          RPT       NN2           ; N TIMES
          BLDD      #INPUT,*BR0+
;
;         FFT CODE for STAGES 1 and 2
;
STAGE1:   SPLK      #7,INDX       ; indexregister = 7
          LAR       AR1,DATAADD   ; pointer to DATA       r1,i1
          LAR       AR2,#DATA+2   ; pointer to DATA + 2   r2,i2
          LAR       AR3,#DATA+4   ; pointer to DATA + 4   r3,i3
          LAR       AR4,#DATA+6   ; pointer to DATA + 6   r4,i4
          COMBO5X 4               ; repeat 4 times
;
;    FFT CODE FOR STAGE 3  /  ARP=AR2
;
STAGE3:   SPLK      #9,INDX       ; index register = 9
          LAR       AR1,DATAADD   ; ar1 -> DATA
          LAR       AR2,#DATA+8   ; ar2 -> DATA+8
          stage3    2             ; repeat 2 times
;
;    FFT CODE FOR STAGE 4  / ARP=ARP
;
STAGE4:   SPLK      #1,INDX       ; index register = 1
          LAR       AR1,DATAADD   ;
          LAR       AR2,#DATA+16  ;
          LAR       AR3,cos4      ; start of cosine in stage 4
          LAR       AR4,sin4      ; start of sine in stage   4
          SPLK      #6,BRCR
          ZEROI                   ; execute ZEROI
          BUTTFLYI                ; execute 7 times BUTTFLYI
          RET
END:      .set      $
FFTLEN    .set      END-FFT+1
          .end
```

A Electrical Specifications

A.1 Pinout and Signal Descriptions

Figure A–1. TMS320C50/C51 Pinout



† See Pin Assignments, Table A–1 (page A-3) for location and description of all pins. The TMS320C50 and TMS320C51 will be packaged in 132-pin plastic QFP in production. See Figure A–18 for mechanical data.

Note: NC = No connect. (These pins are reserved.)

Table A–1. TMS320C50/C51 Pin Assignments (Continued)

Pin	Name	Type	Description
35	V_{SS}	Supply	Ground
36	V_{SS}	Supply	Ground
37	NC†		Reserved
38	$\overline{INT1}$	I	Interrupt #1
39	$\overline{INT2}$	I	Interrupt #2
40	$\overline{INT3}$	I	Interrupt #3
41	$\overline{INT4}$	I	Interrupt #4
42	$\overline{NMI}$	I	Nonmaskable Interrupt
43	DR	I	Serial Port 1 Data Receive
44	TDR	I	Serial Port 2 Data Receive
45	FSR	I	Serial Port 1 Receiver Frame Sync
46	CLKR	I	Serial Port 1 Receiver Clock
47	V_{DD}	Supply	+5 V
48	V_{DD}	Supply	+5 V
49	NC†		Reserved
50	NC†		Reserved
51	NC†		Reserved
52	NC†		Reserved
53	V_{SS}	Supply	Ground
54	V_{SS}	Supply	Ground
55	A0 (LSB)	I/O/Z	Parallel Port Address Bus
56	A1	I/O/Z	(10 pins)
57	A2	I/O/Z	
58	A3	I/O/Z	
59	A4	I/O/Z	
60	A5	I/O/Z	
61	A6	I/O/Z	
62	A7	I/O/Z	
63	A8	I/O/Z	
64	A9	I/O/Z	
65	V_{DD}	Supply	+5 V
66	V_{DD}	Supply	+5 V
67	TDI	I	JTAG Scan Input

† NC = No connect

Table A–1. TMS320C50/C51 Pins (Concluded)

Pin	Name	Type	Description
101	V_{SS}	Supply	Ground
102	V_{SS}	Supply	Ground
103	CLKMD2	I	Clock Mode Pin 2
104	FSX	I/O/Z	Serial Port 1 Transmitter Frame Sync
105	TFSX/TFRM	I/O/Z	Serial Port 2 Transmitter Frame Sync
106	DX	O/Z	Serial Port 1 Transmitter Output
107	TDX	O/Z	Serial Port 2 Transmitter Output
108	$\overline{HOLDA}$	O/Z	Hold Acknowledge
109	XF	O/Z	External Flag
110	CLKOUT1	O/Z	Machine Clock Output
111	$\overline{MSC}$	O/Z	Microstate Complete
112	$\overline{IACK}$	O/Z	Interrupt Acknowledge
113	V_{DD}	Supply	+5 V
114	V_{DD}	Supply	+5 V
115	NC†		Reserved
116	NC†		Reserved
117	NC†		Reserved
118	EMU0	I/O/Z	Emulator Interrupt 0
119	EMU1/$\overline{OFF}$	I/O/Z	Emulator Interrupt 1
120	V_{SS}	Supply	Ground
121	V_{SS}	Supply	Ground
122	TOUT	O/Z	Timer Output
123	TCLKX	I/O/Z	Serial Port 2 Transmitter Clock
124	CLKX	I/O/Z	Serial Port 1 Transmitter Clock
125	TFSR/TADD	I/O/Z	Serial Port 2 Receive Frame/Address
126	TCLKR	I	Serial Port 2 Receiver Clock
127	$\overline{RS}$	I	Device Reset
128	READY	I	External Access Ready to Complete
129	$\overline{HOLD}$	I	Request Access of Local Memory
130	$\overline{BIO}$	I	Bit I/O Pin
131	V_{DD}	Supply	+5 V
132	V_{DD}	Supply	+5 V

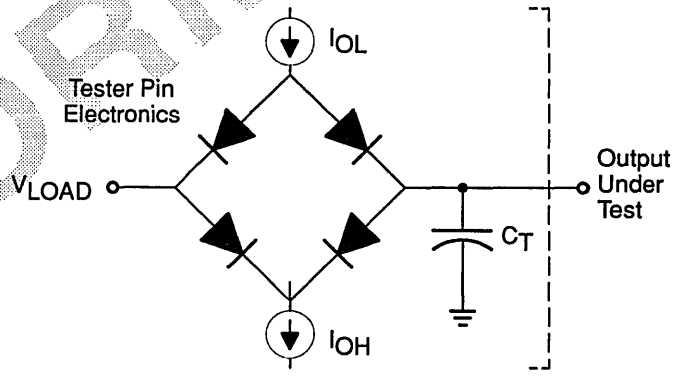
† NC = No connect

Table A–4. Electrical Characteristics Over Specified Free-Air Temperature Range (Unless Otherwise Noted)

Parameter		Test Conditions	Min	Typ†	Max	Unit
V_{OH}	High-level output voltage §	V_{DD}=Min, I_{OH}=Max	2.4	3		V
V_{OL}	Low-level output voltage §	V_{DD}=Min, I_{OL}=Max		0.3	0.6	V
I_Z	Three-state current (V_{DD} = Max)	$\overline{BR}$	–300	‡	20	µA
		All other three-state	–20	‡	20	
I_I	Input current (V_I=V_{SS} to V_{DD})	$\overline{TRST}$ pin	–10	‡	300	µA
		X2CLKIN pin	–10	‡	10	
		All other input only pins	–300	‡	10	
I_{DDC}	Supply current, Core CPU	Operating T_A=0°C, V_{DD}=5.25 V, f_x=40.96 MHz			60	mA
I_{DDP}	Supply current, pins	Operating T_A=0°C, V_{DD}=5.25 V, f_x=40.96 MHz			40	mA
I_{DD}	Supply current, power down modes	IDLE			¶	mA
		IDLE2		500		µA
C_i	Input capacitance			15		pF
C_o	Output capacitance			15		pF

† All typical nominal values are at V_{DD}=5 V, T_A=25°C.
‡ These values are not specified, pending detailed characterization.
§ All input and output voltage levels are TTL-compatible. Figure A-2 shows the test load circuit and Figure A-3 shows the voltage reference levels.
¶ Dependent upon which peripherals are active.

Figure A–2. Test Load Circuit



Where: I_{OL} = 2.0 mA (all outputs)
I_{OH} = 300 µA (all outputs)
V_{LOAD} = 2.15 V
C_T = 80 pF typical load circuit capacitance.

A.3 Clock Characteristics and Timing

The TMS320C50/C51 can use either its internal oscillator or an external frequency source for a clock. The clock mode is determined by the CLKMD1 (pin 71) and CLKMD2 (pin 103) clock mode pins. The following table outlines the selection of the clock mode by these pins.

CLKMD1	CLKMD2	Clock Source
1	0	External divide-by-one clock option.
0	1	Reserved for test purposes.
1	1	External divide-by-two option or internal divide-by-two clock option with an external crystal.
0	0	External divide-by-two option with the internal oscillator disabled.

A.3.1 Internal Divide-by-Two Clock Option With External Crystal

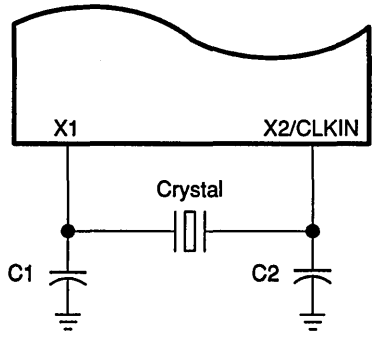
The internal oscillator is enabled by connecting a crystal across X1 and X2/CLKIN. The frequency of CLKOUT1 is one-half the crystal's oscillating frequency. The crystal should be in either fundamental or overtone operation and parallel resonant, with an effective series resistance of 30 ohms and a power dissipation of 1 mW; it should be specified at a load capacitance of 20 pF. Note that overtone crystals require an additional tuned-LC circuit. Figure A-4 shows an external crystal (fundamental frequency) connected to the on-chip oscillator.

Table A–5. Recommended Operating Conditions

Parameter	Test Conditions	Min	Nom	Max	Unit
f_X Input clock frequency	T_A=0° to 70°C	0§		40.96	MHz
C1, C2	T_A=0° to 70°C		10		pF

§ To preserve the internal state of the processor when f_X = 0 Hz, the input clock can be stopped only when both CLKIN and CLKOUT1 are high. In IDLE2 mode, clocks are guaranteed to be stopped properly internal to the device. Therefore, in IDLE2 mode, this constraint is not required.

Figure A–4. Internal Clock Option



A.3.3 External Divide-by-One Clock Option

An external frequency source can be used by injecting the frequency directly into CLKIN2, with X1 left unconnected and X2 connected to V_{DD}. This external frequency is divided by one to generate the internal machine cycle. The divide-by-one option is used when the CLKMD1 pin is strapped high and CLKMD2 is strapped low.

The external frequency injected must conform to specifications listed in the timing requirements table.

Table A–8. Switching Characteristics Over Recommended Operating Conditions (H = 0.5 $t_{c(CO)}$)

	Parameter	Min	Typ	Max	Unit
$t_{c(CO)}$	CLKOUT1 cycle time	48.8	$t_{c(CI)}$		ns
$t_{d(CIH-CO)}$	CLKIN2 low to CLKOUT1 high	10	15	25	ns
$t_{f(CO)}$	CLKOUT1 fall time		5		ns
$t_{r(CO)}$	CLKOUT1 rise time		5		ns
$t_{w(COL)}$	CLKOUT1 low pulse duration	H – 6			ns
$t_{w(COH)}$	CLKOUT1 high pulse duration	H – 7			ns

Table A–9. Timing Requirements Over Recommended Operating Conditions (H = 0.5 $t_{c(CO)}$)

	Parameter	Min	Max	Unit
$t_{c(CI)}$	CLKIN2 cycle time	48.8	‡	ns
$t_{f(CI)}$	CLKIN2 fall time †		5	ns
$t_{r(CI)}$	CLKIN2 rise time †		5	ns
$t_{w(CIL)}$	CLKIN2 low pulse duration, $t_{c(CI)}$ = Min	22		ns
$t_{w(CIH)}$	CLKIN2 high pulse duration, $t_{c(CI)}$ = Min	22		ns
tp	Transitory phase—PLL synchronized after CLKIN2 supplied.	256		cycles
Duty Cycle	$\frac{t_{w(CIL)} + t_{r(CI)}}{t_{c(CI)}}$	35	65	%

† Values deduced from characterization data and not tested.

‡ To preserve the internal state of the processor when f_X = 0 Hz, the input clock can be stopped only when both CLKIN2 and CLKOUT1 are high. In IDLE2 mode, clocks are guaranteed to be stopped properly internal to the device. Therefore, in IDLE2 mode, this constraint is not required.

Figure A–6. Internal Divide-by-One Clock Timing

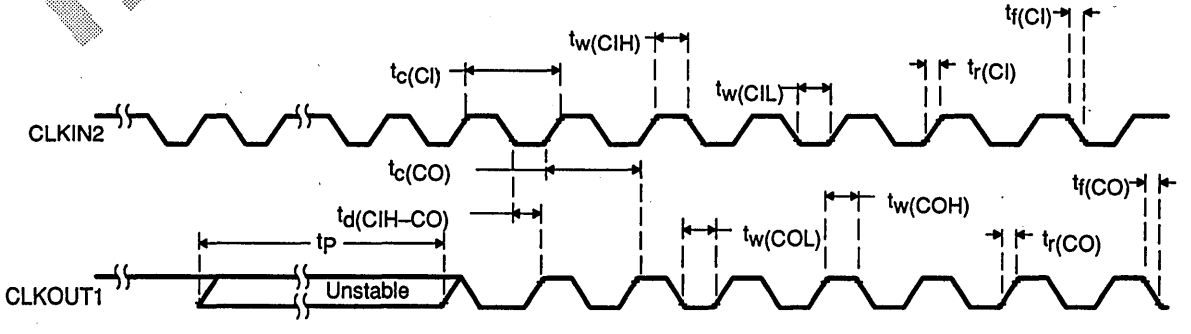
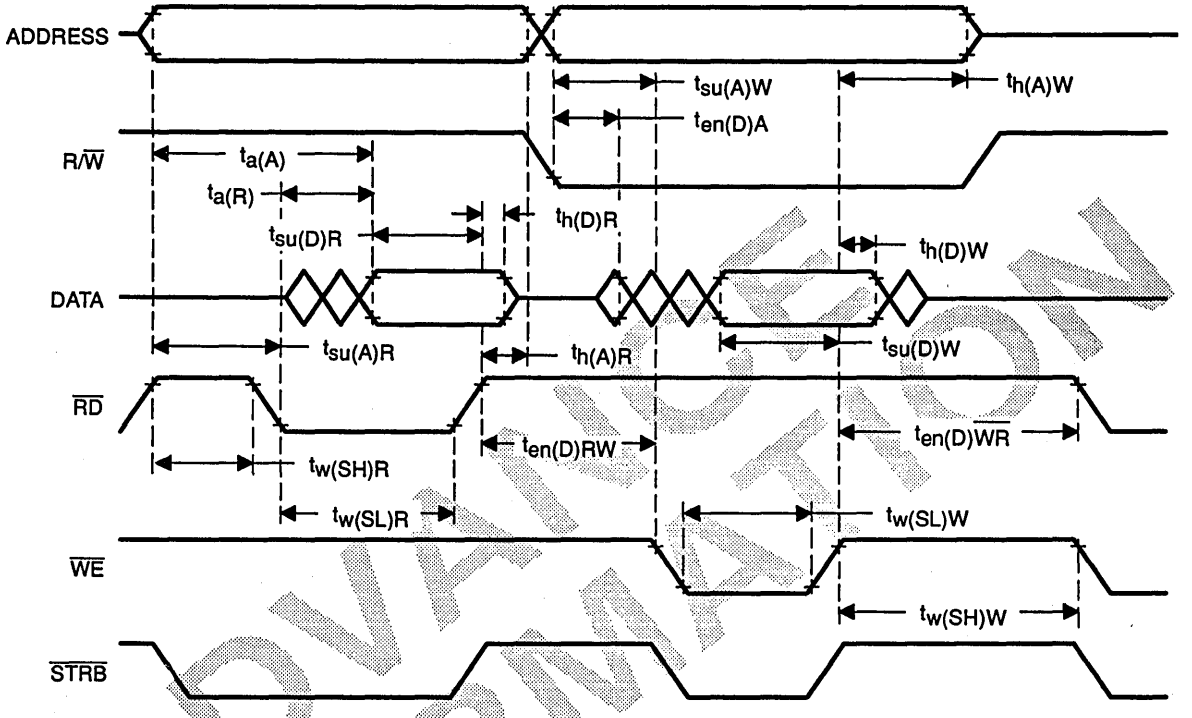
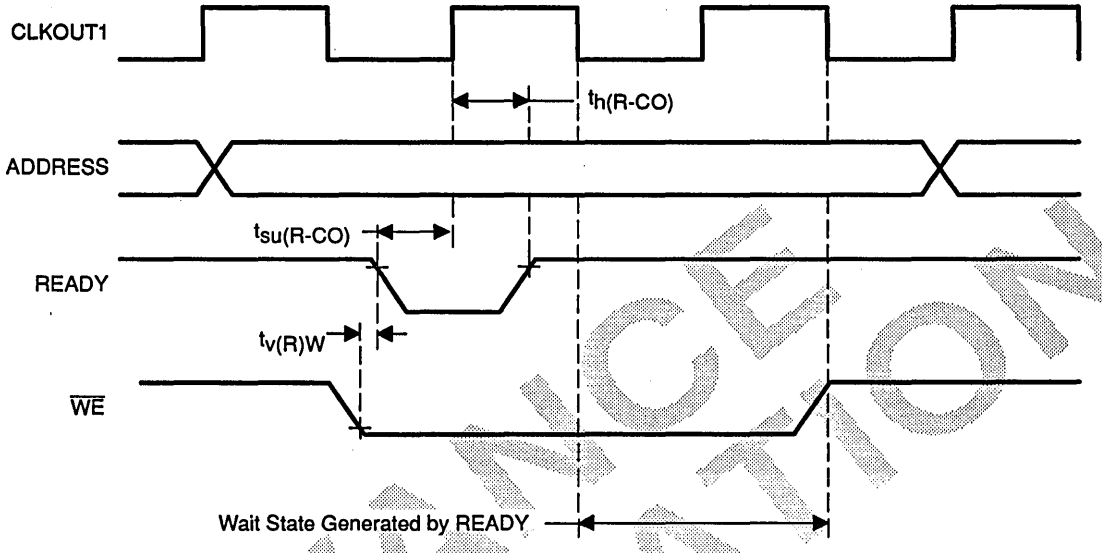


Figure A–7. Memory and Parallel I/O Interface Read and Write Timing



Note: All timings are for 0 wait states. However, external writes always require two cycles to prevent external bus conflicts. The above diagram illustrates a one-cycle read and a two-cycle write and is not drawn to scale. All external writes immediately preceded by an external read or immediately followed by an external read require three machine cycles.

Figure A–9. Ready Timing for Externally Generated Wait States During an External Write Cycle



A.3.8 Instruction Acquisition ($\overline{IAQ}$), Interrupt Acknowledge ($\overline{IACK}$), and External Flag (XF) Timings

Table A–15. Switching Characteristics Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$)

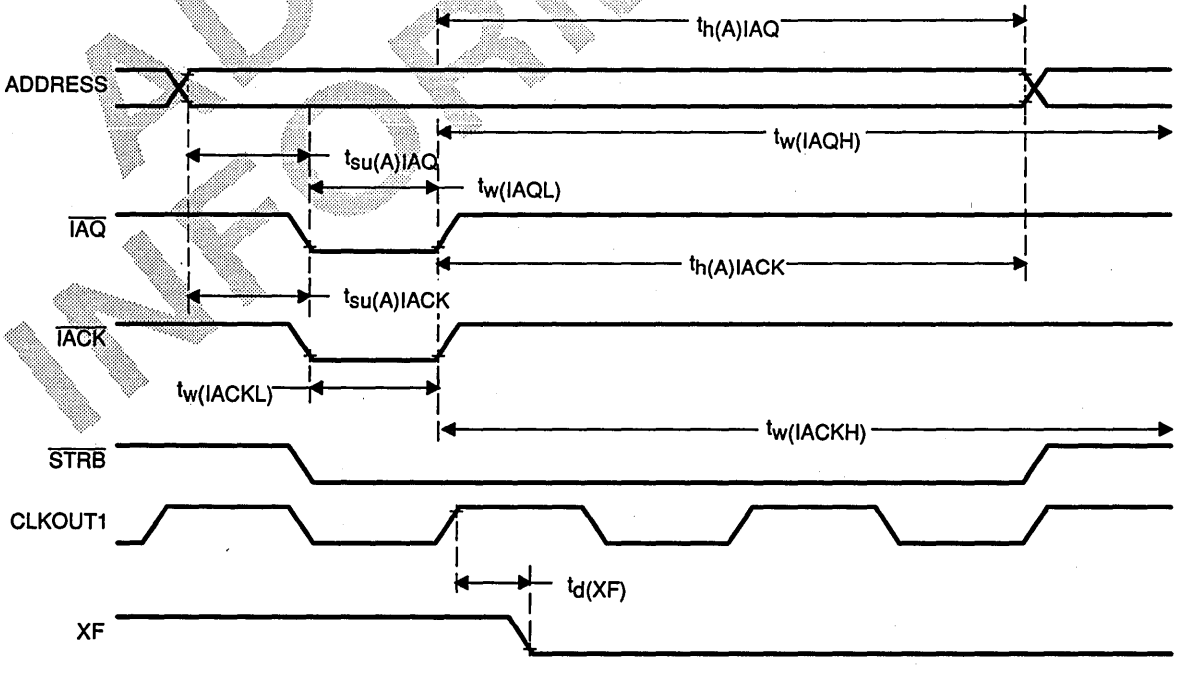
	Parameter	Min	Max	Unit
$t_{su(A)IAQ}$	Setup time, address valid before $\overline{IAQ}$ low †	H – 8		ns
$t_{h(A)IAQ}$	Hold time, address valid after $\overline{IAQ}$ high	H – 8		ns
$t_{w(IAQL)}$	$\overline{IAQ}$ low pulse duration	H – 8		ns
$t_{w(IAQH)}$	$\overline{IAQ}$ high pulse duration	H – 8	§	ns
$t_{su(A)IACK}$	Setup time, address valid before $\overline{IACK}$ low ‡	H – 8		ns
$t_{h(A)IACK}$	Hold time, address valid after $\overline{IACK}$ high ‡	H – 8		ns
$t_{w(IACKL)}$	$\overline{IACK}$ low pulse duration	H – 8		ns
$t_{w(IACKH)}$	$\overline{IACK}$ high pulse duration	H – 8		ns
$t_{d(XF)}$	Delay time, XF valid after CLKOUT1	0	10	ns

† $\overline{IAQ}$ goes low during an instruction acquisition. It goes low only on the first cycle of the read when wait states are used. The falling edge should be used to latch the valid address. The AVIS bit in the PMST register must be set to zero for the address to be valid when the instruction being addressed resides in on-chip memory.

‡ $\overline{IACK}$ goes low during the fetch of the first word of the interrupt vector. It goes low only on the first cycle of the read when wait states are used. Address pins A1 – A4 can be decoded at the falling edge to identify the interrupt being acknowledged. The AVIS bit in the PMST register must be set to zero for the address to be valid when the vectors reside in on-chip memory.

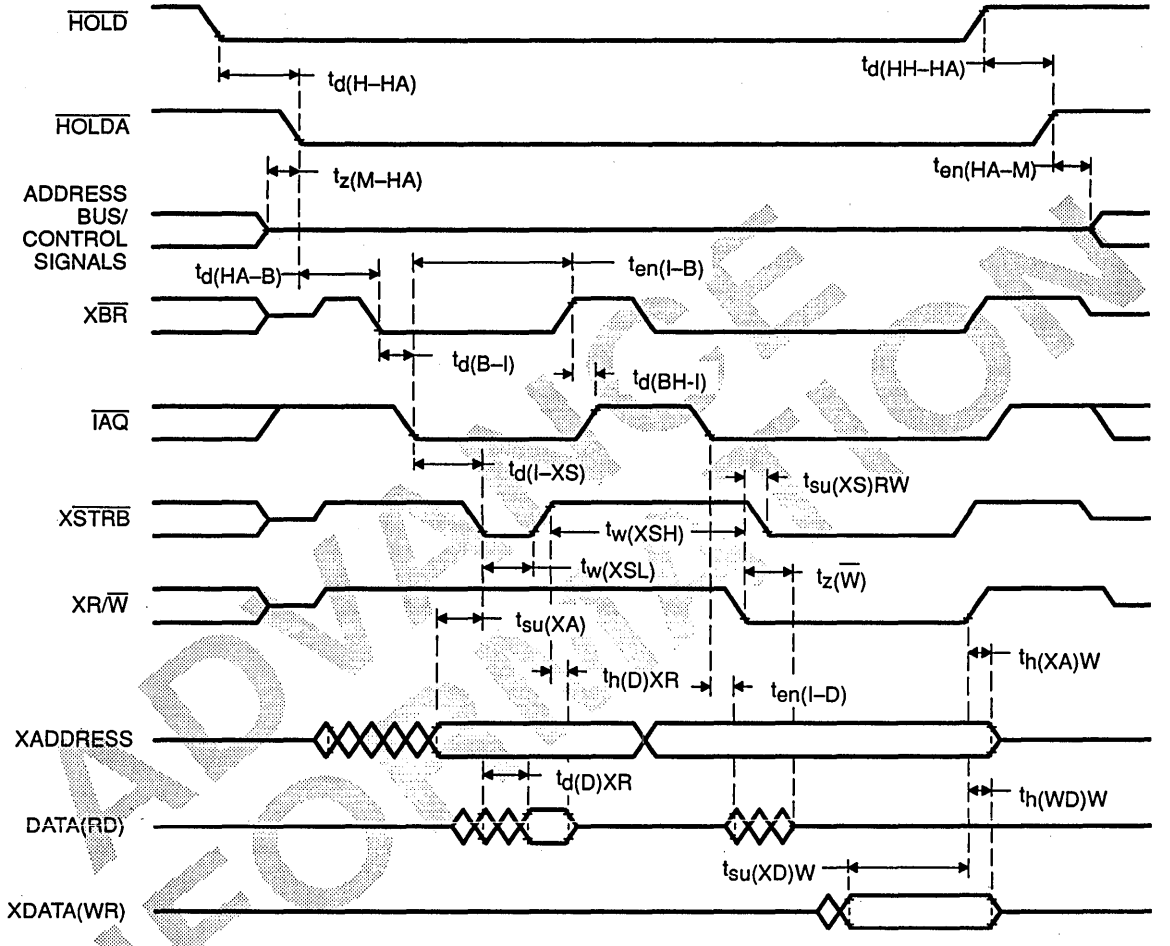
§ Software dependent on instruction cycle count of current instruction being executed.

Figure A–11. $\overline{IAQ}$, $\overline{IACK}$, and XF Timings Example With Two External Wait States



Note: $\overline{IAQ}$ and $\overline{IACK}$ are not affected by wait states.

Figure A–12. External DMA Timing



A.3.11 Serial Port Transmit Timing With External Clocks and Frames

Table A–19. Switching Characteristics Over Recommended Operating Conditions ($S = 0.5t_{c(SCK)}$)

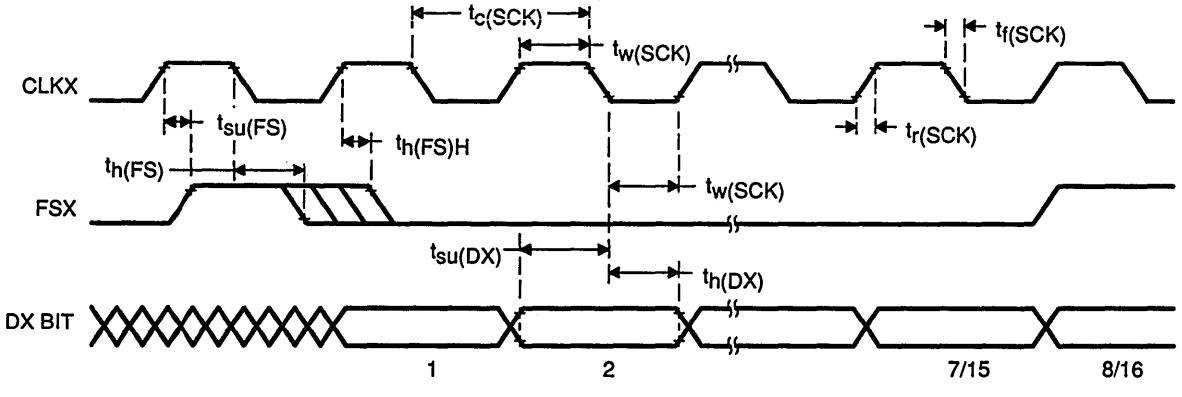
	Parameter	Min	Max	Unit
$t_{su(DX)}$	Setup time, DX valid before CLKX falling	S – 10		ns
$t_{h(DX)}$	Hold time, DX valid after CLKX falling	S – 5		ns

Table A–20. Timing Requirements Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$)

	Parameter	Min	Max	Unit
$t_{c(SCK)}$	Serial port clock cycle time	5.2H		ns
$t_{f(SCK)}$	Serial port clock fall time		8	ns
$t_{r(SCK)}$	Serial port clock rise time		8	ns
$t_{w(SCK)}$	Serial port clock low/high pulse duration	2.1H		ns
$t_{su(FS)}$	FSX setup time before CLKX rising edge	–(2H–8)		ns
$t_{h(FS)}$	FSX hold time after CLKX falling edge	10		ns
$t_{h(FS)H}$	FSX hold time after CLKX rising edge		2H–8 †	ns

† If the FSX pulse does not meet this specification, the first bit of serial data will be driven on the DX pin until the falling edge of FSX. After the falling edge of FSX, data will be shifted out on the DX pin. The transmit buffer empty interrupt will be generated when the $t_{h(RS)}$ and $t_{h(RS)H}$ specification is met.

Figure A–14. Serial Port Transmit Timing With External Clocks and Frames



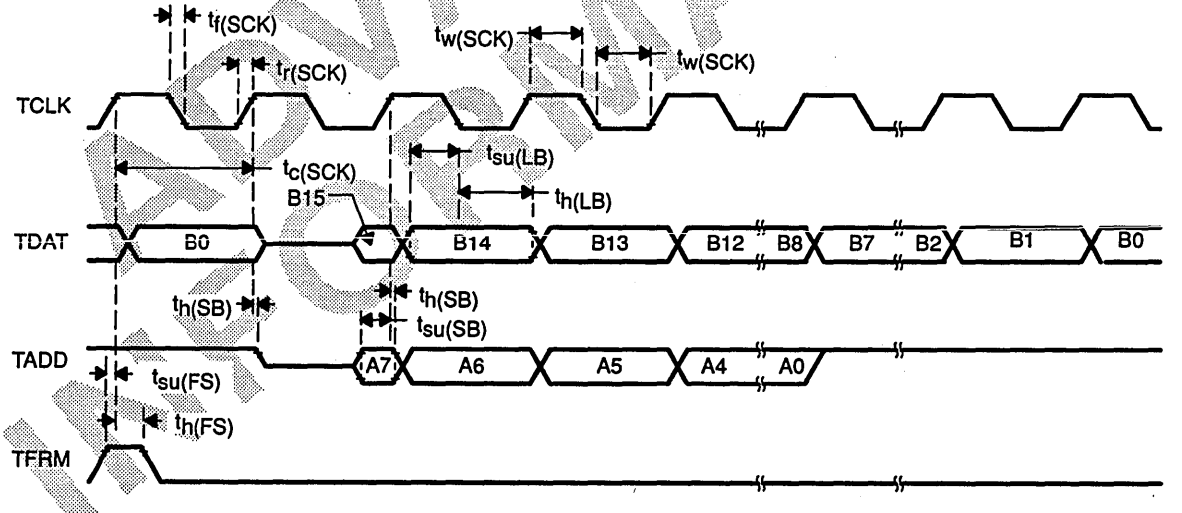
A.3.13 Serial Port Receive Timing in TDM Mode

Table A–22. Timing Requirements Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$)

Parameter		Min	Max	Unit
$t_{c(SCK)}$	Serial port clock cycle time	5.2H		ns
$t_{f(SCK)}$	Serial port clock fall time		8	ns
$t_{r(SCK)}$	Serial port clock rise time		8	ns
$t_{w(SCK)}$	Serial port clock low/high pulse duration	2.1H		ns
$t_{su(LB)}$	TDAT/TADD setup time before TCLK falling edge	S – 30		ns
$t_{h(LB)}$	TDAT/TADD hold time after TCLK falling edge	S – 30		ns
$t_{su(SB)}$	TDAT/TADD setup time before TCLK rising †	S – 10		ns
$t_{h(SB)}$	TDAT/TADD hold time after TCLK rising †	–5		ns
$t_{su(FS)}$	TRFM setup time before TCLK rising edge ‡	10		ns
$t_{h(FS)}$	TRFM hold time after TCLK rising edge ‡	10		ns

† These parameters apply only to the first bits in the serial bit string.
‡ FSX timing and waveforms shown in Figure A–16 are for external FSX. FSX can also be configured as internal. The FSX internal case is illustrated in the transmit timing diagram in Figure A–17.

Figure A–16. Serial Port Timing in TDM Mode

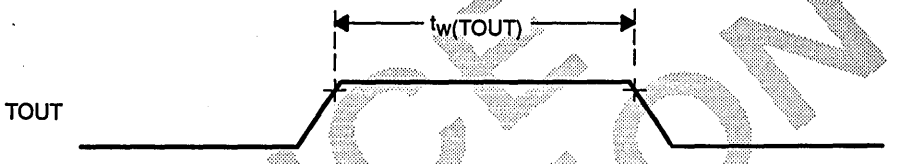


A.3.15 Timer Output

Table A–25. Switching Characteristics Over Recommended Operating Conditions (H = 0.5$t_{c(CO)}$)

Parameter		Min	Nom	Max	Unit
$t_{w(TOUT)}$	TOUT pulse duration	H – 5			ns

Figure A–18. Timer Output



B External Interface Timings

Figure B–1. Memory Interface Operation for Read-Read-Write (0 Wait State)

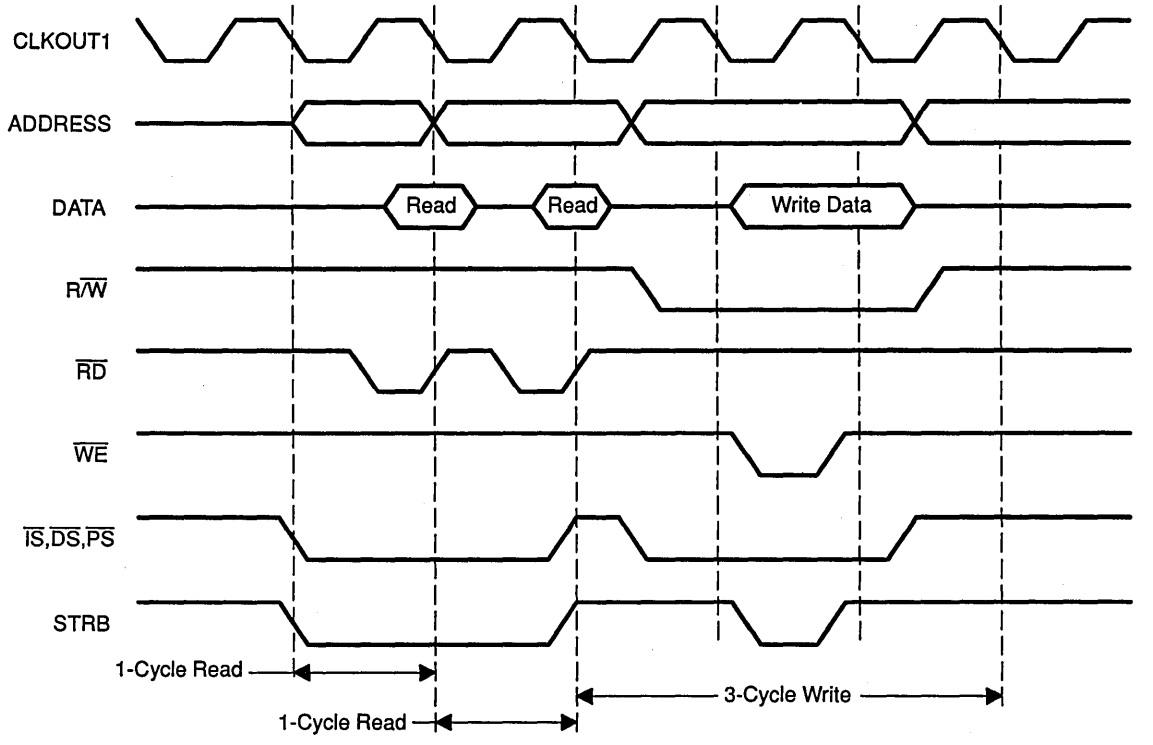
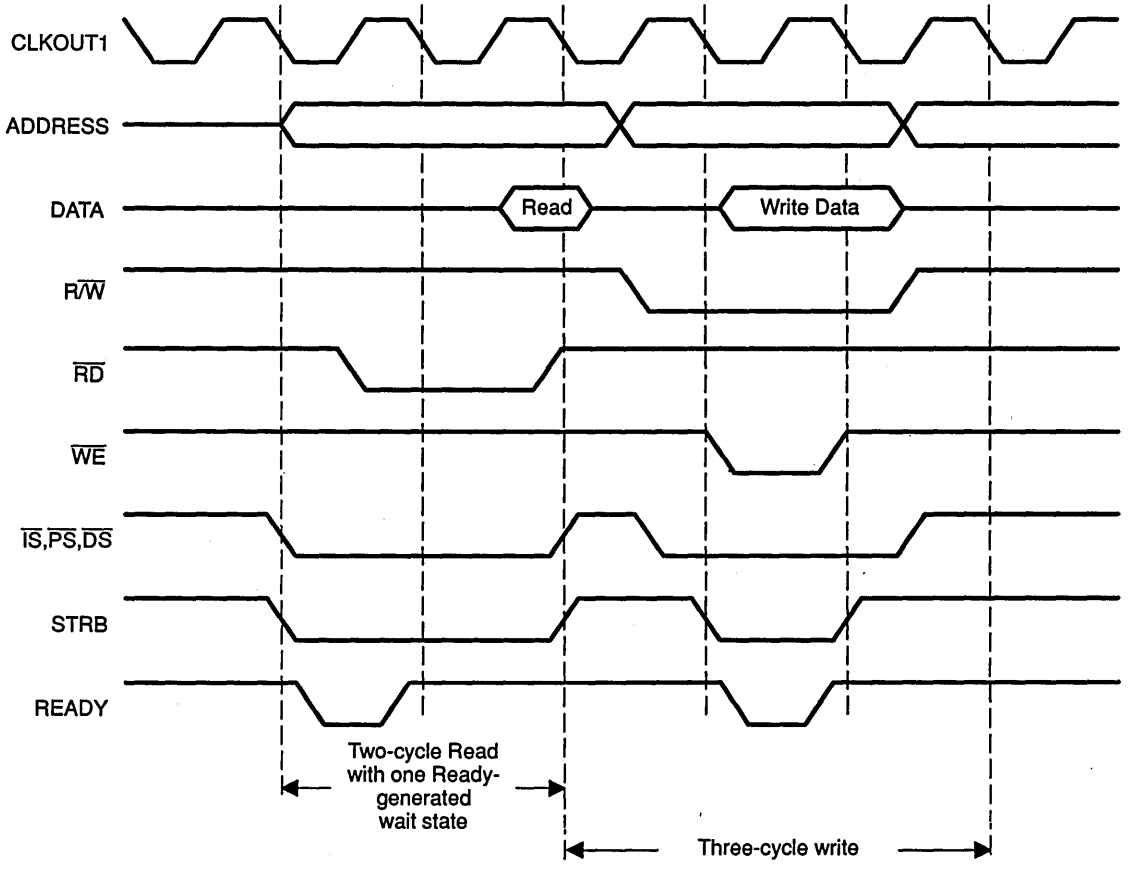


Figure B–3. Memory Interface Operation for Read-Write (1 Wait State)



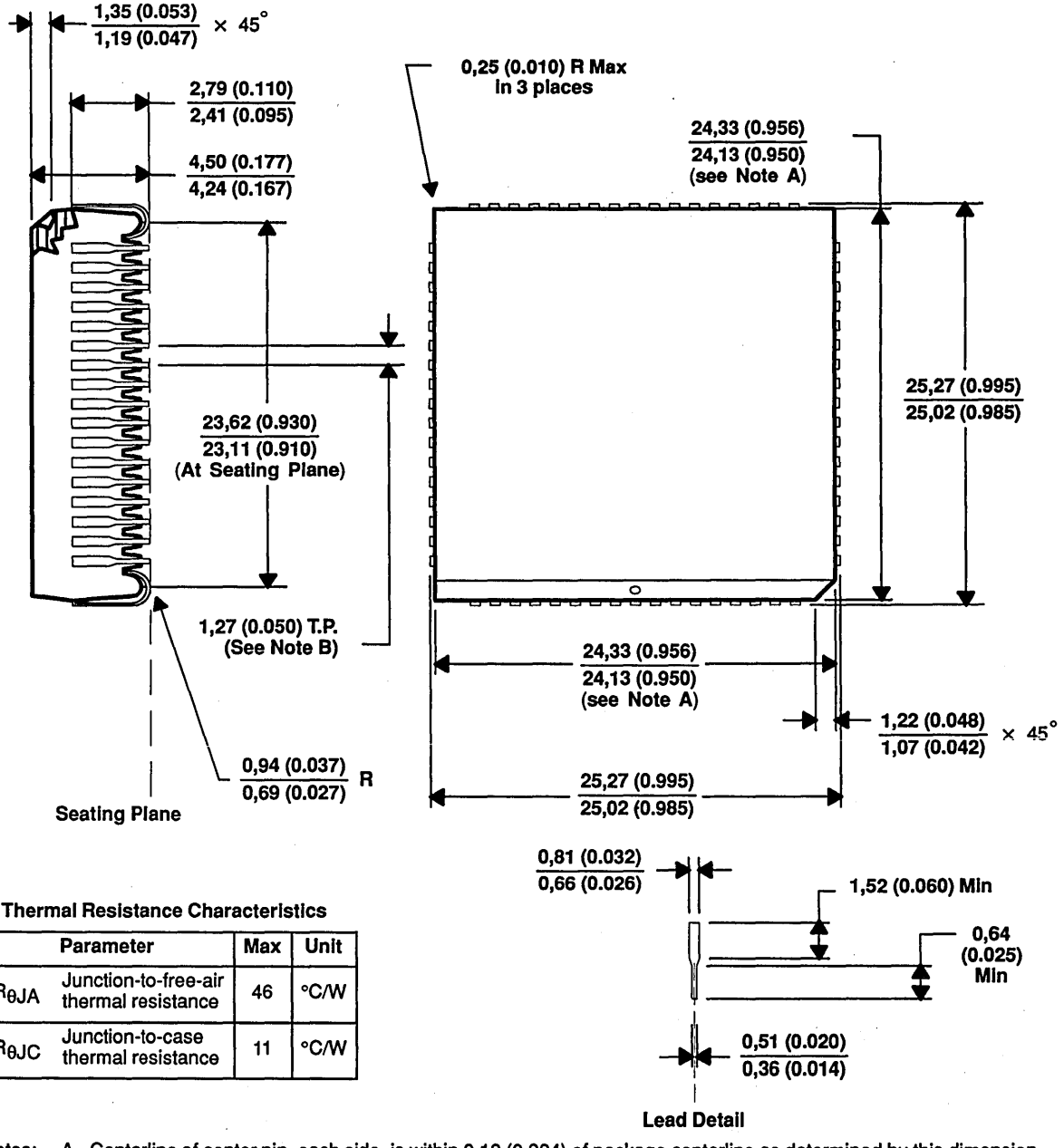
C

TMS320C5x System Migration

C.1 Package and Pin Layout

The TMS320C25 is available in both a 68-pin CPGA and a 68-pin PLCC as shown in Figure C–1 and Figure C–2, respectively. The TMS320C50 and TMS320C51 are packaged in a 132-pin Quad Flat Pack package (QFP). This package is shown in Appendix A.

Figure C–2. TMS320C25 68-Pin Plastic Leaded Chip Carrier



Thermal Resistance Characteristics

Parameter		Max	Unit
$R_{\theta JA}$	Junction-to-free-air thermal resistance	46	°C/W
$R_{\theta JC}$	Junction-to-case thermal resistance	11	°C/W

Notes: A. Centerline of center pin, each side, is within 0,10 (0.004) of package centerline as determined by this dimension.
B. Location of each pin is within 0,127 (0.005) of true position with respect to center pin on each side.

ALL LINEAR DIMENSIONS ARE IN MILLIMETERS AND PARENTHETICALLY IN INCHES.

Only two TMS320C25 signals (CLKOUT2 and $\overline{\text{SYNC}}$) are not present on the TMS320C5x. Because the TMS320C5x operates with a divide-by-two clock, it can be synchronized with reset. Therefore, there is no need for the $\overline{\text{SYNC}}$ signal. With only two phases, there are no external timings that tie to the CLKOUT2 of the TMS320C25.

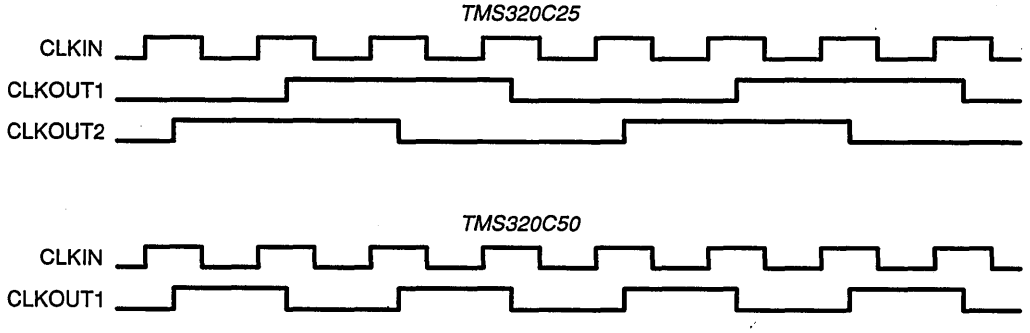
Some of the TMS320C25-equivalent pins have additional capabilities on the TMS320C5x. The TMS320C5x supports external direct memory access of the on-chip single-access RAM block. For this reason, the following signals are now bidirectional:

A0–A15 = address lines
$\overline{\text{STRB}}$ = memory access strobe
R/$\overline{\text{W}}$ = read/write
$\overline{\text{BR}}$ = bus request

The TMS320C5x serial port transmit clock (CLKX) can now be configured as an output that operates at one-fourth the machine clock rate. CLKX is configured as an input by reset. The TMS320C25 CLKX pin is always an input.

The TMS320C25 operates with a four-phase clock. This device's machine rate is one-fourth the CLKIN rate. CLKOUT1 and CLKOUT2 operate at the machine rate and are 90° out of phase. The TMS320C5x operates with a two-phase clock. The device's machine rate is one-half the CLKIN rate. In addition, the TMS320C5x offers a divide-by-one clock input feature so that the device's machine rate equals the CLKIN rate. CLKOUT1 operates at the machine rate. Figure C–4 shows both the TMS320C25 and the TMS320C5x clocking schemes.

Figure C–4. TMS320C25 and TMS320C5x Clocking Schemes



C.2 Timing

The TMS320C25 and the TMS320C5x operate with some timing differences. These timing differences include aspects of the on-chip operation as well as aspects of the external memory interfacing. One key difference is that the TMS320C5x is capable of operating at two to three times the speed of a TMS320C25. Another key difference is that the TMS320C25 operates with a three-deep pipeline, while the TMS320C5x operates with a four-deep pipeline. Key differences in the external memory interface encompass the faster TMS320C5x and include certain external interface enhancements. The final key difference is that some compatible operations execute in a different number of machine cycles. This section describes these differences.

C.2.1 Device Clock Speed

The TMS320C25 operates its machine cycles with a divide-by-four clocking scheme. The TMS320C5x uses a divide-by-two clocking scheme. This means that a TMS320C25, operating with a 40-MHz CLKIN, executes its machine cycles within 100 ns, while the TMS320C5x, which is operating with the same CLKIN, executes its machine cycles in 50 ns. This clocking arrangement changes the way that the signals of the devices are specified. Many of the TMS320C25 timing values, given in the *TMS320 Second-Generation Digital Signal Processor Data Sheet*, are specified as quarter-phase (**Q**) ± **N** ns. The timing values of the TMS320C5x are defined in half-phases (**H**).

C.2.2 Pipeline

The TMS320C25 operates with a three-deep pipeline, while the TMS320C5x operates with a four-deep pipeline. This means that anytime there is a program counter (PC) discontinuity (for example, branch, call, return, interrupt, etc.), it takes four cycles to complete with the TMS320C5x, whereas it takes three cycles on the TMS320C25. The TMS320C5x, however, also has delayed instructions that take only two cycles to complete.

C.2.3 External Memory Interfacing

The TMS320C5x is designed to execute external memory operations with the same signals as the TMS320C25. As mentioned above, the TMS320C5x operates at twice the instruction rate of the TMS320C25 when both operate with the same input clock. The TMS320C5x uses its software wait-state generators to compensate for this interface difference. The TMS320C5x device, operating with one software wait state, has similar memory timing to the TMS320C25 operating with no wait states. However, external writes require two cycles on the TMS320C5x devices. The exact timing of the signals differ because of the more advanced process used with the TMS320C5x.

The TMS320C5x has two additional memory interface signals to reduce the amount of external interfacing circuitries. The $\overline{RD}$ signal can be used to inter-

C.3 Instruction Set

The TMS320C5x instruction set is a superset of the TMS320C25 instruction set. The instruction set of the TMS320C25 is upward source-code compatible. This means that all of the instruction features of the TMS320C25, implemented and code written for the TMS320C25, can be reassembled to run on the TMS320C5x.

The serial port mode control bits have been moved from the status registers to the serial port control register. Because they are no longer part of the CPU registers, they no longer have direct instructions to set or clear them. The bits of the SPC can be manipulated easily with the PLU instructions. The following table shows the instructions used to replace the serial port instructions (note that the data page pointer must be set to zero to execute these new instructions):

TMS320C25	TMS320C5x	
RFSM	APL	#0FFFFh,SPC
SFSM	OPL	#8,SPC
RTXM	APL	#0FFFDh,SPC
STXM	OPL	#2,SPC
FORT0	APL	#0FFFBh,SPC
FORT1	OPL	#4,SPC

Note that any or all three bits can be set in one execution of the OPL instruction, while any or all three bits can be cleared using the APL. The bits can be toggled with the XPL instruction. The I/O ports of the device are addressable in data memory space on the TMS320C5x devices. This means any instruction that can address data memory can also address the I/O ports.

There are a number of new instructions on the TMS320C5x devices. These instructions provide a more orthogonal addressing scheme and exercise the new CPU enhancements. In order to simplify the description of the instruction set, a number of different instructions are combined into single new instructions with additional operand formats, as in this example:

TMS320C25		TMS320C5x	
ADD	*+	ADD	*+
ADDK	0FFh	ADD	#0FFh
ADLK	0FFFFh	ADD	#0FFFFh
ADDH	*+	ADD	*+,16

Refer to Chapter 4 for the detailed discussion of the instruction set.

The IDLE instruction, when executed, stops the CPU from fetching and executing instructions until an unmasked interrupt occurs. The TMS320C25 automatically enables the interrupts globally with the execution of the IDLE instruction; this saves the extra instruction word/cycle required to execute the EINT (enable interrupts globally) instruction. Upon receipt of the interrupt, the TMS320C25 executes the interrupt vector and resumes operations. The

C.4 On-Chip Peripheral Interfacing

The TMS320C5x has more peripherals than the TMS320C25; many TMS320C5x peripherals are enhancements of the TMS320C25 peripherals. The TMS320C25 has three peripheral circuits: serial port, timer, and 16 I/O ports. In addition to these peripherals, the TMS320C5x has software wait states and a divide-by-one clock.

The serial port of the TMS320C5x has been enhanced in that the CLKX pin can be configured as either an input or an output (CLKX is always an input on the TMS320C25). CLKX is configured as an input upon a device reset to maintain compatibility with the TMS320C25. The new serial port status bits are now mapped to a memory-mapped register that is used exclusively for the serial port. The serial port modes are no longer controlled via status register 1. Therefore, serial port modes that are changed by using LST1 instruction will no longer work. The mode bits must be set/reset via the serial port control register (SPC). The data transmit (DXR) and data receive (DRR) registers have been moved in the memory map from locations 1 and 0 to 33 and 32, respectively.

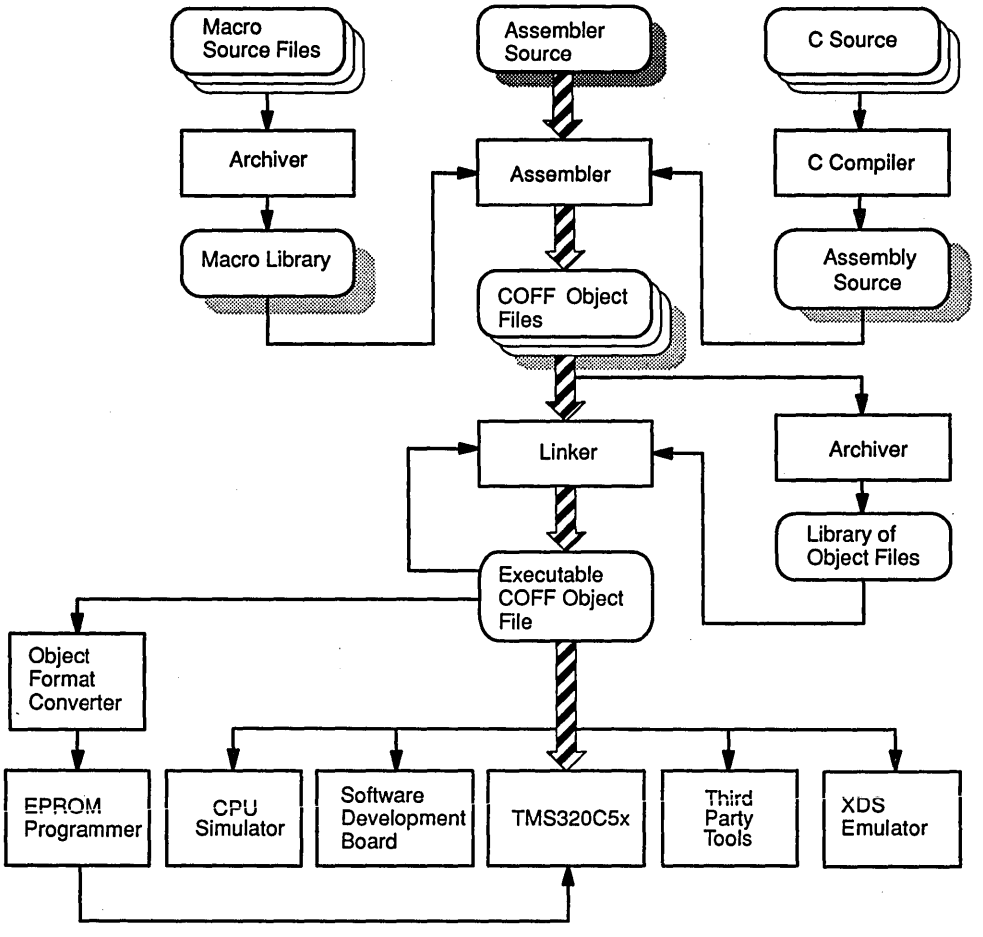
The timer has been enhanced on the TMS320C5x to include a divide-down factor of 1 to 17 and can be stopped or reset via software. These additional features are controlled via the timer control register (TCR). Upon reset, the divide-down factor is set to 1, and the timer is enabled to maintain compatibility with the TMS320C25. The timer (TIM) and period (PRD) registers have been moved in the memory map from locations 2 and 3 to locations 36 and 37, respectively.

The 16 input/output ports of the TMS320C5x are addressable in the data memory space. This allows direct access of the I/O space by the core CPU and supports bit operation in the I/O space via the PLU. The I/O space is increased from 16 ports to 65,536 ports. However, no additional decode circuitry is necessary if only 16 ports are used.

The TMS320C5x includes software wait-state generators that are mapped on 16K-word page sizes in the program and data memory spaces. There are also wait-state generators for the I/O ports. The I/O space wait-state generators can be mapped on two-word or on 4K-word boundaries. These wait-state generators allow the system to be programmed for 0, 1, 2, 3, 4, or 7 wait states, eliminating the need of an off-chip interfacing circuitry. External access wait states can be extended further via the READY signal.

D TMS320C5x Development Tools

Figure D–1. TMS320C5x Development Environment



makes it possible to monitor the state of the simulated device. The simulator accepts object codes that are produced by the macro assembler/linker system. Recent improvements have been made in simulation technology. The TMS320C5x software simulator uses a flexible high-level language debug monitor user interface. This interface allows the user to view both C language and assembly language to be viewed simultaneously. Single-stepping and software breakpoints may be executed in either language, providing a means for a high-level language debug environment. This interface is used on both the SWDS and XDS510, providing an easy transition to other tools.

These are some key features of the TMS320C5x simulator:

- simulates the entire TMS320C5x instruction set
- simulates the key features of the on-chip TMS320C5x peripherals (serial ports and timer)
- has a high-level language debug monitor user interface
- has a windowed, mouse-driven interface, which can be user-customized
- quickly stores/retrieves the simulation parameters from files to facilitate preparation for individual sessions
- performs reverse-assembly on source assembly code and C code, or allows both edit and reassembly of the source statements
- simultaneously displays memory in
  - hexadecimal 16-bit values
  - assembled source code
- offers many execution modes:
  - single/multiple instruction count
  - single/multiple cycle count
  - until condition is met (**UNTIL**)
  - while condition exists (**WHILE**)
  - for set loop count (**FOR**)
  - unrestricted run with halt by key input

During program execution, the internal registers and memory of the simulated TMS320C5x are modified as each instruction is interpreted by the host computer. Execution is suspended when either a breakpoint or an error is encountered or when the user halts execution. Once program execution is suspended, the internal registers and both program and data memory can be inspected and/or modified. Also, the trace memory can be displayed. A record of the simulation session can be maintained in a journal file so that it can be re-executed to regain the same machine state during another simulation session.

The simulator allows verification and monitoring of the states of the processor, without the requirement of hardware. The TMS320C5x software simulator op-

- Watch Window for displaying values of selected variables, registers, or other C expressions. The window automatically displays output of the correct data type.
- Display Windows for displaying all field elements of a selected structure or array. Display windows understand all data types and automatically display values as their correct types. If a member of a structure or array is another substructure or array, a display window can cause children or subwindows to show the substructure or array.

These items are recommended for the interface:

- a color display for easily recognizing the different display elements.
- a graphics display adapter (EGA or VGA board). Some boards produce a larger screen size, which the debugger takes advantage of.
- a mouse to take full advantage of the window and menu feature set.

bility of the TMS320C5x within your target system. The XDS features real-time hardware breakpoint and program execution capabilities from target memory.The TMS320C5x JTAG serial-scan path is used to upload and download both program and data memory and to run all emulator functions. The PC-resident XDS has a cable for connection to your target system. The XDS has the same user interface as the software simulator and the SWDS.

Key features of the XDS510 include

- full-speed execution and monitoring of the TMS320C5x in your target system via a 14-pin target connector
- loading/inspection/modification of all registers
- upload/download of both program memory and data memory
- high-level language debug monitor user interface
- single-step execution
- software breakpoint/trace and timing, with up to thirty software breakpoints
- hardware breakpoint/trace on all program addresses
- emulator portability
- reconnectability for multiprocessing applications
- benchmark of execution time clock cycles in real time

Full-speed emulation and monitoring of the target system is performed serially over a 14-wire cable, which runs from the XDS510 to the target system. Fourteen signals must be brought out of the target system and into a header situated next to the TMS320C5x. The emulation cable is then connected to the header. The 14 signals are 4 JTAG (IEEE standard **P1149.1)** scan path signals, 3 emulation signals, 1 clock signal, and 6 power/ground signals. The JTAG scan path controls the TMS320C5x in the targeted application and provides access to all registers as well as to internal and external memory of the device. Since program execution takes place on the TMS320C5x in the target system, there are no timing differences during emulation. This new emulation technology offers significant advantages over the technology of traditional emulators. These advantages include

- no transmission problems related to cable length
- a nonintrusive system
- no loading problems on signals
- no artificial memory limitations
- a common screen interface for ease of use

E XDS510 Design Considerations

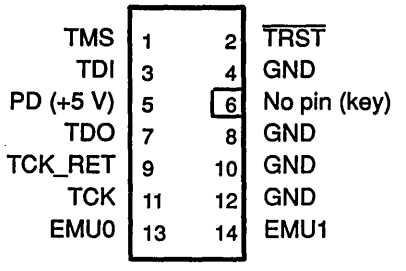
E.1 Header and Header Signals

To perform emulation with the XDS510, your target system must have a 14-pin header (two 7-pin rows) with connections as shown in Figure E–1. Table E–1 describes the emulation signals.

Although you can use other headers, recommended parts include:

Straight header, unshrouded	DuPont Electronics® part number 67996–114
Right-angle header, unshrouded	DuPont Electronics® part number 68405–114

Figure E–1. 14-Pin Header Signals and Header Dimensions



Header Dimensions:
Pin-to-pin spacing: 0.100 in. (X,Y)
Pin width: 0.025 in. square post
Pin length: 0.235 in., nominal

Table E–1. 14-Pin Header Signal Description

XDS510 Signal	XDS510 State	Target State	Description
TMS	O	I	JTAG test mode select.
TDI	O	I	JTAG test data input.
TDO	I	O	JTAG test data output.
TCK	O	I	JTAG test clock. TCK is a 10-MHz clock source from the emulation cable pod. This signal can be used to drive the system test clock.
$\overline{\text{TRST}}$	O	I	JTAG test reset.
EMU0	I	I/O	Emulation pin 0.
EMU1	I	I/O	Emulation pin 1.
PD	I	O	Presence detect. Indicates that the emulation cable is connected and that the target is powered up. PD should be tied to +5 volts in the target system.
TCK_RET	I	O	JTAG test clock return. Test clock input to the XDS510 emulator. May be a buffered or unbuffered version of TCK.

E.3 Cable Pod

Figure E–2 shows a portion of the XDS510 emulator cable pod. These are the functional features of the emulator pod:

- Signals TDO and TCK_RET can be parallel-terminated inside the pod if required by the application. The default is that these signals are not terminated.
- Signal TCK is driven with a 74AS1034 device. Because of the high current drive (48 mA I_{OL}/I_{OH}), this signal can be parallel-terminated. If TCK is tied to TCK_RET, then you can use the parallel terminator in the pod.
- Signals TMS and TDI can be generated from the falling edge of TCK_RET, according to the IEEE 1149.1 bus slave device timing rules. They can also be driven from the rising edge of TCK_RET, which allows a higher TCK_RET frequency. The default is to match the IEEE 1149.1 slave device timing rules. This is an emulator software option that can be selected when the emulator is invoked. In general, single-processor applications can benefit from the higher clock frequency. However, in multiprocessing applications, you may wish to use the IEEE 1149.1 bus slave timing mode to minimize emulation system timing constraints.
- Signals TMS and TDI are series-terminated to reduce signal reflections.
- A 10-MHz test clock source is provided. You may also provide your own test clock for greater flexibility.

Figure E–3. Emulator Pod Timings

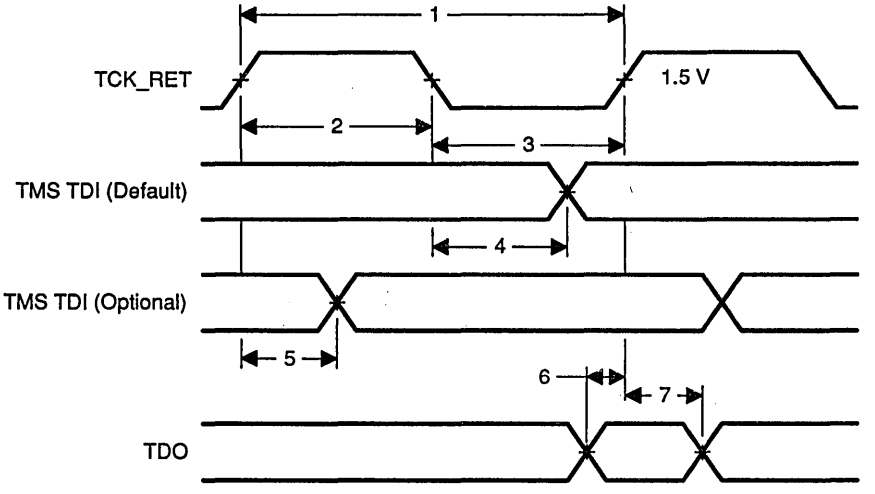


Table E–2. Emulator Pod Timing Parameters

No.	Reference	Description	Min	Max	Unit
1	t_{TCKmin} t_{TCKmax}	TCK_RET period	35	200	ns
2	$t_{TCKhighmin}$	TCK_RET high pulse duration	15		ns
3	$t_{TCKlowmin}$	TCK_RET low pulse duration	15		ns
4	$t_{d(XTMXmin)}$ $t_{d(XTMXmax)}$	TMS/TDI valid from TCK_RET low (default timing)	6	20	ns
5	$t_{d(XTMSmin)}$ $t_{d)XTMSmax)}$	TMS/TDI valid from TCK_RET high (optional timing)	7	24	ns
6	$t_{su(XTDOmin)}$	TDO setup time to TCK_RET high	3		ns
7	$t_{hd(XTDOmin)}$	TDO hold time from TCK_RET high	12		ns

It is extremely important to provide high-quality signals between the emulator and the target processor. If the distance between the emulation header and the processor is greater than 6 inches, the emulation signals should be buffered. Sections E.4 and E.5 illustrate typical connections between the target processor and the emulation header.

E.5 Multiprocessor Configuration

Figure E–5. Multiprocessor Connections

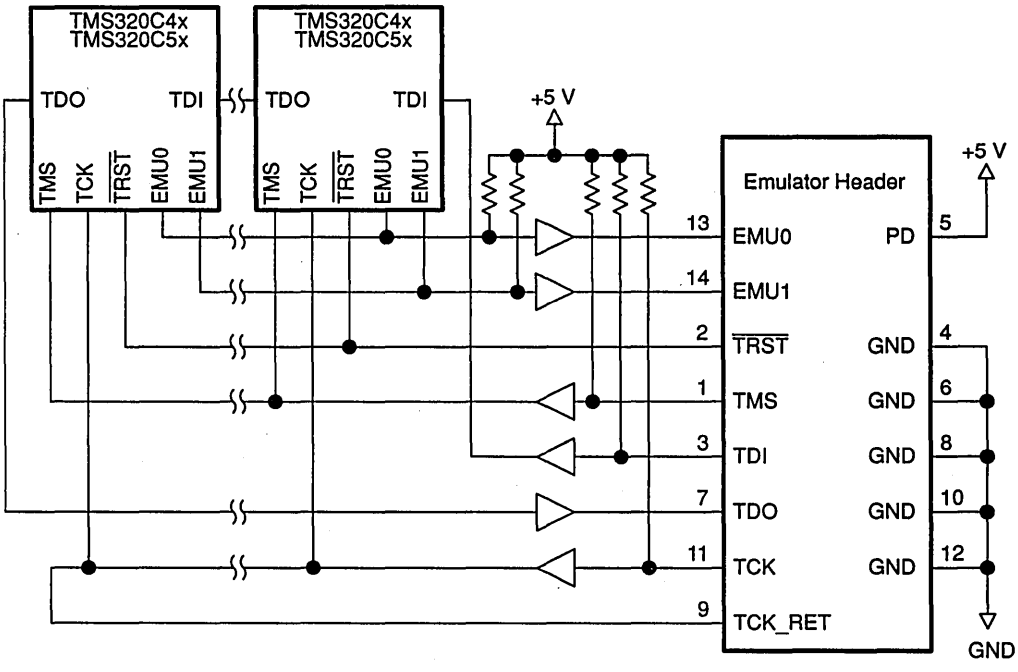


Figure E–5 shows a typical multiprocessor configuration. This is a daisy-chained configuration (TDO-TDI daisy-chained), which meets the minimum requirements of the IEEE 1149.1 specification. The emulation signals in this example are buffered to isolate the processors from the emulator and provide adequate signal drive for the target system. One of the benefits of a JTAG test interface is that you can generally slow down the test clock to eliminate timing problems. Several key points to multiprocessor support are as follows:

- The processor TMS, TDI, TDO, and TCK should be buffered through the same physical package to control timing skew better.
- The input buffers for TMS, TDI, and TCK should have pullups to 5 volts. This will hold these signals at a known value when the emulator is not connected. A pullup of 4.7 kΩ or greater is suggested.
- Buffering EMU0 and EMU1 is optional, but highly recommended to provide isolation. These are not critical signals and do not need to be buffered through the same physical package as TMS, TCK, TDI, and TDO. Buffered and unbuffered signals are shown in Figure E–6 and Figure E–7.

No signal buffering. In this situation, the distance between the header and the processor should be no more than 6 inches.

- It is extremely important to provide high quality signals, especially on the processor TCK and the emulator TCK_RET signal. In some cases, this may require you to provide special PWB trace routing and to use termination resistors to match the trace impedance. The emulator pod does provide optional internal parallel terminators on the TCK_RET and TDO. TMS and TDI provide fixed series termination.

Case 1: Single processor, direct connection, TMS/TDI timed from TCK_RET low (default timing).

$$t_{prdtck\_TMS} = [t_{d(XTMSmax)} + t_{su(TTMS)}] / t_{tckfactor}$$
$$= (20\text{ ns} + 10\text{ ns}) / 0.4$$
$$= 75\text{ ns} \quad (13.3\text{ MHz})$$

$$t_{prdtck\_TDO} = [t_{d(TTDO)} + t_{su(XTDOmin)}] / t_{tckfactor}$$
$$= (15\text{ ns} + 3\text{ ns}) / 0.4$$
$$= 45\text{ ns} \quad (22.2\text{ MHz})$$

In this case, the TCK/TMS path is the limiting factor.

Case 2: Single processor, direct connection, TMS/TDI timed from TCK_RET high (optional timing).

$$t_{prdtck\_TMS} = t_{d(XTMSmax)} + t_{su(TTMS)}$$
$$= (24\text{ ns} + 10\text{ ns})$$
$$= 34\text{ ns (29.4 MHz)}$$

$$t_{prdtck\_TDO} = [t_{d(TTDO)} + t_{su(XTDOmin)}] / ttckfactor$$
$$= (15 + 3) / 0.4$$
$$= 45\text{ ns} \quad (22.2\text{ MHz})$$

In this case, the TCK/TDO path is the limiting factor. One other thing to consider in this case is the TMS/TDI hold time. The minimum hold time for the XDS510 cable pod is 7 ns, which meets the 5-ns hold time of the target device.

Case 3: Single/multiple processor, TMS/TDI buffered input; TCK_RET/TDO buffered output, TMS/TDI timed from TCK_RET high (optional timing).

$$t_{prdtck\_TMS} = t_{d(XTMSmax)} + t_{su(TTMS)} + 2\, t_{d(bufmax)}$$
$$= 24\text{ ns} + 10\text{ ns} + 2\ (10)$$
$$= 54\text{ ns (18.5 MHz)}$$

$$t_{prdtck\_TDO} = \frac{t_{d(TTDO)} + t_{su(XTDOmin)} + t_{bufskew}}{t_{tckfactor}}$$

$$= (15\text{ ns} + 3\text{ ns} + 1.35\text{ ns}) / 0.4$$
$$= 58.4\text{ ns} \quad (20.7\text{ MHz})$$

In this case, the TCK/TMS path is the limiting factor. The hold time on TMS/TDI is also reduced by the buffer skew (1.35 ns) but still meets the minimum device hold time.

F Memories, Analog Converters, Sockets, and Crystals

F.1 Memories and Analog Converters

This section provides product information for EPROM memories, codecs, analog interface circuits, and A/D and D/A converters.

All of these devices can be interfaced with TMS320C5x processors (see Chapter NO TAG for hardware interface designs). Refer to *Digital Signal Processing Applications with the TMS320* Family for additional information on interfaces using memories and analog conversion devices.

The following paragraphs give the name of each device and the location of the data sheet for that device in order to obtain further specification information if desired.

Data sheets for EPROM memories are located in the *MOS Memory Data Book* (literature number SMYD008).

TMS27C64
TMS27C128
TMS27C256
TMS27C512

Another EPROM memory, TMS27C291/292, is described in a data sheet (literature number SMLS291A).

The TCM29C13/14/16/17 codecs and filters are described in the data sheet beginning on page 2–111 of the *Telecommunications Circuits Data Book* (literature number SCT001). An analog interface for the DSP using a codec and filter is provided by the TCM29C18/19 data sheet (literature number SCT021).

The data sheet for the TLC32040 analog interface circuit is provided in the *Interface Circuits Data Book* (literature number SLYD002).

In the same book are data sheets for A/D and D/A converters. The names of the devices are as follows:

TLC0820
TLC1205/1225
TLC7524

F.3 Crystals

This section lists the commonly used crystal frequencies, crystal specification requirements, and the names of suitable vendors.

Table F–1 lists the commonly used crystal frequencies and the devices with which they can be used.

Table F–1. Commonly Used Crystal Frequencies

Device	Frequency
TMS320C25	40.96 MHz
TMS320C5x	20.48 MHz 40.96 MHz

When connected across X1 and X2/CLKIN of the TMS320 processor, a crystal enables the internal oscillator. Crystal specification requirements are listed below.

Load capacitance = 20 pF
Series resistance = 30 ohm
Power dissipation = 1 mW

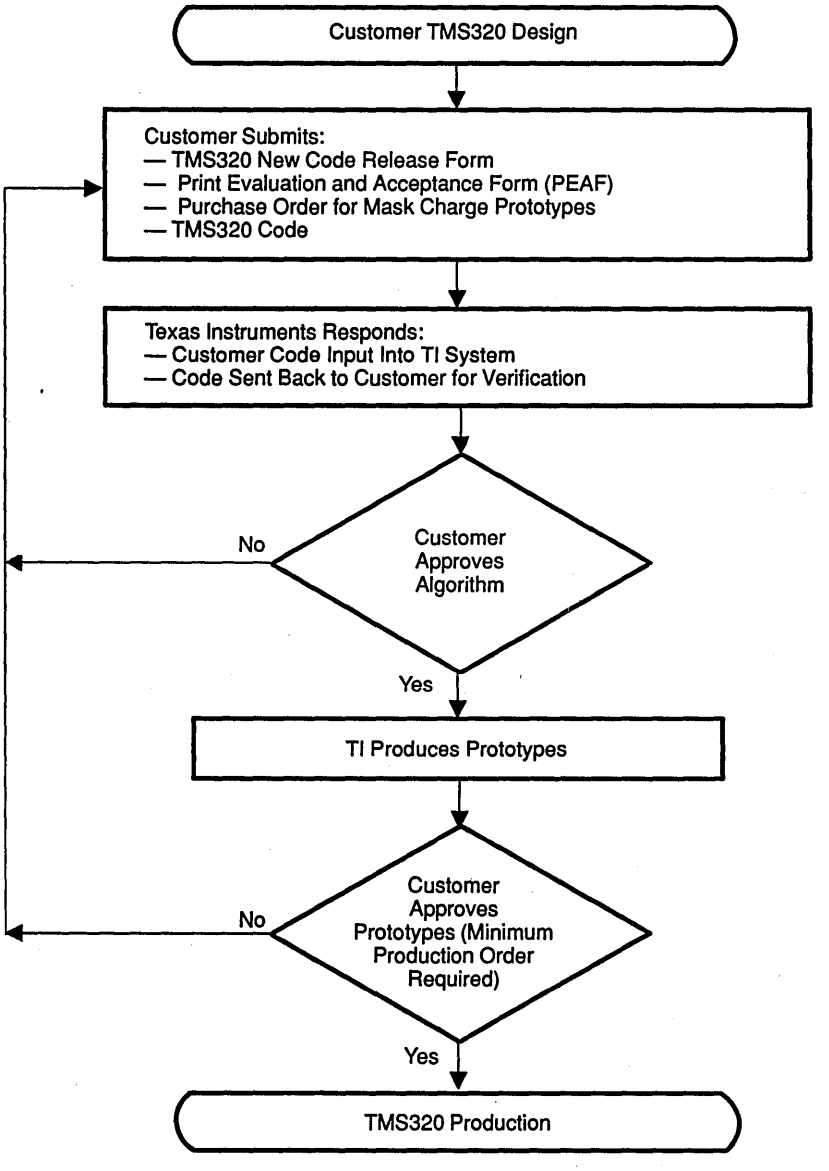
Vendors of crystals suitable for use with TMS320 devices are listed below.

RXD, Inc.
Norfolk, NB
(800) 228–8108

N.E.L. Frequency Controls, Inc.
Burlington, WI
(414) 763–3591

CTS Knight, Inc.
Contact the local distributor.

Figure G–1. TMS320 ROM Code Flowchart



H Device and Development Support Tool Nomenclature

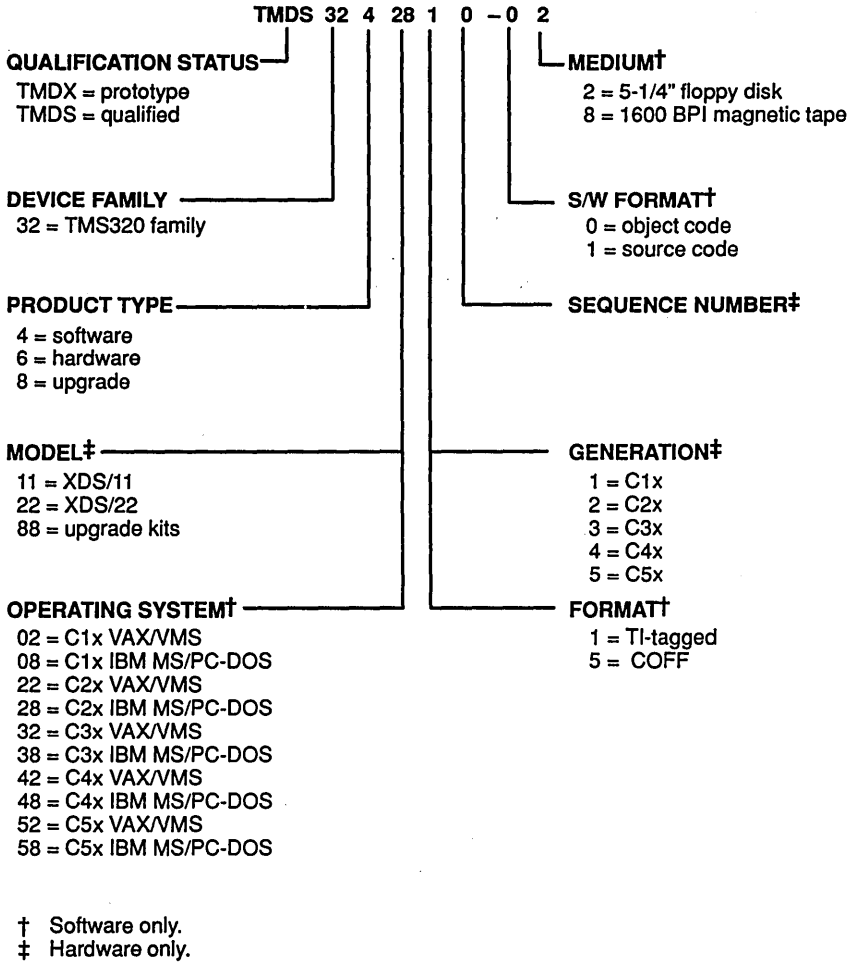
Note:

Predictions show that prototype devices (TMX or TMP) will have a greater failure rate than the standard production devices. Texas Instruments recommends that these devices *not* be used in any production system, because their expected end-use failure rate is still undefined. Only qualified production devices are to be used.

TI device nomenclature also includes a suffix with the device family name. This suffix indicates the package type (for example, N, FN, or GB) and temperature range (for example, L). Figure H–1 provides a legend for reading the complete device name for any TMS320 family member.

Figure H–2 provides a legend for reading the part number for any TMS320 hardware or software development tool.

Figure H–2. TMS320 Development Tool Nomenclature



† Software only.
‡ Hardware only.

D

E

F

G

H

O

P

R

S

TI North American Sales Offices

ALABAMA: Huntsville: (205) 837-7530
ARIZONA: Phoenix: (602) 995-1007
CALIFORNIA: Irvine: (714) 660-1200
Roseville: (916) 786-9208
San Diego: (619) 278-9601
Santa Clara: (408) 980-9000
Woodland Hills: (818) 704-8100
COLORADO: Aurora: (303) 368-8000
CONNECTICUT: Wallingford: (203) 269-0074
FLORIDA: Altamonte Springs: (407) 260-2116
Fort Lauderdale: (305) 973-8502
Tampa: (813) 882-0017
GEORGIA: Norcross: (404) 662-7900
ILLINOIS: Arlington Heights: (708) 640-3000
INDIANA: Carmel: (317) 573-6400
Fort Wayne: (219) 482-3311
IOWA: Cedar Rapids: (319) 395-9551
KANSAS: Overland Park: (913) 451-4511
MARYLAND: Columbia: (301) 964-2003
MASSACHUSETTS: Waltham: (617) 895-9100
MICHIGAN: Farmington Hills: (313) 553-1500
Grand Rapids: (616) 957-4202
MINNESOTA: Eden Prairie: (612) 828-9300
MISSOURI: St. Louis: (314) 821-8400
NEW JERSEY: Iselin: (201) 750-1050
NEW MEXICO: Albuquerque: (505) 291-0495
NEW YORK: East Syracuse: (315) 463-9291
Fishkill: (914) 897-2900
Melville: (516) 454-6600
Pittsford: (716) 385-6770
NORTH CAROLINA: Charlotte: (704) 527-0930
Raleigh: (919) 876-2725
OHIO: Beachwood: (216) 464-6100
Beavercreek: (513) 427-6200
OREGON: Beaverton: (503) 643-6758
PENNSYLVANIA: Blue Bell: (215) 825-9500
PUERTO RICO: Hato Rey: (809) 753-8700
TEXAS: Austin: (512) 250-7655
Dallas: (214) 917-1264
Houston: (713) 778-6592
UTAH: Salt Lake City: (801) 466-8973
WASHINGTON: Redmond: (206) 881-3080
WISCONSIN: Waukesha: (414) 798-1001
CANADA: Nepean: (613) 726-1970
Richmond Hill: (416) 884-9181
St. Laurent: (514) 335-8392

TI Regional Technology Centers

CALIFORNIA: Irvine: (714) 660-8140
Santa Clara: (408) 748-2220
GEORGIA: Norcross: (404) 662-7950
ILLINOIS: Arlington Heights: (708) 640-2909
INDIANA: Indianapolis: (317) 573-6400
MASSACHUSETTS: Waltham: (617) 895-9196
MEXICO: Mexico City: 491-70834
MINNESOTA: Minneapolis: (612) 828-9300
TEXAS: Dallas: (214) 917-3881
CANADA: Nepean: (613) 726-1970

Customer Response Center

TOLL FREE: (800) 336-5236
OUTSIDE USA: (214) 995-6611
(8:00 a.m. - 5:00 p.m. CST)

TI Authorized North American Distributors

Alliance Electronics, Inc. (military product only)
Almac Electronics
Arrow/Kierulff Electronics Group
Arrow (Canada)
Future Electronics (Canada)
GRS Electronics Co., Inc.
Hall-Mark Electronics
Lex Electronics
Marshall Industries
Newark Electronics
Wyle Laboratories
Zeus Components
Rochester Electronics, Inc. (obsolete product only (508) 462-9332)

TI Distributors

ALABAMA: Arrow/Kierulff (205) 837-6955; Hall-Mark (205) 837-8700; Marshall (205) 881-9235; Lex (205) 895-0480.

ARIZONA: Arrow/Kierulff (602) 437-0750; Hall-Mark (602) 437-1200; Marshall (602) 496-0290; Lex (602) 431-0030; Wyle (602) 437-2088.

CALIFORNIA: Los Angeles/Orange County: Arrow/Kierulff (818) 701-7500, (714) 838-5422; Hall-Mark (818) 773-4500, (714) 727-6000; Marshall (818) 407-4100, (714) 458-5301; Lex (818) 880-9686, (714) 863-0200; Wyle (818) 880-9000, (714) 863-9953; Zeus (714) 921-9000, (818) 889-3838;

Sacramento: Hall-Mark (916) 624-9781; Marshall (916) 635-9700; Lex (916) 364-0230; Wyle (916) 638-5282;

San Diego: Arrow/Kierulff (619) 565-4800; Hall-Mark (619) 268-1201; Marshall (619) 578-9600; Lex (619) 495-0015; Wyle (619) 565-9171; Zeus (619) 277-9681;

San Francisco Bay Area: Arrow/Kierulff (408) 441-9700; Hall-Mark (408) 432-4000; Marshall (408) 942-4600; Lex (408) 432-7171; Wyle (408) 727-2500; Zeus (408) 629-4789.

COLORADO: Arrow/Kierulff (303) 373-5616; Hall-Mark (303) 790-1662; Marshall (303) 451-8383; Lex (303) 799-0258; Wyle (303) 457-9953.

CONNECTICUT: Arrow/Kierulff (203) 265-7741; Hall-Mark (203) 271-2844; Marshall (203) 265-3822; Lex (203) 264-4700.

FLORIDA: Fort Lauderdale: Arrow/Kierulff (305) 429-8200; Hall-Mark (305) 971-9280; Marshall (305) 977-4880; Lex (305) 977-7511;

Orlando: Arrow/Kierulff (407) 333-9300; Hall-Mark (407) 830-5855; Marshall (407) 767-8585; Lex (407) 331-7555; Zeus (407) 365-3000;

Tampa: Hall-Mark (813) 541-7440; Marshall (813) 573-1399; Lex (813) 541-5100.

GEORGIA: Arrow/Kierulff (404) 497-1300; Hall-Mark (404) 623-4400; Marshall (404) 923-5750; Lex (404) 449-9170.

ILLINOIS: Arrow/Kierulff (708) 250-0500; Hall-Mark (708) 860-3800; Marshall (708) 490-0155; Newark (312)784-5100; Lex (708) 330-2888.

INDIANA: Arrow/Kierulff (317) 299-2071; Hall-Mark (317) 872-8875; Marshall (317) 297-0483; Lex (317) 843-1050.

IOWA: Arrow/Kierulff (319) 395-7230; Lex (319) 373-1417.

KANSAS: Arrow/Kierulff (913) 541-9542; Hall-Mark (913) 888-4747; Marshall (913) 492-3121; Lex (913) 492-2922.

MARYLAND: Arrow/Kierulff (301) 995-6002; Hall-Mark (301) 988-9800; Marshall (301) 622-1118; Lex (301) 596-7800; Zeus (301) 997-1118.

MASSACHUSETTS: Arrow/Kierulff (508) 658-0900; Hall-Mark (508) 667-0902; Marshall (508) 658-0810; Lex (508) 694-9100; Wyle (617) 272-7300; Zeus (617) 863-8800.

MICHIGAN: Detroit: Arrow/Kierulff (313) 462-2290; Hall-Mark (313) 462-1205; Marshall (313) 525-5850; Newark (313) 967-0600; Lex (313) 525-8100;

Grand Rapids: Arrow/Kierulff (616) 243-0912.

MINNESOTA: Arrow/Kierulff (612) 830-1800; Hall-Mark (612) 941-2600; Marshall (612) 559-2211; Lex (612) 941-5280.

MISSOURI: Arrow/Kierulff (314) 567-6888; Hall-Mark (314) 291-5350; Marshall (314) 291-4650; Lex (314) 739-0526.

NEW HAMPSHIRE: Lex (800) 833-3557.

NEW JERSEY: Arrow/Kierulff (201) 538-0900, (609) 596-8000; GRS (609) 964-8560; Hall-Mark (201) 515-3000, (609) 235-1900; Marshall (201) 882-0320, (609) 234-9100; Lex (201) 227-7880, (609) 273-7900.

NEW MEXICO: Alliance (505) 292-3360.

NEW YORK: Long Island: Arrow/Kierulff (516) 231-1000; Hall-Mark (516) 737-0600; Marshall (516) 273-2424; Lex (516) 231-2500; Zeus (914) 937-7400;

Rochester: Arrow/Kierulff (716) 427-0300; Hall-Mark (716) 425-3300; Marshall (716) 235-7620; Lex (716) 383-8020;

Syracuse: Marshall (607) 798-1611.

NORTH CAROLINA: Arrow/Kierulff (919) 876-3132; (919) 725-8711; Hall-Mark (919) 872-0712; Marshall (919) 878-9882; Lex (919) 876-0000.

OHIO: Cleveland: Arrow/Kierulff (216) 248-3990; Hall-Mark (216) 349-4632; Marshall (216) 248-1788; Lex (216) 464-2970;

Columbus: Hall-Mark (614) 888-3313;

Dayton: Arrow/Kierulff (513) 435-5563; Marshall (513) 898-4480; Lex (513) 439-1800; Zeus (513) 293-6162.

OKLAHOMA: Arrow/Kierulff (918) 252-7537; Hall-Mark (918) 254-6110; Lex (918) 622-8000.

OREGON: Almac (503) 629-8090; Arrow/Kierulff (503) 627-7667; Marshall (503) 644-5050; Wyle (503) 643-7900.

PENNSYLVANIA: Arrow/Kierulff (215) 928-1800; GRS (215) 922-7037; Marshall (412) 788-0441; Lex (412) 963-6804.

TEXAS: Austin: Arrow/Kierulff (512) 835-4180; Hall-Mark (512) 258-8848; Lex (512) 339-0088; Wyle (512) 345-8853;

Dallas: Arrow/Kierulff (214) 380-6464; Hall-Mark (214) 553-4300; Marshall (214) 233-5200; Lex (214) 247-6300; Wyle (214) 235-9953; Zeus (214) 783-7010;

Houston: Arrow/Kierulff (713) 530-4700; Hall-Mark (713) 781-6100; Marshall (713) 895-9200; Lex (713) 784-3600; Wyle (713) 879-9953.

UTAH: Arrow/Kierulff (801) 973-6913; Marshall (801) 485-1551; Wyle (801) 974-9953.

WASHINGTON: Almac (206) 643-9992, (509) 924-9500; Arrow/Kierulff (206) 643-4800; Marshall (206) 486-5747; Wyle (206) 881-1150.

WISCONSIN: Arrow/Kierulff (414) 792-0150; Hall-Mark (414) 797-7844; Marshall (414) 797-8400; Lex (414) 784-9451.

CANADA: Calgary: Future (403) 235-5325;

Edmonton: Future (403) 438-2858;

Montreal: Arrow Canada (514) 735-5511; Future (514) 694-7710; Marshall (514) 694-8142;

Ottawa: Arrow Canada (613) 226-6903; Future (613) 820-8313; **Quebec City:** Arrow Canada (418) 871-7500;

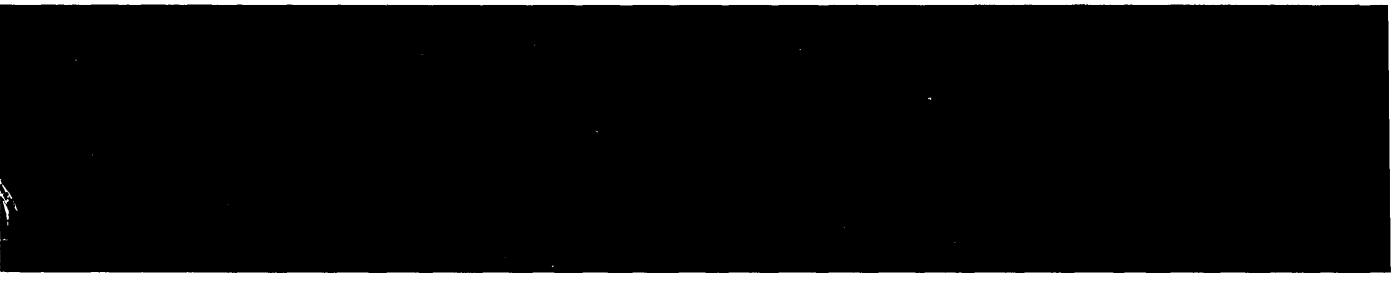
Toronto: Arrow Canada (416) 670-7769; Future (416) 612-9200; Marshall (416)458-8046;

Vancouver: Arrow Canada (604) 421-2333; Future (604) 294-1166.

TEXAS INSTRUMENTS

D0291

Printed in the U.S.A.



TEXAS INSTRUMENTS

Printed in U.S.A.
2547301-9761 revision C

SPRU056A