SPRU012

TMS320C25 User's Guide

Preliminary

Digital Signal Processor Products

TEXAS INSTRUMENTS

TMS320C25
User's Guide

Digital Signal Processor Products

Preliminary

TEXAS
INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes in the devices or the device specifications identified in this publication without notice. TI advises its customers to obtain the latest version of device specifications to verify, before placing orders, that the information being relied upon by the customer is current.

In the absence of written agreement to the contrary, TI assumes no liability for TI applications assistance, customer's product design, or infringement of patents or copyrights of third parties by or arising from use of semiconductor devices described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor devices might be or are used.

Contents

Illustrations

Figure *Page*

Tables

Table *Page*

1. Introduction

The TMS320C25 Digital Signal Processor is a member of the TMS320 family of VLSI digital signal processors and peripherals. The TMS320 family supports realtime digital signal processing (DSP) and computation-intensive applications in the areas of telecommunications, modems, speech processing, graphics/image processing, spectrum analysis, audio processing, digital filtering, high-speed control, instrumentation, and numeric processing.

The architectural investment made in the TMS320 family provides the user with a choice of five distinct processors (TMS32010, TMS320C10, TMS32011, TMS32020, TMS320C25) to best support a wide spectrum of DSP applications. Software compatibility is maintained throughout the family to protect the user's investment in the architecture. Each processor has software and hardware tools to facilitate rapid design.

The first processor in the TMS320 family is the TMS32010, a microcomputer with a 32-bit internal Harvard architecture and a 16-bit external interface capable of executing five million instructions per second. The TMS32020 is the next processor in the family with an architecture based on that of the TMS32010. Major architectural changes made on the TMS32020 enable the device to lower system cost and improve throughput by two to three times over the TMS32010 for DSP applications. The TMS32020 instruction set is a superset of that of the TMS32010, thus maintaining software compatibility.

The TMS320C25 is a pin-compatible CMOS version of the TMS32020 with a faster instruction cycle time and the inclusion of additional hardware and software features. The TMS320C25 is completely object code-compatible with the TMS32020 so that TMS32020 programs run unmodified on the TMS320C25. Some of the major enhancements of the TMS320C25 over the TMS32020 are as follows:

- Faster instruction cycle time: 100 ns
- Low-power CMOS technology with powerdown mode
- 4K words of on-chip masked ROM
- Eight auxiliary registers with a dedicated arithmetic unit
- Eight-level hardware stack
- Fully static double-buffered serial port
- Concurrent DMA using an extended hold operation
- Bit-reversed addressing modes for radix-2 FFTs
- Extended-precision arithmetic and adaptive filtering support
- Full-speed operation of MAC/MACD from external memory
- Accumulator carry bit and related instructions

Development tools and applications support are key advantages to using the TMS320C25. Full-speed emulators, software simulators and assemblers, and extensive documentation including over 735 pages of application reports provide for rapid design and development cycles. Texas Instruments regional technology centers, system application engineers, and third-party support are available for DSP education, training, and design.

1.1 General Description

The TMS320C25 architecture is based upon that of the TMS32020 digital signal processor. The TMS320C25 increases performance of DSP algorithms through a faster instruction cycle time and innovative additions to the TMS320 family architecture. The TMS320C25 is object code-compatible with the TMS32020, thus enabling current TMS32020 programs to run unmodified on the TMS320C25.

Two versions of the TMS320C25 are available to support price and performance requirements for different applications: 100-ns and 125-ns instruction cycle time versions.

The 100-ns instruction cycle time provides double the throughput for existing applications. Since most instructions are capable of executing in a single cycle, the processor is capable of executing ten million instructions per second (10 MIPS). Increased throughput on the TMS320C25 for many DSP applications is attained by means of single-cycle multiply/accumulate instructions with a data move option, eight auxiliary registers with a dedicated arithmetic unit, instruction set support for adaptive filtering and extended-precision arithmetic, bit-reversal addressing, and faster I/O necessary for data-intensive signal processing.

The architectural design of the TMS320C25 emphasizes overall system speed, communication, and flexibility in processor configuration. Control signals and instructions provide block memory transfers, communication to slower off-chip devices, multiprocessing implementations, and floating-point support.

Two large on-chip data RAM blocks (a total of 544 words), one of which is configurable either as program or data memory, provide increased flexibility in system design. An off-chip 64K-word directly addressable data memory address space is included to facilitate implementations of DSP algorithms. The large on-chip 4K-word masked ROM can be used to cost-reduce systems, thus providing for a true single-chip DSP solution. Programs of up to 4K words can be masked into the internal program ROM. The remainder of the 64K-word program memory space is located externally. Large programs can execute at full speed from this memory space. Programs may also be downloaded from slow external memory to on-chip RAM for full-speed operation. The VLSI implementation of the TMS320C25 incorporates all of these features as well as many others such as a hardware timer, serial port, and block data transfer capabilities.

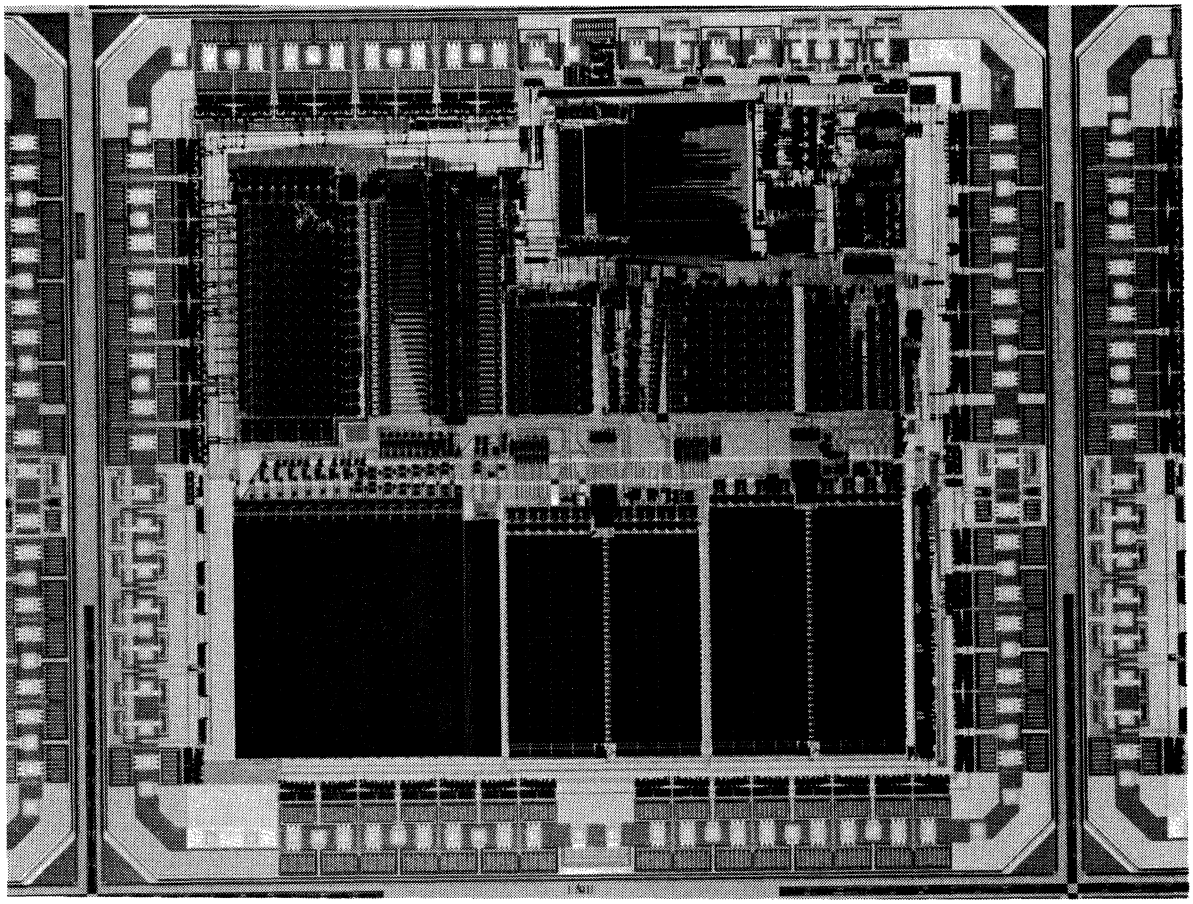


Figure 1-1. TMS320C25 Digital Signal Processor

1.2 Typical Applications

The TMS320 family's unique versatility and power offer a new approach to a variety of sophisticated applications. Table 1-1 lists some typical applications of the TMS320 family.

Table 1-1. Typical Applications of the TMS320 Family

GENERAL-PURPOSE DSP	GRAPHICS/IMAGING	INSTRUMENTATION
Digital Filtering Convolution Correlation Hilbert Transforms Fast Fourier Transforms Adaptive Filtering Windowing Waveform Generation	3-D Rotation Robot Vision Image Transmission/ Compression Pattern Recognition Image Enhancement Homomorphic Processing Workstations Animation/Digital Map	Spectrum Analysis Function Generation Pattern Matching Seismic Processing Transient Analysis Digital Filtering Phase-Locked Loops
VOICE/SPEECH	**CONTROL**	**MILITARY**
Voice Mail Speech Vocoding Speech Recognition Speaker Verification Speech Enhancement Speech Synthesis Text to Speech	Disk Control Servo Control Robot Control Laser Printer Control Engine Control Motor Control	Secure Communications Radar Processing Sonar Processing Image Processing Navigation Missile Guidance Radio Frequency Modems
TELECOMMUNICATIONS		**AUTOMOTIVE**
Echo Cancellation ADPCM Transcoders Digital PBXs Line Repeaters Channel Multiplexing 1200 to 19200-bps Modems Adaptive Equalizers DTMF Encoding/Decoding Data Encryption	FAX Cellular Telephones Speaker Phones Digital Speech Interpolation (DSI) X.25 Packet Switching Video Conferencing Spread Spectrum Communications	Engine Control Vibration Analysis Antiskid Brakes Adaptive Ride Control Global Positioning Navigation Voice Commands Digital Radio Cellular Telephones
CONSUMER	**INDUSTRIAL**	**MEDICAL**
Radar Detectors Power Tools Digital Audio/TV Music Synthesizer Educational Toys	Robotics Numeric Control Security Access Power Line Monitors	Hearing Aids Patient Monitoring Ultrasound Equipment Diagnostic Tools Prosthetics Fetal Monitors

Many of the TMS320C25's features, such as single-cycle multiply/accumulate instructions, 32-bit arithmetic unit, large auxiliary register file with a separate arithmetic unit, and large on-chip RAM and ROM, make the device particularly applicable in digital signal processing systems. At the same time, general-purpose applications of the TMS320C25 are greatly enhanced by its large address spaces, on-chip timer, serial port, multiple interrupt structure, provision for external wait states, and multiprocessor interface capability.

The flexibility of the TMS320C25 allows it to be configured to satisfy a wide range of system requirements. This allows the device to be applied in systems currently using costly bit-slice processors or custom ICs. Some of the system configurations are:

- A standalone system using 4K words of on-chip ROM and 544 words of on-chip RAM
- Parallel multiprocessing systems with shared global data memory
- Host/peripheral coprocessing using interface control signals.

1.3 Key Features

The TMS320C25 Digital Signal Processor offers a cost-effective alternative to custom VLSI and bit-slice devices. It has the following significant key features:

- 100-ns instruction cycle time
- 544 words of on-chip data RAM
- 4K words of on-chip masked ROM
- 128K words of data/program space
- Single-cycle multiply/accumulate instructions
- Object code-compatible with the TMS32020
- 16-bit instruction and data words
- 32-bit ALU and accumulator
- 16-bit parallel shifter
- Block moves for efficient data/program management
- Unsigned multiply instruction for extended-precision arithmetic
- Carry bit with associated add and subtract instructions
- Instructions for floating-point operations and adaptive filtering
- Eight auxiliary registers and a dedicated arithmetic unit
- Bit-reversed indexed addressing mode for radix-2 FFTs
- Wait states for communication to slow off-chip memories/peripherals
- Double-buffered static serial port for direct codec interface
- Three external, maskable user interrupts
- Synchronization capability between multiple processors
- On-chip clock generator
- 1.8-micron CMOS technology; single 5-volt supply
- 68-pin plastic leaded chip carrier (PLCC)
- Two versions available:
  - 40-MHz clock
  - 32-MHz clock
- Commercial and military versions supported.

1.4 How To Use This Manual

The purpose of this user's guide is to serve as a reference book for the TMS320C25 Digital Signal Processor. Sections 2 through 6 provide specific information about the architecture and operation of the device, and Sections 7 through 9 describe how to use the macro assembler/linker support software. TMS320C25 electrical specifications and mechanical data can be found in the data sheet (Appendix A).

This user's guide is designed to provide information that assists managers and hardware/software engineers in application development. The Introduction and Architectural Overview sections provide managers with basic information that describes the capabilities of the TMS320C25 for a particular application. The hardware engineer will find the Architectural Overview, Device Operation, and Hardware Applications sections and the Data Sheet and System Migration appendices most helpful. The Assembly Language Instructions, Software Applications, Assembler Directives, Macros, and Link Editor sections and the Instruction Cycle Timings, Development Support, and Software Installation appendices will aid the software engineer.

The following table lists each section and briefly describes the section contents.

Section 2. Architectural Overview. Brief description of the TMS320C25 hardware components and their functions. Block diagram, pinout of the 68-pin plastic leaded chip carrier (PLCC) package, a table of signal descriptions, and a list of TMS320C25 instructions.

Section 3. Device Operation. TMS320C25 design description, hardware components, and their functions. Functional block diagram and internal hardware summary table.

Section 4. Assembly Language Instructions. Addressing modes and format descriptions. Instruction set summary listed according to function. Alphabetized individual instruction descriptions with examples.

Section 5. Software Applications. Software application examples for the use of various TMS320C25 instruction set features.

Section 6. Hardware Applications. Hardware design techniques and application examples for interfacing to codecs or external memory.

Section 7. Assembler Directives. Description of assembly language source statement, source listing, and object code format. Individual assembler directive descriptions in alphabetical order. Assembler error diagnostics.

Section 8. Assembler Macros. Description of macro assembly language elements. Individual macro verb descriptions. Several macro examples given. Macro error diagnostics.

Section 9. Link Editor. Description of link editor and its files. Individual linker command descriptions in alphabetical order. Examples of simple linking, ROM/RAM partitioning, partial linking, and library creation given. Linker error diagnostics.

Seven appendices are included to provide additional information.

Appendix A. TMS320C25 Data Sheet. Electrical specifications, timing, and mechanical data for the TMS320C25.

Appendix B. TMS32020 Data Sheet. Electrical specifications, timing, and mechanical data for the TMS32020 Digital Signal Processor.

Appendix C. TMS320C10 Data Sheet. Electrical specifications, timing, and mechanical data for the TMS320C10 Digital Signal Processor.

Appendix D. TMS32020/TMS320C25 System Migration. Information for upgrading a TMS32020-based system to a TMS320C25-based system.

Appendix E. TMS320C25 Instruction Cycle Timings. Listing of the number of cycles for an instruction to execute in a given memory configuration.

Appendix F. TMS320C25 Development Support/Part Order Information. Listings of the hardware and software available to support the TMS320C25.

Appendix G. TMS320C25 Macro Assembler and Link Editor Installation. Series of procedures used to install and verify the TMS320C25 Macro Assembler and Link Editor on a VAX or TI/IBM PC.

1.5 References

The following reference list contains useful information regarding functions, operations, and applications of digital signal processing. These books also list other references to many useful technical papers. The references are organized into categories of general DSP, speech, image processing, and digital control theory.

General Digital Signal Processing:

Antoniou, Andreas, *Digital Filters: Analysis and Design*. New York, NY: McGraw-Hill Company, Inc., 1979.

Brigham, E. Oran, *The Fast Fourier Transform*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1974.

Burrus, C.S. and Parks, T.W., *DFT/FFT and Convolution Algorithms*. New York, NY: John Wiley & Sons, Inc., 1984.

Digital Signal Processing Applications with the TMS320 Family, Texas Instruments, 1986.

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

Gold, Bernard and Rader, C.M., *Digital Processing of Signals*. New York, NY: McGraw-Hill Company, Inc., 1969.

Hamming, R.W., *Digital Filters*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1977.

IEEE ASSP DSP Committee (Editor), *Programs for Digital Signal Processing*. New York, NY: IEEE Press, 1979.

Jackson, Leland B., *Digital Filters and Signal Processing*. Hingham, MA: Kluwer Academic Publishers, 1986.

Morris, L. Robert, *Digital Signal Processing Software*. Ottawa, Canada: Carleton University, 1983.

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

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

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

Parks, T.W. and Burrus, C.S., *Digital Filter Design*. New York, NY: John Wiley & Sons, Inc., 1986.

Speech:

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

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

Papamichalis, Panos, *Practical Approaches to Speech Coding*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986.

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

Image Processing:

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

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

Pratt, William K., *Digital Image Processing*. New York, NY: John Wiley & Sons, 1978.

Digital Control Theory:

Jacquot, R., *Modern Digital Control Systems*. New York, NY: Marcel Dekker, Inc., 1981.

Katz, P., *Digital Control Using Microprocessors*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

Kuo, B.C., *Digital Control Systems*. New York, NY: Holt, Reinholt and Winston, Inc., 1980.

Moroney, P., *Issues in the Implementation of Digital Feedback Compensators*. Cambridge, MA: The MIT Press, 1983.

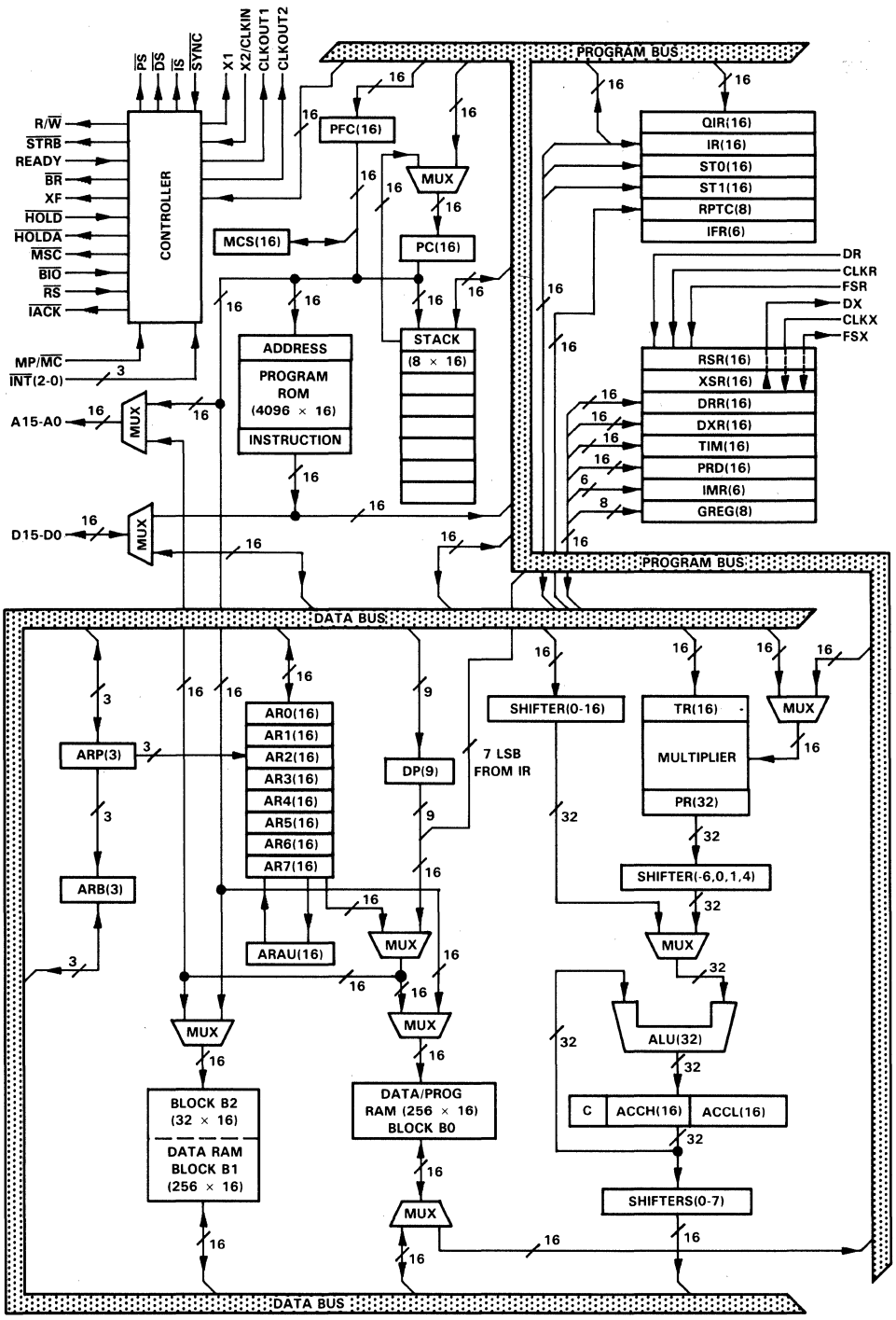
Phillips, C. and Nagle, H., *Digital Control System Analysis and Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

2. Architectural Overview

The TMS320C25 high-performance digital signal processor implements a single-accumulator, Harvard-type architecture in which program and data memory reside in separate address spaces. This allows a full overlap of instruction fetch and execution. Instructions are included to provide data transfers between the two spaces. Externally, the program and data memory spaces are multiplexed over the same bus so as to maximize the address range for both spaces while minimizing the pin count of the device. Internally, the TMS320C25 architecture maximizes processing power by maintaining two separate bus structures, program and data, for full-speed execution. Increased flexibility in system design is provided by two large on-chip data RAM blocks, one of which is configurable either as program or data memory.

The TMS320C25 incorporates a separate level of pipelining for instruction decoding. The instruction fetch-decode-execute pipeline is essentially invisible to the user, except in some cases where the pipeline must be broken (such as for branch instructions). In this case, the instructions will have slightly different timing characteristics than the TMS32020. Other instructions, such as those that operate with external data memory, have improved cycle timings compared to the TMS32020. The device executes the majority of its instructions in a single machine cycle when sufficiently fast memory is utilized. The device may also communicate to slower off-chip memories or peripherals by utilizing the READY signal. In those cases, the instructions become multicycle.

The major topics discussed in this section are as follows:



LEGEND:

ACCH	= Accumulator high	IFR	= Interrupt flag register	PC	= Program counter
ACCL	= Accumulator low	IMR	= Interrupt mask register	PFC	= Prefetch counter
ALU	= Arithmetic logic unit	IR	= Instruction register	RPTC	= Repeat instruction counter
ARAU	= Auxiliary register arithmetic unit	MCS	= Microcall stack	GREG	= Global memory allocation register
ARB	= Auxiliary register pointer buffer	QIR	= Queue instruction register	RSR	= Serial port receive shift register
ARP	= Auxiliary register pointer	PR	= Product register	XSR	= Serial port transmit shift register
DP	= Data memory page pointer	PRD	= Period register for timer	AR0-AR7	= Auxiliary registers
DRR	= Serial port data receive register	TIM	= Timer	ST0,ST1	= Status registers
DXR	= Serial port data transmit register	TR	= Temporary register		

Figure 2-1. TMS320C25 Block Diagram

2.1 Functional Block Diagram

The functional block diagram of the TMS320C25, shown in Figure 2-1, outlines the principal blocks and data paths within the processor. The diagram also shows all of the TMS320C25 interface pins.

The TMS320C25 architecture is built around two major buses: the program bus and the data bus. The program bus carries the instruction code and immediate operands from program memory. The data bus interconnects various elements, such as the Central Arithmetic Logic Unit (CALU) and the auxiliary register file, to the data RAM. Together, the program and data buses can carry data from on-chip data RAM and internal or external program memory to the multiplier in a single cycle for multiply/accumulate operations.

The TMS320C25 has a high degree of parallelism; e.g., while the data is being operated upon by the CALU, arithmetic operations may also be implemented in the Auxiliary Register Arithmetic Unit (ARAU). Such parallelism results in a powerful set of arithmetic, logic, and bit-manipulation operations that may all be performed in a single machine cycle.

2.2 Pinout and Signal Descriptions

The TMS320C25 is packaged in a 68-pin plastic leaded chip carrier (PLCC). The electrical specifications and mechanical data are given in Appendix A, the TMS320C25 Data Sheet. Figure 2-2 shows a pinout of the TMS320C25 PLCC package. Table 2-1 lists each TMS320C25 signal, its pin location, function, and input, output, or high-impedance state (I/O/Z). The signals in Table 2-1 are grouped according to function and alphabetized within that grouping.

Adaptor sockets are commercially available to convert a TMS320C25 PLCC package to a TMS32020-like 68-pin grid array (PGA) footprint, thus maintaining plug-in compatibility.

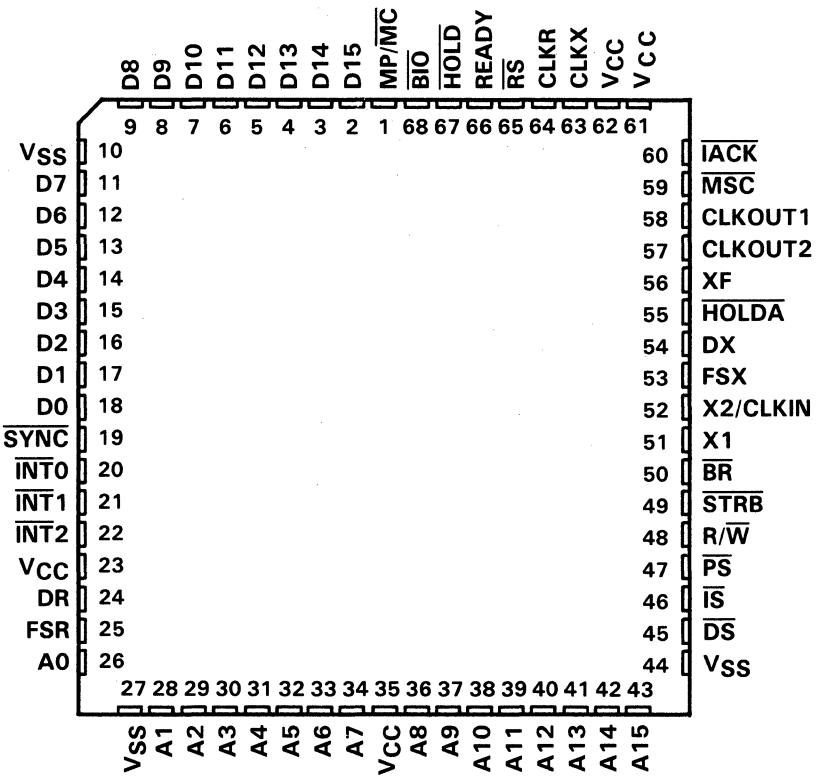


Figure 2-2. TMS320C25 Pin Assignments

Table 2-1. TMS320C25 Signal Descriptions

SIGNAL	PIN	I/O/Z†	DESCRIPTION
			ADDRESS/DATA BUSES
A15 MSB A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0 LSB	43 42 41 40 39 38 37 36 34 33 32 31 30 29 28 26	O/Z	Parallel address bus A15 (MSB) through A0 (LSB). Multiplexed to address external data/program memory or I/O. Placed in high-impedance state in the hold mode.
D15 MSB D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0 LSB	2 3 4 5 6 7 8 9 11 12 13 14 15 16 17 18	I/O/Z	Parallel data bus D15 (MSB) through D0 (LSB). Multiplexed to transfer data between the TMS320C25 and external data/program memory or I/O devices. Placed in high-impedance state when not outputting or when $\overline{RS}$ or $\overline{HOLD}$ is asserted.
			INTERFACE CONTROL SIGNALS
$\overline{DS}$ $\overline{PS}$ $\overline{IS}$	45 47 46	O/Z	Data, program, and I/O space select signals. Always high unless low level asserted for communicating to a particular external space. Placed in high-impedance state in the hold mode.
READY	66	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 = 0), the TMS320C25 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/$\overline{W}$	48	O/Z	Read/write signal. Indicates transfer direction when communicating to an external device. Normally in read mode (high), unless low level asserted for performing a write operation. Placed in high-impedance state in the hold mode.
$\overline{STRB}$	49	O/Z	Strobe signal. Always high unless asserted low to indicate an external bus cycle. Placed in high-impedance state in the hold mode.
			MULTIPROCESSING SIGNALS
$\overline{BR}$	50	O	Bus request signal. Asserted when the TMS320C25 requires access to an external global data memory space. READY is asserted to the device when the bus is available and the global data memory is available for the bus transaction.
$\overline{HOLD}$	67	I	Hold input. When asserted, the TMS320C25 places the data, address, and control lines in the high-impedance state.
$\overline{HOLDA}$	55	O	Hold acknowledge signal. Indicates that the TMS320C25 has gone into the hold mode and that an external processor may access the local external memory of the TMS320C25.
$\overline{SYNC}$	19	I	Synchronization input. Allows clock synchronization of two or more TMS320C25s. $\overline{SYNC}$ is an active-low signal and must be asserted on the rising edge of CLKIN.

† Input/Output/High-impedance state

Table 2-1. TMS320C25 Signal Descriptions (Concluded)

SIGNAL	PIN	I/O/Z†	DESCRIPTION
			INTERRUPT AND MISCELLANEOUS SIGNALS
$\overline{BIO}$	68	I	Branch control input. Polled by BIOZ instruction. If low, the TMS320C25 executes a branch. This signal must be active during the BIOZ instruction fetch.
$\overline{IACK}$	60	O	Interrupt acknowledge signal. Output is only valid while CLKOUT1 is low. Indicates receipt of an interrupt and that the program is branching to the interrupt-vector location indicated by A15-A0.
$\overline{INT2}$ $\overline{INT1}$ $\overline{INT0}$	22 21 20	I	External user interrupt inputs. Prioritized and maskable by the interrupt mask register and the interrupt mode bit.
MP/$\overline{MC}$	1	I	Microprocessor/microcomputer mode select pin. When asserted low, the pin causes the internal ROM to be mapped into the lower 4K words of the program memory map.
$\overline{MSC}$	59	O	Microstate complete signal. Asserted low and valid only during CLKOUT1 low when the TMS320C25 has just completed a memory operation, such as an instruction fetch or a data memory read/write. $\overline{MSC}$ can be used to generate a one wait-state READY signal for slow memory.
$\overline{RS}$	65	I	Reset input. Causes the TMS320C25 to terminate execution and forces the program counter to zero. When brought to a high level, execution begins at location zero of program memory. $\overline{RS}$ affects various registers and status bits.
XF	56	O	External flag output (latched software-programmable signal). Used for signalling other processors in multiprocessor configurations or as a general-purpose output pin.
			SUPPLY/OSCILLATOR SIGNALS
CLKOUT1	58	O	Master clock output signal (CLKIN frequency/4). Rises at the beginning of quarter-phase 3 (Q3) and falls at the beginning of quarter-phase 1 (Q1).
CLKOUT2	57	O	A second clock output signal. Rises at the beginning of quarter-phase 2 (Q2) and falls at beginning of quarter-phase 4 (Q4).
V_{CC}	23 35 61 62	I	Four 5-V supply pins, tied together externally.
V_{SS}	10 27 44	I	Three ground pins, tied together externally.
X1	51	O	Output pin from the internal oscillator for the crystal. If a crystal is not used, this pin should be left unconnected.
X2/CLKIN	52	I	Input pin to the internal oscillator from the crystal. If a crystal is not used, a clock may be input to the device on this pin.
			SERIAL PORT SIGNALS
CLKR	64	I	Receive clock input. External clock signal for clocking data from the DR (data receive) pin into the RSR (serial port receive shift register). Must be present during serial port transfers.
CLKX	63	I	Transmit clock input. External clock signal for clocking data from the XSR (serial port transmit shift register) to the DX (data transmit) pin. Must be present during serial port transfers.
DR	24	I	Serial data receive input. Serial data is received in the RSR (serial port receive shift register) via the DR pin.
DX	54	O/Z	Serial data transmit output. Serial data transmitted from the XSR (serial port transmit shift register) via the DX pin. Placed in high-impedance state when not transmitting.
FSR	25	I	Frame synchronization pulse for receive input. The falling edge of the FSR pulse initiates the data-receive process by gating the clock for receive (CLKR) input to the DRR (serial port data receive register), and beginning the clocking of the RSR.
FSX	53	I/O	Frame synchronization pulse for transmit input/output. The falling edge of the FSX pulse initiates the data-transmit process by gating the clock for transmit (CLKX) input to the shift register associated with DXR (serial port data transmit register), and beginning the clocking of the XSR. The FSX is normally an input, but this pin is an output when the TXM in the status register is set to 1.

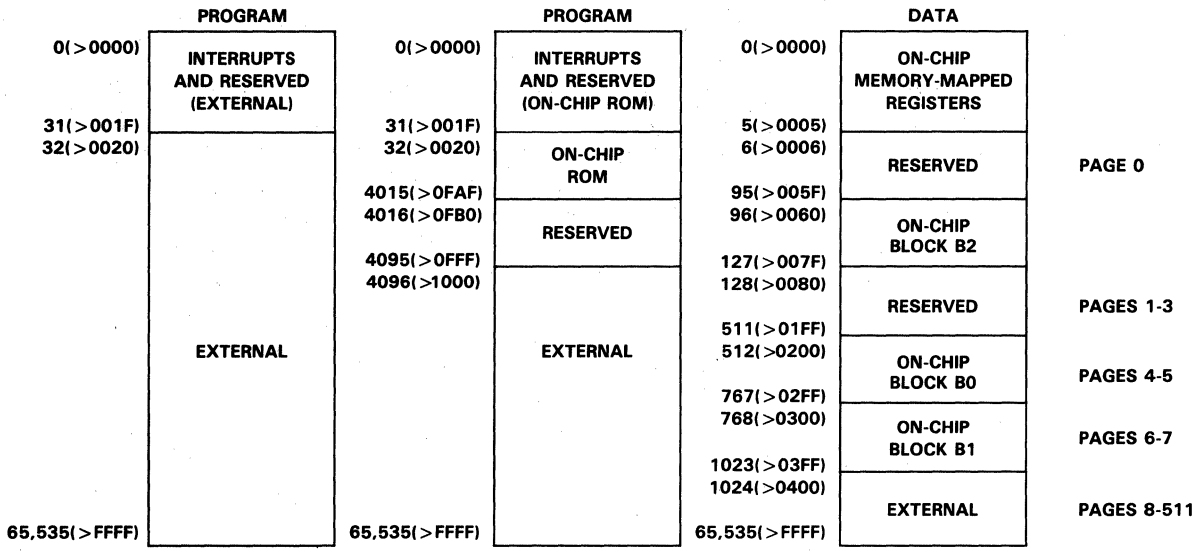
† Input/Output/High-impedance state

2.3 Memory

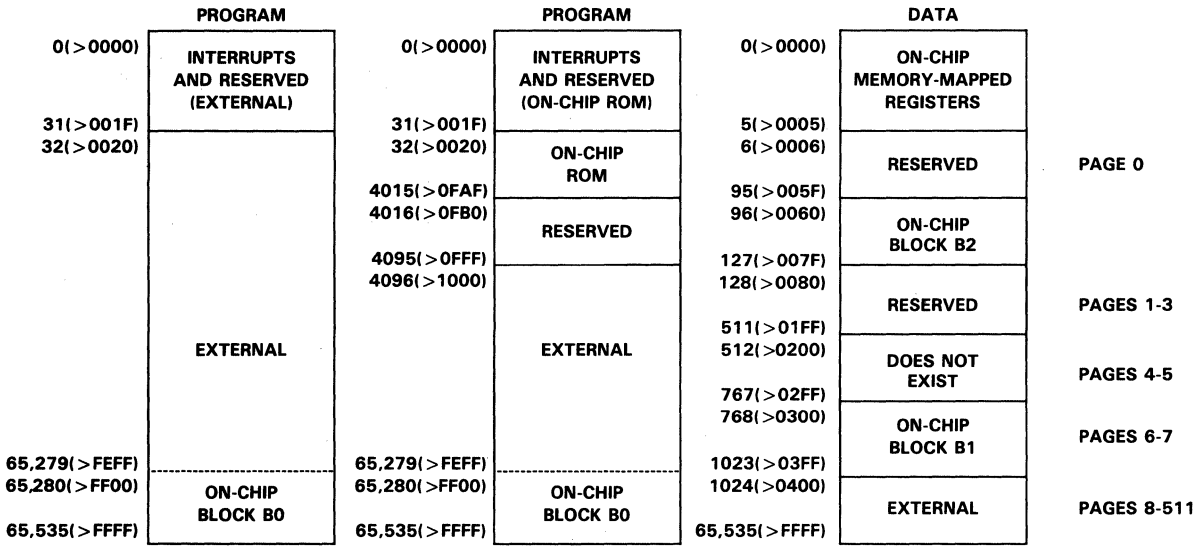
The TMS320C25 provides a total of 544 16-bit words of on-chip data RAM, which is divided into three separate blocks (B0, B1, and B2). Of the 544 words, 256 words (block B0) are configurable as either data or program memory by CNFD or CNFP instructions provided for that purpose; 288 words (blocks B1 and B2) are always data memory. A data memory size of 544 words allows the TMS320C25 to handle a data array of 512 words while still leaving 32 locations for intermediate storage. The TMS320C25 provides 64K words of off-chip directly addressable data memory space.

The TMS320C25 is equipped with a 4096-word on-chip ROM that can be mask-programmed at the factory with a customer's program. The ROM may be mapped in or out of the TMS320C25's memory space by an external pin on the device, MicroProcessor/MicroComputer select (MP/$\overline{MC}$). This permits the designer to accelerate time-to-market with a TMS320C25-based product by using external ROM, and cost-reducing it later with the large 4K internal ROM on the device without any PC-board redesign. The TMS320C25 provides 64K words of off-chip program memory space in which programs can be executed at full speed with sufficiently fast memory or with wait states inserted for slower memories. Block B0 may also be used as program memory. Instructions can be downloaded from slow (inexpensive) external program memory by using block repeats from program to data memory (RPTK and BLKP). The block can then be configured as program memory using the CNFP instruction. In this way, small time-critical blocks of program memory can be stored inexpensively yet executed at full speed.

The TMS320C25 provides three separate address spaces for program memory, data memory, and I/O. In addition to blocks B0, B1, and B2, the data memory map (see Figure 2-3) includes the memory-mapped registers and reserved locations. Six peripheral registers including the serial port registers, timer register, period register, interrupt mask register, and global memory allocation register have been mapped into the data memory space for easy modification. Reserved locations may not be used for storage, and their contents are undefined when read.



(a) MEMORY MAPS AFTER A CNFD INSTRUCTION



(b) MEMORY MAPS AFTER A CNFP INSTRUCTION

Figure 2-3. TMS320C25 Memory Maps

The TMS320C25 provides a register file containing eight Auxiliary Registers (AR0-AR7), which may be used for indirect addressing of data memory or for temporary storage. These registers may be either directly addressed by an instruction or indirectly addressed by a three-bit Auxiliary Register Pointer (ARP). The auxiliary registers and the ARP may be loaded from either data memory or by an immediate operand defined in the instruction. The contents of these registers may also be stored into data memory.

The auxiliary register file is connected to the Auxiliary Register Arithmetic Unit (ARAU). The ARAU may autoindex the current auxiliary register while the data memory location is being addressed. Indexing by either +/-1 or the contents of AR0 may be performed. As a result, accessing tables of information does not require the CALU for address manipulation, thus freeing it for other operations.

Although the ARAU is useful for address manipulation in parallel with other operations, it may also serve as an additional general-purpose arithmetic unit since the auxiliary register file can directly communicate with data memory. The ARAU implements 16-bit unsigned arithmetic, whereas the CALU implements 32-bit two's-complement arithmetic. Branches dependent on the comparison of AR0 with the auxiliary register pointed to by ARP are also provided.

The TMS320C25 contains a 16-bit Program Counter (PC), a 16-bit Prefetch Counter (PFC), a MicroCall Stack (MCS) register, and an eight-level hardware stack for PC storage. The program counter contains the address of the currently executing instruction, either on-chip or off-chip, and the prefetch counter is used for fetching instructions. The eight-level stack is used during interrupts and subroutines, and the MCS is used to store the contents of the PFC during BLKD/BLKP, MAC/MACD, and TBLR/TBLW instructions.

The contents of the accumulator may be loaded into the PC in order to implement "computed go to" operations. The TMS320C25 includes push and pop instructions for nesting of subroutines/interrupts beyond eight levels by allowing a stack to be built in data memory. These instructions store the top of the stack into data memory or load it into the accumulator.

The TMS320C25 local memory interface consists of a 16-bit parallel data bus (D15-D0), a 16-bit program address bus (A15-A0), three pins for data/program memory or I/O space select ($\overline{DS}$, $\overline{PS}$, and $\overline{IS}$), and various system control signals. The R/$\overline{W}$ signal controls the direction of a data transfer, and $\overline{STRB}$ provides a timing signal to control the transfer. When using on-chip program RAM, ROM, or high-speed external program memory, the TMS320C25 runs at full speed without wait states. The use of a READY signal allows wait-state generation for communicating with slower off-chip memories.

The TMS320C25 supports Direct Memory Access (DMA) to its external program/data memory using the $\overline{HOLD}$ and $\overline{HOLDA}$ signals. Another processor can take complete control of the TMS320C25's external memory by asserting $\overline{HOLD}$ low. This causes the TMS320C25 to place its address, data, and control lines in the high-impedance state. Signaling between the external processor and the TMS320C25 can be performed using interrupts. Two modes are available on the device. In the TMS32020-like mode, execution is suspended during assertion of $\overline{HOLD}$. In the new "concurrent DMA" mode, the TMS320C25 continues to execute its program while operating from internal RAM or ROM, thus greatly increasing throughput in data-intensive applications.

2.4 Central Arithmetic Logic Unit (CALU)

The TMS320C25 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, and some additional scalers available at the outputs of both the accumulator and the multiplier.

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 from the accumulator, and the other input may be transferred from the Product Register (PR) of the multiplier or the scaling shifter which is loaded from data memory.

The TMS320C25 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. The LSBs of the output are filled with zeros, and the MSBs may be either filled with zeros or sign-extended, depending upon the state of the sign-extension mode bit of status register ST1. Additional shift capabilities enable the processor to perform numerical scaling, bit extraction, extended arithmetic, and overflow prevention.

The TMS320C25 32-bit ALU and accumulator perform a wide range of arithmetic and logical instructions, the majority of which execute in a single clock cycle. The overflow saturation mode may be programmed through the SOVM and ROVM (set/reset overflow mode) instructions. When the accumulator is in the overflow saturation mode and an overflow occurs, an overflow flag is set and the accumulator is loaded with the most positive/negative number depending upon the direction of overflow.

The 32-bit accumulator is split into two 16-bit segments for storage in data memory: ACCH (accumulator high) and ACCL (accumulator low). Additional shifters at the output of the accumulator provide a shift of 0 to 7 places to the left. This shift is performed while the data is being transferred to the data bus for storage. The contents of the accumulator remain unchanged. The accumulator also has an in-place one-bit shift to the left or right (SFL or SFR instructions) and rotate through carry (ROL or ROR instructions) for shifting the contents of the accumulator.

A carry bit has been added to the TMS320C25 to facilitate multiple-precision arithmetic. The carry bit is affected by all add and subtract instructions. Two new instructions, ADDC (add with carry) and SUBB (subtract wth borrow), use the carry bit when computing a result.

The TMS320C25 utilizes a 16 x 16-bit hardware multiplier, which is capable of computing a 32-bit product during every machine cycle. Two registers are associated with the multiplier:

- A 16-bit Temporary Register (TR) that holds one of the operands for the multiplier, and
- A 32-bit Product Register (PR) that holds the product.

The output of the product register can be left-shifted 1 or 4 bits. This is useful for implementing fractional arithmetic or justifying fractional products. The output of the PR can also be right-shifted 6 bits to enable the execution of up to 128 consecutive multiply/accumulates without overflow.

An unsigned multiply (MPYU) instruction facilitates extended-precision multiplication. The unsigned contents of the T register are multiplied by the unsigned

contents of the addressed data memory location, with the result placed in the P register.

Two multiply/accumulate instructions (MAC and MACD) fully utilize the computational bandwidth of the multiplier, allowing both operands to be processed simultaneously. For MAC and MACD, two operands are transferred to the multiplier each cycle via the program and data buses. This provides for single-cycle multiply/accumulates when used with repeat (RPT or RPTK) instructions. The program bus can supply data from internal or external memory (RAM or ROM) and still maintain single-cycle operation. The SQRA (square/add) and SQRS (square/subtract) instructions pass the same value to both inputs of the multiplier for squaring a data memory value.

The TMS320C25 supports floating-point operations for applications requiring a large dynamic range. A normalization (NORM) instruction is used to normalize fixed-point numbers contained in the accumulator by performing left shifts. The LACT (load accumulator with shift specified by the T register) instruction denormalizes a floating-point number by arithmetically left-shifting the mantissa through the input scaling shifter. The ADDT and SUBT instructions have also been provided to allow additional arithmetic operations with shift specified by the T register. Floating-point numbers with 16-bit mantissas and 4-bit exponents can thus be manipulated.

The device has a variety of branch instructions that are interpreted according to the status of the ALU. Bit test instructions (BIT and BITT) have also been included, which do not affect the accumulator but allow the testing of a specified bit of a word in data memory.

2.5 System Control

Control operations are provided on the TMS320C25 by an on-chip timer, a repeat counter, three external maskable user interrupts, and internal interrupts generated by serial port operations or by the timer.

The TMS320C25 provides a memory-mapped 16-bit timer (TIM) register that is a down counter continuously clocked by CLKOUT1. A timer interrupt (TINT) is generated whenever the timer decrements to zero. The timer is reloaded with the value contained in the period (PRD) register within the next cycle after it reaches zero so that interrupts may be programmed to occur at regular intervals of (PRD + 1) × CLKOUT1 cycles. This feature is useful for control operations and for synchronously sampling or writing to peripherals.

The TMS320C25 design includes a repeat feature that allows a single instruction to be performed up to 256 times. The repeat counter (RPTC) is loaded with either a data memory value (in the case of the RPT instruction) or an immediate value (in the case of the RPTK instruction). The repeat feature can be used with instructions such as multiply/accumulates, block moves, I/O transfers, and table read/writes. Those instructions that are normally multicycle are pipelined when using the repeat feature, and effectively become single-cycle instructions. For example, the table read (TBLR) instruction ordinarily takes four cycles, but when repeated, a table location can be read every cycle.

The TMS320C25 has three external maskable user interrupts ($\overline{\text{INT}}$2-$\overline{\text{INT}}$0) available for external devices that interrupt the processor. Internal interrupts are generated by either the serial port, the timer, or the software interrupt instruction. Interrupts are prioritized with reset having the highest priority and the serial port transmit interrupt having the lowest priority.

The conditions and modes of the TMS320C25 are stored in the two status registers, ST0 and ST1. Instructions allow for storing and loading the status registers into and from data memory. In this manner, the current status of the device may be saved during interrupts and subroutine calls.

2.6 I/O Interface

The TMS320C25 supports a wide range of system interfacing requirements. Three separate address spaces (program, data, and I/O) provide interfacing to memory and I/O, thus maximizing system throughput. I/O design is simplified by having I/O treated the same way as memory. I/O devices are mapped into the I/O address space using the processor's external address and data buses in the same manner as memory-mapped devices. Interfacing to memory and I/O devices of varying speeds is accomplished by using the READY line.

The TMS320C25 I/O space consists of 16 input and 16 output ports. These ports provide the full 16-bit parallel I/O interface via the data bus on the device. A single input or output operation typically takes two cycles; however, when used with the repeat counter, the operation becomes single-cycle.

An on-chip serial port provides direct communication with serial devices such as codecs, serial A/D converters, and other serial systems. The interface signals are compatible with codecs and many other serial devices with a minimum of external hardware. The two serial port memory-mapped registers (the data transmit/receive registers) may be operated in either an 8-bit byte or 16-bit word mode. The transmit framing synchronization pulse can be generated internally or externally. The maximum speed of the serial port is 5 MHz.

The primary enhancements of the TMS320C25's serial port over the TMS32020 are:

- Double-buffering for both receive and transmit operations, thus allowing a continuous bit stream even if FSX is an output,
- No minimum CLKR/CLKX frequency (f_{min} = 0 Hz), and
- Frame sync mode (FSM) bit, which allows continuous operation with no frame synchronization pulses.

The frame sync mode is useful in communicating to "PCM highways." For AT&T T1 and CCITT G711/712 lines, the TMS320C25 can easily be made to communicate directly in these formats by counting the transmitted/received bytes in software and performing SFSM/RFSM instructions as needed to set/reset the FSM bit.

2.7 System Configurations

The flexibility of the TMS320C25 allows configurations to satisfy a wide range of system requirements. The TMS320C25 can be used as follows:

- A standalone system (a single processor using 4K words of on-chip ROM and 544 words of on-chip RAM),
- Parallel multiprocessing systems with shared global data memory, or
- Host/peripheral coprocessing using interface control signals.

The standalone hardware system interface consists of a 16-bit parallel data bus, a 16-bit address bus, three pins for memory space select, and various system control signals. In Figure 2-4, an external data RAM and a PROM/EPROM have been added to the minimum processing system. The READY signal allows wait-state generation for communicating with slower off-chip memories. All the memories and I/O devices

are directly controlled by the TMS320C25, thus minimizing external hardware requirements.

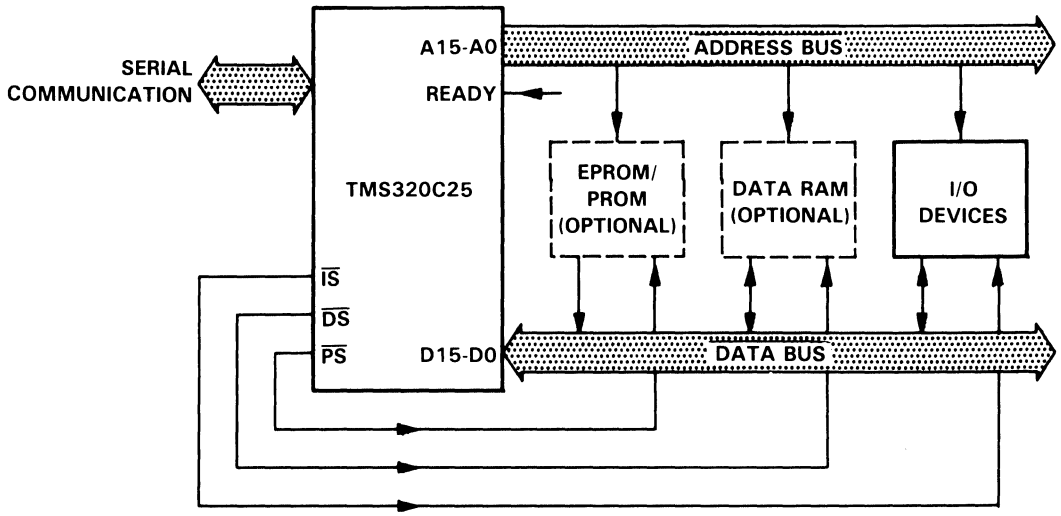


Figure 2-4. A Minimum Processing System

The serial port can interface to serial devices such as codecs and serial A/D converters. Serial communication can also be used between processors, e.g., to connect two minimal systems together to make a multiprocessing system.

For multiprocessing applications, the TMS320C25 has the capability of allocating global data memory space and communicating with that space via the $\overline{BR}$ (bus request) and READY control signals. The 8-bit memory-mapped global memory allocation register (GREG) specifies up to 32K words of the TMS320C25's data memory as global external memory. The contents of the register determine the size of the global memory space. If the current instruction addresses an operand within that space, $\overline{BR}$ is asserted to request control of the bus. The length of the memory cycle is controlled by the READY line.

In a multiprocessing system using global memory, the address space of each processor is divided into local and global sections. Global memory can be used for common data memory storage.

Figure 2-5 shows a configuration for a parallel processing system using global memory. Two TMS320C25s share a global data memory while executing from local program memory. The arbitration for the global memory is handled in software by using the XF and $\overline{BIO}$ pins. The XF pin acts as an external flag, and the $\overline{BIO}$ pin can be polled by a branch (BIOZ) instruction whose condition depends on the state of $\overline{BIO}$.

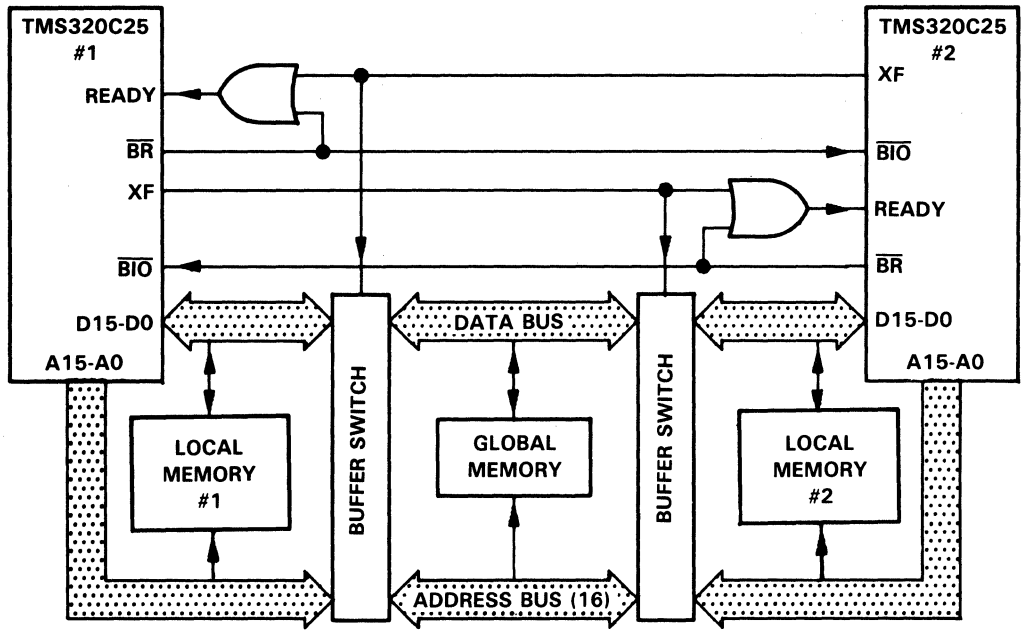


Figure 2-5. Global Memory Parallel Processing

Multiprocessing with the TMS320C25 may also be accomplished through the use of two sets of interface control signals: $\overline{\text{HOLD}}/\overline{\text{HOLDA}}$ and interrupts. $\overline{\text{HOLD}}/\overline{\text{HOLDA}}$ (hold/hold acknowledge) signals allow another microprocessor to read from or write to the local off-chip data/program memory of the temporarily halted processor. Using these signals to implement direct memory access is useful for downloading to or initializing the TMS320C25. In interrupt-driven multiprocessing, time-critical operations can be protected by masking out interrupts.

The TMS320C25 has been enhanced to provide a new hold mode that provides the ability to perform concurrent DMA. The new hold mode has been defined so that if the device is executing from on-chip program memory (ROM or RAM) and $\overline{\text{HOLD}}$ is asserted, the device is not halted, but instead proceeds with program execution until an external access must be made. This greatly enhances system throughput in multiprocessing applications.

Many applications require a digital signal processing-type peripheral interface to a general-purpose 16- or 16/32-bit microcomputer. Such configurations are often useful when a general-purpose system is already available. A host/peripheral configuration using the interface control signals of $\overline{\text{HOLD}}/\overline{\text{HOLDA}}$ is shown in Figure 2-6.

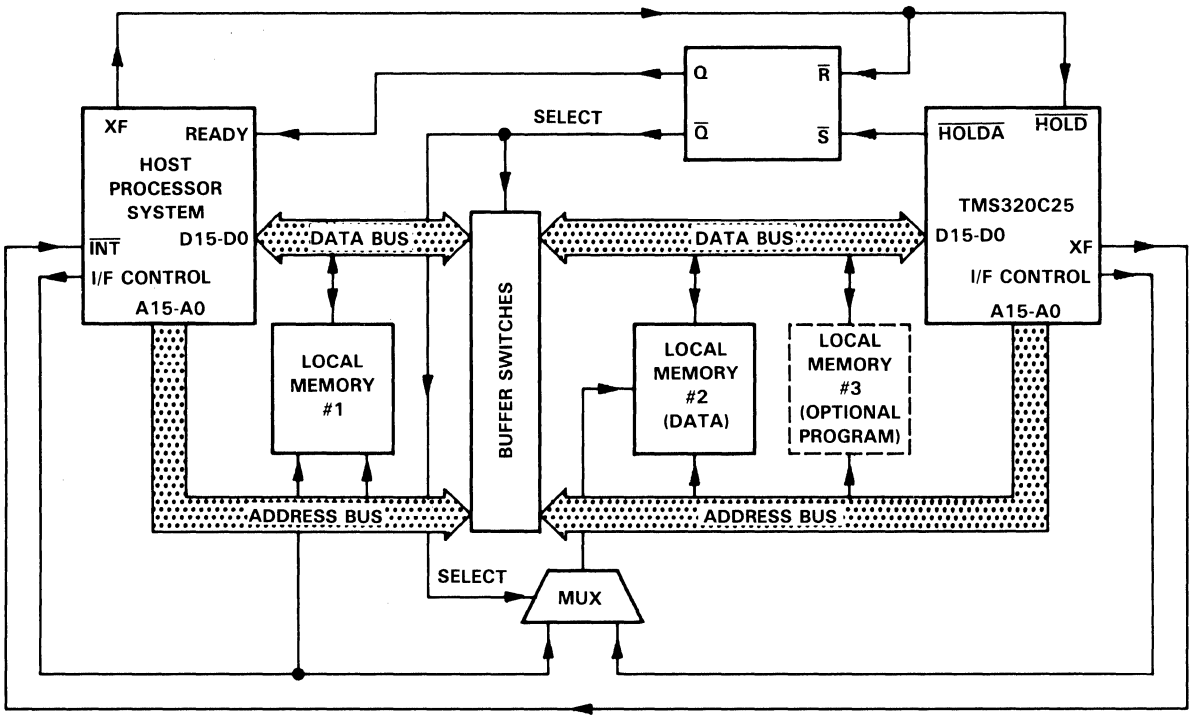


Figure 2-6. Host/Peripheral Coprocessing Using Interface Control Signals

A great advantage to using the TMS320C25 in a multiprocessor system is its ability to be synchronized to an external signal. A special $\overline{\text{SYNC}}$ pin allows the internal clocks of two or more TMS320C25s to be synchronized. Since the processors operate on the same internal clock phase, all external signals will also be synchronized, eliminating the need for external logic to synchronize interprocessor signals.

2.8 Addressing Modes and Instructions

The TMS320C25 instruction set supports numeric-intensive signal processing operations as well as general-purpose applications such as multiprocessing and high-speed control. The TMS320C25 is completely object code upward-compatible with the TMS32020 so that TMS32020 programs run unmodified on the TMS320C25. The TMS32010 source code is upward-compatible with the TMS320C25 source code.

For maximum throughput, the current instruction is executed while the next instruction is decoded and the one following that is prefetched. Since the same data lines are used to communicate to external data/program or I/O space, the number of cycles may vary depending upon whether the next data operand fetch is from internal or external memory. Highest throughput is achieved by maintaining data memory on-chip and using either internal or fast external program memory.

Three memory addressing modes are available with the TMS320C25 instruction set: direct, indirect, and immediate addressing. Both direct and indirect addressing can be used to access data memory. When using direct addressing, seven bits of the instruction word are concatenated with the nine bits of the Data memory page Pointer (DP) to form the 16-bit data memory address. With a 128-word page length, the DP register points to one of 512 possible data memory pages to obtain a 64K total data memory space. The seven-bit address in the instruction points to the specific location within the data memory page. Direct addressing can be used with all instructions except CALL, the branch instructions, immediate operand instructions, and instructions with no operands.

Flexible and powerful indirect addressing is provided by the eight auxiliary registers (AR0-AR7). The data address to be used in an instruction is placed into one of eight auxiliary registers. To select a specific auxiliary register, the Auxiliary Register Pointer (ARP) is loaded with a value from 0 through 7, designating AR0 through AR7, respectively. The ARAU implements 16-bit unsigned arithmetic, performing auxiliary register arithmetic operations in the same cycle as the execution of the instruction.

There are seven types of indirect addressing: indexing with either increment or decrement, indexing by either adding or subtracting the contents of AR0, indexing by either adding or subtracting the contents of AR0 with the carry propagation reversed (for FFTs), or no indexing (see Table 2-2). All indexing operations are performed on the current auxiliary register in the same cycle as the original instruction, with an optional new ARP value being loaded.

Bit-reversed indexed addressing modes allow efficient I/O to be performed for 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 AR0 is added to/subtracted from the current auxiliary register. Typical use of this addressing mode requires that AR0 first be set to a value corresponding to one-half of the array size, and AR (ARP) be set to the base address of the data (the first data point).

Table 2-2. Addressing Modes

ADDRESSING MODE	OPERATION
OP A	Direct addressing
OP *(,NARP)	Indirect; no change to AR.
OP *+(,NARP)	Indirect; current AR is incremented.
OP *-(,NARP)	Indirect; current AR is decremented.
OP *0+(,NARP)	Indirect; AR0 is added to current AR.
OP *0-(,NARP)	Indirect; AR0 is subtracted from current AR.
OP *BR0+(,NARP)	Indirect; AR0 is added to current AR (with reverse carry propagation).
OP *BR0-(,NARP)	Indirect; AR0 is subtracted from current AR (with reverse carry propagation).

NOTE: The optional NARP field specifies a new value of the ARP.

In immediate addressing, the instruction word contains the value of the immediate operand. The TMS320C25 has both single-word (8-bit and 13-bit constant) short immediate instructions and two-word (16-bit constant) long immediate instructions. In the case of long (16-bit constant) immediate instructions, the word following the instruction opcode is used as the immediate operand. Included in the TMS320C25's instruction set are 17 immediate operand instructions.

Table 2-3 defines the symbols and abbreviations used in the operation portion of the list of TMS320C25 instructions (Table 2-4).

Table 2-3. Instruction Symbols

SYMBOL	MEANING
ACC	Accumulator
ARB	Auxiliary register pointer buffer
ARn	Auxiliary Register n (AR0 through AR7 are predefined assembler symbols equal to 0 through 7, respectively.)
ARP	Auxiliary register pointer
BIO	Branch control input
C	Carry bit
CM	2-bit field specifying compare mode
CNF	On-chip RAM configuration control bit
dma	Data memory address
DP	Data page pointer
FO	Format status bit
FSM	Frame synchronization mode bit
HM	Hold mode bit
INTM	Interrupt mode flag bit
>nn	Indicates nn is a hexadecimal number. (All others are assumed to be decimal values.)
OV	Overflow flag bit
OVM	Overflow mode bit
P	Product register
PA	Port address. (PA0 through PA15 are predefined assembler symbols equal to 0 through 15, respectively.)
PC	Program counter
PM	2-bit field specifying P register output shift code
pma	Program memory address
Preg	Product register
RPTC	Repeat counter
STn	Status Register n (ST0 or ST1)
SXM	Sign-extension mode bit
T	Temporary register
TC	Test control bit
TOS	Top of stack
Treg	Temporary register
TXM	Transmit mode bit
Usgn	Unsigned value
XF	XF pin status bit
→	Is assigned to
| |	An absolute value
[ ]	Optional items
( )	Contents of

Twenty-four new instructions have been added to the TMS320C25 instruction set to improve overall processor throughput and ease of use. These new instructions can be categorized into the following four groups:

- Extended-precision arithmetic (ADDC, SUBB, MPYU, BC, BNC, SC, RC)
- Adaptive filtering (MPYA, MPYS, ZALR)
- Control and I/O (SHM, RHM, STC, RTC, SFSM, RFSM)
- Accumulator and register instructions (SPH, SPL, ADDK, SUBK, ADRK, SBRK, ROL, ROR)

The list of TMS320C25 instructions in Table 2-4 is organized according to function and alphabetized within each functional grouping. The symbol (†) indicates instructions that are not included in the TMS32010 instruction set, and the symbol (‡) those not included in the TMS32020 instruction set.

Table 2-4. TMS320C25 Instructions

ACCUMULATOR MEMORY REFERENCE INSTRUCTIONS			
MNEMONIC	DESCRIPTION	NO. WORDS	OPERATION
ABS	Absolute value of accumulator	1	|(ACC)| → ACC
ADD	Add to accumulator with shift	1	(ACC) + [(dma) × 2^{shift}] → ACC
ADDC‡	Add to accumulator with carry	1	(ACC) + (dma) + (C) → ACC
ADDH	Add to high accumulator	1	(ACC) + [(dma) × 2^{16}] → ACC
ADDK‡	Add to accumulator short immediate	1	(ACC) + 8-bit constant → ACC
ADDS	Add to low accumulator with sign extension suppressed	1	(ACC) + (dma) → ACC
ADDT†	Add to accumulator with shift specified by T register	1	(ACC) + [(dma) × $2^{(Treg)}$] → ACC
ADLK†	Add to accumulator long immediate with shift	2	(ACC) + [16-bit constant × 2^{shift}] → ACC
AND	AND with accumulator	1	(ACC(15-0)).AND.(dma) → ACC(15-0), 0 → ACC(31-16)
ANDK†	AND immediate with accumulator with shift	2	(ACC)30-0)).AND.[16-bit constant × 2^{shift}] → ACC(30-0), 0 → ACC(30-0)
CMPL†	Complement accumulator	1	$(\overline{ACC})$ → ACC
LAC	Load accumulator with shift	1	(dma) × 2^{shift} → ACC
LACK	Load accumulator immediate short	1	8-bit constant → ACC
LACT†	Load accumulator with shift specified by T register	1	(dma) × $2^{(Treg)}$ → ACC
LALK†	Load accumulator long immediate with shift	2	(16-bit constant) × 2^{16} → ACC
NEG†	Negate accumulator	1	−(ACC) → ACC
NORM†	Normalize contents of accumulator	1	
OR	OR with accumulator	1	(ACC(15-0)).OR. (dma) → ACC(15-0)
ORK†	OR immediate with accumulator with shift	2	(ACC(30-0)).OR.[16-bit constant × 2^{shift}] → ACC(30-0)
ROL‡	Rotate accumulator left	1	(ACC(30-0)) → ACC(31-1), (C) → ACC(0), (ACC(31)) → C
ROR‡	Rotate accumulator right	1	(ACC(31-1)) → ACC(30-0), (C) → ACC(31), (ACC(0)) → C
SACH	Store high accumulator with shift	1	[(ACC) × 2^{shift}] → dma
SACL	Store low accumulator with shift	1	[(ACCL) × 2^{shift}] → dma
SBLK†	Subtract from accumulator long immediate with shift	2	(ACC) − [16-bit constant × 2^{shift}] → ACC
SFL†	Shift accumulator left	1	(ACC(30-0)) → ACC(31-1), 0 → ACC(0)
SFR†	Shift accumulator right	1	(ACC(31-1)) → ACC(30-0), (ACC(31)) → ACC(31)
SUB	Subtract from accumulator with shift	1	(ACC) − [(dma) × 2^{shift}] → ACC
SUBB‡	Subtract from accumulator with borrow	1	(ACC) − (dma) − $(\overline{C})$ → ACC
SUBC	Conditional subtract	1	
SUBH	Subtract from high accumulator	1	(ACC) − [(dma) × 2^{16}] → ACC
SUBK‡	Subtract from accumulator short immediate	1	(ACC) − 8-bit constant → ACC
SUBS	Subtract from low accumulator with sign extension suppressed	1	(ACC) − (dma) → ACC
SUBT†	Subtract from accumulator with shift specified by T register	1	(ACC) − [(dma) × $2^{(Treg)}$] → ACC
XOR	Exclusive-OR with accumulator	1	(ACC(15-0)).XOR.(dma) → ACC(15-0)
XORK†	Exclusive-OR immediate with accumulator with shift	2	(ACC(30-0)).XOR.[16-bit constant × 2^{shift}] → ACC(30-0)
ZAC	Zero accumulator	1	0 → ACC
ZALH	Zero low accumulator and load high accumulator	1	(dma) × 2^{16} → ACC
ZALR‡	Zero low accumulator and load high accumulator with rounding	1	(dma) × 2^{16} + >8000→ ACC
ZALS	Zero accumulator and load low accumulator with sign extension suppressed	1	(dma) → ACCL, 0 → ACCH

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

Table 2-4. TMS320C25 Instructions (Continued)

AUXILIARY REGISTERS AND DATA PAGE POINTER INSTRUCTIONS			
MNEMONIC	**DESCRIPTION**	**NO. WORDS**	**OPERATION**
ADRK‡	Add to auxiliary register short immediate	1	(ARn) + 8-bit constant → ARn
CMPR†	Compare auxiliary register with auxiliary register AR0	1	If ARn | CM | AR0, then 1 → TC; else 0 → TC
LAR	Load auxiliary register	1	(dma) → (ARn)
LARK	Load auxiliary register short immediate	1	8-bit constant → ARn
LARP	Load auxiliary register pointer	1	3-bit constant → ARP, (ARP) → ARB
LDP	Load data memory page pointer	1	(dma) → DP
LDPK	Load data memory page pointer immediate	1	9-bit constant → DP
LRLK†	Load auxiliary register long immediate	2	16-bit constant → ARn
MAR	Modify auxiliary register	1	
SAR	Store auxiliary register	1	(ARn) → dma
SBRK‡	Subtract from auxiliary register short immediate	1	(ARn) − 8-bit constant → ARn
T REGISTER, P REGISTER, AND MULTIPLY INSTRUCTIONS			
MNEMONIC	**DESCRIPTION**	**NO. WORDS**	**OPERATION**
APAC	Add P register to accumulator	1	(ACC) + (shift Preg) → ACC
LPH†	Load high P register	1	(dma) → Preg (31-16)
LT	Load T register	1	(dma) → Treg
LTA	Load T register and accumulate previous product	1	(dma) → Treg, (ACC) + (shifted Preg) → ACC
LTD	Load T register, accumulate previous product, and move data	1	(dma) → Treg, (dma) → dma + 1, (ACC) + (shifted Preg) → ACC
LTP†	Load T register and store P register in accumulator	1	(dma) → Treg, (shifted Preg) → ACC
LTS†	Load T register and subtract previous product	1	(dma) → Treg, (ACC) − (shifted Preg) → ACC
MAC†	Multiply and accumulate	2	(ACC) + (shifted Preg) → ACC, (pma) × (dma) → Preg
MACD†	Multiply and accumulate with data move	2	(ACC) + (shifted Preg) → ACC, (pma) × (dma) → Preg, (dma) → dma + 1
MPY	Multiply (with T register, store product in P register)	1	(Treg) × (dma) → Preg
MPYA‡	Multiply and accumulate previous product	1	(ACC) + (shifted Preg) → ACC, (Treg) × (dma) → Preg
MPYK	Multiply immediate	1	(Treg) × 13-bit constant → Preg
MPYS‡	Multiply and subtract previous product	1	(ACC) − (shifted Preg) → ACC, (Treg) × (dma) → Preg
MPYU‡	Multiply unsigned	1	Usgn (Treg) × Usgn (dma) → Preg
PAC	Load accumulator with P register	1	(shifted Preg) → ACC
SPAC	Subtract P register from accumulator	1	(ACC) − (shifted Preg) → ACC
SPH‡	Store high P register	1	(shifted Preg (31-16)) → dma
SPL‡	Store low P register	1	(shifted Preg (15-0)) → dma
SPM†	Set P register output shift mode	1	2-bit constant → PM
SQRA†	Square and accumulate	1	(ACC) + (shifted Preg) → ACC, (dma) × (dma) → Preg
SQRS†	Square and subtract previous product	1	(ACC) − (shifted Preg) → ACC, (dma) × (dma) → Preg

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

Table 2-4. TMS320C25 Instructions (Continued)

BRANCH/CALL INSTRUCTIONS			
MNEMONIC	**DESCRIPTION**	**NO. WORDS**	**OPERATION**
B	Branch unconditionally	2	pma → PC
BACC†	Branch to address specified by accumulator	1	(ACC(15-0)) → PC
BANZ	Branch on auxiliary register not zero	2	If (AR(ARP)) ≠ 0, then pma → PC; else (PC) + 2 → PC
BBNZ†	Branch if TC bit ≠ 0	2	If (TC) = 1, then pma → PC; else (PC) + 2 → PC
BBZ†	Branch if TC bit = 0	2	If (TC) = 0, then pma → PC; else (PC) + 2 → PC
BC‡	Branch on carry	2	If (C) = 1, then pma → PC; else (PC) + 2 → PC
BGEZ	Branch if accumulator ≥ 0	2	If (ACC) ≥ 0, then pma → PC; else (PC) + 2 → PC
BGZ	Branch if accumulator > 0	2	If (ACC) > 0, then pma → PC; else (PC) + 2 → PC
BIOZ	Branch on I/O status = 0	2	If ($\overline{BIO}$) = 0, then pma → PC; else (PC) + 2 → PC
BLEZ	Branch if accumulator ≤ 0	2	If (ACC) ≤ 0, then pma → PC; else (PC) + 2 → PC
BLZ	Branch if accumulator < 0	2	If (ACC) < 0, then pma → PC; else (PC) + 2 → PC
BNC‡	Branch on no carry	2	If (C) = 0, then pma → PC; else (PC) + 2 → PC
BNV†	Branch if no overflow	2	If (OV) ≠ 0, then pma → PC; else (PC) + 2 → PC
BNZ	Branch if accumulator ≠ 0	2	If (ACC) ≠ 0, then pma → PC; else (PC) + 2 → PC
BV	Branch on overflow	2	If (OV) = 0, then pma → PC; else (PC) + 2 → PC
BZ	Branch if accumulator = 0	2	If (ACC) = 0, then pma → PC; else (PC) + 2 → PC
CALA	Call subroutine indirect	1	(ACC(15-0)) → PC, (PC) + 1 → TOS
CALL	Call subroutine	2	(PC) + 2 → TOS, pma → PC
RET	Return from subroutine	1	(TOS) → PC
I/O AND DATA MEMORY OPERATIONS			
MNEMONIC	**DESCRIPTION**	**NO. WORDS**	**OPERATION**
BLKD†	Block move from data memory to data memory	2	(dma1, addressed by PC) → dma2
BLKP†	Block move from program memory to data memory	2	(pma, addressed by PC) → dma
DMOV	Data move in data memory	1	(dma) → dma + 1
FORT†	Format serial port registers	1	1-bit constant → FO
IN	Input data from port	1	(data bus, addressed by PA) → dma
OUT	Output data to port	1	(dma) → data bus, addressed by PA
RFSM‡	Reset serial port frame synchronization mode	1	0 → FSM
RTXM†	Reset serial port transmit mode	1	0 → TXM
RXF†	Reset external flag	1	0 → XF
SFSM‡	Set serial port frame synchronization mode	1	1 → FSM
STXM†	Set serial port transmit mode	1	1 → TXM
SXF†	Set external flag	1	1 → XF
TBLR	Table read	1	(pma, addressed by ACC (15-0)) → dma
TBLW	Table write	1	(dma) → pma, addressed by ACC (15-0)

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

Table 2-4. TMS320C25 Instructions (Concluded)

CONTROL INSTRUCTIONS			
MNEMONIC	**DESCRIPTION**	**NO. WORDS**	**OPERATIONS**
BIT†	Test bit	1	(dma bit at (15-bit code)) → TC
BITT†	Test bit specified by T register	1	(dma bit at (15-Treg)) → TC
CNFD†	Configure block as data memory	1	0 → CNF
CNFP†	Configure block as program memory	1	1 → CNF
DINT	Disable interrupt	1	1 → INTM
EINT	Enable interrupt	1	0 → INTM
IDLE†	Idle until interrupt	1	(PC) + 1 → PC, powerdown
LST	Load status register ST0	1	(dma) → ST0
LST1†	Load status register ST1	1	(dma) → ST1
NOP	No operation	1	(PC) + 1 → PC
POP	Pop top of stack to low accumulator	1	(TOS) → ACC
POPD†	Pop top of stack to data memory	1	(TOS) → dma
PSHD†	Push data memory value onto stack	1	(dma) → TOS
PUSH	Push low accumulator onto stack	1	(ACCL) → TOS
RC‡	Reset carry bit	1	0 → C
RHM‡	Reset hold mode	1	0 → HM
ROVM	Reset overflow mode	1	0 → OVM
RPT†	Repeat instruction as specified by data memory value	1	(dma) → RPTC
RPTK†	Repeat instruction as specified by immediate value	1	8-bit constant → RPTC
RSXM†	Reset sign-extension mode	1	0 → SXM
RTC‡	Reset test/control flag	1	0 → TC
SC‡	Set carry bit	1	1 → C
SHM‡	Set hold mode	1	1 → HM
SOVM	Set overflow mode	1	1 → OVM
SST	Store status register ST0	1	ST0 → dma
SST1†	Store status register ST1	1	ST1 → dma
SSXM†	Set sign-extension mode	1	1 → SXM
STC‡	Set test/control flag	1	1 → TC
TRAP†	Software interrupt	1	(PC) + 1 → TOS, 30 → PC

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

2.9 Development Support

Texas Instruments offers extensive development support and documentation for the TMS320 family (see Figure 2-7). Sophisticated development operations are performed with the TMS320C25 Macro Assembler/Linker, Simulator, and Emulator to evaluate the performance of the processor, develop algorithms, and fully integrate the design's software and hardware modules. Since the TMS320C25 is pin-compatible with the TMS32020, development can begin immediately by utilizing the broad base of TMS32020 support tools (see Appendix F).

Extensive documentation, including application reports, user's guides, and textbooks, is available to support DSP design, research, and education. When questions arise, additional support can be obtained by contacting the Texas Instruments Customer Response Center (CRC) hotline number, 1-800-232-3200.

Figure 2-7. TMS320 Family Development Support

TMS320C25 MACRO ASSEMBLER/LINKER

The TMS320C25 Macro Assembler translates TMS320C25 assembly language source code into executable object code. The assembler allows the programmer to work with mnemonics rather than hexadecimal machine instructions and to reference memory locations with symbolic addresses. The macro assembler supports macro calls and definitions along with conditional assembly.

The TMS320C25 Linker permits a program to be designed and implemented in separate modules that will later be linked together to form the complete program. The linker resolves external definitions and references for relocatable code, creating an object file that can be executed by the TMS320C25 Simulator, TMS320C25 Emulator, or TMS320C25 processor.

The TMS320C25 Macro Assembler/Linker is supported on the VAX/VMS, TI PC/MS-DOS, and IBM PC/PC-DOS operating systems.

TMS320C25 SIMULATOR

The TMS320C25 Simulator is a software program that simulates operation of the TMS320C25 to allow program verification. The debug mode enables the user to monitor the state of the simulated TMS320C25 while the program is executing. The simulator uses the TMS320C25 object code produced by the TMS320C25 Macro Assembler/Linker. During program execution, the internal registers and memory of the simulated TMS320C25 are modified as each instruction is interpreted by the host computer. Once program execution is suspended, the internal registers and both program and data memories can be inspected and/or modified.

The TMS320C25 Simulator is supported on the VAX/VMS, TI PC/MS-DOS, and IBM PC/PC-DOS operating systems.

TMS320C25 EMULATOR

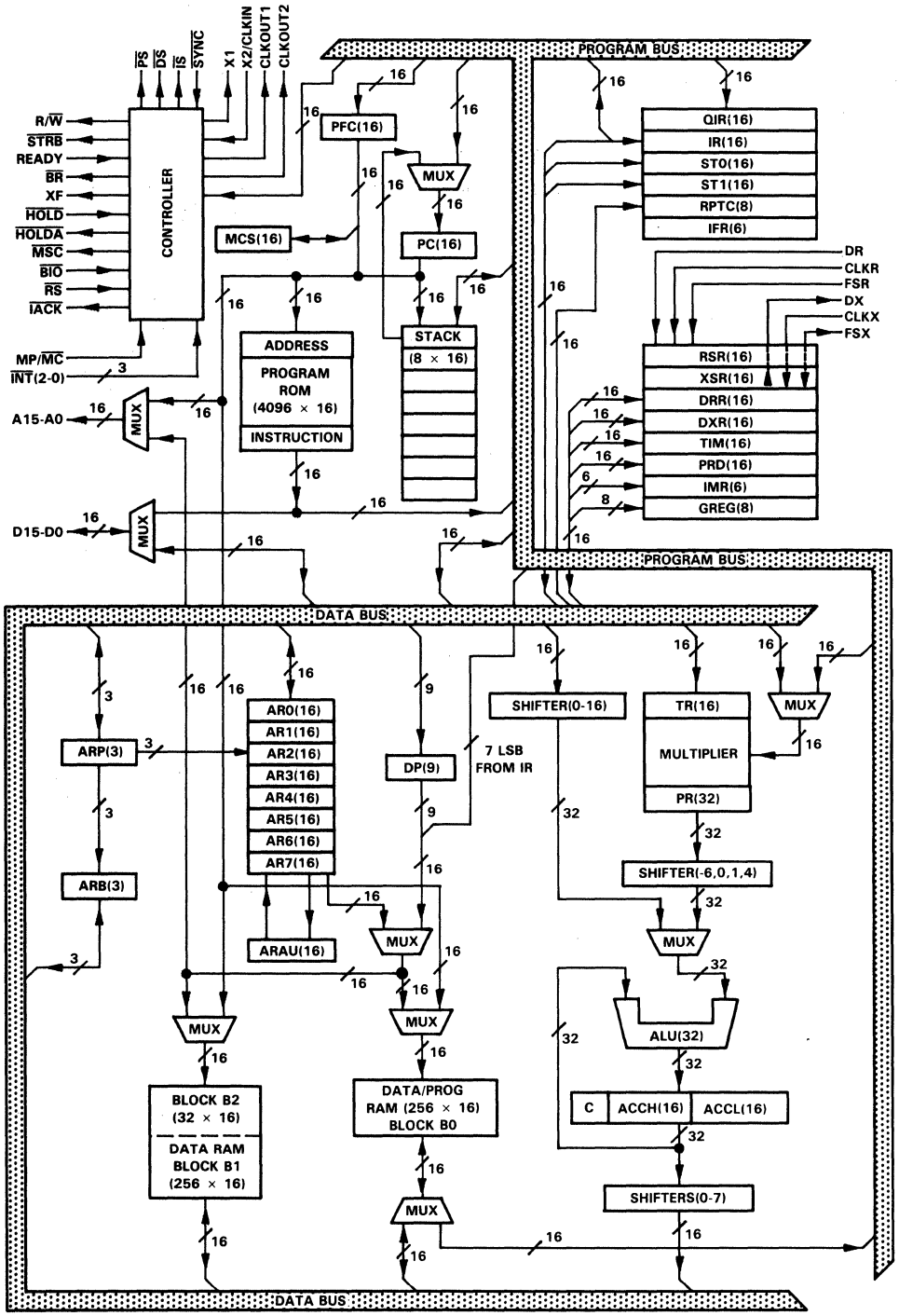
The TMS320C25 Emulator (XDS/22) is a user-friendly system that has all the features necessary for realtime in-circuit emulation. This allows integration of the hardware and software modules in the debug mode. By setting breakpoints based on internal conditions or external events, execution of the program can be suspended and control given to the debug mode. In the debug mode, all registers and memory locations can be inspected and modified. Single-step execution is available. Full trace capabilities at full speed and a reverse assembler that translates machine code back into assembly instructions also increase debugging productivity.

The TMS320C25 Emulator is a completely self-contained system. With three RS-232-C ports, it can be interfaced to a terminal, host computer for source or object downloading/uploading capabilities, and printer or PROM programmer. The emulator has 4K x 16-bit words of high-speed static RAM (zero wait states) for program memory. The XDS/22 also supports memory expansion by including 64K words of DRAM. This slower memory is configurable by the user as either all program memory, all data memory, or 32K words of each.

3. Device Operation

The TMS320C25 microprocessor architectural design emphasizes overall speed, communication, and flexibility in processor configuration. Control signals and instructions provide block-memory transfers, communication to slower off-chip devices, and multiprocessing implementations. Increased throughput for many digital signal processing (DSP) applications is accomplished by single-cycle multiply/accumulate instructions, two large on-chip RAM blocks, eight auxiliary registers with a dedicated arithmetic unit, a serial port, hardware timer, faster I/O for data-intensive signal processing, and many other features. Figure 2-1 shows the functional block diagram of the TMS320C25 processor.

Major topics discussed in this section are listed below.



LEGEND:

ACCH	= Accumulator high	IFR	= Interrupt flag register	PC	= Program counter
ACCL	= Accumulator low	IMR	= Interrupt mask register	PFC	= Prefetch counter
ALU	= Arithmetic logic unit	IR	= Instruction register	RPTC	= Repeat instruction counter
ARAU	= Auxiliary register arithmetic unit	MCS	= Microcall stack	GREG	= Global memory allocation register
ARB	= Auxiliary register pointer buffer	QIR	= Queue instruction register	RSR	= Serial port receive shift register
ARP	= Auxiliary register pointer	PR	= Product register	XSR	= Serial port transmit shift register
DP	= Data memory page pointer	PRD	= Period register for timer	AR0-AR7	= Auxiliary registers
DRR	= Serial port data receive register	TIM	= Timer	ST0,ST1	= Status registers
DXR	= Serial port data transmit register	TR	= Temporary register		

Figure 3-1. TMS320C25 Block Diagram

3.1 Internal Hardware Summary

The TMS320C25 internal hardware implements functions that other processors typically perform in software or microcode. For example, the device contains hardware for single-cycle 16 x 16-bit multiplication, data shifting, and address manipulation. This hardware-intensive approach provides computing power previously unavailable on a single chip.

Table 3-1 presents a summary of the TMS320C25 internal hardware. This summary table, which includes the internal processing elements, registers, and buses, is alphabetized within each functional grouping. All of the symbols used in this table correspond to the symbols used in the block diagram of Figure 3-1, the succeeding block diagrams in this section, and the text throughout this document.

Table 3-1. Internal Hardware

UNIT	SYMBOL	FUNCTION
PROCESSING ELEMENTS		
Arithmetic Logic Unit	ALU	A 32-bit two's-complement arithmetic logic unit having two 32-bit input ports and one 32-bit output port feeding the accumulator.
Central Arithmetic Logic Unit	CALU	The grouping of the ALU, multiplier, accumulator, and scaling shifter.
Multiplier	MULT	A 16 x 16-bit parallel multiplier.
Period Register	PRD (15-0)	A 16-bit memory-mapped register used to reload the timer.
Program Counter	PC (15-0)	A 16-bit program counter used to address program memory. The PC always contains the address of the next instruction to be executed. The PC contents are updated following each instruction decode operation.
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. The PFC is updated when a new prefetch is initiated. The PFC is also used to address data memory when using the block move (BLKD and BLKP), multiply/accumulate (MAC and MACD), and table read/write (TBLR and TBLW) instructions.
Random Access Memory (data or program)	RAM (B0)	A RAM block with 256 x 16 locations configured either as data or program memory.
Random Access Memory (data only)	RAM (B1)	A data RAM block, organized as 256 x 16 locations.
Random Access Memory (data only)	RAM (B2)	A data RAM block, organized as 32 x 16 locations.
Auxiliary Register Arithmetic Unit	ARAU	A 16-bit unsigned arithmetic unit used to perform operations on auxiliary register data.
Repeat Counter	RPTC (7-0)	An 8-bit counter to control the repeated execution of a single instruction.
Shifters	SFL, SFR	Shifters SFL (left) and SFR (right) are located at the ALU input, the accumulator output, and the product register output. Also an in-place shifter within the accumulator.
Timer	TIM (15-0)	A 16-bit memory-mapped timer (counter) for timing control.
Accumulator	ACC (31-0) ACCH(31-16) ACCL(15-0)	A 32-bit accumulator split in two halves: ACCH (accumulator high) and ACCL (accumulator low). Used for storage of ALU output.
Auxiliary Register File	AR0,AR1,AR2 AR3,AR4,AR5 AR6,AR7 (15-0)	A register file containing 8 16-bit auxiliary registers (AR0-AR7), used for addressing data memory, for temporary storage, or for integer arithmetic processing through the ARAU.
Auxiliary Register Pointer	ARP(2-0)	A 3-bit register used to select one of the eight auxiliary registers.

Table 3-1. Internal Hardware (Continued)

UNIT	SYMBOL	FUNCTION
REGISTERS		
Auxiliary Register Pointer Buffer	ARB(2-0)	A 3-bit register used to buffer the ARP. Each time the ARP is loaded, the old value is written to the ARB, except during an LST (load status register) instruction. When the ARB is loaded with an LST1, the same value is also copied into ARP.
Data Memory Page Pointer	DP(8-0)	A 9-bit register pointing to 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).
Global Memory Allocation Register	GREG(7-0)	An 8-bit memory-mapped register for allocating the size of the global memory space.
Instruction Register	IR(15-0)	A 16-bit register used to store the currently executing instruction.
Queue Instruction Register	QIR(15-0)	A 16-bit register used to store prefetched instructions.
Interrupt Flag Register	IFR(5-0)	A 6-bit flag register used to latch the active-low external user interrupts $\overline{\text{INT}}$(2-0) and the internal interrupts XINT/RINT (serial port transmit/receive interrupts) and TINT (timer interrupt). The IFR is not accessible through software.
Interrupt Mask Register	IMR(5-0)	A 6-bit memory-mapped register used to mask interrupts.
Product Register	PR(31-0) PH(31-16) PL(15-0)	A 32-bit product register used to hold the multiplier product. The PR can also be accessed as the most or least significant words using the SPH (store P register high) or SPL (store P register low) instructions.
Stack	Stack(15-0)	An 8 x 16 hardware stack used to store the PC during interrupts or calls. The ACCL and data memory values may also be pushed onto and popped from the stack.
MicroCall Stack	MCS (15-0)	A single-word stack that temporarily stores the contents of the PFC while the PFC is being used to address data memory with the block move (BLKD and BLKP), multiply/accumulate (MAC and MACD), and table read/write (TBLR and TBLW) instructions.
Serial Port Data Receive Register	DRR(15-0)	A 16-bit memory-mapped serial port data receive register. Only the eight LSBs are used in the byte mode.
Serial Port Data Transmit Register	DXR(15-0)	A 16-bit memory-mapped serial port data transmit register. Only the eight LSBs are used in the byte mode.
Serial Port Receive Shift Register	RSR(15-0)	A 16-bit register used to shift in serial port data from the RX pin. RSR contents are sent to the DRR after a serial transfer is completed. RSR is not directly accessible through software.
Serial Port Transmit Shift Register	XSR(15-0)	A 16-bit register used to shift out serial port data onto the DX pin. XSR contents are loaded from DXR at the beginning of a serial port transmit operation. XSR is not directly accessible through software.
Status Registers	ST0,ST1 (15-0)	Two 16-bit status registers that contain status and control bits.
Temporary Register	TR(15-0)	A 16-bit register that holds either an operand for the multiplier or a shift code for the scaling shifter.

Table 3-1. Internal Hardware (Concluded)

UNIT	SYMBOL	FUNCTION
BUSES		
Auxiliary Register File Bus	AFB(15-0)	A 16-bit bus that carries data from the AR pointed to by the ARP.
Data Bus	D(15-0)	A 16-bit bus used to route data.
Data Memory Address Bus	DAB(15-0)	A 16-bit bus that carries the data memory address.
Direct Data Memory Address Bus	DRB(15-0)	A 16-bit bus that carries the 'direct' address for the data memory, which is the concatenation of the DP register with the seven LSBs of the instruction.
Program Bus	P(15-0)	A 16-bit bus used to route instructions (and data for the MAC and MACD instructions).
Program Memory Address Bus	PAB(15-0)	A 16-bit bus that carries the program memory address.

3.2 Memory Organization

The TMS320C25 provides a total of 544 16-bit words of on-chip data RAM and 4K words of maskable program ROM. Of the 544 words of on-chip data RAM, 288 are always data memory and the remaining 256 words may be configured as either program or data memory. This section explains memory management using the on-chip program ROM and data RAM, external memory, memory maps, memory-mapped registers, auxiliary registers, data memory addressing, and memory-to-memory moves.

3.2.1 On-Chip Program ROM

The 4K words of on-chip program ROM allow program execution at full speed without the need for high-speed external program memory. The use of this memory also allows the external data bus to be freed for access of external data memory. In addition, there is the added benefit of increased security for the algorithms contained in on-chip memory, which may be proprietary.

Mapping of the first 4K-word block of program memory is user-selectable by means of the MP/$\overline{MC}$ (microprocessor/microcomputer) pin. Setting MP/$\overline{MC}$ high maps in the block of off-chip memory while holding the pin low maps in the block of on-chip ROM. The XF (external flag) pin can be used to toggle the MP/$\overline{MC}$ pin to dynamically enable or disable the on-chip ROM. The MP/$\overline{MC}$ pin is in the location of a VCC pin on the TMS32020. This allows substitution of a TMS320C25 for a TMS32020 since the TMS320C25 automatically operates in the microprocessor mode and therefore is directly compatible in the system. See Section 3.2.3 for the location of the on-chip program ROM in the memory map configurations.

3.2.2 On-Chip Data RAM Blocks

The 544 words of on-chip data RAM are divided into three separate blocks (B0, B1, and B2), as shown in Figure 3-2. Of the 544 words, 256 words (block B0) are configurable as either data or program memory by instructions provided for that purpose; 288 words (blocks B1 and B2) are always data memory. A data memory size of 544 words allows the TMS320C25 to handle a data array of 512 words (256 words if on-chip RAM is used for program memory), while still leaving 32 locations for intermediate storage. See Section 3.2.3 for memory map configurations.

When using block B0 as program memory, instructions can be downloaded from external program memory using the RPTK (repeat instruction as specified by immediate value) and BLKP (block move from program memory to data memory) instructions.

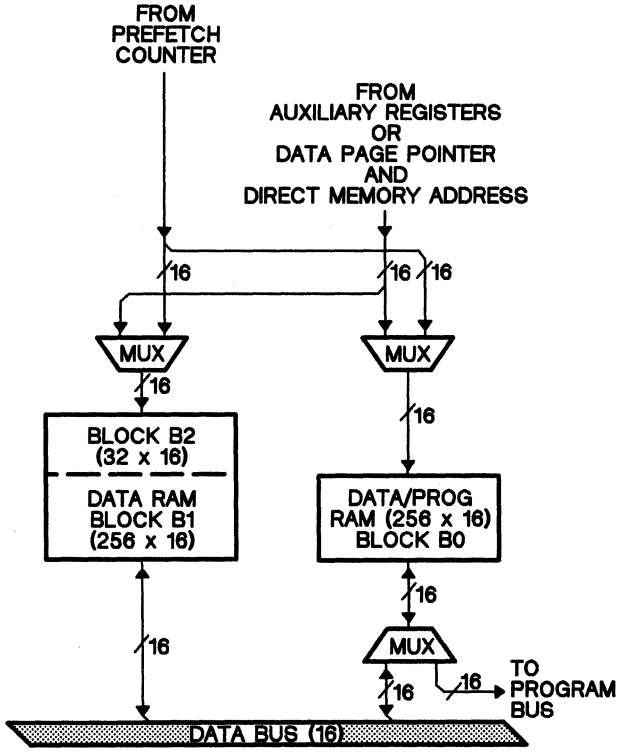


Figure 3-2. On-Chip Data Memory

When using on-chip program RAM, ROM, or high-speed external program memory, the TMS320C25 runs at full speed without wait states. However, the READY line can be used to interface the TMS320C25 to slower, less-expensive external memory. Downloading programs from slow off-chip memory to on-chip program RAM speeds processing while cutting system costs. See Section 3.5 for a description of instruction execution using various memory configurations.

3.2.3 Memory Maps

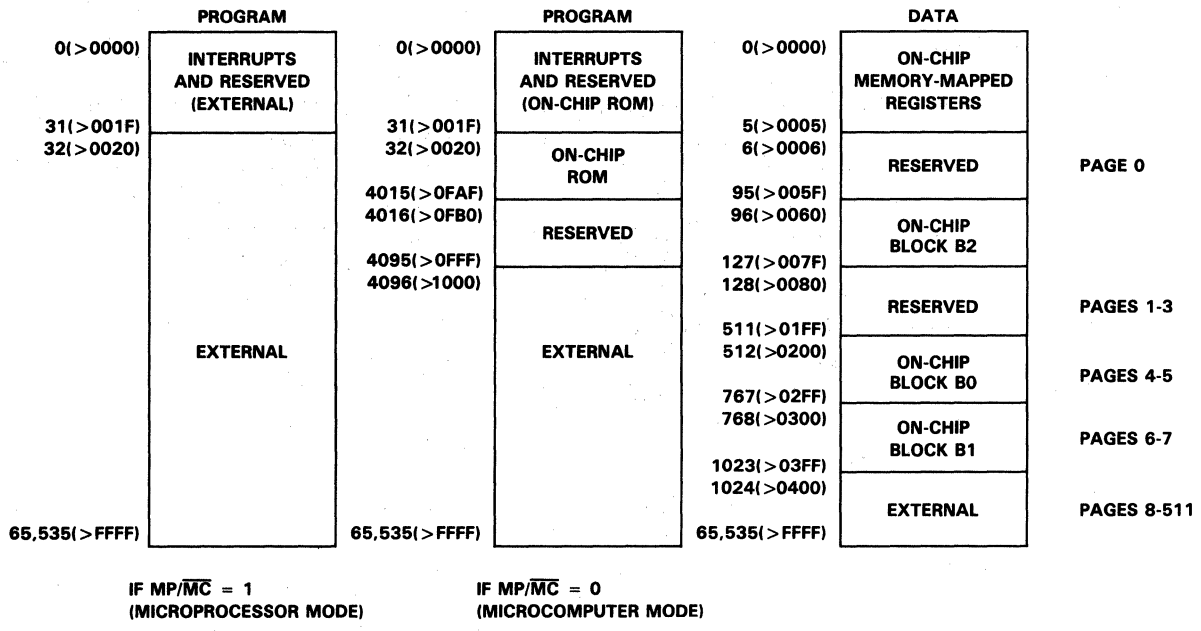
The TMS320C25 provides three separate address spaces for program memory, data memory, and I/O, as shown in Figure 3-3. These spaces are distinguished externally by means of the $\overline{PS}$, $\overline{DS}$, and $\overline{IS}$ (program, data, and I/O space select) signals. The on-chip memory blocks B0, B1, and B2 are comprised of a total of 544 words of RAM. Program/data RAM block B0 (256 words) resides in pages 4 and 5 of the data memory map when configured as data RAM and at addresses >FF00 to >FFFF when configured as program RAM. Block B1 (always data RAM) resides in pages 6 and 7, while block B2 resides in the upper 32 words of page 0. Note that the remainder of page 0 is composed of the memory-mapped registers and reserved locations, and pages 1-3 of the data memory map consist of reserved locations. Reserved locations may not be used for storage, and their contents are undefined when read. See Section 3.2.4 for further information on the memory-mapped registers.

The CNFD/CNFP instructions are used to configure block B0 as either data or program memory, respectively. The BLKP (block move from program memory to data memory) instruction may be used to download program information to block B0 when it is configured as data RAM, and then a CNFP (configure block as program memory) instruction may be used to convert it to program RAM (see the code example in Section 5.4.2).

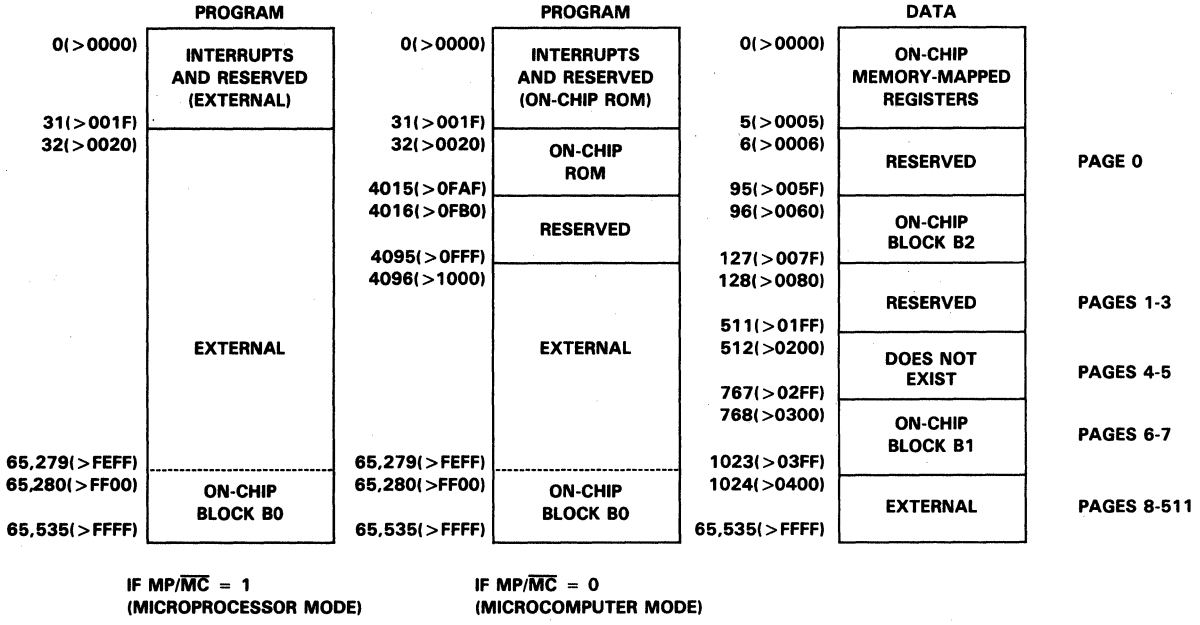
Reset configures block B0 as data RAM. Note that, due to internal pipelining, when the CNFD or CNFP instruction is used to remap RAM block B0, there is a delay before the new configuration becomes effective. This delay is one fetch cycle if execution is from internal program RAM and two fetch cycles if execution is from ROM or external program memory. This is particularly important if program execution is from the locations around 65280. Accordingly, a CNFP instruction must be placed at location 65277 in external memory if execution is to continue from the first location in block B0.

If a CNFP is placed at location 65278, and the instruction at location 65279 is a two-word instruction, the second word of the instruction will be fetched from the first location in block B0. If execution is from above location 65280 and block B0 is reconfigured, care must be taken to assure that execution resumes at the appropriate point in a new configuration. See Section 3.4.1 for a detailed description of pipeline operation.

On-chip program ROM is located in the lower 4K words of program memory when selected by setting MP/$\overline{MC}$ = 0. When MP/$\overline{MC}$ = 1, the lower 4K words of program memory are external.



(a) MEMORY MAPS AFTER A CNFD INSTRUCTION



(b) MEMORY MAPS AFTER A CNFP INSTRUCTION

Figure 3-3. Memory Maps

3.2.4 Memory-Mapped Registers

The six registers mapped into the data memory space are listed in Table 3-2 and are shown in the block diagram of Figure 3-1.

The memory-mapped registers may be accessed in the same manner as any other data memory location, with the exception that block moves using the BLKD (block move from data memory to data memory) instruction cannot be performed from the memory-mapped registers.

Table 3-2. Memory-Mapped Registers

REGISTER NAME	ADDRESS LOCATION	DEFINITION
DRR(15-0)	0	Serial port data receive register
DXR(15-0)	1	Serial port data transmit register
TIM(15-0)	2	Timer register
PRD(15-0)	3	Period register
IMR (5-0)	4	Interrupt mask register
GREG(7-0)	5	Global memory allocation register

3.2.5 Auxiliary Registers

The TMS320C25 provides a register file containing eight auxiliary registers (AR0-AR7). This section discusses each register's function and how an auxiliary register is selected and stored.

The auxiliary registers may be used for indirect addressing of data memory or for temporary data storage. Indirect auxiliary register addressing (see Figure 3-4) allows placement of the data memory address of an instruction operand into one of eight 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 either from data memory or by an immediate operand defined in the instruction. The contents of these registers may also be stored in data memory. (Section 4 describes the programming of the indirect addressing mode.)

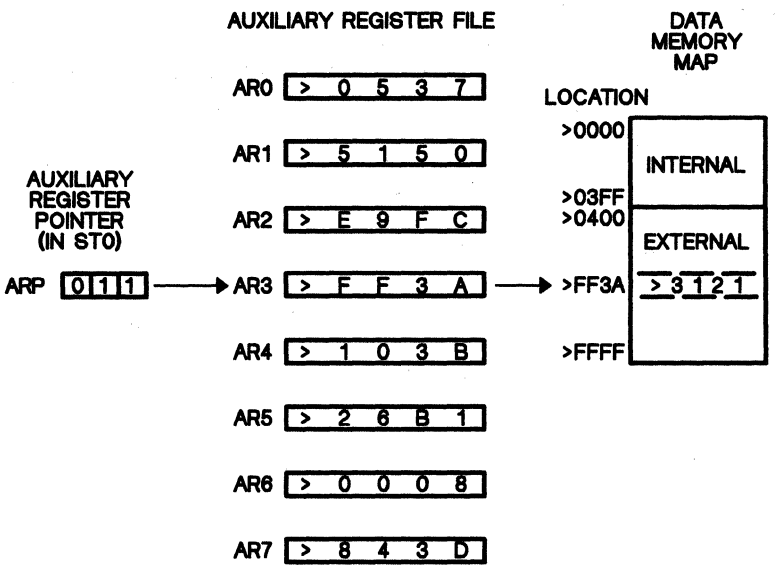


Figure 3-4. Indirect Auxiliary Register Addressing Example

The auxiliary register file (AR0-AR7) is connected to the Auxiliary Register Arithmetic Unit (ARAU), shown in Figure 3-5. The ARAU may autoindex the current auxiliary register while the data memory location is being addressed. Indexing by either ±1 or by the contents of AR0 may be performed. As a result, accessing tables of information does not require the Central Arithmetic Logic Unit (CALU) for address manipulation, thus freeing it for other operations.

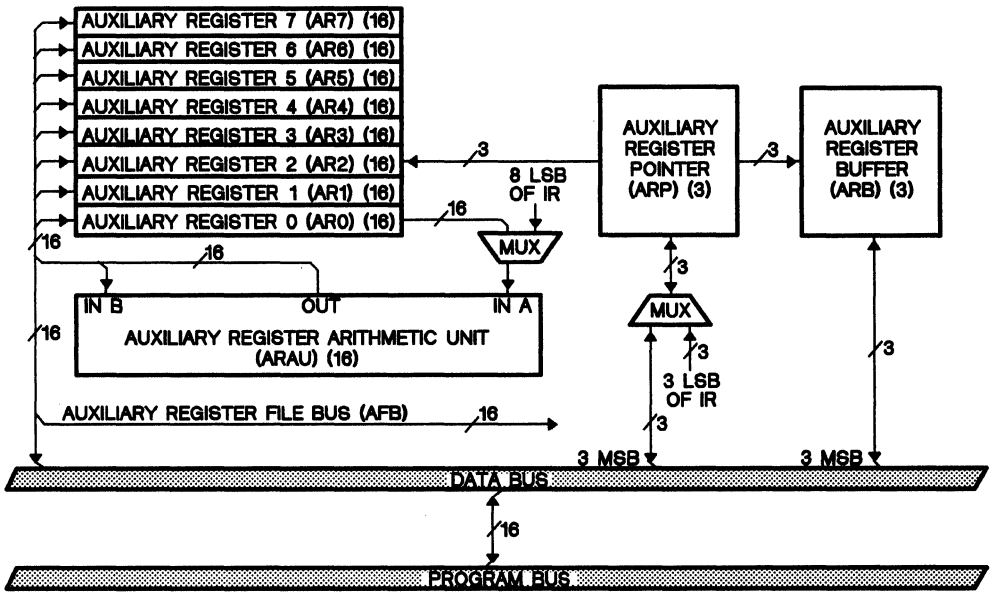


Figure 3-5. Auxiliary Register File

As shown in Figure 3-5, auxiliary register 0 (AR0) or the eight LSBs of the instruction registers 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)	+	AR0	→	AR(ARP)	Index the current AR by adding a 16-bit integer contained in AR0.
AR(ARP)	-	AR0	→	AR(ARP)	Index the current AR by subtracting a 16-bit integer contained in AR0.
AR(ARP)	+	1	→	AR(ARP)	Increment the current AR by one.
AR(ARP)	-	1	→	AR(ARP)	Decrement the current AR by one.
AR(ARP)			→	AR(ARP)	AR(ARP) is unchanged.
AR(ARP)	+	IR(7-0)	→	AR(ARP)	Add 8-bit immediate value to the current AR.
AR(ARP)	-	IR(7-0)	→	AR(ARP)	Subtract 8-bit immediate value to the current AR.
AR(ARP)	+	rcAR0	→	AR(ARP)	Bit-reversed indexing with reverse-carry propagation (see Section 4.1.2).
AR(ARP)	-	rcAR0	→	AR(ARP)	Bit-reversed indexing with reverse-carry propagation (see Section 4.1.2).

Although the ARAU is useful for address manipulation in parallel with other operations, it may also serve as an additional general-purpose arithmetic unit since the auxiliary register file can directly communicate with data memory. The ARAU implements 16-bit unsigned arithmetic, whereas the CALU implements 32-bit two's-complement arithmetic. Instructions provide branches dependent on the comparison of the auxiliary register pointed to by ARP with AR0.

Figure 3-5 also shows the three-bit Auxiliary Register pointer Buffer (ARB) that provides storage for the ARP on subroutine calls and interrupts.

3.2.6 Addressing Modes

The TMS320C25 can address a total of 64K words of program memory and 64K words of data memory. The on-chip data memory is mapped into the 64K data memory space. The memory maps, which change with the configuration of block B0, are described in detail in Section 3.2.4.

The 16-bit Data Address Bus (DAB) addresses data memory in one of the following two ways:

1) By the DiRect address Bus (DRB) using the direct addressing mode (e.g., ADD >10), or

2) By the Auxiliary register File Bus (AFB) using the indirect addressing mode (e.g., ADD *).

Operands are also addressed by the contents of the program counter in the immediate addressing mode.

Figure 3-6 illustrates operand addressing in the direct, indirect, and immediate addressing modes.

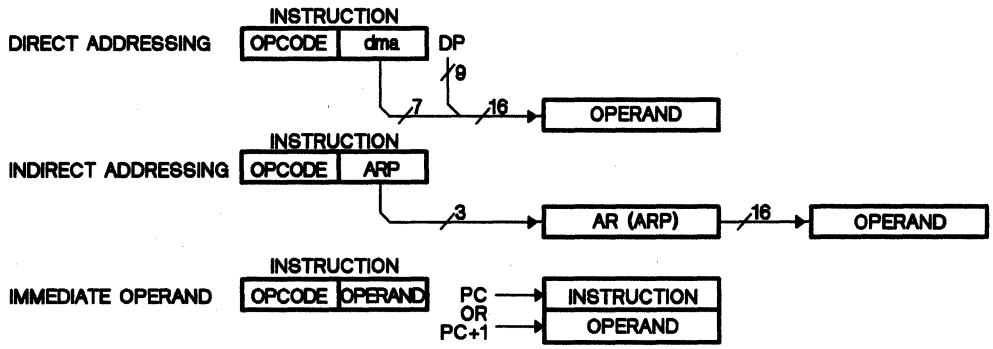


Figure 3-6. Methods of Instruction Operand Addressing

If the direct addressing mode is used, the 9-bit Data memory page Pointer (DP) points to one of 512 128-word pages. 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 direct address bus (DRB) is formed by concatenating the 9-bit DP with the 7-bit dma.

When using the indirect addressing mode, the currently selected 16-bit auxiliary register AR(ARP) addresses the data memory through the auxiliary register file bus (AFB). While the selected auxiliary register provides the data memory address and the data is being manipulated by the Central Arithmetic Logic Unit (CALU), the contents of the auxiliary register may be manipulated through the Auxiliary Register Arithmetic Unit (ARAU). See Figure 3-4 for an example of indirect auxiliary register addressing. The direct and indirect addressing modes are described in detail in Section 4.1.

When an immediate operand is used, it is either contained within the instruction word itself or, in the case of 16-bit immediate operands, the word following the instruction opcode.

3.2.7 Memory-to-Memory Moves

The TMS320C25 provides instructions for data and program block moves and for data move functions that efficiently utilize the configurable on-chip RAM.

The BLKD instruction moves a block within data memory, and the BLKP instruction moves a block from program memory to data memory. When used with the repeat instructions (RPT and RPTK), the BLKD and BLKP instructions efficiently perform block moves from on- or off-chip memory.

The DMOV (data move) function is useful for implementing algorithms that use the z^{-1} delay operation, such as convolutions and digital filtering where data is being passed through a time window. The data move function can be used anywhere within blocks B0, B1, or B2. It is continuous across the boundary of blocks B0 and B1 but cannot be used with off-chip data memory.

Implemented in on-chip RAM, the DMOV function is equivalent to that of the TMS32010 and TMS32020. DMOV allows a word to be copied from the currently addressed data memory location in on-chip RAM to the next higher location while the data from the addressed location is being operated upon in the same cycle (e.g., by the CALU). An ARAU operation may also be performed in the same cycle when using the indirect addressing mode. The MACD (multiply and accumulate with data move) and the LTD (load T register, accumulate previous product, and move data) instructions use the data move function.

3.3 Central Arithmetic Logic Unit (CALU)

The TMS320C25 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), and additional shifters at the outputs of both the accumulator and the multiplier. This section describes the CALU components and their functions. Figure 3-7 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 from the accumulator, and the other input may be fed from the Product Register (PR) of the multiplier or the scaling shifter that is loaded from data memory.

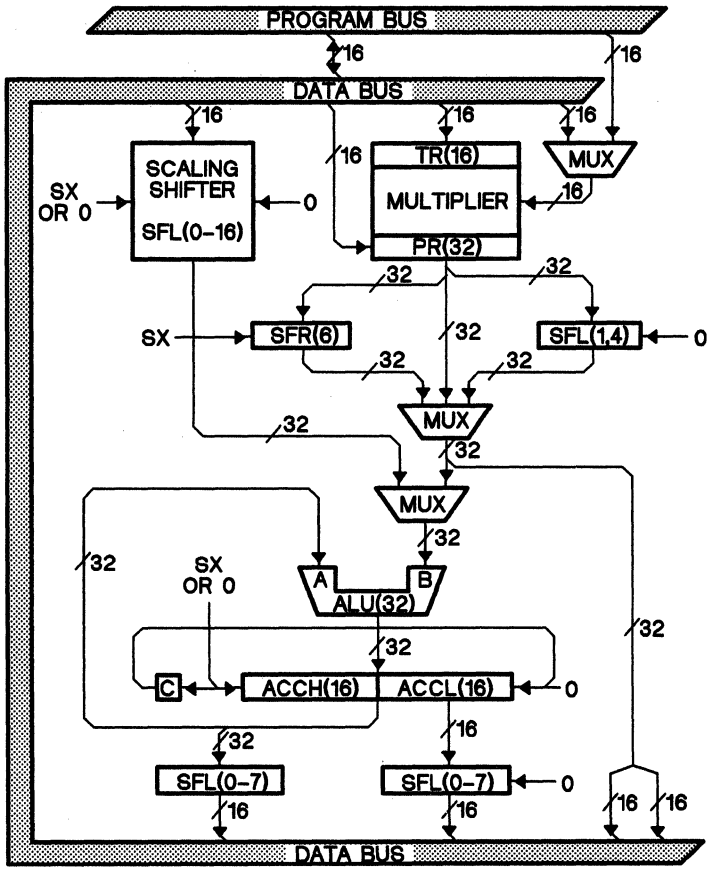


Figure 3-7. Central Arithmetic Logic Unit (CALU)

3.3.1 Scaling Shifter

The TMS320C25 scaling shifter has a 16-bit input connected to the data bus and a 32-bit output connected to the ALU (see Figure 3-7). The scaling shifter produces a left shift of 0 to 16 bits on the input data, as programmed in the instruction. The LSBs of the output are filled with zeros, and the MSBs may be either filled with zeros or sign-extended, depending upon the status programmed into the SXM (sign-extension mode) bit of status register ST0.

The TMS320C25 also contains several other shifters, which allow it to perform numerical scaling, bit extraction, extended arithmetic, and overflow prevention. These shifters are connected to the output of the multiplier and the accumulator.

3.3.2 ALU and Accumulator

The TMS320C25 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 scaling shifter.

The 32-bit accumulator (see Figure 3-7) is split into two 16-bit segments for storage in data memory: ACCH (accumulator high) and ACCL (accumulator low). 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 ACCH data is shifted left, the LSBs are transferred from the ACCL, and the MSBs are lost. When ACCL is shifted left, the LSBs are zero-filled, and the MSBs are lost.

The accumulator also has an associated carry bit that is set or reset depending on various operations within the TMS320C25. The carry bit allows more efficient computation of extended-precision products and additions or subtractions. It is affected by most arithmetic instructions as well as the shift and rotate instructions. It is not affected by loading the accumulator, logical operations, or other such nonarithmetic or control instructions. It is also not affected by the multiply (MPY, MPYK, and MPYU) instructions, but is affected by the accumulation process in the MAC and MACD instructions. Examples of carry bit operation are shown in Figure 3-8.

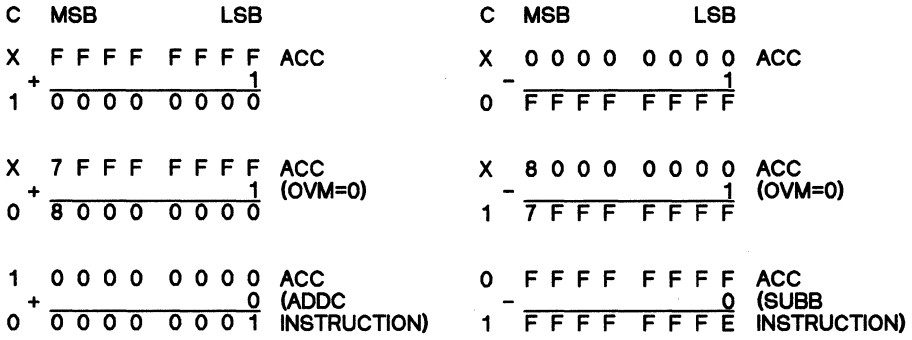


Figure 3-8. Examples of Carry Bit Operation

The value added to or subtracted from the accumulator, shown in the examples of Figure 3-8, may come from either the input scaling shifter or the shifter at the output of the P register. The carry bit is set if the result of an addition or accumulation process generates a carry, or reset to zero if the result of a subtraction generates a borrow. Otherwise, it is reset 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 (see these instructions in Section 4 for more detailed information).

The one exception to operation of the carry bit, as shown in Figure 3-8, is in the use of the ADDH (add to high accumulator) and SUBH (subtract from high accumulator) instructions. The ADDH instruction can only set the carry bit if a carry is generated, and the SUBH instruction can only reset the carry bit if a borrow is generated; otherwise, neither instruction can affect it.

Two branch instructions, BC and BNC, have been provided for branching on the status of the carry bit. The SC, RC, and LST1 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 or right) and ROL and ROR (rotate to the left or right) instructions implement shifting or rotating of the contents of the accumulator through the carry bit. The SXM bit affects the definition of the SFR (shift accumulator right) instruction. When SXM = 1, SFR performs an arithmetic right shift, maintaining the sign of the accumulator data. When SXM = 0, SFR performs a logical shift, shifting out the LSB and shifting in a zero for the MSB. The SFL (shift accumulator left) instruction is not affected by the SXM bit and behaves the same in both cases, shifting out the MSB and shifting in a zero. Repeat (RPT or RPTK) instructions may be used with the shift and rotate instructions for multiple shift counts.

The TMS320C25 supports floating-point operations for applications requiring a large dynamic range. The NORM (normalization) instruction is used to normalize fixed-point numbers contained in the accumulator by performing left shifts. The LACT (load accumulator with shift specified by the T register) instruction denormalizes a floating-point number by arithmetically left-shifting the mantissa through the input scaling shifter. The shift count, in this case, is the value of the exponent specified by the four low-order bits of the T register (TR). ADDT and SUBT (add to/subtract from accumulator with shift specified by T register) instructions have also been provided to allow additional arithmetic operations. Floating-point numbers with 16-bit mantissas and 4-bit exponents can thus be manipulated.

The accumulator overflow saturation mode may be programmed through the SOVM and ROVM (set/reset overflow mode) instructions. When the accumulator is in the overflow saturation mode and an overflow occurs, the overflow flag is set and the accumulator is loaded with the most positive or negative number depending upon the direction of overflow.

The TMS320C25 also has the capacity of executing branch instructions that depend on the status of the ALU and accumulator. These instructions include the BC (branch on carry), BV (branch on overflow), and BZ (branch on accumulator equal to zero) instructions. (Refer to Section 4 for a complete list of TMS320C25 instructions.) In addition, the BACC (branch to address in accumulator) instruction provides the ability to branch to an address specified by the accumulator.

3.3.3 Multiplier, T and P Registers

The TMS320C25 utilizes a 16 x 16-bit hardware multiplier, which is capable of computing a 32-bit product in a single machine cycle. As shown in Figure 3-7, the following two registers are associated with the multiplier:

- A 16-bit Temporary Register (TR) that holds one of the operands for the multiplier, and
- A 32-bit Product Register (PR) that holds the product.

Normally, an LT (load T register) instruction loads the TR to provide one operand (from the data bus), and the MPY (multiply) instruction provides the second operand (also from the data bus). Alternatively, a multiplication can be performed with an immediate operand using the MPYK instruction. In either case, a product can be obtained every two cycles.

Two multiply/accumulate instructions (MAC and MACD) fully utilize the computational bandwidth of the multiplier, allowing both operands to be processed simultaneously. For MAC and MACD, two operands are transferred to the multiplier each

cycle via the program and data buses. This provides for single-cycle multiply/accumulates when used with repeat (RPT or RPTK) instructions. The MAC and MACD instructions can be used with operands in either internal or external memory. The SQRA (square/add) and SQRS (square/subtract) instructions pass the same value to both inputs of the multiplier for squaring a data memory value.

All multiply instructions, except the MPYU (multiply unsigned) instruction, perform a signed multiply operation in the multiplier. That is, the two numbers being multiplied are treated as two's-complement numbers, and the result is a 32-bit two's-complement number. The MPYU instruction performs an unsigned multiplication, which greatly facilitates multiple-precision arithmetic operations. This allows operands of greater than 16 bits to be broken down into 16-bit words and processed separately to generate products of greater than 32 bits.

After the multiplication of two 16-bit numbers, the 32-bit product is loaded into the 32-bit Product Register (PR). The product from the PR may be transferred to the ALU directly or optionally shifted before it is transferred to the ALU.

Four product shift modes (PM) are available. The PM field of status register ST1 specifies the PM shift mode, as shown in Table 3-3.

Table 3-3. PM Shift Modes

IF PM IS:	RESULT
00	No shift
01	Left shift of 1 bit
10	Left shift of 4 bits
11	Right shift of 6 bits

Left shifts specified by the PM value are useful for implementing fractional arithmetic or justifying fractional products. For example, the product of either two normalized, 16-bit, two's-complement numbers or two Q15 numbers contains two sign bits, one of which is redundant (see Section 5.6.6 for an explanation of Q15 representation). The single-bit left shift allows this extra sign bit to be shifted off of the product when it is passed to the accumulator. This results in the accumulator contents being formatted in the same manner as the multiplicands. Similarly, the product of either a normalized, 16-bit, two's-complement or Q15 number and a 13-bit, two's-complement constant contains five sign bits, four of which are redundant. This is the case, for example, when using the MPYK instruction. Here the four-bit shift properly aligns the result as it is transferred to the accumulator.

Use of the right-shift PM value allows the execution of up to 128 consecutive multiply/accumulate operations without the threat of an arithmetic overflow, thereby avoiding the overhead of overflow management. The shifter can be disabled to cause no shift in the product when working with integer or 32-bit precision operations. This allows compatibility with TMS32010 code to be maintained. Note that the PM right shift is always sign-extended regardless of the state of SXM.

The four least significant bits of the T register (TR) also define a variable shift through the scaling shifter for the LACT/ADDT/SUBT (load/add-to/subtract-from accumulator with shift specified by the TR) instructions. These instructions are useful in floating-point arithmetic where a number needs to be denormalized, i.e., floating-point to fixed-point conversion. The BITT (bit test) instruction allows testing of a single bit of a word in data memory based on the value contained in the four LSBs of the TR.

3.4 System Control

System control on the TMS320C25 is provided by the program counter and related hardware, the external reset signal, the status registers, the on-chip timer, and the repeat counter. The following sections describe the function of each of these components in system control.

3.4.1 Program Counter and Related Hardware

The description of the TMS320C25 Program Counter (PC) and its related hardware contained in this section provides information useful in understanding the sequence of external bus operations that occurs during instruction execution. It should be noted, however, that in virtually all cases, operation of this hardware and its effects on operation of the internal pipeline are transparent to the user and are included in the instruction cycle timings provided in Appendix E.

The PC and related hardware (see Figure 3-9) direct program execution on the TMS320C25. Included in the related hardware are the eight-level PC stack, the Prefetch Counter (PFC), the 16-bit MicroCall Stack (MCS) register, the Instruction Register (IR), and the Queue Instruction Register (QIR). The operation of these components and their function in instruction pipelining is now described in detail.

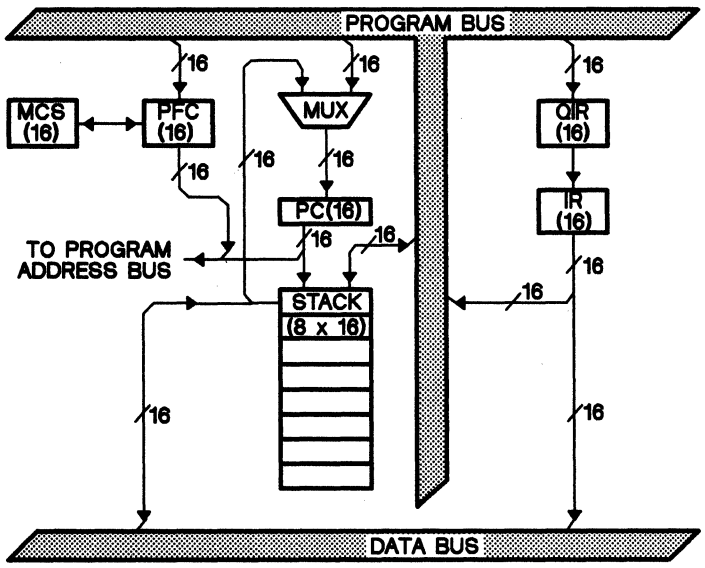


Figure 3-9. Program Counter and Related Hardware

In order to speed instruction execution, the TMS320C25 utilizes a three-level internal pipeline, which divides an instruction cycle into three operations: prefetch, decode, and execution. The PFC contains the address of the next instruction to be prefetched. Once an instruction is prefetched, the instruction is loaded into the IR, unless the IR still contains an instruction currently executing, in which case the prefetched instruction is stored in the QIR. The PFC is then incremented, and after the current instruction has completed execution, the instruction in the QIR is loaded into the IR to be executed.

The PC contains the address of the next instruction to be executed, and is not used directly in instruction fetch operations, but merely serves as a reference pointer to the current position within the program. The PC is incremented as each instruction is executed. When interrupts or subroutine call instructions occur, the contents of the PC are pushed onto the stack to preserve return linkage to the previous program context.

In the operation of the pipeline, the prefetch, decode, and execute operations are independent, which allows instruction executions to overlap. Thus, during any given cycle, three different instructions can be active, each at a different stage of completion, resulting in the three-instruction pipeline. Figure 3-10 shows the operation of the three-level pipeline for single-word, single-cycle instructions executing from either internal program ROM or external memory with no wait states.

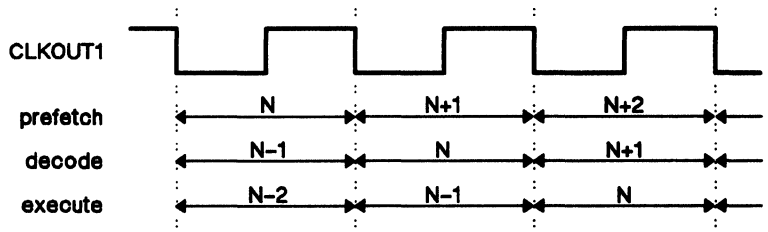


Figure 3-10. Three-Level Pipeline Operation

Pipelining is reduced to two levels when execution is from internal program RAM due to the fact that an instruction in internal RAM can be fetched and decoded in the same cycle. Thus, separate prefetch and decode operations are not required, as shown in Figure 3-11.

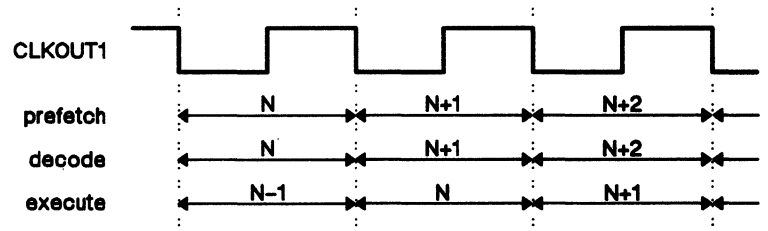


Figure 3-11. Two-Level Pipeline Operation

Note that the difference in pipeline levels does not necessarily affect instruction execution speed, but merely changes the fetch/decode sequence. Most instructions execute in the same number of cycles regardless of whether they are executed from internal RAM, ROM, or external program memory. Also note that the effects of pipelining are included in the instruction cycle timings listed in Appendix E.

When branches, subroutine calls, or interrupts occur, the pipeline flow shown in Figure 3-12 and Figure 3-13 is disrupted because the pipeline prefetches sequentially, and cannot in general detect that a transfer of control will occur until an instruction reaches execution. This is especially true with conditional branches.

During branch or call instructions, both the PC and PFC are loaded with the destination address, and the pipeline must be refilled. This causes these instructions to require at least three cycles to execute when the destination address is located externally or in internal program ROM. When the destination address is located in internal program RAM, branch instructions generally execute in two cycles, due to the two-level pipeline for internal RAM. In either case, some instructions that have been prefetched may be discarded as control passes to the branch destination address. Operation of the pipeline during interrupts is similar and is described in Section 3.6.2.

Operation of the pipeline during execution of a conditional branch instruction such as a BANZ (branch on auxiliary register not zero), located in external program memory with no wait states, is shown in Figure 3-12. The diagram shows the sequence that occurs when the branch is taken, and the destination address is also in external memory. When the branch is not taken, instruction execution continues sequentially, and the branch instruction requires only two cycles to execute. Operation of unconditional branches is identical to that of conditional branches with the exception that in conditional branches, the N+2 instruction is not fetched, although the address bus is still driven with N+2.

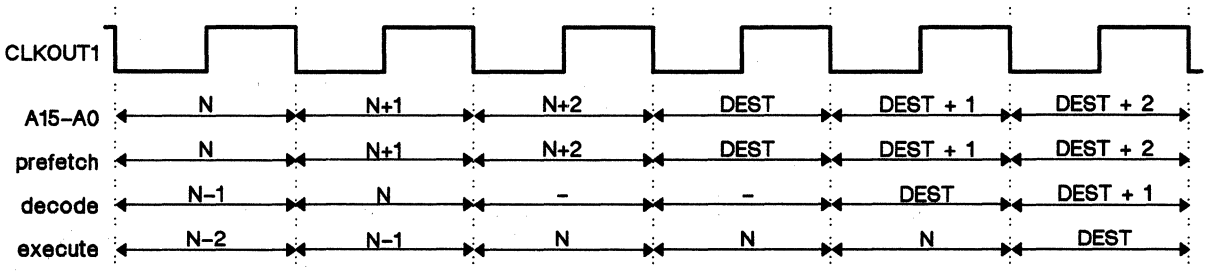


Figure 3-12. Pipeline Operation During BANZ Instruction

There is one additional condition under which the pipeline becomes disrupted. This is when execution of single-cycle instructions changes from internal RAM to external program memory or internal ROM. This occurs in only the following two cases:

1) When execution of single-cycle instructions wraps around from >FFFF in block B0 (when configured as program memory) to location >0000.

2) When execution of single-cycle instructions is from block B0 (configured as program memory) and a CNFD instruction is executed, converting block B0 to data memory.

Under these conditions, one dummy execute cycle occurs as the pipeline is refilled. This situation is depicted in Figure 3-13. Note that this condition occurs only under the above circumstances, and its effects are not included in the instruction cycle timings given in Appendix E.

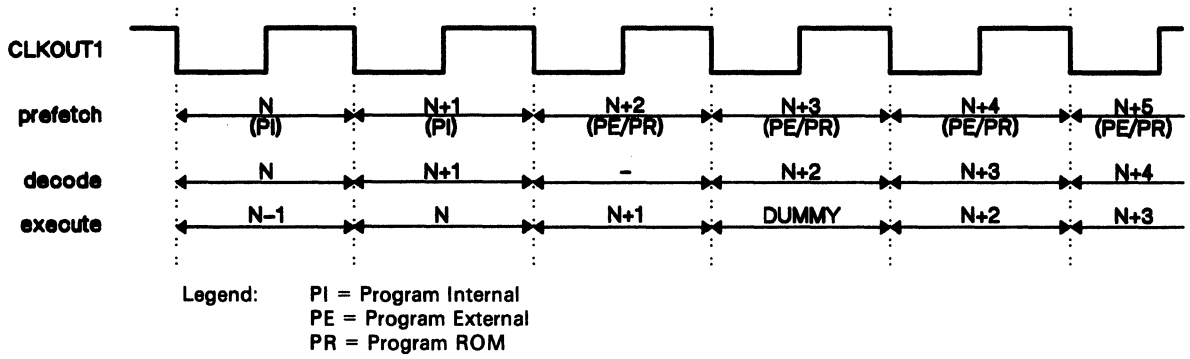


Figure 3-13. Pipeline Operation When Crossing Program Boundaries

The contents of the accumulator may be loaded into the PC and PFC in order to implement "computed go to" operations. This can be accomplished using the BACC (branch to address in accumulator) or CALA (call subroutine indirect) instructions.

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 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 eighth stack level. All eight 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.

The 16-bit MicroCall Stack (MCS) register is used expressly for temporary storage of the PFC contents during execution of the TBLR/TBLW, MAC/MACD, and BLKD/BLKP instructions. In these instructions, two operand addresses are required: one provided through either direct or indirect addressing, and the other loaded into the PFC. When execution of the instruction is completed, the contents of the MCS are transferred back to the PFC.

3.4.2 Reset

Reset ($\overline{RS}$) is a non-maskable external interrupt that can be used at any time to put the TMS320C25 into a known state. Reset is typically applied after powerup when the machine is in a random state.

Driving the $\overline{RS}$ signal low causes the TMS320C25 to terminate execution and forces the program counter to zero. $\overline{RS}$ affects various registers and status bits. At powerup, the state of the processor is undefined. For correct system operation after powerup, a reset signal must be asserted low for at least three clock cycles to guarantee a reset of the device. Processor execution begins at location 0, which normally contains a B (branch) statement to direct program execution to the system initialization routine (see Section 5.1 for an initialization routine example).

Upon receiving an $\overline{RS}$ signal, the following actions take place:

1) A logic 0 is loaded into the CNF (configuration control) bit in status register ST1, causing all RAM to be configured as data memory.

2) The Program Counter (PC) is set to 0, and the address bus A15-A0 is driven with all zeroes while $\overline{RS}$ is low.

3) The data bus D15-D0 is placed in the high-impedance state.

4) All memory and I/O space control signals ($\overline{PS}$, $\overline{DS}$, $\overline{IS}$, R/$\overline{W}$, $\overline{STRB}$, and $\overline{BR}$) are de-asserted by setting them to high levels while $\overline{RS}$ is low.

5) All interrupts are disabled by setting the INTM (interrupt mode) bit to a high level. (Note that $\overline{RS}$ is non-maskable). The interrupt flag register (IFR) is reset to all zeroes.

6) Status bits:
   0 → OV; 1 → XF; 1 → SXM; 0 → PM; 1 → HM; 0 → FO;1 → C
   1 → FSM (Remaining status bits are unchanged).

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

8) The RPTC (repeat counter) is cleared.

9) The DX (data transmit) pin is placed in the high-impedance state. Any transmit/receive operations on the serial port are terminated, and the TXM (transmit mode) bit is reset to a low level. This configures the FSX framing pulse to be an input. A transmit/receive operation may be started by framing pulses only after the removal of $\overline{RS}$.

10) The timer (TIM) and period (PRD) registers are both set to >FFFF and TIM does not begin decrementing until $\overline{RS}$ is de-asserted.

11) The $\overline{IACK}$ (interrupt acknowledge) signal is generated in the same manner as a maskable interrupt.

Execution starts from location 0 of program memory when the $\overline{RS}$ signal is taken high. Note that if $\overline{RS}$ is asserted while in the hold mode, normal reset operation occurs internally, but all buses and control lines remain in the high-impedance state. Upon release of $\overline{HOLD}$ and $\overline{RS}$, execution starts from location zero.

Note that the ARB, ARP, DP, IMR, OVM, and TC bits are not initialized by reset.

3.4.3 Status Registers

Two status registers, ST0 and ST1, contain the status of various conditions and modes. The SST and SST1 instructions provide for storing the status registers into data memory. The LST and LST1 instructions load the status registers from data memory. In this manner, the current status of the device may be saved on interrupts and subroutine calls.

Figure 3-14 shows the organization of both status registers, indicating all status bits contained in each. Note that the DP, ARP, and ARB registers are shown as separate registers in the processor block diagram of Figure 3-1. Because these registers do not have separate instructions for storing them into RAM, they are included in the status registers.

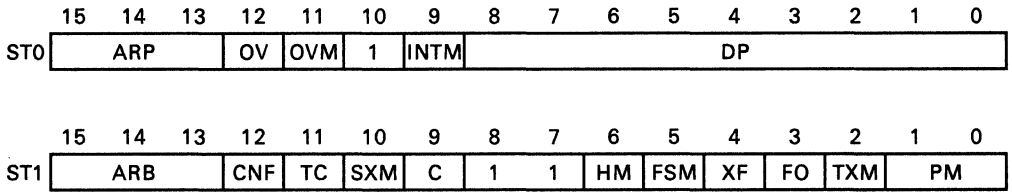


Figure 3-14. Status Register Organization

The capability of storing the status registers into data memory and loading them from data memory allows the status of the machine to be saved and restored for interrupts and subroutines. All status bits are written to and read from using LST/LST1 and SST/SST1 instructions, respectively (with the exception of INTM, which cannot be loaded via an LST instruction). However, some additional instructions or functions may affect those bits, as indicated in Table 3-4.

As shown in Figure 3-14, several bits in the status registers are reserved and are read as logic '1's by the LST and LST1 instructions.

Table 3-4. Status Register Field Definitions

FIELD	FUNCTION
ARB	Auxiliary Register Pointer Buffer. Whenever the ARP is loaded, the old ARP value is copied to the ARB except during an LST instruction. When the ARB is loaded via an LST1 instruction, the same value is also copied to the ARP.
ARP	Auxiliary Register Pointer. This three-bit field selects the AR to be used in indirect addressing. When ARP is loaded, the old ARP value is copied to the ARB register. ARP may be modified by memory-reference instructions when using indirect addressing, and by the LARP, MAR, and LST instructions. ARP is also loaded with the same value as ARB when an LST1 instruction is executed.
C	Carry bit. This bit is set to '1' if the result of an addition generates a carry, or reset to '0' if the result of a subtraction generates a borrow. Otherwise, it is reset after an addition or set after a subtraction, except if the instruction is an ADDH or a SUBH. ADDH can only set and SUBH only reset the carry bit, but cannot affect it otherwise. The shift and rotate instructions also affect this bit, as well as the SC, RC, and LST1 instructions. Two branch instructions, BC and BNC, have been provided to branch on the status of C. C is set to '1' on a reset. The carry bit is useful in implementing multiple-precision arithmetic and in overflow management.
CNF	On-Chip RAM Configuration Control bit. If set to '0', block B0 is configured as data memory; otherwise, block B0 is configured as program memory. The CNF may be modified by the CNFD, CNFP, and LST1 instructions. $\overline{RS}$ resets 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, LDP, and LDPK instructions.
FO	Format bit. A '0' configures the serial port registers as 16-bit registers. A '1' configures the port registers to receive and transmit eight-bit bytes. FO may be modified by the FORT and LST1 instructions. FO is reset to '0'.

Table 3-4. Status Register Field Definitions (Concluded)

FIELD	FUNCTION
FSM	Frame Synchronization Mode bit. This bit indicates whether the serial port will operate with or without frame sync pulses. When FSM = 1, the serial port operation will be initiated following a frame sync pulse on the FSX/FSR inputs. When FSM = 0, the FSX/FSR inputs are ignored and the serial port operates continuously with no frame sync pulses required. The bit is set to one by a reset.
HM	Hold Mode bit. When HM = 1, the TMS320C25 halts internal execution when acknowledging an active $\overline{\text{HOLD}}$. 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 one by a reset.
INTM	Interrupt Mode. A '0' enables all unmasked interrupts. A '1' disables all maskable interrupts. INTM is set and reset by the DINT and EINT instructions. $\overline{\text{RS}}$ and $\overline{\text{IACK}}$ also set INTM. INTM has no effect on the unmaskable $\overline{\text{RS}}$ interrupt. Note that INTM is unaffected by the LST instruction.
OV	Overflow Flag. 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, BV, BNV, or LST instruction clears the OV.
OVM	Overflow Mode. A '0' causes overflowed results to overflow normally in the accumulator. A '1' causes the accumulator to be set to either its most positive or negative value upon encountering an overflow. The SOVM and ROVM instructions set and reset this bit. LST may also be used to modify the OVM.
PM	Product Shift Mode. If these two bits are 00, the multiplier's 32-bit product is loaded into the ALU with no shift. If PM = 01, the PR output is left-shifted one place and loaded into the ALU, with the LSBs zero-filled. If PM = 10, the PR output is left-shifted by four bits and loaded into the ALU, with the LSBs zero-filled. PM = 11 produces a right shift of six bits, sign-extended. Note that the PR contents remain unchanged. The shift takes place when transferring the contents of the PR to the ALU. PM is loaded with the SPM and LST1 instructions. The PM bits are cleared by $\overline{\text{RS}}$.
SXM	Sign-Extension Mode bit. A '1' produces sign extension on data as it is passed into the accumulator through the scaling shifter. A '0' suppresses sign extension. Note that SXM does not affect the definition of certain instructions. For example, the ADDS instruction suppresses sign extension regardless of SXM. This bit is set and reset by the SSXM and RSXM instructions and may also be loaded by LST1. SXM is set to '1' by $\overline{\text{RS}}$.
TC	Test/Control Flag bit. The TC bit is affected by the BIT, BITT, CMPR, LST1, and NORM instructions. The TC bit is set to a '1' if: (1) a bit tested by BIT or BITT is a '1', (2) a compare condition tested by CMPR exists between AR0 and another AR pointed to by ARP, or (3) the exclusive-OR function of the two MSBs of the accumulator is true when tested by a NORM instruction. Two branch instructions, BBZ and BBNZ, provide branching on the status of the TC.
TXM	Transmit Mode bit. A '1' configures the serial port's FSX pin to be an output. In this mode, a pulse is produced on FSX when DXR is loaded. Transmission then starts on the DX pin. A '0' configures the FSX pin to be an input. TXM is set and reset by the STXM and RTXM instructions and may also be loaded by LST1. $\overline{\text{RS}}$ resets TXM to a '0'.
XF	XF pin status. A status bit indicating the state of the XF pin, a general-purpose output pin. XF is set and reset by the SXF and RXF instructions or may be loaded by LST1. XF is set to '1' by $\overline{\text{RS}}$.

3.4.4 Timer Operation

The TMS320C25 provides a memory-mapped timer (TIM) register and a period (PRD) register. The timer register is a down counter continuously clocked by CLKOUT1. Reset sets the timer and period registers (see Figure 3-15) to their maximum value (>FFFF). Upon release of reset, the timer begins decrementing. Following this, the TIM and PRD registers may be reloaded under program control. See Section 3.4.2 for reset information.

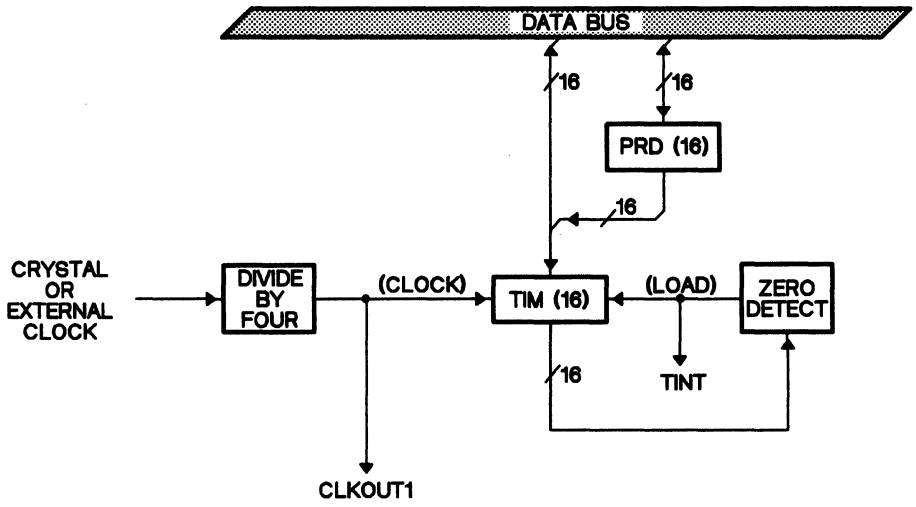


Figure 3-15. Timer Block Diagram

The TIM register, data memory location 2, holds the current count of the timer. At every CLKOUT1 cycle, the TIM register is decremented by one. The PRD register, data memory location 3, holds the starting count for the timer. When the TIM register decrements to zero, a TINT (timer interrupt) is generated. In the next cycle, the contents of the PRD register are loaded into the TIM register. In this way, a TINT is generated every PRD + 1 cycles of CLKOUT1. By programming the PRD register from 1 to 65,535 (>FFFF), a TINT can be generated every 2 to 65,536 cycles, respectively. A PRD register value of zero is not allowed.

The timer and period registers can be read from or written to on any cycle. The count can be monitored by reading the TIM register. A new counter period can be written to the period register without disturbing the current timer count. The timer will then start the new period after the current count is complete. If both the PRD and TIM registers are loaded with a new period, the timer begins decrementing the new period without generating an interrupt. Thus, the programmer has complete control of the current and next periods of the timer.

If the timer is not used, TINT should be masked or all maskable interrupts disabled by a DINT instruction. The PRD register can then be used as a general-purpose data memory location. If TINT is used, the PRD and TIM registers should be programmed before unmasking the TINT.

3.4.5 Repeat Counter

The repeat counter (RPTC) is an 8-bit counter, which when loaded with a number N, causes the next single instruction to be executed N + 1 times. The RPTC can be loaded with a number from 0 to 255 using either the RPT (repeat) or RPTK (repeat immediate) instructions. This results in a maximum of 256 executions of a given instruction. RPTC is cleared by reset.

The repeat feature can be used with instructions such as multiply/accumulates (MAC/MACD), block moves (BLKD/BLKP), I/O transfers (IN/OUT), and table read/writes (TBLR/TBLW). These instructions, which are normally multicycle, are pipelined when using the repeat feature, and effectively become single-cycle instructions. For example, the table read instruction may take three or more cycles to execute, but when repeated, a table location can be read every cycle. Note that not all instructions can be repeated (see Section 4.3 and Appendix E for more information).

3.4.6 Powerdown Mode

When operated in the powerdown mode, the TMS320C25 enters a dormant state and requires only a fraction of the power normally needed to supply the device. Powerdown mode is invoked either by executing an IDLE instruction or by driving the $\overline{\text{HOLD}}$ input low with the HM status bit set to one.

While in powerdown mode, all of the internal contents of the processor are maintained to allow operation to continue unaltered when powerdown mode is terminated. Powerdown mode is terminated upon receipt of an interrupt when an IDLE instruction is being executed or by removal of the $\overline{\text{HOLD}}$ input. (Refer to the IDLE instruction description in Section 4 and the hold function description in Section 3.8.3 for further information.) Actual power supply current requirements in powerdown mode are specified in the TMS320C25 Data Sheet (Appendix A).

3.5 External Memory and I/O Interface

Data, program, and I/O address spaces provide interfacing to memory and I/O, thus maximizing system throughput. The local memory interface consists of:

- A 16-bit parallel data bus (D15-D0),
- A 16-bit address bus (A15-A0),
- Data, program, and I/O space select ($\overline{\text{DS}}$, $\overline{\text{PS}}$, and $\overline{\text{IS}}$) signals, and
- Various system control signals.

The R/$\overline{\text{W}}$ (read/write) signal controls the direction of the transfer, and $\overline{\text{STRB}}$ (strobe) provides a timing signal to control the transfer.

I/O design is simplified by having I/O treated the same way as memory. I/O devices are mapped into the I/O address space using the processor's external address and data buses in the same manner as memory-mapped devices.

Interfacing to memory and I/O devices of varying speeds is accomplished by using the READY line. When communicating with slower devices, the TMS320C25 processor waits until the other device completes its function, signals the processor via the READY line, and continues execution.

3.5.1 Memory Combinations

The exact sequence of operations performed as instructions execute depends on the areas in memory where the instructions and operands are located. There are six possible combinations of program and data memory since information can be located in either internal RAM, external memory, or internal ROM. The six possible combinations are:

- Program Internal RAM/Data Internal (PI/DI)
- Program Internal RAM/Data External (PI/DE)
- Program External/Data Internal (PE/DI)
- Program External/Data External (PE/DE)
- Program Internal ROM/Data Internal (PR/DI)
- Program Internal ROM/Data External (PR/DE)

Appendix E provides cycle timings for instructions both when repeated and when not repeated. The following is a summary of program execution, organized according to memory configuration.

PI/DI or PR/DI — When both program and data memory are on-chip, the processor runs at full speed with no wait states. Note that IN and OUT instructions have different cycle timings when program memory is internal; IN requires two cycles to execute while OUT requires only one.

PE/DI — This memory mode can run at full speed if external program memory is sufficiently fast since internal data operations can occur coincident with external program memory accesses. If external program memory is not fast enough, wait states may be generated using the READY input.

PI/DE, PE/DE, or PR/DE — Additional cycles are required to execute instructions that reference an external data memory space. At least two cycles are required to execute 'read from external data memory' instructions such as ADD, LAR, etc. Further additional cycles may be required due to wait states if external data memory is not fast enough to be accessed within a single cycle. Note, however, that the TMS320C25 has the capability of executing 'write to external data memory' instructions in a single cycle when program memory is internal (two cycles are required if program memory is also external). Additional cycles are also required in this case if external data memory is not sufficiently fast.

Note that in all memory configurations where the same bus is used to communicate with external data, program, or I/O space, the number of cycles required to execute a particular instruction may further vary depending on whether the next instruction fetch is from internal or external program memory. Instruction execution and operation of the pipeline are discussed in detail in the following sections and in Section 3.4.1.

3.5.2 Internal Clock Timing Relationships

The crystal or external clock source frequency is divided to produce an internal four-phase clock. The four phases are defined by CLKOUT1 and CLKOUT2, as shown in Figure 3-16.

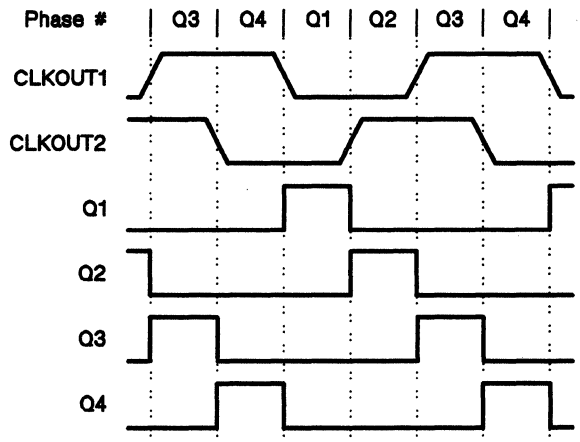


Figure 3-16. Four-Phase Clock

Figure 3-16 shows the start of quarter-phase 3 (Q3) on the rising edge of CLKOUT1. To help facilitate the description of the TMS320C25's operation, this nomenclature is used throughout this document.

3.5.3 External Read Cycle

Each time an external read cycle is performed, a specific sequence of events occurs. This sequence of events is as follows:

1) During clock quarter-phase 1, the processor begins driving the address bus and one of the memory space select signals. R/$\overline{W}$ is driven high to indicate an external memory read.

2) At the beginning of quarter-phase 2, $\overline{STRB}$ is asserted to indicate that the address bus is valid. $\overline{STRB}$, in conjunction with R/$\overline{W}$, may be used to gate a read enable signal.

3) After decoding the addressed memory area, the user's memory interface must set up the appropriate READY signal during quarter-phase 2. READY is sampled by the processor at the beginning of quarter-phase 3.

4) If READY was high at the proper time, the data is clocked in at the end of quarter-phase 3.

5) $\overline{STRB}$ is deasserted at the beginning of quarter-phase 4. The processor ends the memory access by deactivating the address bus and $\overline{PS}$, $\overline{DS}$, or $\overline{IS}$.

Note that the control signals $\overline{PS}$, $\overline{DS}$, $\overline{IS}$, $\overline{STRB}$, and R/$\overline{W}$ are only asserted when an external address location is being accessed.

Figure 3-17 shows the timing for several read operations. Two instructions are shown executing completely, an ADD and a SUB instruction. Note that a previous instruction is being executed while ADD and SUB are being fetched. Also note that while the SUB instruction is being executed, the next instruction, LAC, is being fetched even though execution of the LAC is not shown.

The ADD instruction takes one cycle to execute because both the next instruction and the ADD's data are internal. The SUB instruction that is fetched during ADD execution takes two cycles to execute because its data is external. The LAC instruction is fetched externally, but no wait state is needed since fast program memory is being used. $\overline{STRB}$ going high (inactive) signals the end of the read cycle. Data is clocked into the processor at the beginning of clock quarter-phase 4 if the READY signal was active at the beginning of quarter-phase 3 and satisfied the required setup time.

Note that one dummy execute cycle occurs in the sequence of instructions because program execution changes from PI to PE. This is discussed in detail in Section 3.4.1.

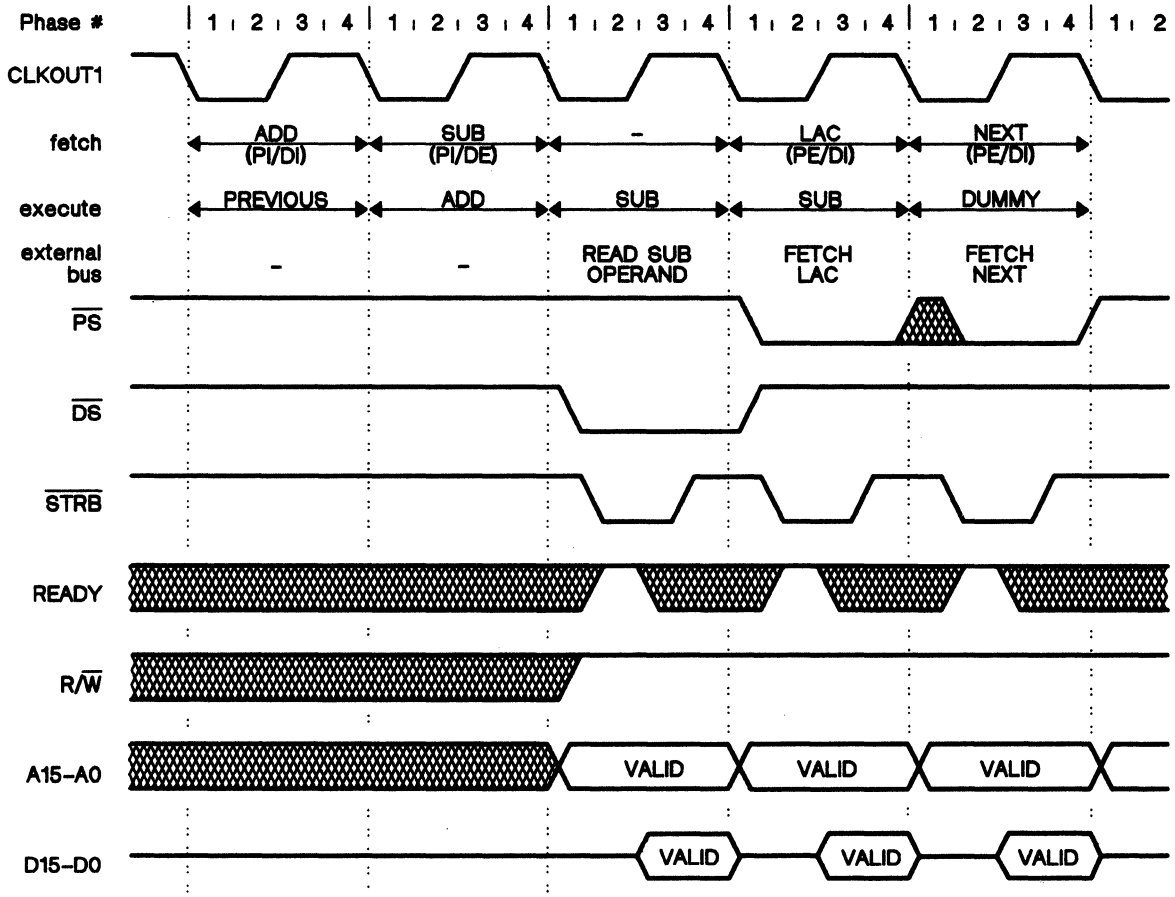


Figure 3-17. Read Cycle Functional Timing

3.5.4 External Write Cycle

The sequence of events that occurs each time an external write cycle is performed, is as follows:

1) During clock quarter-phase 1, the TMS320C25 begins driving the address bus and one of the memory space select signals. R/$\overline{W}$ is driven low to indicate an external memory write.

2) At the beginning of quarter-phase 2, $\overline{STRB}$ is asserted to indicate that the address bus is valid. $\overline{STRB}$, in conjunction with R/$\overline{W}$, may be used to gate a write enable signal.

3) After decoding the addressed memory area, the user's memory interface must provide the appropriate logic level to the READY signal input during quarter-phase 2. READY is sampled by the processor at the beginning of quarter-phase 3.

4) The data bus begins to be driven at the start of quarter-phase 2.

5) $\overline{STRB}$ is then deasserted at the beginning of quarter-phase 4. The processor ends the memory access by deactivating the address bus and $\overline{PS}$, $\overline{DS}$, or $\overline{IS}$.

The number of cycles in a memory or I/O access is determined by the state of the READY input. At the start of quarter-phase 3, the TMS320C25 samples the READY input. If READY is high, the memory access ends at the next falling edge of CLKOUT1. If READY is low, the memory cycle is extended by one machine cycle, and all other signals remain valid. At the beginning of the next quarter-phase 3, this sequence is repeated. Note that for on-chip program and data memory accesses, the READY input is ignored.

Figure 3-18 illustrates the functional timing for write operations and wait states. The timing for three instructions, SACL, SAR, and SACH, is shown. The SACL instruction stores data in external data memory, and the next instruction fetched is in internal program memory. Therefore, the SACL instruction's memory references are PI/DE. SACL only takes one cycle to complete, because the instruction writes to the zero wait-state external data memory. The SAR instruction references a PI/DI memory configuration. This instruction only takes one cycle to execute, because the data and program are internal. The SACH instruction uses slow external data memory (one wait state) and fast external program memory. SACH takes three cycles: one for the write to external data memory, one for a wait state since the external data is slow, and one for the external program fetch. External logic holds the READY line inactive during quarter-phase 2 to indicate a wait state. For write operations, $\overline{STRB}$ going high can be used to clock data into the external memory.

One dummy execute cycle also occurs in this sequence of instructions, because program execution changes from PI to PE (see Section 3.4.1).

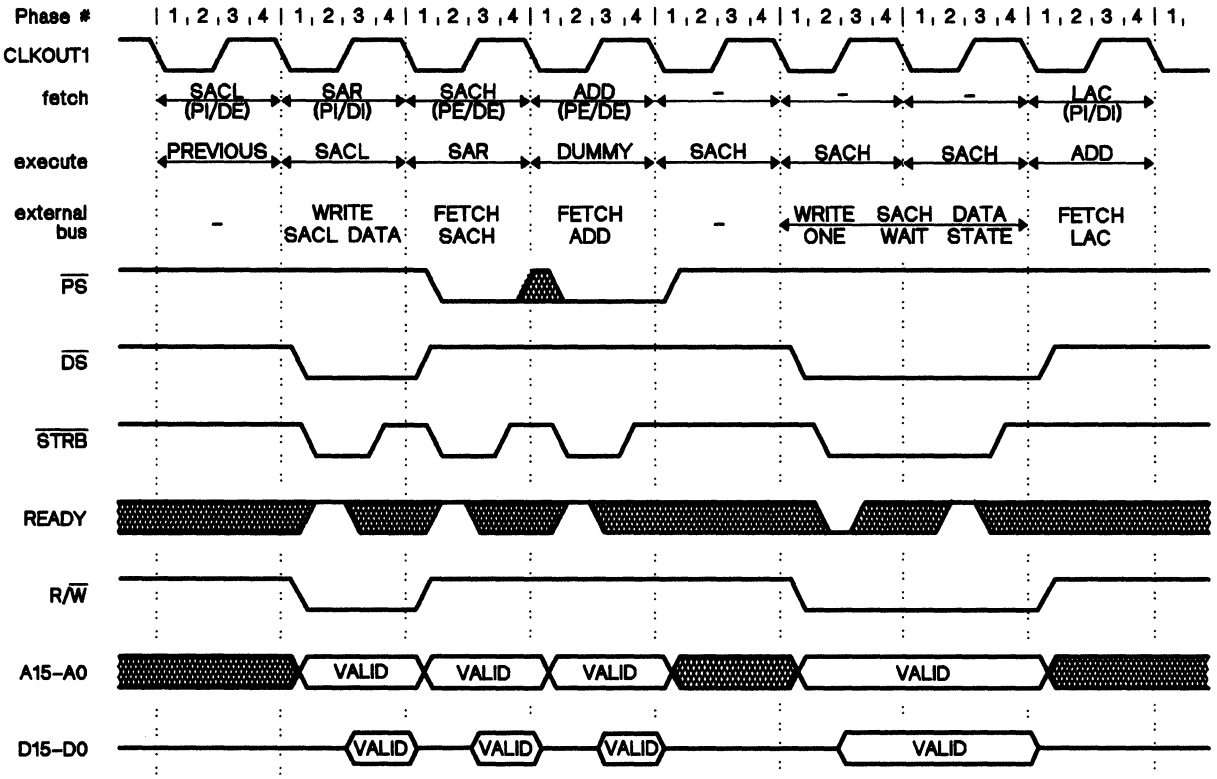


Figure 3-18. Functional Timing of Write Cycles and Wait States

3.6 Interrupts

The TMS320C25 has three external maskable user interrupts ($\overline{INT2}$-$\overline{INT0}$), available for external devices that interrupt the processor. Internal interrupts are generated by the serial port (RINT and XINT), by the timer (TINT), and by the software interrupt (TRAP) instruction. Interrupts are prioritized with reset having the highest priority and the serial port transmit interrupt having the lowest priority.

3.6.1 Interrupt Operation

This subsection details interrupt organization and management. Vector locations and priorities for all internal and external interrupts are shown in Table 3-5. The TRAP instruction, used for software interrupts, is not prioritized but is included here since 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.

Table 3-5. Interrupt Locations and Priorities

INTERRUPT NAME	MEMORY LOCATION	PRIORITY	FUNCTION
$\overline{RS}$	0	1 (highest)	External reset signal
$\overline{INT0}$	2	2	External user interrupt #0
$\overline{INT1}$	4	3	External user interrupt #1
$\overline{INT2}$	6	4	External user interrupt #2
	8-23		Reserved locations
TINT	24	5	Internal timer interrupt
RINT	26	6	Serial port receive interrupt
XINT	28	7 (lowest)	Serial port transmit interrupt
TRAP	30	N/A	TRAP instruction address

When an interrupt occurs, it is stored in the 6-bit Interrupt Flag Register (IFR). This register is set by the external user interrupts $\overline{INT}$(2-0) and the internal interrupts RINT, XINT, and TINT. Each interrupt is stored until it is recognized and then cleared by the $\overline{IACK}$ (interrupt acknowledge) signal or the $\overline{RS}$ (reset) signal. The $\overline{RS}$ signal is not stored in the IFR. No instructions are provided for reading from or writing to the IFR.

The TMS320C25 has a memory-mapped Interrupt Mask Register (IMR) for masking external and internal interrupts. The layout of the register is shown in Figure 3-19. A '1' in bit positions 5 through 0 of the IMR enables the corresponding interrupt, provided that INTM = 0. The IMR is accessible with both read and write operations but cannot be read using BLKD. When the IMR is read, the unused bits (15 through 6) will be read as '1's. The lower six bits are used to write to or read from the IMR. Note that $\overline{RS}$ is not included in the IMR, and therefore the IMR has no effect on reset.

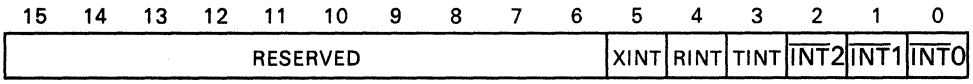


Figure 3-19. Interrupt Mask Register (IMR)

The INTM (interrupt mode) bit, which is bit 9 of status register ST0, enables or disables all maskable interrupts. A '0' in INTM enables all the unmasked interrupts, and a '1' disables these interrupts. The INTM is set to a '1' by the $\overline{IACK}$ (interrupt acknowledge) signal, the DINT instruction, or a reset. This bit is reset to a '0' by the EINT instruction. Note that the INTM does not actually modify the IMR or IFR.

The TMS320C25 has a built-in mechanism for protecting multicycle instructions. If an interrupt occurs during a multicycle instruction, the interrupt is not processed until the instruction is completed. This also includes instructions that become multicycle due to the READY signal.

In addition, the device also does not allow interrupts to be processed when an instruction is being repeated via the RPT or RPTK instructions. The interrupt is stored in the IFR until the repeat counter (RPTC) decrements to zero, and then the interrupt is processed. Note that even if the interrupt is de-asserted while the TMS320C25 is processing the RPT or RPTK, the interrupt will still be latched by IFR and be pending until RPTC decrements to zero.

If both the $\overline{HOLD}$ line and an interrupt go active during a multicycle instruction or a repeat loop, the $\overline{HOLD}$ takes control of the processor at the end of the instruction or loop. When $\overline{HOLD}$ is released, the interrupt is acknowledged.

Interrupts cannot be processed between EINT and the next instruction in a program sequence. For example, if an interrupt occurs during an EINT instruction execution, the device always completes EINT as well as the following instruction before the pending interrupt is processed. This insures that a RET can be executed before the next interrupt is processed, assuming that a RET instruction follows the EINT. The state of the machine, upon receiving an interrupt, may be saved and restored (see Section 5.3.1).

3.6.2 External Interrupt Interface

$\overline{INT}$(2-0) may be asynchronous edges or levels. The functional logic organization for $\overline{INT}$(2-0) is shown in Figure 3-20. As shown in the figure, the external interrupt $\overline{INT0}$ is connected to an edge-triggered flip-flop. The $\overline{INT0}$ signal is ORed with the interrupt edge flip-flop Q output and synchronized with internal quarter-phases 1 and 2 to produce an interrupt signal. In this way, the device can handle both edge-triggered and level-triggered interrupts.

If the INTM bit and flag register have been properly enabled, the interrupt signal is accepted by the processor. An $\overline{IACK}$ (interrupt acknowledge) signal is then generated. The $\overline{IACK}$ clears the appropriate interrupt edge flip-flop and disables the INTM latch. The logic is the same for $\overline{INT1}$ and $\overline{INT2}$.

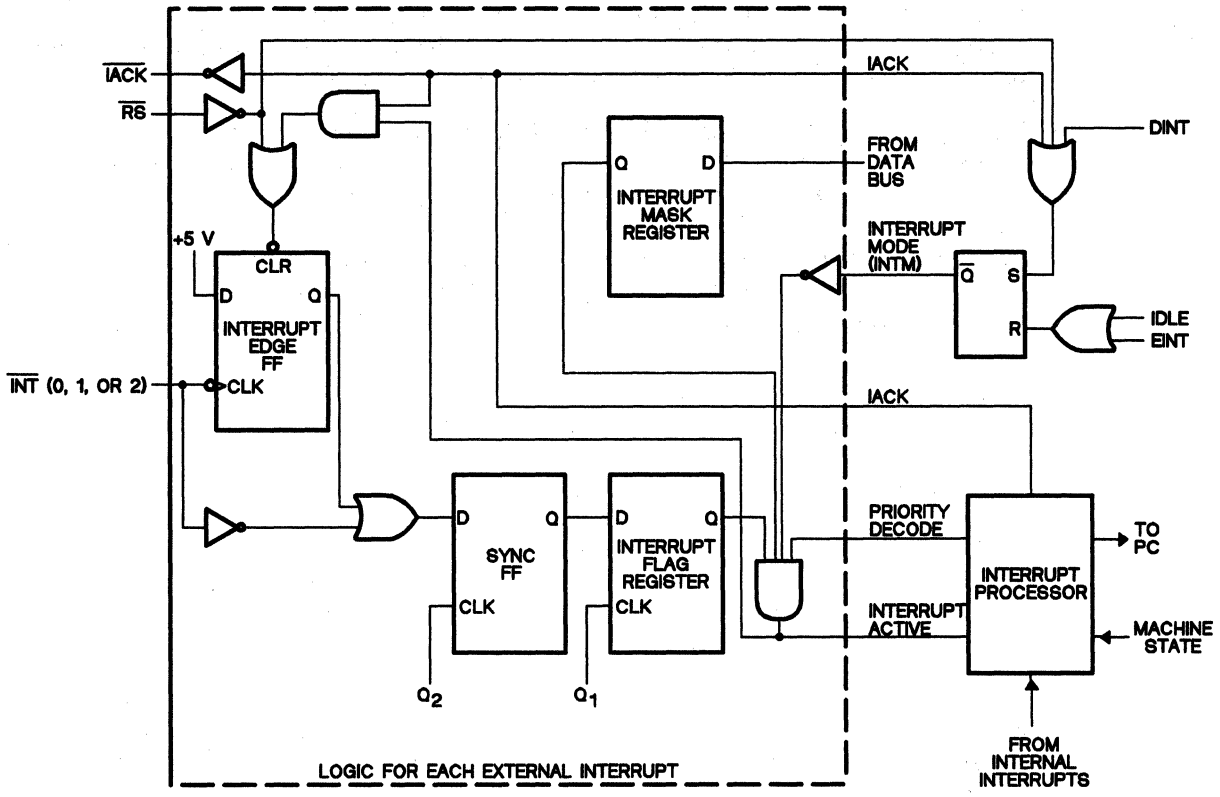
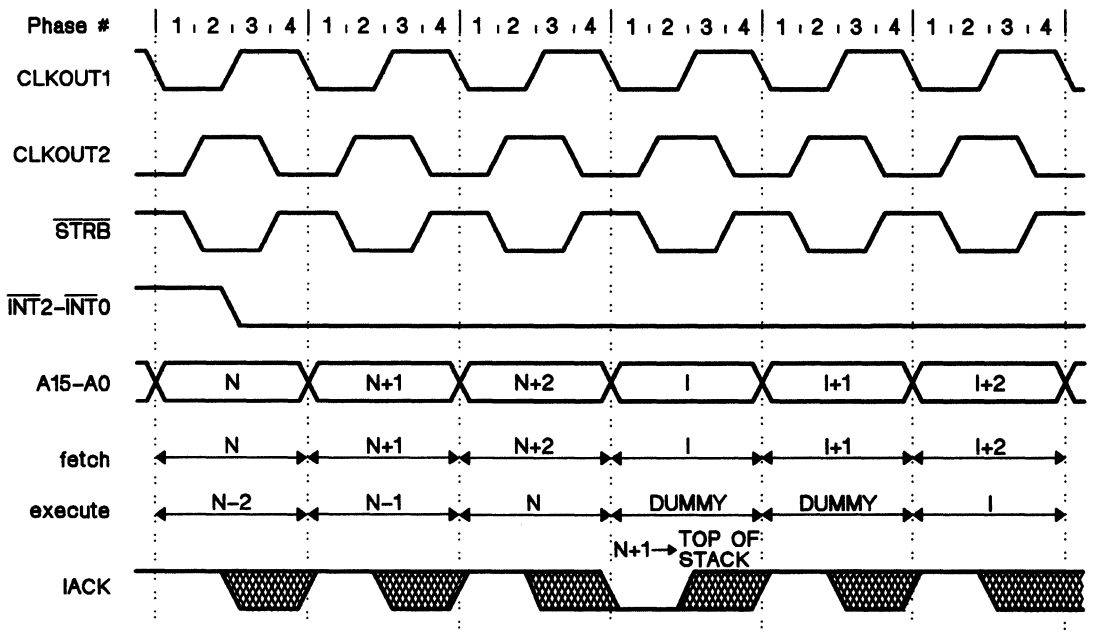


Figure 3-20. Internal Interrupt Logic Diagram

In a typical interrupt ($\overline{INT}2$-$\overline{INT}0$) operation, the interrupt is generated by a negative-going edge and the IFR bit is set. Since INTM is disabled when the interrupt is acknowledged, the level may continue to be present on the $\overline{INT}$ input without generating further interrupts. If the level is removed before an EINT instruction is executed, no further interrupts are generated. If a low level continues to be present after the EINT, another interrupt is generated after the EINT/next instruction sequence. In addition, if the $\overline{INT}$ pin is pulsed between the previous $\overline{IACK}$ and EINT, another interrupt is generated after EINT/RET, because the corresponding IFR bit is again set.

The timing diagram of Figure 3-21, shows an interrupt, interrupt acknowledge, and various other signals for the special case of single-cycle instructions. An interrupt generated during the current (N) fetch cycle still allows the fetch and execution of that instruction. The N+1 and N+2 instructions are also fetched, then discarded, and the address N+1 is pushed onto the top of the stack. The instruction is fetched again upon a return command from the interrupt routine.

As shown in Figure 3-21, two dummy execute cycles occur on an interrupt. The $\overline{IACK}$ signal is asserted low during CLKOUT1 low when the device initiates a fetch from interrupt location I. Therefore, an external device can determine the interrupt that occurred by latching the address bus value present on A4-A1 with the rising edge of CLKOUT2 when $\overline{IACK}$ is low.



Notes:
1. N is the program memory location for the current instruction.
2. I is the interrupt vector location in program memory for the active interrupt.
3. For simplicity, this example only shows the execution of single-cycle instructions fetched from external program memory, rather than multicycle instructions.

Figure 3-21. Interrupt Timing Diagram

3.7 Serial Port

The on-chip serial port provides direct communication with serial devices such as codecs, serial A/D converters, and other serial systems. The interface signals are compatible with codecs and many other serial devices with a minimum of external hardware. The serial port may also be used for intercommunication between processors in multiprocessing applications.

Serial port operation is controlled by the following registers and mode bits:

- Data Transmit Register (DXR)
- Transmit Shift Register (XSR)
- Data Receive Register (DRR)
- Receive Shift Register (RSR)
- Format bit (FO)
- Transmit Mode bit (TXM)
- Frame Synchronization Mode bit (FSM)

The serial port uses two memory-mapped registers: the DXR register that holds the data to be transmitted by the serial port, and the DRR register that holds the received data (see Figure 3-22). Any instruction accessing data memory can be used to read from or write to these registers; however, the BLKD (block move from

data memory to data memory) instruction cannot be used to read these registers. The DXR and DRR registers are mapped into locations 0 and 1 in the data address space. The XSR and RSR registers are not directly accessible through software.

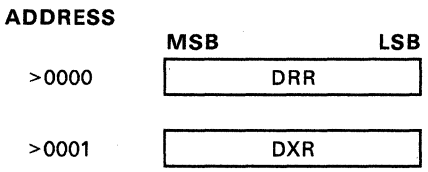


Figure 3-22. The DRR and DXR Registers

The transmit and receive sections of the serial port are implemented separately to allow independent transmit and receive operations, as shown in Figure 3-23. Externally, the serial port interface is implemented using the following six pins on the TMS320C25 device:

- Transmitted serial data (DX)
- Transmit clock (CLKX)
- Transmit framing synchronization signal (FSX)
- Received serial data (RX)
- Receive clock (CLKR)
- Receive framing synchronization signal (FSR)

The data on the DX and DR pins is clocked out of or into the XSR or RSR by the CLKX or CLKR signal, respectively. CLKX and CLKR are only required to be present during actual serial port transfers, and may be stopped when no data is being transferred. Data bits can be transferred in either 8-bit bytes or 16-bit words. Data is clocked out of XSR on the rising edges of CLKX, while data is clocked into RSR on the falling edges of CLKR. The MSB of the data is transferred first.

The XSR and RSR are connected to the DXR and DRR, respectively. For transmit operations, the contents of DXR are transferred to XSR when a new transmission begins. For a receive operation, the contents of RSR are transferred to DRR when all of the bits have been received. Thus, the serial port is double-buffered since data may be transferred to or from the DXR or DRR while another transmit or receive operation is being performed.

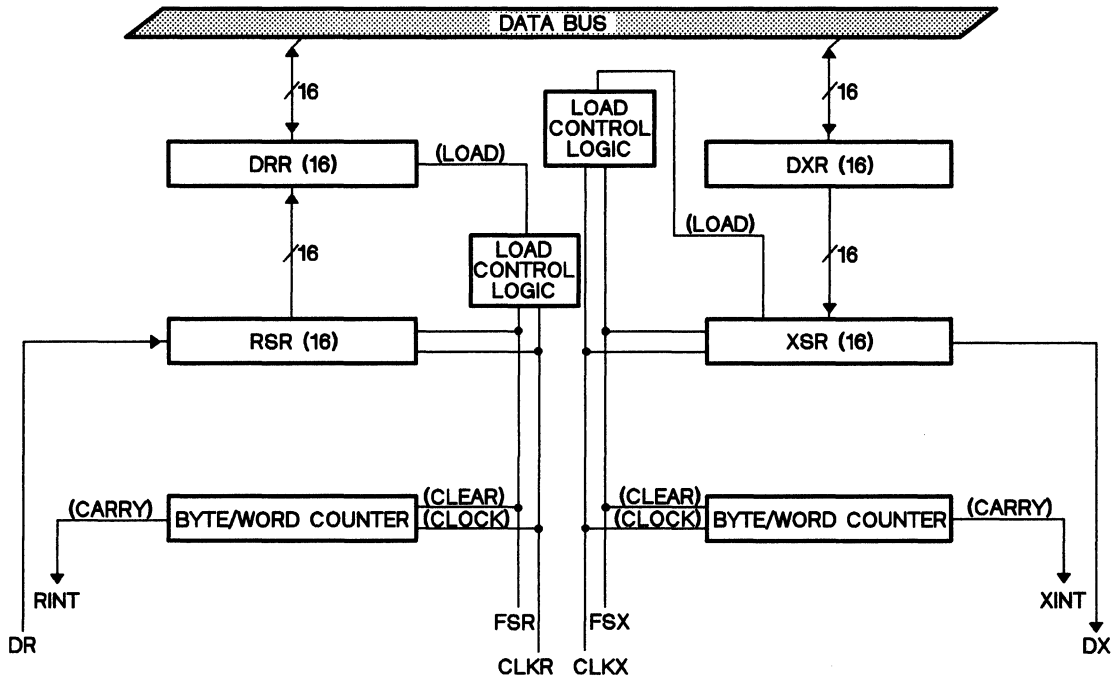


Figure 3-23. Serial Port Block Diagram

Serial port transfers are generally initiated by a frame sync pulse. The exception to this is when the continuous mode of operation is used with FSM=0, as described in a subsequent paragraph. Frame sync pulses are input on FSX for transmit operations and on FSR for receive operations.

Upon completion of a serial port transfer, an internal interrupt is generated. The RINT interrupt is generated for a receive operation, and XINT is generated for a transmit operation. RINT and XINT are generated on the rising edge of CLKR and CLKX, respectively, after the last bit is transferred. Note that if DRR is read before a RINT is received, it will contain the data from the previous operation. Similarly, if DXR is loaded more than once after an XINT is generated (in the continuous transmission mode), only the last value written will be loaded into XSR for the next transmit operation.

When the TMS320C25 is reset, TXM is cleared to zero, and DX is placed in a high-impedance state. Any transmit or receive operation that is in progress when the reset occurs is terminated.

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

The FO (format) bit, located in status register ST1, is used to define whether data to be transmitted and received is an 8-bit byte or a 16-bit word. If FO = 0, the data is formatted in 16-bit words. If FO = 1, the data is formatted in 8-bit bytes. In the 8-bit mode, only the lower eight least-significant bits are used for transmit/receive operations. The FO bit is loaded by the FORT (format serial port registers) instruction. On reset, FO is set to a '0'.

The TXM (transmit mode) bit, also located in status register ST1, is used to determine if the frame sync pulse for the transmit operation is generated internally or externally. If TXM=0, FSX is an input, but if TXM=1, FSX becomes an output and frame sync pulses are produced on FSX at the beginning of a serial port transmission. The TXM bit can be loaded by the LST1, STXM, or RTXM instructions.

The FSM (frame synchronization mode) status register bit is used to select whether frame sync pulses are required for each serial port transfer. If FSM=1, frame sync pulses are required; if FSM=0, they are not required. FSM is set by the SFSM (set frame synchronization mode) instruction and cleared by the RFSM (reset frame synchronization mode) instruction.

The timing of the serial port signals is compatible with the TI/Intel 2910 series codecs. The timing is also compatible with the AMI S3506 series codecs if the frame synchronization signals are inverted.

3.7.1 Burst-Mode Operation

In burst-mode serial port operation, transfers are separated in time by periods of no serial port activity (the serial port does not operate continuously). For burst-mode operation, FSM must be set to one. Timing of the serial port in this mode of operation is shown in Figure 3-24 and Figure 3-25.

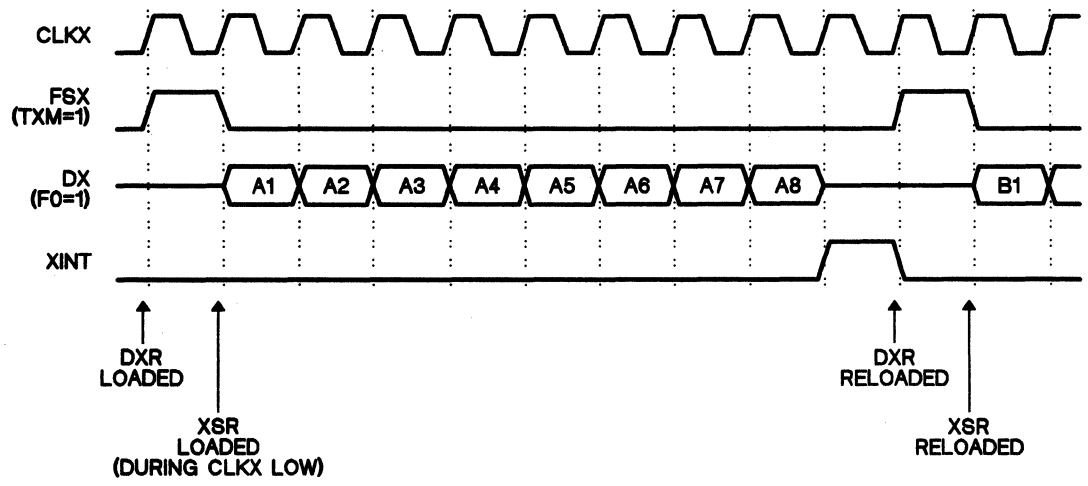


Figure 3-24. Burst-Mode Serial Port Transmit Operation

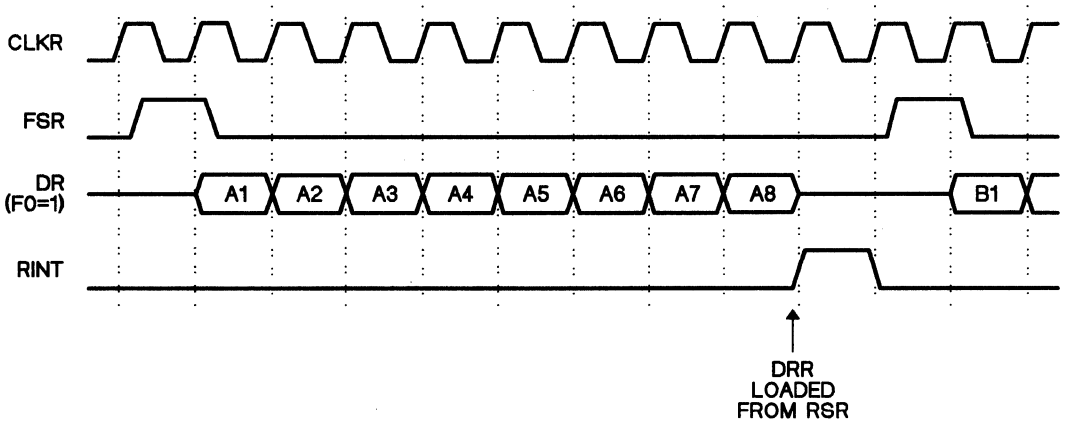


Figure 3-25. Burst-Mode Serial Port Receive Operation

When TXM=1 (in the transmit operation) and the serial port register DXR is loaded, a framing pulse is generated on the next rising edge of CLKX. XSR is loaded with the current contents of DXR while FSX is high and CLKX is low. Transmission begins when FSX goes low while CLKX is high or is going high. Figure 3-24 shows the timing for the byte mode (FO=1). XINT is generated on the rising edge of CLKX after all 8 or 16 bits have been transmitted and DX is placed in the high-impedance state. If DXR is reloaded before the next rising edge of CLKX after XINT, FSX will again be generated as shown, and XSR will be reloaded.

The receive operation is very similar to the transmit operation. The contents of RSR are loaded into DRR while CLKR is low, just after reception of the last bit sent by the transmitting device (see Figure 3-25). RINT is generated on the next rising edge of CLKR, and DRR may be read at any time before the reception of the final bit of the next transmission. When operating in the byte mode, the eight most-significant bits of the DRR are the contents of the eight least-significant bits of the DRR prior to reception of the current byte, as shown in Figure 3-26.

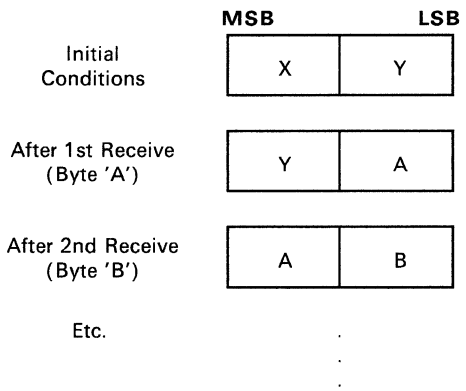


Figure 3-26. Byte-Mode DRR Operation

3.7.2 Continuous-Mode Operation Using Frame Sync Pulses

The TMS320C25 provides two modes of operation that allow the use of a continuous stream of serial data. When FSM=1, frame sync pulses are required, but since DXR is double-buffered, continuous operation is achieved even if TXM=1. Writing to DXR during a serial port transmission does not abort the transmission in progress, but instead DXR stores that data until XSR can be reloaded. As long as DXR is reloaded before the CLKX rising edge on the final bit being transmitted, the FSX pulse will go high on the rising edge of CLKX during the transmission of the final bit and fall on the next rising edge when transmission of the word just loaded begins. If DXR is not reloaded within this period and FSM =1, the DX pin will be placed in a high-impedance state for at least one CLKX cycle until DXR is reloaded (as described in the previous section). Figure 3-27 and Figure 3-28 show the timing diagrams for the continuous operation with frame sync pulses.

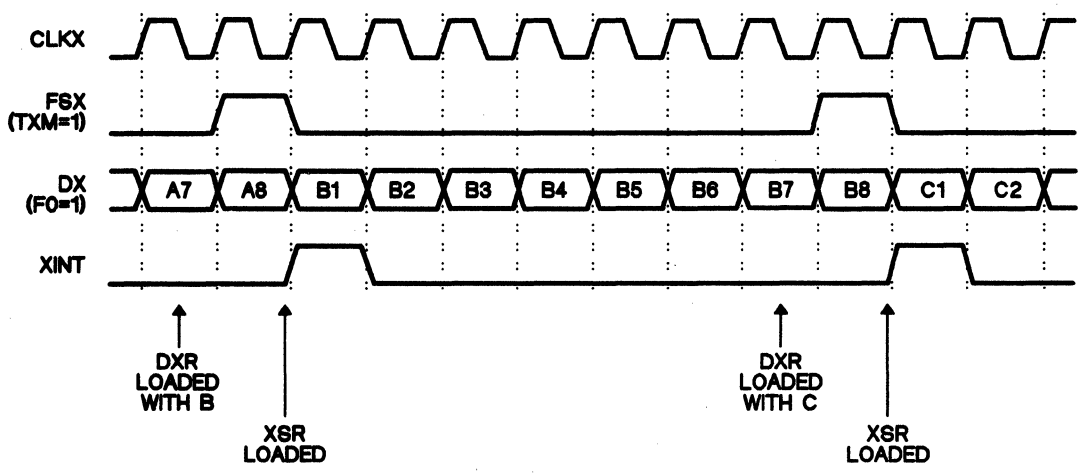


Figure 3-27. Serial Port Transmit Continuous Operation (FSM=1)

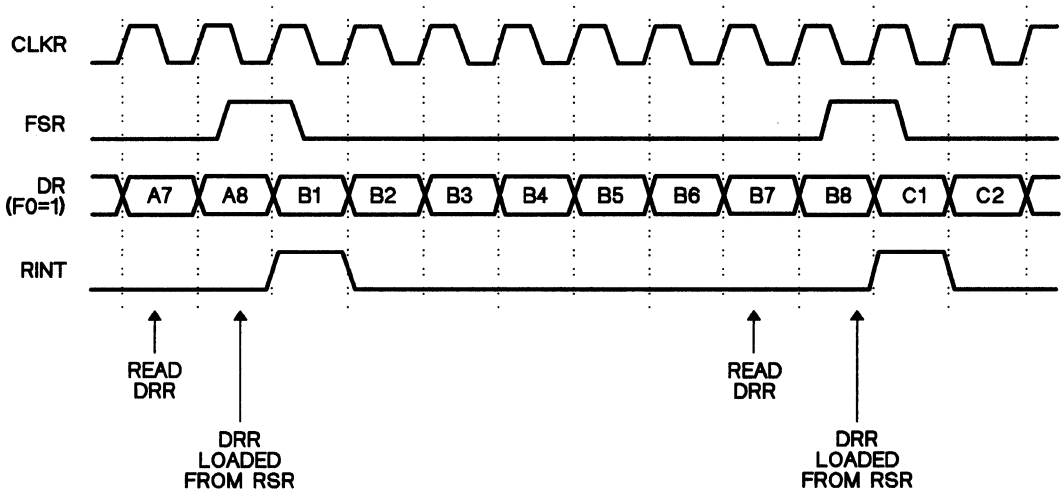


Figure 3-28. Serial Port Receive Continuous Operation (FSM=1)

Continuous receive operation with FSM=1 is identical to that of burst-mode operation with the exception that FSR is pulsed during reception of the final bit.

3.7.3 Continuous-Mode Operation Without Frame Sync Pulses

The continuous mode of operation allows transmission and reception of a continuous bit stream without requiring frame sync pulses every 8 or 16 bits. This mode is selected by setting FSM=0.

Figure 3-29 and Figure 3-30 show operation of the serial port for both states of FSM to illustrate differences in operation for each case. FSM is initially set to one, and frame sync pulses are required to initiate serial transfers. During processing of the next serial port interrupt (XINT or RINT), FSM is reset to zero by means of an RFSM (reset FSM) instruction. RFSM can occur either before or after the write to DXR or read from DRR. From this point on, the FSX and FSR inputs are ignored, with transmission occurring every CLKX cycle and reception occurring every CLKR cycle as long as those clocks are present.

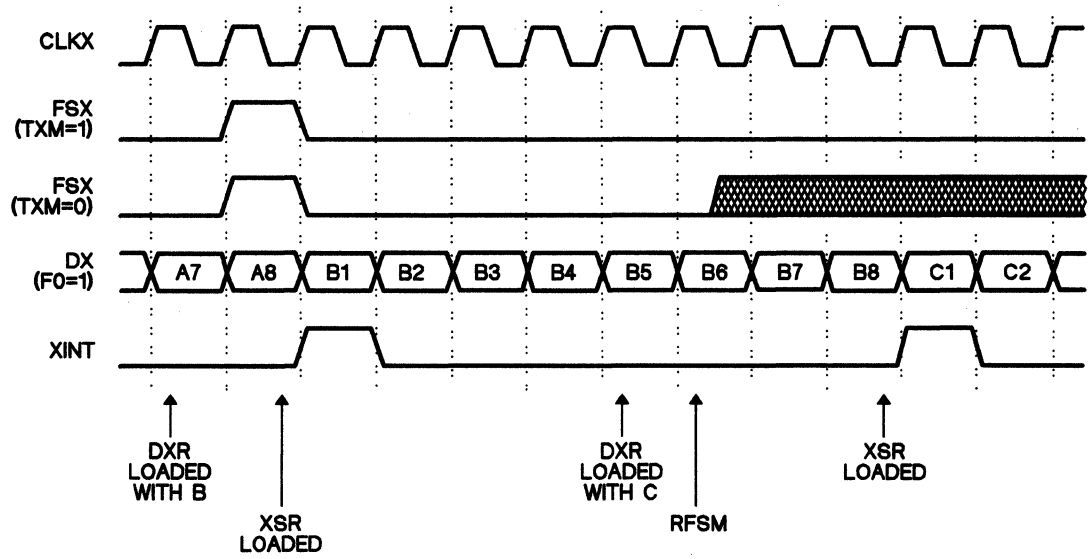


Figure 3-29. Serial Port Transmit Continuous Operation (FSM=0)

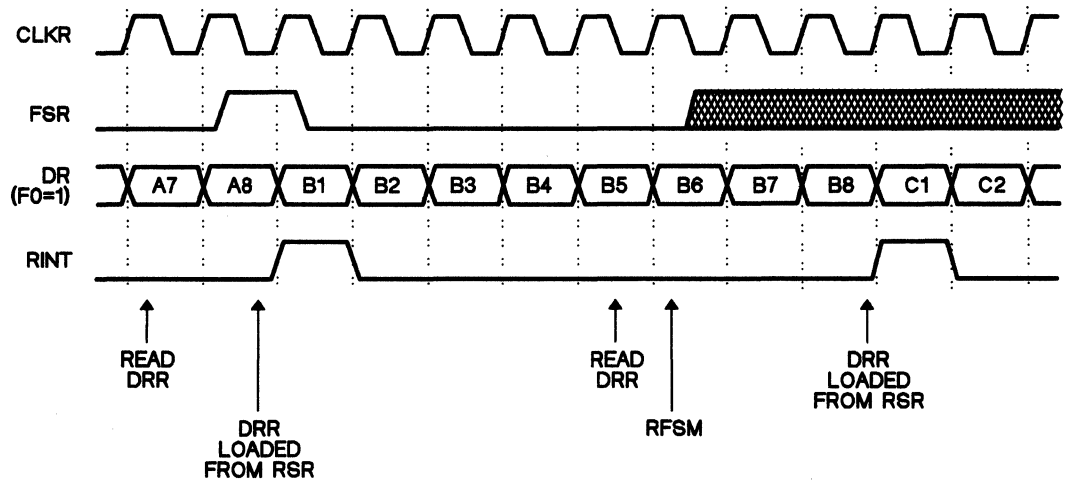


Figure 3-30. Serial Port Receive Continuous Operation (FSM=0)

If FSX is configured as an output, it will remain low until FSM is set back to one and DXR is reloaded. If DXR is not reloaded with new data every XINT (every 8 or 16 CLKX cycles depending on FO), the last value loaded will be transmitted on DX continuously. Note that this is different from the case with FSM=1 where DX is placed into a high-impedance state if DXR is not reloaded before transmission of the last bit of the current word in XSR. For example,if byte C is not loaded into DXR as indicated in Figure 3-29, bits B1-B8 will be retransmitted instead of bits C1 and C2 as shown.

For receive operations, DRR is loaded from RSR (and an RINT is generated) every 8 or 16 CLKR cycles (depending on FO), regardless of whether or not DRR has been read. An overrun of DRR is also possible with FSM=1 if DRR is not read before the next RINT. The only way to stop continuous transmission or reception once started, when FSM=0, is to either stop CLKX or CLKR or to perform an SFSM (set FSM) instruction.

Continuous transmission without frame sync pulses is very useful in communicating directly to telephone system PCM highways. For AT&T T1 and CCITT G711/712 lines, FSX and FSR pulses are generated only every 24 or 32 bytes. By counting the transmitted and received bytes in software after an initial FSX or FSR and performing SFSM and RFSM instructions as required, the TMS320C25 can easily be made to communicate in these formats.

3.7.4 Initialization of Continuous-Mode Operation Without Frame Sync Pulses

FSM is normally initialized during an XINT or RINT service routine to enable or disable FSX and FSR, respectively, for the next serial port operation. However, in order to initialize the continuous-mode operation, it is permissible to reset FSM to zero before a serial port transmit or receive is initiated. As shown in Figure 3-31 and Figure 3-32, RFSM may occur before a write to DXR, regardless of the state of TXM. If TXM=1, FSX is generated in a normal manner on the next rising edge of CLKX, but only once. If TXM=0, the TMS320C25 waits to transmit until FSX is pulsed, but from then on, the FSX input is ignored. Note that just as in the case of continuous-mode operation without sync pulses described in Section 3.7.3, the first data written to DXR (byte A) is output twice unless DXR is reloaded before the second transmission is started. It is important to consider this dummy cycle when using continuous-mode serial operation.

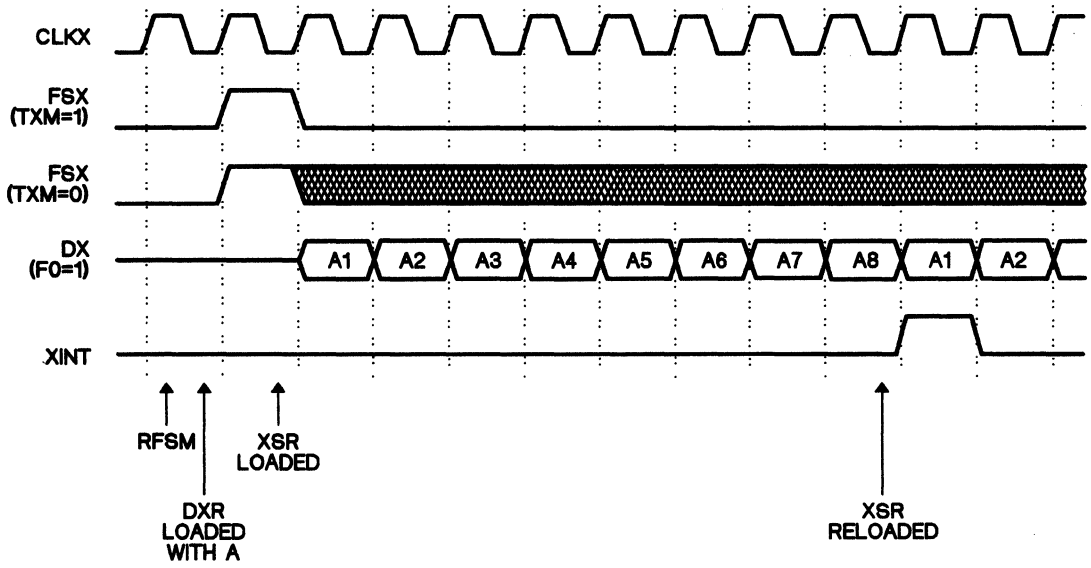


Figure 3-31. Continuous Transmit Operation Initialization

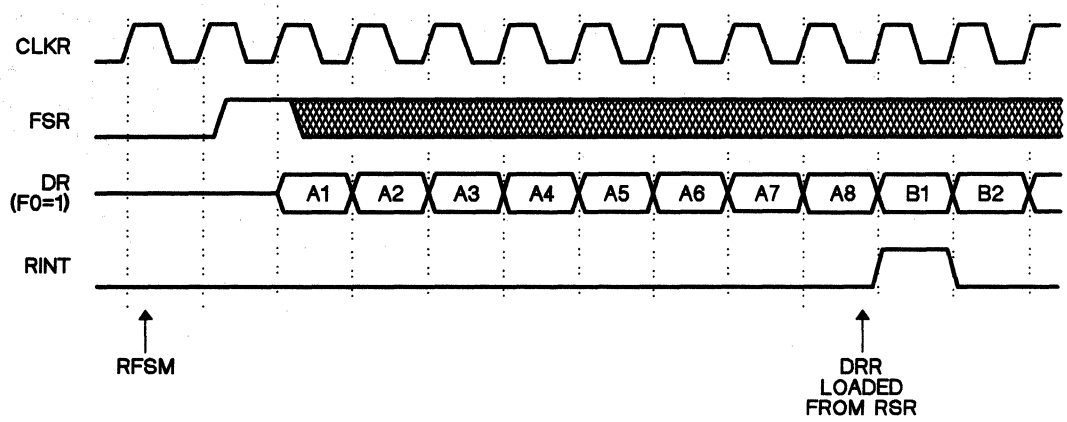


Figure 3-32. Continuous Receive Operation Initialization

The receive timings are the same as those for the transmit operations with TXM=0. The TMS320C25 waits to receive data until FSR is pulsed, but thereafter the FSR input is ignored. No dummy cycle is associated with the receive operation due to the postbuffering nature of DRR as opposed to the prebuffering nature of DXR.

3.8 Multiprocessing and Direct Memory Access (DMA)

The flexibility of the TMS320C25 allows configurations to satisfy a wide range of system requirements. Some of the system configurations using the TMS320C25 are as follows:

- A standalone system (single processor)
- A host/slave or parallel multiprocessing system with shared global data memory
- A host/peripheral coprocessor configuration using interface control signals.

These system configurations are made possible by three specialized features of the TMS320C25. These three features are the synchronization function utilizing the $\overline{\text{SYNC}}$ input, the global memory interface, and the hold function implemented with the $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$ pins. The following sections describe these functions in detail.

3.8.1 Synchronization

In a multiprocessor environment, the $\overline{\text{SYNC}}$ input can be used to greatly ease interface between processors. This input is used to cause each of the TMS320C25s in the system to synchronize their internal clocks, thereby allowing the processors to run in lock-step operation.

Multiple TMS320C25s are synchronized by using common $\overline{\text{SYNC}}$ and external clock inputs. A negative transition on $\overline{\text{SYNC}}$ sets each processor to internal quarter-phase one (Q1). This transition must occur synchronously with the rising edge of CLKIN. The timing diagram for the $\overline{\text{SYNC}}$ input is shown in Figure 3-33. Note that there is a two CLKIN cycle delay following the cycle in which $\overline{\text{SYNC}}$ goes low, before the synchronized Q1 occurs.

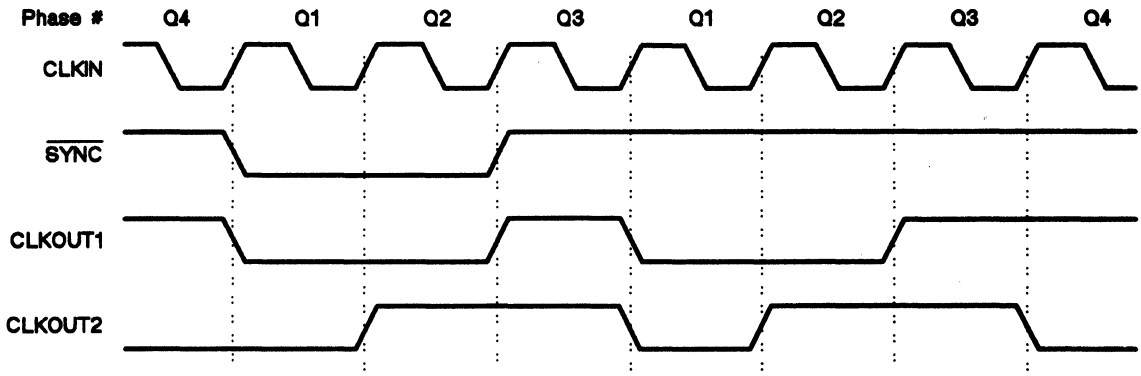


Figure 3-33. Synchronization Timing Diagram

Normally, $\overline{\text{SYNC}}$ is applied while $\overline{\text{RS}}$ is active. If $\overline{\text{SYNC}}$ is asserted after a reset, the following can occur:

1) The processor machine cycle is reset to Q1, provided that the timing requirements for $\overline{\text{SYNC}}$ are met. If $\overline{\text{SYNC}}$ is asserted at the beginning of Q1, Q3, or Q4, the current instruction is improperly executed. If $\overline{\text{SYNC}}$ is asserted at the beginning of Q2, the current instruction is executed properly.

2) If $\overline{\text{SYNC}}$ does not meet the timing requirements, unpredictable processor operation occurs. A reset should then be executed to place the processor back in a known state.

3.8.2 Global Memory

For multiprocessing applications, the TMS320C25 has the capability of allocating global data memory space and communicating with that space via the $\overline{\text{BR}}$ (bus request) and READY control signals.

Global memory is memory shared by more than one processor; therefore, access to it must be arbitrated. When using global memory, the processor's address space is divided into local and global sections. The local section is used by the processor to perform its individual function, and the global section is used to communicate with other processors.

A memory-mapped register (GREG) is provided that allows part of data memory to be specified as global external memory. GREG, which is memory-mapped at data memory address location 5, is an eight-bit register connected to the eight LSBs of the internal D bus. The upper eight bits of location 5 are nonexistent and read as '1's.

The contents of GREG determine the size of the global memory space. The legal values of GREG and corresponding global memory spaces are shown in Table 3-6. Note that values other than those listed in the table lead to fragmented memory maps.

Table 3-6. Global Data Memory Configurations

GREG VALUE	LOCAL MEMORY RANGE	LOCAL MEMORY # WORDS	GLOBAL MEMORY RANGE	GLOBAL MEMORY # WORDS
000000XX	>0 - >FFFF	65,536	----------	0
10000000	>0 - >7FFF	32,768	>8000 - >FFFF	32,768
11000000	>0 - >BFFF	49,152	>C000 - >FFFF	16,384
11100000	>0 - >DFFF	57,344	>E000 - >FFFF	8,192
11110000	>0 - >EFFF	61,440	>F000 - >FFFF	4,096
11111000	>0 - >F7FF	63,488	>F800 - >FFFF	2,048
11111100	>0 - >FBFF	64,512	>FC00 - >FFFF	1,024
11111110	>0 - >FDFF	65,024	>FE00 - >FFFF	512
11111111	>0 - >FEFF	65,280	>FF00 - >FFFF	256

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 TMS320C25 has control. One wait-state timing is shown in Figure 3-34. Note that all signals not shown have the same timing as in the normal read or write case.

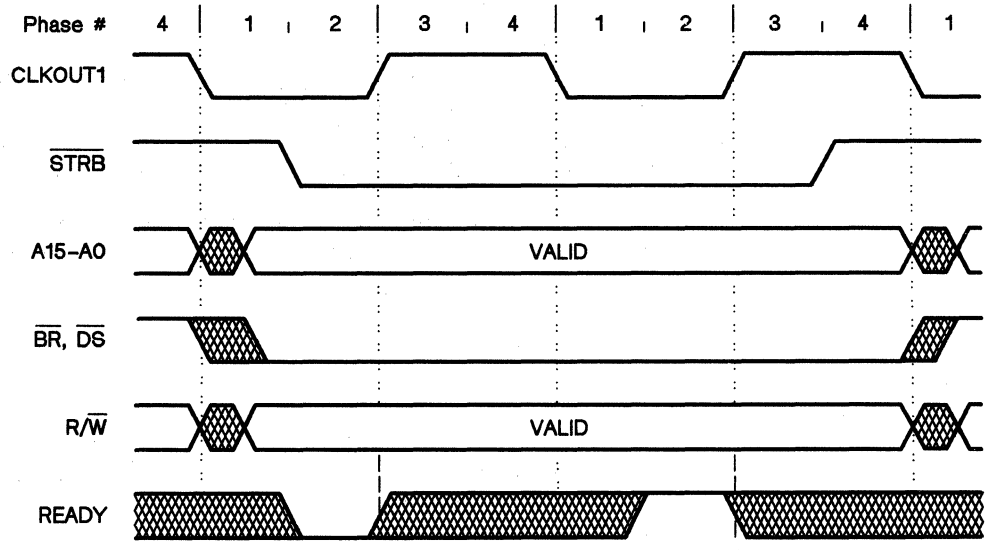


Figure 3-34. Global Memory Access Timing

3.8.3 The Hold Function

The TMS320C25 supports Direct Memory Access (DMA) to its local (off-chip) program, data, and I/O spaces. Two signals, $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$, are provided to allow another device to take control of the processor's buses. Upon receiving a $\overline{\text{HOLD}}$ signal from an external device, the processor acknowledges by bringing $\overline{\text{HOLDA}}$ low. The processor then places its address and data buses as well as all control signals ($\overline{\text{PS}}$, $\overline{\text{DS}}$, $\overline{\text{IS}}$, R/$\overline{\text{W}}$, and $\overline{\text{STRB}}$) in the high-impedance state. The serial port output pins, DX and FSX, are not affected by $\overline{\text{HOLD}}$.

The timing for the $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$ signals is shown in Figure 3-35. $\overline{\text{HOLD}}$ has the same setup time as READY and is sampled at the beginning of quarter-phase 3. If the setup time is met, it takes three machine cycles before the buses and control signals go to the high-impedance state. Note that unlike the external interrupts $\overline{\text{INT}}$(2-0), $\overline{\text{HOLD}}$ is not a latched input. The external device must keep $\overline{\text{HOLD}}$ low until it receives a $\overline{\text{HOLDA}}$ from the TMS320C25.

The hold function has two distinct operating modes, which are selected by the HM (hold mode) status register bit. The $\overline{\text{HOLD}}$ signal is pulled low, as shown in the first part of Figure 3-35. When HM=1, the TMS320C25 halts program execution and enters the hold state directly. When HM=0, the processor enters the hold state directly, as shown in Figure 3-35, if program execution is from external memory or if external data memory is being accessed. If program execution is from internal memory, however, and if no external data memory accesses are required, the processor enters the hold state externally, but program execution continues internally. This allows more efficient system operation since a program may continue executing while an external DMA operation is being performed.

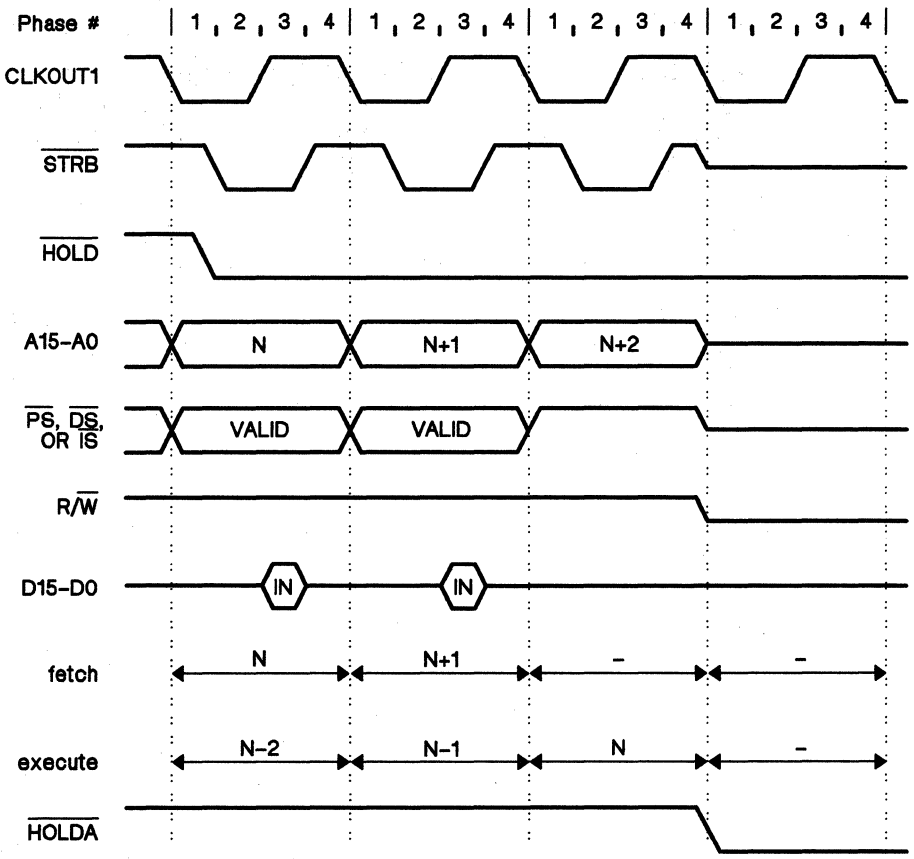
Note that if the processor is in a hold state with HM=0 and an internally executing program requires an external access, or if the program branches to an external address, program execution ceases until $\overline{\text{HOLD}}$ is removed. Also, if a repeat instruction that requires the use of the external bus is executing with HM=0 and a hold occurs, the hold state is entered after the current bus cycle. If this situation occurs with HM=1, the hold state will not be entered until the repeat count is completed. HM is set and reset by the SHM (set hold mode) and RHM (reset hold mode) instructions, respectively.

If the TMS320C25 is in the middle of a multicycle instruction, it will finish the instruction before entering the hold state. After the instruction is completed, the buses are placed in the high-impedance state. This also applies to instructions that become multicycle due to insertion of wait states.

After $\overline{\text{HOLD}}$ is de-asserted, program execution resumes from the same point at which it was halted. $\overline{\text{HOLDA}}$ is removed synchronously with $\overline{\text{HOLD}}$, as shown in Figure 3-35. If the setup time is met, two machine cycles are required before the buses and control signals become valid.

All interrupts are disabled while $\overline{\text{HOLD}}$ is active with HM=1. If an interrupt is received during this period, the interrupt is latched and remains pending. $\overline{\text{HOLD}}$ itself does not affect any interrupt flags or registers. If HM=0, interrupts function normally.

$\overline{\text{HOLD}}$ is not treated as an interrupt. If the TMS320C25 was executing the IDLE instruction before entering the hold state, it resumes executing IDLE once it leaves the hold state.



Notes:
1. N is the program memory location for the current instruction.
2. This example only shows the execution of single-cycle instructions fetched from external program memory.

Figure 3-35. Hold Timing Diagram

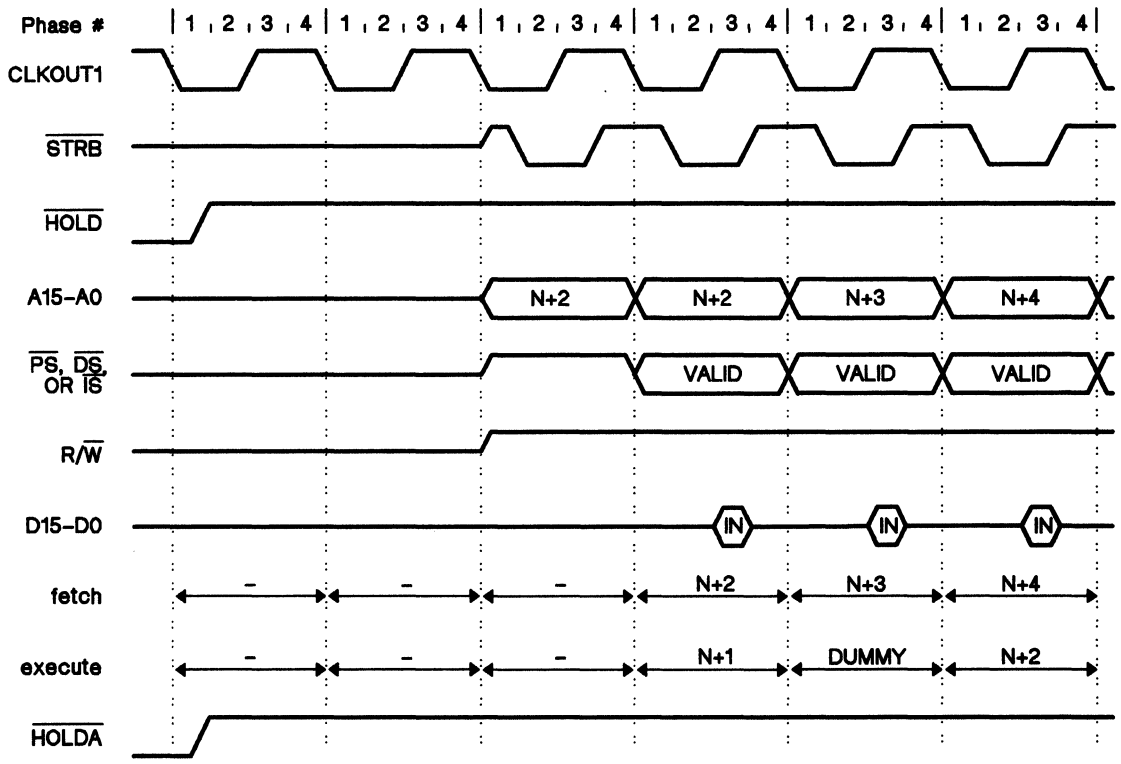


Figure 3-35. Hold Timing Diagram (Concluded)

3.9 General-Purpose I/O Pins

The TMS320C25 has two general-purpose pins that are software-controlled. The $\overline{\text{BIO}}$ pin is a branch control input pin, and the XF pin is an external flag output pin.

3.9.1 $\overline{\text{BIO}}$ Input

When the $\overline{\text{BIO}}$ input pin is active (low), execution of the BIOZ instruction causes a branch to occur.

The $\overline{\text{BIO}}$ pin is useful for monitoring peripheral device status. It is especially useful as an alternative to using an interrupt when it is necessary not to disturb time-critical loops.

Figure 3-36 shows the $\overline{\text{BIO}}$ timing diagram. $\overline{\text{BIO}}$ is sampled at the end of quarter-phase 4. Note that the timing diagram shown is for a sequence of single-cycle, single-word instructions without branches located in external memory. Because of variations in pipelining due to instructions prior to and following the BIOZ instruction, this timing may vary. Therefore, it is recommended that several cycles of setup be provided if $\overline{\text{BIO}}$ is to be recognized on a particular cycle.

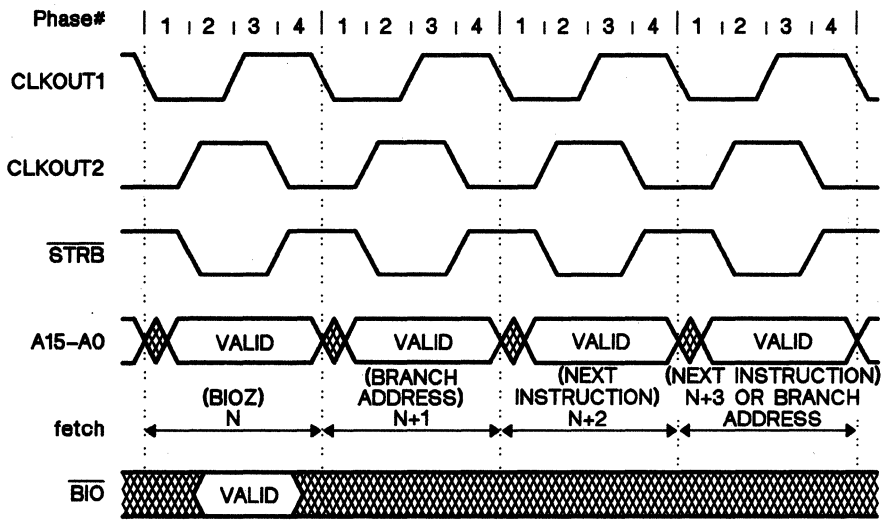
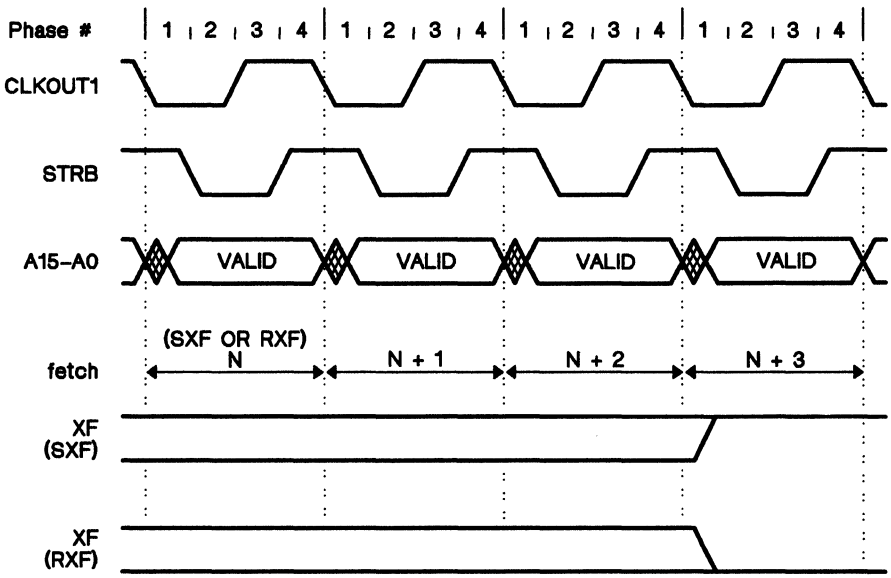


Figure 3-36. $\overline{\text{BIO}}$ Timing Diagram

3.9.2 External Flag Output

The XF (external flag) output pin is set to a high level by the SXF (set external flag) instruction and reset to a low level by the RXF (reset external flag) instruction. XF is set high by $\overline{\text{RS}}$.

The relationship between the time the SXF/RXF instruction is fetched before the XF pin is set or reset is shown in Figure 3-37. As with $\overline{\text{BIO}}$, 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.



Notes: 1. N is the program memory location for the current instruction.
2. This example only shows the execution of single-cycle instructions fetched from external program memory.

Figure 3-37. External Flag Timing Diagram

4. Assembly Language Instructions

The TMS320C25 instruction set supports numeric-intensive signal processing operations as well as general-purpose applications, such as multiprocessing and high-speed control. TMS32010 source code is upward-compatible with TMS320C25 source code. TMS32020 object code is upward-compatible with TMS320C25 object code.

This section describes the assembly language instructions for the TMS320C25 microprocessor. Included in this section are the following major topics:

- Memory Addressing Modes (Section 4.1 on page 4-2)
  - Direct addressing
  - Indirect addressing (using eight auxiliary registers)
  - Immediate addressing
- Instruction Set (Section 4.2 on page 4-8)
  - Symbols and abbreviations used in the instructions
  - Instruction set summary (listed according to function)
- Individual Instruction Descriptions (Section 4.3 on page 4-13)
  - Presented in alphabetical order and providing the following:
    - Assembler syntax
    - Operands
    - Execution
    - Encoding
    - Description
    - Words
    - Cycles
    - Repeatability
    - Example(s)

4.1 Memory Addressing Modes

The TMS320C25 instruction set provides three memory addressing modes:

- Direct addressing mode
- Indirect addressing mode
- Immediate 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 the eight auxiliary registers. In immediate addressing, the data is based on a portion of the instruction word(s). The following sections 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 word 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 through the LDP (load data memory page pointer), LDPK (load data memory page pointer immediate), or LST (load status register ST0) instructions. Figure 4-1 illustrates how the 16-bit data address is formed.

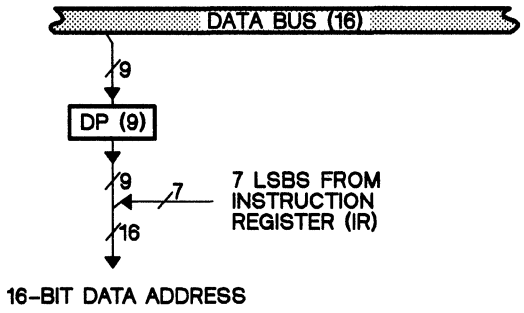


Figure 4-1. Direct Addressing Block Diagram

Direct addressing can be used with all instructions except CALL, the branch instructions, immediate operand instructions, and instructions with no operands. The direct addressing format is as follows:

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

Bits 15 through 8 contain the opcode. Bit 7 = 0 defines the addressing mode as direct, and bits 6 through 0 contain the data memory address (dma).

Example of Direct Addressing Format:

ADD 9,5 Add to accumulator the contents of data memory location 9 left-shifted 5 bits.

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

The opcode of the ADD 9,5 instruction is >05 and appears in bits 15 through 8. The notation >nn indicates nn is a hexadecimal number. The shift count of >5 appears in bits 11 through 8 of the opcode. The data memory address >09 appears in bits 6 through 0.

4.1.2 Indirect Addressing Mode

The eight auxiliary registers (AR0-AR7) provide flexible and powerful indirect addressing. To select a specific auxiliary register, the Auxiliary Register Pointer (ARP) is loaded with a value from 0 through 7, designating AR0 through AR7, respectively (see Figure 4-2).

The contents of the auxiliary registers may be operated upon by the Auxiliary Register Arithmetic Unit (ARAU), which implements 16-bit unsigned arithmetic. The ARAU performs auxiliary register arithmetic operations in the same cycle as the execution of the instruction. (Note that the increment or decrement of the indicated AR is always executed after the use of that AR in the instruction.)

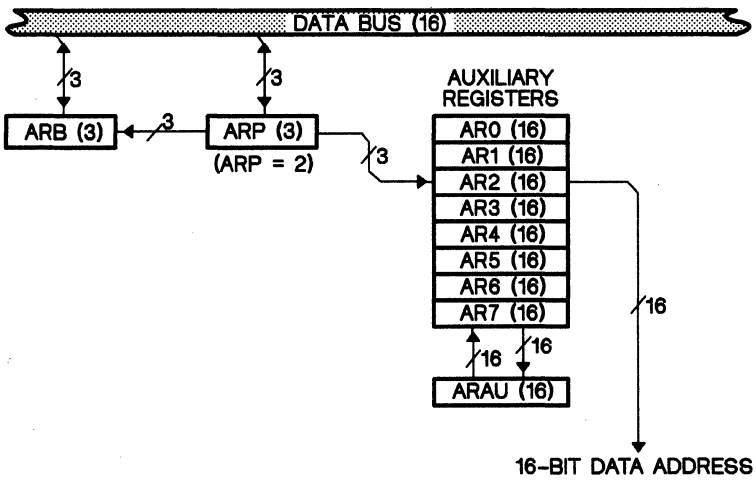


Figure 4-2. Indirect Addressing Block Diagram

In indirect addressing, any location in the 64K data memory space can be accessed via the 16-bit addresses contained in the auxiliary registers. These may be loaded by the instructions LAR (load auxiliary register), LARK (load auxiliary register immediate), and LRLK (load auxiliary register long immediate). The auxiliary registers may be modified by ADRK (add to auxiliary register short immediate) or SBRK (subtract from auxiliary register short immediate). The auxiliary registers may also be modified by the MAR (modify auxiliary register) instruction or, equivalently, by

the indirect addressing field of any instruction supporting indirect addressing. AR(ARP) denotes the auxiliary register selected by ARP.

The following symbols are used in indirect addressing:

*	Contents of AR(ARP) are used as the data memory address.
*-	Contents of AR(ARP) are used as the data memory address, then decremented after the access.
*+	Contents of AR(ARP) are used as the data memory address, then incremented after the access.
*0-	Contents of AR(ARP) are used as the data memory address, and the contents of AR0 subtracted from it after the access.
*0+	Contents of AR(ARP) are used as the data memory address, and the contents of AR0 added to it after the access.
*BR0-	Contents of AR(ARP) are used as the data memory address, and the contents of AR0 subtracted from it (with reverse carry propagation) after the access.
*BR0+	Contents of AR(ARP) are used as the data memory address, and the contents of AR0 added to it (with reverse carry propagation) after the access.

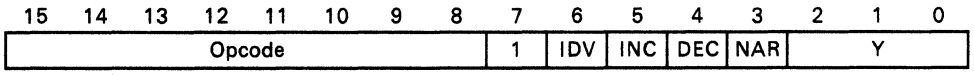
There are two main types of indirect addressing with indexing:

- Regular indirect addressing with increment or decrement, and
- Indirect addressing with indexing based on the value of AR0.

In either case, the contents of the auxiliary register pointed to by the ARP register are used as the address of the data memory operand. Then, the ARAU performs the specified mathematical operation on the indicated auxiliary register. Additionally, the ARP may be loaded with a new value.

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 based upon the contents of AR0.

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



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 determines whether AR0 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, AR0 may be 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 AR0. 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.

Table 4-1. Indirect Addressing Arithmetic Operations

BITS 6	5	4	ARITHMETIC OPERATION
0	0	0	No operation on AR(ARP)
0	0	1	AR(ARP) - 1 → AR(ARP)
0	1	0	AR(ARP) + 1 → AR(ARP)
0	1	1	Not used
1	0	0	AR(ARP) - AR0 → AR(ARP) [reverse carry propagation]
1	0	1	AR(ARP) - AR0 → AR(ARP)
1	1	0	AR(ARP) + AR0 → AR(ARP)
1	1	1	AR(ARP) + AR0 → AR(ARP) [reverse carry propagation]

Bit 3 and bits 2 through 0 control the Auxiliary Register Pointer (ARP). Bit 3 (NAR) determines if a new value is loaded into the ARP. If bit 3 = 1, the contents of bits 2 through 0 (Y = next ARP) are loaded into the ARP. If bit 3 = 0, the contents of the ARP remain unchanged.

Table 4-2. Bit Fields for Indirect Addressing

INSTRUCTION FIELD BITS 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	NOTATION	OPERATION
← OPCODE → 1 0 0 0 0 ← Y →	*	No manipulation of ARs/ARP
← OPCODE → 1 0 0 0 1 ← Y →	*,Y	Y → ARP
← OPCODE → 1 0 0 1 0 ← Y →	*-	AR(ARP)-1 → AR(ARP)
← OPCODE → 1 0 0 1 1 ← Y →	*-,Y	AR(ARP)-1 → AR(ARP); Y → ARP
← OPCODE → 1 0 1 0 0 ← Y →	*+	AR(ARP)+1 → AR(ARP)
← OPCODE → 1 0 1 0 1 ← Y →	*+,Y	AR(ARP)+1 → AR(ARP); Y → ARP
← OPCODE → 1 1 0 0 0 ← Y →	*BR0-	AR(ARP)-rcAR0 → AR(ARP)†
← OPCODE → 1 1 0 0 1 ← Y →	*BR0-,Y	AR(ARP)-rcAR0 → AR(ARP); Y → ARP†
← OPCODE → 1 1 0 1 0 ← Y →	*0-	AR(ARP)-AR0 → AR(ARP)
← OPCODE → 1 1 0 1 1 ← Y →	*0-,Y	AR(ARP)-AR0 → AR(ARP); Y → ARP
← OPCODE → 1 1 1 0 0 ← Y →	*0+	AR(ARP)+AR0 → AR(ARP)
← OPCODE → 1 1 1 0 1 ← Y →	*0+,Y	AR(ARP)+AR0 → AR(ARP); Y → ARP
← OPCODE → 1 1 1 1 0 ← Y →	*BR0+	AR(ARP)+rcAR0 → AR(ARP)†
← OPCODE → 1 1 1 1 1 ← Y →	*BR0+,Y	AR(ARP)+rcAR0 → AR(ARP); Y → ARP†

†rc = reverse carry propagation

For some instructions, the notation in Table 4-2 includes a shift code, e.g., *0+,8,3 where 8 is the shift code and Y = 3.

The CMPR (compare auxiliary register with AR0), and BBZ/BBNZ (branch if TC bit equal/not equal to zero) instructions facilitate conditional branches based on comparisons between the contents of AR0 and the contents of AR(ARP).

The auxiliary registers may also be used for temporary storage via the load and store auxiliary register instructions, LAR and SAR, respectively.

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 opcode is >08A0, as shown below.

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

Example 2:

ADD *,8 As in Example 1, but with no autoincrement; the opcode is >0880.

Example 3:

ADD *-,8 As in Example 1, except that the current auxiliary register is decremented by one; the opcode is >0890.

Example 4:

ADD *0+,8 As in Example 1, except that the contents of auxiliary register AR0 are added to the current auxiliary register; the opcode is >08E0.

Example 5:

ADD *0-,8 As in Example 1, except that the contents of auxiliary register AR0 are subtracted from the current auxiliary register; the opcode is >08D0.

Example 6:

ADD *+,8,3 As in Example 1, except that the auxiliary register pointer (ARP) is loaded with the value 3 for subsequent instructions; the opcode is >08AB.

Example 7:

ADD *BR0-,8 The opcode is >08C0. The contents of auxiliary register AR0 are subtracted from the current auxiliary register with reverse carry propagation.

Example 8:

ADD *BR0+,8 The opcode is >08F0. The contents of auxiliary register AR0 are added to the current auxiliary register with reverse carry propagation.

4.1.3 Immediate Addressing Mode

In immediate addressing, the instruction word(s) contains the value of the immediate operand. The immediate operand may be contained within the instruction word itself or in the word following the opcode.

The following instructions contain the immediate operand in the instruction word and execute within a single instruction cycle. The length of the constant operand is instruction-dependent.

ADDK Add to accumulator short immediate (8-bit absolute constant)

ADRK Add to auxiliary register short immediate (8-bit absolute constant)

LACK Load accumulator immediate short (8-bit absolute constant)

LARK Load auxiliary register immediate short (8-bit absolute constant)

LARP Load auxiliary register pointer (3-bit constant)

LDPK Load data memory page pointer immediate (9-bit constant)

MPYK Multiply immediate (13-bit two's-complement constant)

RPTK Repeat instruction as specified by immediate value (8-bit constant)

SBRK Subtract from auxiliary register short immediate (8-bit absolute constant)

SUBK Subtract from accumulator short immediate (8-bit absolute constant).

For the other immediate instructions, the constant is a 16-bit value in the word following the opcode. The 16-bit value can be optionally used as an absolute constant or as a two's-complement value.

ADLK Add to accumulator long immediate with shift (absolute or two's complement)

ANDK AND immediate with accumulator with shift

LALK Load accumulator long immediate with shift (absolute or two's complement)

LRLK Load auxiliary register long immediate

ORK OR immediate with accumulator with shift

SBLK Subtract from accumulator long immediate with shift (absolute or two's complement)

XORK Exclusive-OR immediate with accumulator with shift.

The following examples illustrate immediate addressing format:

Example 1:

ADLK 16384,2 Add to the accumulator the value 16384 with a shift to the left of two, effectively adding 65536 to the contents of the accumulator.

The ADLK instruction uses the word following the instruction opcode as the immediate operand. The instruction format for ADLK is as follows:

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

Example 2:

RPTK 99 Execute the instruction following this instruction 100 times.

With the RPTK instruction, the immediate operand is contained as a part of the instruction opcode. The instruction format for RPTK is as follows:

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

4.2 Instruction Set

The following sections list the symbols and abbreviations used in the instruction set summary and in the instruction descriptions. The complete instruction set summary is organized according to function. A detailed description of each instruction is listed in the instruction set summary.

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.

Table 4-3. Instruction Symbols

SYMBOL	MEANING
ACC	Accumulator
ARB	Auxiliary register pointer buffer
ARn	Auxiliary Register n (AR0 through AR7 are predefined assembler symbols equal to 0 through 7, respectively.)
ARP	Auxiliary register pointer
B	4-bit field specifying a bit code
BIO	Branch control input
C	Carry bit
CM	2-bit field specifying compare mode
CNF	On-chip RAM configuration control bit
D	Data memory address field
DATn	Label assigned to data memory location n
dma	Data memory address
DP	Data page pointer
FO	Format status bit
FSM	Frame synchronization mode bit
HM	Hold mode bit
I	Addressing mode bit
INTM	Interrupt mode flag bit
K	Immediate operand field
>nn	Indicates nn is a hexadecimal number. (All others are assumed to be decimal values.)
OV	Overflow mode flag bit
OVM	Overflow mode bit
P	Product register
PA	Port address (PA0 through PA15 are predefined assembler symbols equal to 0 through 15, respectively.)
PC	Program counter
PM	2-bit field specifying P register output shift code
pma	Program memory address
PRGn	Label assigned to program memory location n
R	3-bit operand field specifying auxiliary register
RPTC	Repeat counter
S	4-bit left-shift code
STn	Status register n (ST0 or ST1)
SXM	Sign-extension mode bit
T	Temporary register
TC	Test control bit
TOS	Top of stack
TXM	Transmit mode bit
X	3-bit accumulator left-shift field
XF	XF pin status bit
→	Is assigned to
| |	An absolute value
< >	User-defined items
[ ]	Optional items
( )	Contents of
{ }	Alternative items, one of which must be entered
	Blanks or spaces must be entered where shown.

4.2.2 Instruction Set Summary

The instruction set summary of Table 4-4 is arranged according to function and alphabetized within each functional grouping. Additional information is presented in the individual instruction descriptions in the following section. The symbol † indicates instructions that are not included in the TMS32010 instruction set. The symbol ‡ indicates instructions that are not included in the TMS32020 instruction set.

Table 4-4. Instruction Set Summary

ACCUMULATOR MEMORY REFERENCE INSTRUCTIONS			
MNEMONIC	DESCRIPTION	NO. WORDS	INSTRUCTION BIT CODE 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ABS	Absolute value of accumulator	1	1 1 0 0 1 1 1 0 0 0 0 1 1 0 1 1
ADD	Add to accumulator with shift	1	0 0 0 0 ◄—S—► I ◄———D———►
ADDC‡	Add to accumulator with carry	1	0 1 0 0 0 0 1 1 I ◄———D———►
ADDH	Add to high accumulator	1	0 1 0 0 1 0 0 0 I ◄———D———►
ADDK‡	Add to accumulator short immediate	1	1 1 0 0 1 1 0 0 ◄———K———►
ADDS	Add to low accumulator with sign extension suppressed	1	0 1 0 0 1 0 0 1 I ◄———D———►
ADDT†	Add to accumulator with shift specified by T register	1	0 1 0 0 1 0 1 0 I ◄———D———►
ADLK†	Add to accumulator long immediate with shift	2	1 1 0 1 ◄—S—► 0 0 0 0 0 0 1 0
AND	AND with accumulator	1	0 1 0 0 1 1 1 0 I ◄———D———►
ANDK†	AND immediate with accumulator with shift	2	1 1 0 1 ◄—S—► 0 0 0 0 0 1 0 0
CMPL†	Complement accumulator	1	1 1 0 0 1 1 1 0 0 0 1 0 0 1 1 1
LAC	Load accumulator with shift	1	0 0 1 0 ◄—S—► I ◄———D———►
LACK	Load accumulator immediate short	1	1 1 0 0 1 0 1 0 ◄———K———►
LACT†	Load accumulator with shift specified by T register	1	0 1 0 0 0 0 1 0 I ◄———D———►
LALK†	Load accumulator long immediate with shift	2	1 1 0 1 ◄—S—► 0 0 0 0 0 0 0 1
NEG†	Negate accumulator	1	1 1 0 0 1 1 1 0 0 0 1 0 0 0 1 1
NORM†	Normalize contents of accumulator	1	1 1 0 0 1 1 1 0 1 ◄———D———►
OR	OR with accumulator	1	0 1 0 0 1 1 0 1 I ◄———D———►
ORK†	OR immediate with accumulator with shift	2	1 1 0 1 ◄—S—► 0 0 0 0 0 1 0 1
ROL‡	Rotate accumulator left	1	1 1 0 0 1 1 1 0 0 0 1 1 0 1 0 0
ROR‡	Rotate accumulator right	1	1 1 0 0 1 1 1 0 0 0 1 1 0 1 0 1
SACH	Store high accumulator with shift	1	0 1 1 0 1 ◄—X—► I ◄———D———►
SACL	Store low accumulator with shift	1	0 1 1 0 0 ◄—X—► I ◄———D———►
SBLK†	Subtract from accumulator long immediate with shift	2	1 1 0 1 ◄—S—► 0 0 0 0 0 0 1 1
SFL†	Shift accumulator left	1	1 1 0 0 1 1 1 0 0 0 0 1 1 0 0 0
SFR†	Shift accumulator right	1	1 1 0 0 1 1 1 0 0 0 0 1 1 0 0 1
SUB	Subtract from accumulator with shift	1	0 0 0 1 ◄—S—► I ◄———D———►
SUBB‡	Subtract from accumulator with borrow	1	0 1 0 0 1 1 1 1 I ◄———D———►
SUBC	Conditional subtract	1	0 1 0 0 0 1 1 1 I ◄———D———►
SUBH	Subtract from high accumulator	1	0 1 0 0 0 1 0 0 I ◄———D———►
SUBK‡	Subtract from accumulator short immediate	1	1 1 0 0 1 1 0 1 ◄———K———►
SUBS	Subtract from low accumulator with sign extension suppressed	1	0 1 0 0 0 1 0 1 I ◄———D———►
SUBT†	Subtract from accumulator with shift specified by T register	1	0 1 0 0 0 1 1 0 I ◄———D———►
XOR	Exclusive-OR with accumulator	1	0 1 0 0 1 1 0 0 I ◄———D———►
XORK†	Exclusive-OR immediate with accumulator with shift	2	1 1 0 1 ◄—S—► 0 0 0 0 0 1 1 0
ZAC	Zero accumulator.	1	1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0
ZALH	Zero low accumulator and load high accumulator	1	0 1 0 0 0 0 0 0 I ◄———D———►
ZALR‡	Zero low accumulator and load high accumulator with rounding	1	0 1 1 1 1 0 1 1 I ◄———D———►
ZALS	Zero accumulator and load low accumulator with sign extension suppressed	1	0 1 0 0 0 0 0 1 I ◄———D———►

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

Table 4-4. Instruction Set Summary (Continued)

AUXILIARY REGISTERS AND DATA PAGE POINTER INSTRUCTIONS																	
MNEMONIC	DESCRIPTION	NO. WORDS	INSTRUCTION BIT CODE														
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADRK‡	Add to auxiliary register short immediate	1	0	1	1	1	1	1	1	0	◄	K						►
CMPR†	Compare auxiliary register with auxiliary register AR0	1	1	1	0	0	1	1	1	0	0	1	0	1	0	0	◄	CM ►
LAR	Load auxiliary register	1	0	0	1	1	0	◄	R	►	I	◄		D				►
LARK	Load auxiliary register short immediate	1	1	1	0	0	0	◄	R	►	◄			K				►
LARP	Load auxiliary register pointer	1	0	1	0	1	0	1	0	1	1	0	0	0	1	◄	R	►
LDP	Load data memory page pointer	1	0	1	0	1	0	0	1	0	I	◄		D				►
LDPK	Load data memory page pointer immediate	1	1	1	0	0	1	0	0	◄			DP					►
LRLK†	Load auxiliary register long immediate	2	1	1	0	1	0	◄	R	►	0	0	0	0	0	0	0	0
MAR	Modify auxiliary register	1	0	1	0	1	0	1	0	1	I	◄		D				►
SAR	Store auxiliary register	1	0	1	1	1	0	◄	R	►	I	◄		D				►
SBRK‡	Subtract from auxiliary register short immediate	1	0	1	1	1	1	1	1	1	◄			K				►
T REGISTER, P REGISTER, AND MULTIPLY INSTRUCTIONS																		
MNEMONIC	DESCRIPTION	NO. WORDS	INSTRUCTION BIT CODE															
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
APAC	Add P register to accumulator	1	1	1	0	0	1	1	1	0	0	0	0	1	0	1	0	1
LPH†	Load high P register	1	0	1	0	1	0	0	1	1	I	◄		D				►
LT	Load T register	1	0	0	1	1	1	1	0	0	I	◄		D				►
LTA	Load T register and accumulate previous product	1	0	0	1	1	1	1	0	1	I	◄		D				►
LTD	Load T register, accumulate previous product, and move data	1	0	0	1	1	1	1	1	1	I	◄		D				►
LTP†	Load T register and store P register in accumulator	1	0	0	1	1	1	1	1	0	I	◄		D				►
LTS†	Load T register and subtract previous product	1	0	1	0	1	1	0	1	1	I	◄		D				►
MAC†	Multiply and accumulate	2	0	1	0	1	1	1	0	1	I	◄		D				►
MACD†	Multiply and accumulate with data move	2	0	1	0	1	1	1	0	0	I	◄		D				►
MPY	Multiply (with T register, store product in P register)	1	0	0	1	1	1	0	0	0	I	◄		D				►
MPYA‡	Multiply and accumulate previous product	1	0	0	1	1	1	0	1	0	I	◄		D				►
MPYK	Multiply immediate	1	1	0	1	◄							K					►
MPYS‡	Multiply and subtract previous product	1	0	0	1	1	1	0	1	1	I	◄		D				►
MPYU‡	Multiply unsigned	1	1	1	0	0	1	1	1	1	I	◄		D				►
PAC	Load accumulator with P register	1	1	1	0	0	1	1	1	0	0	0	0	1	0	1	0	0
SPAC	Subtract P register from accumulator	1	1	1	0	0	1	1	1	0	0	0	0	1	0	1	1	0
SPH‡	Store high P register	1	0	1	1	1	1	1	0	1	I	◄		D				►
SPL‡	Store low P register	1	0	1	1	1	1	1	0	0	I	◄		D				►
SPM†	Set P register output shift mode	1	1	1	0	0	1	1	1	0	0	0	0	0	1	0	◄	PM ►
SQRA†	Square and accumulate	1	0	0	1	1	1	0	0	1	I	◄		D				►
SQRS†	Square and subtract previous product	1	0	1	0	1	1	0	1	0	I	◄		D				►

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

Table 4-4. Instruction Set Summary (Continued)

BRANCH/CALL INSTRUCTIONS			
MNEMONIC	DESCRIPTION	NO. WORDS	INSTRUCTION BIT CODE
			15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
B	Branch unconditionally	2	1 1 1 1 1 1 1 1 1 ◄——D——►
BACC†	Branch to address specified by accumulator	1	1 1 0 0 1 1 1 0 0 0 1 0 0 1 0 1
BANZ	Branch on auxiliary register not zero	2	1 1 1 1 1 0 1 1 1 ◄——D——►
BBNZ†	Branch if TC bit ≠ 0	2	1 1 1 1 1 0 0 1 1 ◄——D——►
BBZ†	Branch if TC bit = 0	2	1 1 1 1 1 0 0 0 1 ◄——D——►
BC‡	Branch on carry	2	0 1 0 1 1 1 1 0 1 ◄——D——►
BGEZ	Branch if accumulator ≥ 0	2	1 1 1 1 0 1 0 0 1 ◄——D——►
BGZ	Branch if accumulator > 0	2	1 1 1 1 0 0 0 1 1 ◄——D——►
BIOZ	Branch on I/O status = 0	2	1 1 1 1 1 0 1 0 1 ◄——D——►
BLEZ	Branch if accumulator ≤ 0	2	1 1 1 1 0 0 1 0 1 ◄——D——►
BLZ	Branch if accumulator < 0	2	1 1 1 1 0 0 1 1 1 ◄——D——►
BNC‡	Branch on no carry	2	0 1 0 1 1 1 1 1 1 ◄——D——►
BNV†	Branch if no overflow	2	1 1 1 1 0 1 1 1 1 ◄——D——►
BNZ	Branch if accumulator ≠ 0	2	1 1 1 1 0 1 0 1 1 ◄——D——►
BV	Branch on overflow	2	1 1 1 1 0 0 0 0 1 ◄——D——►
BZ	Branch if accumulator = 0	2	1 1 1 1 0 1 1 0 1 ◄——D——►
CALA	Call subroutine indirect	1	1 1 0 0 1 1 1 0 0 0 1 0 0 1 0 0
CALL	Call subroutine	2	1 1 1 1 1 1 1 0 1 ◄——D——►
RET	Return from subroutine	1	1 1 0 0 1 1 1 0 0 0 1 0 0 1 1 0

I/O AND DATA MEMORY OPERATIONS			
MNEMONIC	DESCRIPTION	NO. WORDS	INSTRUCTION BIT CODE
			15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
BLKD†	Block move from data memory to data memory	2	1 1 1 1 1 1 0 1 I ◄——D——►
BLKP†	Block move from program memory to data memory	2	1 1 1 1 1 1 0 0 I ◄——D——►
DMOV	Data move in data memory	1	0 1 0 1 0 1 1 0 I ◄——D——►
FORT†	Format serial port registers	1	1 1 0 0 1 1 1 0 0 0 0 0 1 1 1 FO
IN	Input data from port	1	1 0 0 0 ◄—PA—► I ◄——D——►
OUT	Output data to port	1	1 1 1 0 ◄—PA—► I ◄——D——►
RFSM‡	Reset serial port frame synchronization mode	1	1 1 0 0 1 1 1 0 0 0 1 1 0 1 1 0
RTXM†	Reset serial port transmit mode	1	1 1 0 0 1 1 1 0 0 0 1 0 0 0 0 0
RXF†	Reset external flag	1	1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 0
SFSM‡	Set serial port frame synchronization mode	1	1 1 0 0 1 1 1 0 0 0 1 1 0 1 1 1
STXM†	Set serial port transmit mode	1	1 1 0 0 1 1 1 0 0 0 1 0 0 0 0 1
SXF†	Set external flag	1	1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 1
TBLR	Table read	1	0 1 0 1 1 0 0 0 I ◄——D——►
TBLW	Table write	1	0 1 0 1 1 0 0 1 I ◄——D——►

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

Table 4-4. Instruction Set Summary (Concluded)

CONTROL INSTRUCTIONS																	
MNEMONIC	DESCRIPTION	NO. WORDS	INSTRUCTION BIT CODE														
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIT†	Test bit	1	1	0	0	1	◄	B	►		I	◄		D			►	
BITT†	Test bit specified by T register	1	0	1	0	1	0	1	1	1	I	◄		D			►	
CNFD†	Configure block as data memory	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	0	0
CNFP†	Configure block as program memory	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	0	1
DINT	Disable interrupt	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	1
EINT	Enable interrupt	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0
IDLE†	Idle until interrupt	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	1	1
LST	Load status register STO	1	0	1	0	1	0	0	0	0	I	◄		D			►	
LST1†	Load status register ST1	1	0	1	0	1	0	0	0	1	I	◄		D			►	
NOP	No operation	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0
POP	Pop top of stack to low accumulator	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	0	1
POPD†	Pop top of stack to data memory	1	0	1	1	1	1	0	1	0	I	◄		D			►	
PSHD†	Push data memory value onto stack	1	0	1	0	1	0	1	0	0	I	◄		D			►	
PUSH	Push low accumulator onto stack	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	0	0
RC‡	Reset carry bit	1	1	1	0	0	1	1	1	0	0	0	1	1	0	0	0	0
RHM‡	Reset hold mode	1	1	1	0	0	1	1	1	0	0	0	1	1	1	0	0	0
ROVM	Reset overflow mode	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	1	0
RPT†	Repeat instruction as specified by data memory value	1	0	1	0	0	1	0	1	1	I	◄		D			►	
RPTK†	Repeat instruction as specified by immediate value	1	1	1	0	0	1	0	1	1	◄			K			►	
RSXM†	Reset sign-extension mode	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	1	0
RTC‡	Reset test/control flag	1	1	1	0	0	1	1	1	0	0	0	1	1	0	0	1	0
SC‡	Set carry bit	1	1	1	0	0	1	1	1	0	0	0	1	1	0	0	0	1
SHM‡	Set hold mode	1	1	1	0	0	1	1	1	0	0	0	1	1	1	0	0	1
SOVM	Set overflow mode	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	1	1
SST	Store status register STO	1	0	1	1	1	1	0	0	0	I	◄		D			►	
SST1†	Store status register ST1	1	0	1	1	1	1	0	0	1	I	◄		D			►	
SSXM†	Set sign-extension mode	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	1	1
STC‡	Set test/control flag	1	1	1	0	0	1	1	1	0	0	0	1	1	0	0	1	1
TRAP†	Software interrupt	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	1	0

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

4.3 Individual Instruction Descriptions

Each instruction in the instruction set summary is described in the following pages. Instructions are listed in alphabetical order. Information, such as assembler syntax, operands, operation, encoding, description, words, cycles, repeatability, and examples, is provided for each instruction. An example instruction is provided to familiarize the user with the special format used and explain its content. Refer to Section 4.1 for further information on memory addressing. Code examples using many of the instructions are given in Section 5 on Software Applications.

Direct Addressing: [<label>] EXAMPLE <dma>[,<shift>]
Indirect Addressing: [<label>] EXAMPLE {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<shift>[,<nextARP>]]
Immediate Addressing: [<label>] EXAMPLE [<constant>]

Each instruction begins with an assembler syntax expression. The optional comment field that concludes the syntax is not included in the syntax expression. Space(s) are required between each field (label, command, operand, and comment fields) as shown in the syntax. The syntax example illustrates both direct and indirect addressing, as well as immediate addressing in which the operand field includes <constant>.

Operands

$0 \leq$ dma ≤ 127
$0 \leq$ next ARP ≤ 7
$0 \leq$ constant ≤ 255

Operands may be constants or assembly-time expressions referring to memory, I/O and register addresses, pointers, shift counts, and a variety of constants. The operand values used in the example syntax are shown.

Execution

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

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

Affects C and OV; affected by OVM and SXM.

This section provides an example of the instruction operation sequence, describing the processing that takes place when the instruction is executed. Conditional effects of status register specified modes are also given. In addition, those bits in the status registers that are affected by the instruction are listed.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	0	0	0	Shift				0	Data Memory Address						
Indirect	0	0	0	0	Shift				1	See Section 4.1						
Short Immediate	1	1	0	0	1	0	1	1	8-Bit Constant							
Long Immediate	1	1	0	1	Shift				0	0	0	0	0	0	1	0
	16-Bit Constant															

Opcode examples are shown of both direct and indirect addressing or of the use of short or long immediate operands.

Description

This section decribes the instruction execution and its effect on the rest of the processor or memory contents. Any constraints on the operands imposed by the processor or the assembler are also described here. The description parallels and supplements the information given by the execution block.

Words

1

The digit specifies the number of memory words required to store the instruction and its extension words.

Cycles

Class I (1)

Instructions are classified according to the number of cycles required for each instruction. The single digit value enclosed in parentheses represents the cycle execution time of the instruction when not repeated. The instruction is assumed to be executed from on-chip ROM and use on-chip RAM. Repeatable multicycle instructions will execute in one cycle on all repeat executions. Refer to Appendix E for detailed information on instruction cycle timings.

Repeatability

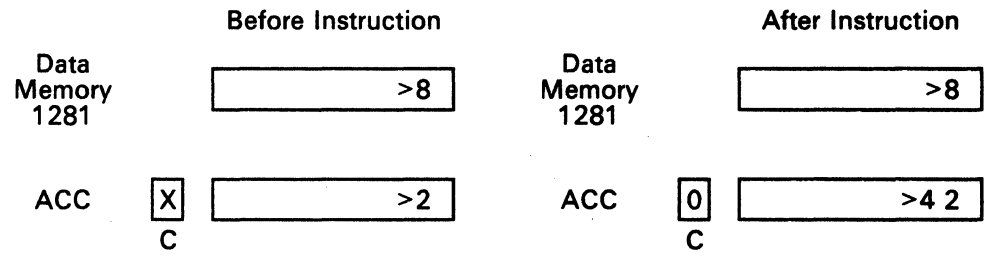
Category B

The repeatability of each instruction (using RPT or RPTK) is classified as to A, B, C, or X according to the following:

A	Instruction repeatable; useful if repeated.
B	Instruction repeatable; may be of some use if repeated.
C	Instruction repeatable; not useful to repeat the instruction.
X	Instruction not repeatable.

Example

```
ADD   DAT1,3     (DP = 10)
or
ADD   *,3        If current auxiliary register contains 1281.
```



The sample code presented in the above format shows the effect of the code on memory and/or registers.

Assembler Syntax [<label>] ABS

Operands None

Execution
(PC) + 1 → PC
|(ACC)| → ACC

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

Encoding

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

Description

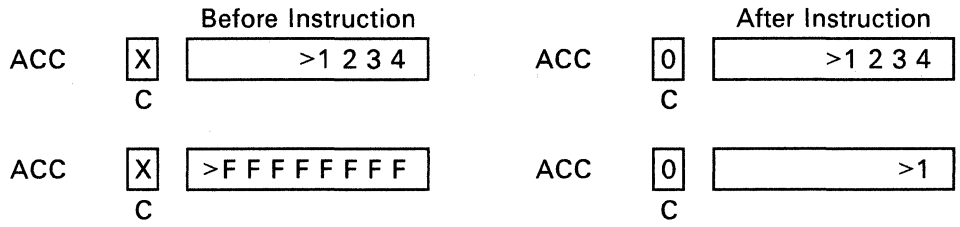
If the contents of the accumulator are greater than or equal to zero, the accumulator is unchanged by the execution of ABS. If the contents of the accumulator is less than zero, the accumulator is replaced by its two's-complement value.

Note that >80000000 is a special case. When the overflow mode is not set, the ABS of >80000000 is >80000000. When in the overflow mode, the ABS of >80000000 is >7FFFFFFF. In either case, the OV status bit is set. Also note that the carry bit C is always reset to zero by the execution of this instruction.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
ABS
```



Assembler Syntax

Direct Addressing: [<label>] ADD <dma>,[<shift>]
Indirect Addressing: [<label>] ADD {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<shift>[,<next ARP>]]

Operands

$0 \le$ dma ≤ 127
$0 \le$ next ARP ≤ 7
$0 \le$ shift ≤ 15 (defaults to 0)

Execution

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

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

Affects C and OV; affected by OVM and SXM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	0	0	0	Shift				0	Data Memory Address						
Indirect	0	0	0	0	Shift				1	See Section 4.1						

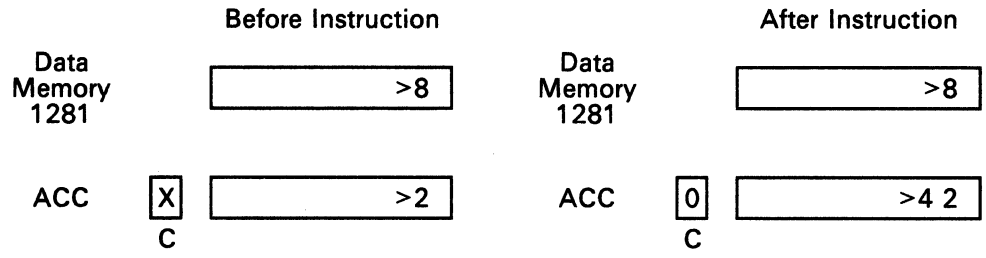
Description

The contents of the addressed data memory location 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.

Words 1
Cycles Class I (1)
Repeatability Category A

Example

```
ADD   DAT1,3     (DP = 10)
or
ADD   *,3        If current auxiliary register contains 1281.
```



Assembler Syntax

Direct Addressing: [<label>] ADDC <dma>
Indirect Addressing: [<label>] ADDC {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

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

Affects C and OV; affected by OVM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	0	0	1	1	0	Data Memory Address						
Indirect	0	1	0	0	0	0	1	1	1	See Section 4.1						

Description

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

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

Words 1
Cycles Class I (1)
Repeatability Category B

Example 1

```
ADDC  DAT5     (DP = 8)
or
ADDC  *        If current auxiliary register contains 1029.
```

	Before Instruction		After Instruction	
Data Memory 1029	>4		>4	
ACC	>1 3	1 (C)	>1 8	0 (C)

Example 2

```
ADDC  DAT5     (DP = 8)
or
ADDC  *        If current auxiliary register contains 1029.
```

	Before Instruction		After Instruction	
Data Memory 1029	>0		>0	
ACC	>F F F F F F F F	1 (C)	>0	1 (C)

Assembler Syntax

Direct Addressing: [<label>] ADDH <dma>
Indirect Addressing: [<label>] ADDH {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 1 → PC
(ACC) + [(dma) x 2^{16}] → ACC

Affects C and OV; affected by OVM.
Low-order bits of the ACC not affected.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	1	0	0	0	0	Data Memory Address						
Indirect	0	1	0	0	1	0	0	0	1	See Section 4.1						

Description

The contents of the addressed data memory location are added to the upper half of the accumulator (bits 31 through 16). Low-order bits are unaffected by ADDH. The carry bit C is set if the result of the addition generates a carry; otherwise, C is unaffected. The carry bit can only be set, not reset, by the ADDH instruction.

The ADDH instruction may be used in performing 32-bit arithmetic.

Words 1
Cycles Class I (1)
Repeatability Category B

Example

```
ADDH   DAT5    (DP = 8)
or
ADDH   *       If current auxiliary register contains 1029.
```

	Before Instruction		After Instruction	
Data Memory 1029	>4		>4	
ACC	1 (C)	>1 3	1 (C)	>4 0 0 1 3

Assembler Syntax [<label>] ADDK <constant>

Operands 0 ≤ constant ≤ 255

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

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

Encoding

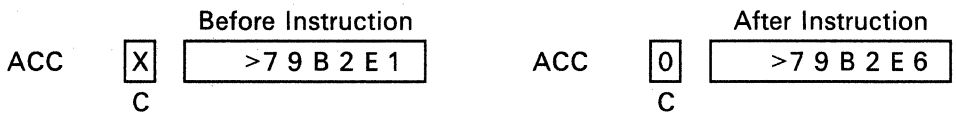
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	0	0	8-Bit Constant							

Description The 8-bit immediate value is added, right-justified, to the accumulator with the result replacing the accumulator contents. The immediate value is treated as an 8-bit positive number, regardless of the value of SXM.

Words 1
Cycles Class IV (1)
Repeatability Category X

Example

```
ADDK    >5
```



Assembler Syntax

Direct Addressing: [<label>] ADDS <dma>
Indirect Addressing: [<label>] ADDS {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 1 → PC
(ACC) + (dma) → ACC
(dma) is a 16-bit unsigned number.

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

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	1	0	0	1	0	Data Memory Address						
Indirect	0	1	0	0	1	0	0	1	1	See Section 4.1						

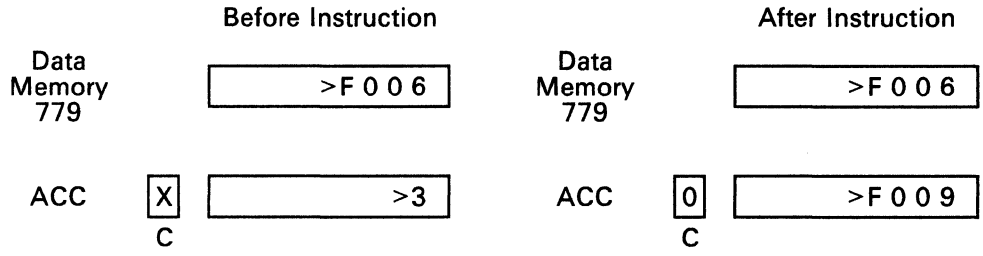
Description

The contents of the specified data memory location are added with sign-extension suppressed. The data is treated as a 16-bit unsigned number, regardless of SXM. The accumulator behaves as a signed number. Note that ADDS produces the same results as an ADD instruction with SXM = 0 and a shift count of 0.

Words 1
Cycles Class I (1)
Repeatability Category B

Example

```
ADDS    DAT11   (DP = 6)
or
ADDS    *       If current auxiliary register contains 779.
```



Assembler Syntax

Direct Addressing: [<label>] ADDT <dma>
Indirect Addressing: [<label>] ADDT {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 1 → PC
(ACC) + [(dma) x $2^{T\ register(3-0)}$] → (ACC)

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

Affects C and OV; affected by SXM and OVM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	1	0	1	0	0	Data Memory Address						
Indirect	0	1	0	0	1	0	1	0	1	See Section 4.1						

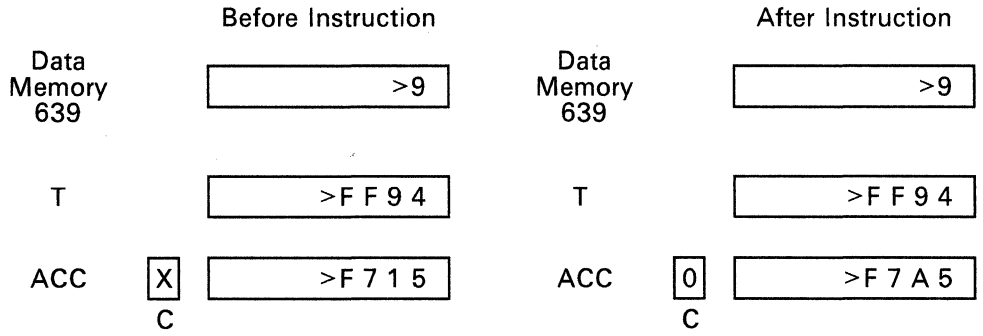
Description

The data memory value, left-shifted as defined by the four LSBs of the T register, is added to the accumulator, with the result replacing the accumulator contents. Sign extension on the data memory value is controlled by SXM.

Words 1
Cycles Class I (1)
Repeatability Category A

Example

```
ADDT   DAT127   (DP = 4)
or
ADDT   *        If current auxiliary register contains 639.
```



Assembler Syntax [<label>] ADLK <constant>[,<shift>]

Operands
16-bit constant
0 ≤ shift ≤ 15 (defaults to 0)

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

If SXM = 1:
Then -32768 ≤ constant ≤ 32767.
If SXM = 0:
Then 0 ≤ constant ≤ 65535.

Affects C and OV; affected by OVM and SXM.

Encoding

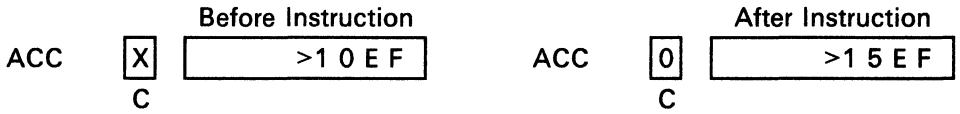
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Shift				0	0	0	0	0	0	1	0
16-bit Constant															

Description The 16-bit immediate value, left-shifted as specified, is added to the accumulator. The result replaces the accumulator contents. SXM determines whether the constant is treated as a signed two's-complement number or as an unsigned number. The shift count is optional and defaults to zero.

Words 2
Cycles Class V (2)
Repeatability Category X

Example

```
ADLK   5,8
```



Assembler Syntax [<label>] ADRK <constant>

Operands 0 ≤ constant ≤ 255

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

Encoding

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

Description The 8-bit immediate value is added, right-justified, to the currently selected auxiliary register 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.

Words 1
Cycles Class IV (1)
Repeatability Category X

Example

```
ADRK    >80     (ARP = 5)
```

	Before Instruction		After Instruction
AR5	>4 3 2 1	AR5	>4 3 A 1

Assembler Syntax

Direct Addressing: [<label>] AND <dma>
Indirect Addressing: [<label>] AND {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 1 → PC
(ACC(15-0)).AND.(dma) → ACC(15-0)
0 → ACC(31-16)

Not affected by SXM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	1	1	1	0	0	Data Memory Address						
Indirect	0	1	0	0	1	1	1	0	1	See Section 4.1						

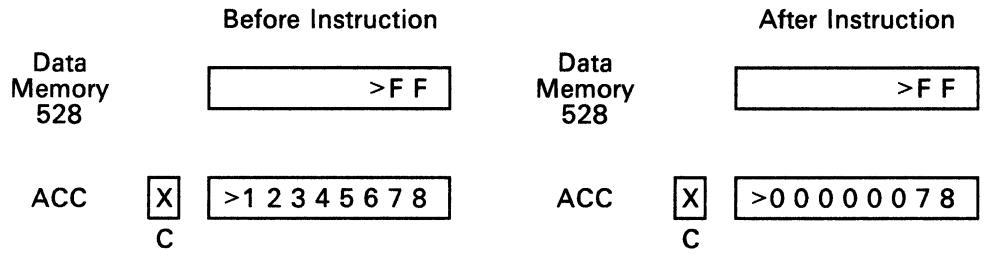
Description

The lower half of the accumulator is ANDed with the contents of the addressed data memory location. The upper half of the accumulator is ANDed with all zeroes. Therefore, the upper half of the accumulator is always zeroed by the AND instruction.

Words 1
Cycles Class I (1)
Repeatability Category B

Example

```
AND     DAT16    (DP = 4)
or
AND     *        If current auxiliary register contains 528.
```



Assembler Syntax [<label>] ANDK <constant>[,<shift>]

Operands
16-bit constant
0 ≤ shift ≤ 15 (defaults to 0)

Execution
(PC) + 2 → PC
(ACC(30-0)).AND.[(constant x 2^{shift})] → ACC(30-0)
0 → ACC(31) and all other bit positions not occupied by the shifted constant.

Not affected by SXM.

Encoding

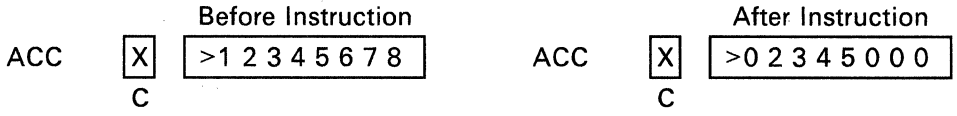
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Shift				0	0	0	0	0	1	0	0
16-bit Constant															

Description
The 16-bit immediate constant is left-shifted as specified and ANDed with the accumulator. The result is left in the accumulator. Low-order bits below and high-order bits above the shifted value are treated as zeroes, clearing the corresponding bits in the accumulator. Note that the accumulator's most-significant bit is always zeroed regardless of the shift-code value.

Words 2
Cycles Class V (2)
Repeatability Category X

Example

```
ANDK    >FFFF,12
```



Assembler Syntax [<label>] APAC

Operands None

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

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

Encoding

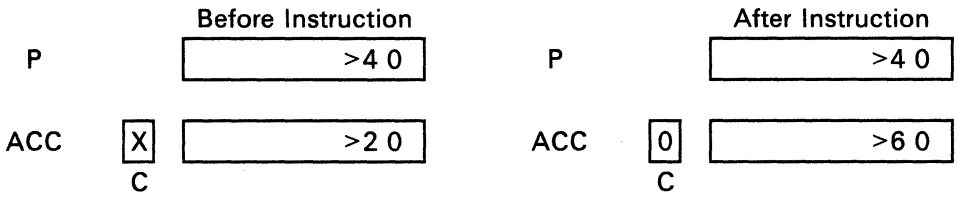
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	0	0	1	0	1	0	1

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 left in the accumulator. APAC is not affected by the SXM bit of the status register; the P register is always sign-extended. Note that APAC is a subset of the LTA, LTD, MAC, MACD, MPYA, and SQRA instructions.

Words 1
Cycles Class IV (1)
Repeatability Category B

Example

```
APAC    (PM = 0)
```



Assembler Syntax [<label>] B <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

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

Execution
pma → PC
Modify AR(ARP) and ARP as specified.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified, and control passes to the designated program memory address (pma). Note that no AR or ARP modification occurs if nothing is specified in those fields. Pma can be either a symbolic or a numeric address.

Words 2
Cycles ~~Class VIII (3)~~
Repeatability Category X

Example

```
B     PRG191      191 is loaded into the program counter, and
                  the program continues running from that
                  location.
```

Assembler Syntax [<label>] BACC

Operands None

Execution (ACC(15-0)) → PC

Encoding

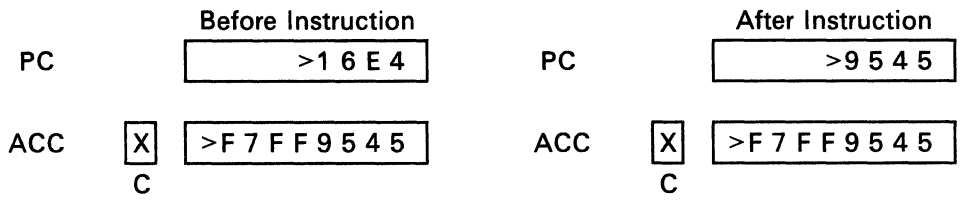
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	0	1	0	0	1	0	1

Description The branch uses the lower half of the accumulator (bits 15-0) for the branch address.

Words 1
Cycles Class VIII (3)
Repeatability Category X

Example

```
BACC
```



Assembler Syntax [<label>] BANZ <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

Operands
$0 \leq$ pma ≤ 65535
$0 \leq$ next ARP ≤ 7

Execution
If AR(ARP) $\neq 0$:
Then pma → PC;
Else (PC) + 2 → PC.
Modify AR(ARP) as specified.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	See Section 4.1						
Program Memory Address															

Description
Control is passed to the designated program memory address (pma) if the current auxiliary register is not equal to zero. Otherwise, control passes to the next instruction. The current auxiliary register and ARP are also modified as specified.

The current auxiliary register is either incremented or decremented from zero when the branch is not taken. Note that the AR modification defaults to *- (decrement current AR by one) when nothing is specified, making it compatible with the TMS32010. Pma can be either a symbolic or a numeric address.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example 1

```
BANZ    PRG35,*-
```

	Before Instruction		After Instruction
AR	>1	AR	>0
PC	>4 6	PC	>3 5
or			
AR	>0	AR	>F F F F
PC	>4 6	PC	>4 8

Example 2

```
BANZ    PRG64,*+
```

	Before Instruction		After Instruction
AR	>F F F F	AR	>0
PC	>1 1 7	PC	>6 4
or			
AR	>0	AR	>1
PC	>1 1 7	PC	>1 1 9

Note:

BANZ is designed for loop control using the auxiliary registers as loop counters. Using *0+ or *0- allows modification of the loop counter by a variable step size. Care must be exercised when doing this, however, because the auxiliary registers behave as modulo 65536 counters, and zero may be passed without being detected if AR0 > 1.

Assembler Syntax [<label>] BBNZ <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

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

Execution
If test/control (TC) status bit = 1:
Then pma → PC;
Else (PC) + 2 → PC.
Modify AR (ARP) and ARP as specified.

Affected by TC.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified. Control then passes to the designated program memory address if TC = 1. Otherwise, control passes to the next instruction. Note that no AR or ARP modification occurs if nothing is specified in those fields. Pma can be either a symbolic or a numeric address. Note that the TC bit may be affected by the BIT, BITT, CMPR, LST1, NORM, RTC, and STC instructions.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example

```
BBNZ    PRG650      If TC = 1, 650 is loaded into the program
                    counter; otherwise, the program counter
                    is incremented by 2.
```

Assembler Syntax [<label>] BBZ <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

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

Execution
If test/control (TC) status bit = 0:
Then pma → PC;
Else (PC) + 2 → PC.
Modify AR(ARP) and ARP as specified.

Affected by TC bit.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified. Control then passes to the designated program memory address if TC = 0. Otherwise, control passes to the next instruction. No AR or ARP modification occurs if nothing is specified in those fields. Pma can be either a symbolic or a numeric address. Note that the TC bit is affected by the BIT, BITT, CMPR, LST1, NORM, RTC, and STC instructions.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example

```
BBZ     PRG325    If TC = 0, 325 is loaded into the program
                  counter; otherwise, the program counter
                  is incremented by 2.
```

Assembler Syntax [<label>] BC <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

Operands
$0 \leq$ pma ≤ 65535
$0 \leq$ next ARP ≤ 7

Execution
If carry bit C = 1:
Then pma → PC;
Else (PC) + 2 → PC.
Modify AR(ARP) and ARP as specified.

Affected by C.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified. Control then passes to the designated program memory address if the carry bit C = 1. Otherwise, control passes to the next instruction. Note that no AR or ARP modification occurs if nothing is specified in those fields. Pma can be either a symbolic or a numeric address.

Note that the carry bit C is affected by all add, subtract, and accumulate instructions as well as the ABS, LST1, NEG, RC, SC, rotate, and shift instructions. The carry bit is not affected by execution of BC, BNC, or nonarithmetic instructions.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example

```
BC      PRG512      If the carry bit C = 1, 512 is loaded into
                    the program counter; otherwise, the program
                    counter is incremented by 2.
```

Assembler Syntax [<label>] BGEZ <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

Operands
$0 \leq$ pma ≤ 65535
$0 \leq$ next ARP ≤ 7

Execution
If (ACC) ≥ 0:
Then pma → PC;
Else (PC) + 2 → PC.
Modify AR (ARP) and ARP as specified.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified. Control then passes to the designated program memory address if the contents of the accumulator are greater than or equal to zero. Otherwise, control passes to the next instruction. Note that no AR or ARP modification occurs if nothing is specified in those fields. Pma can be either a symbolic or a numeric address.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example

```
BGEZ    PRG217      217 is loaded into the program counter if
                    the accumulator is greater than or equal
                    to zero.
```

Assembler Syntax [<label>] BGZ <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

Operands
$0 \leq$ pma ≤ 65535
$0 \leq$ next ARP ≤ 7

Execution
If (ACC) > 0:
Then pma → PC;
Else (PC) + 2 → PC.
Modify AR(ARP) and ARP as specified.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified. Control then passes to the designated program memory address if the contents of the accumulator are greater than zero. Otherwise, control passes to the next instruction. Note that no AR or ARP modification occurs if nothing is specified in those fields. Pma can be either a symbolic or a numeric address.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example

```
BGZ     PRG342    342 is loaded into the program counter if
                  the accumulator is greater than zero.
```

Assembler Syntax [<label>] BIOZ <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

Operands
$0 \leq$ pma ≤ 65535
$0 \leq$ next ARP ≤ 7

Execution
If $\overline{BIO} = 0$:
Then pma → PC;
Else (PC) + 2 → PC.
Modify AR(ARP) and ARP as specified.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified. Control then passes to the designated program memory address if the $\overline{BIO}$ pin is low. Otherwise, control passes to the next instruction. Note that no AR or ARP modification occurs if nothing is specified in those fields. Pma can be either a symbolic or a numeric address.

BIOZ in conjunction with the $\overline{BIO}$ pin can be used to test if a peripheral is ready to send or receive data. Polling the $\overline{BIO}$ pin using BIOZ may be preferable to an interrupt when executing time-critical loops.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example

```
BIOZ    PRG64    If the BIO- pin is active (low), then
                 a branch to location 64 occurs.
```

Assembler Syntax

Direct Addressing: [<label>] BIT <dma>,<bit code>
Indirect Addressing: [<label>] BIT {*|*+|*-|*0+|*0-|*BR0+|*BR0-},<bit code>[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ ARP ≤ 7
0 ≤ bit code ≤ 15

Execution

(PC) + 1 → PC
(dma bit at bit address (15-bit code)) → TC.

Affects TC.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	1	0	0	1	Bit Code				0	Data Memory Address						
Indirect	1	0	0	1	Bit Code				1	See Section 4.1						

Description

The BIT instruction copies the specified bit of the data memory value to the TC bit of status register ST1. Note that the BITT, CMPR, LST1, and NORM instructions also affect the TC bit in status register ST1. A bit code value is specified that corresponds to a certain bit address in the instruction, as given by the following table:

Bit Address	Bit Code 11 10 9 8
(LSB) 0	1 1 1 1
1	1 1 1 0
2	1 1 0 1
3	1 1 0 0
4	1 0 1 1
5	1 0 1 0
6	1 0 0 1
7	1 0 0 0
8	0 1 1 1
9	0 1 1 0
10	0 1 0 1
11	0 1 0 0
12	0 0 1 1
13	0 0 1 0
14	0 0 0 1
(MSB) 15	0 0 0 0

Words 1
Cycles Class I (1)
Repeatability Category C

Example

```
BIT     >0,>8     (DP = 488)
or
BIT     *,8       If current auxiliary register contains >F400.
```

	Before Instruction	After Instruction
Data Memory >F400	>7 E 9 8	>7 E 9 8
TC	>0	>1

Assembler Syntax

Direct Addressing: [<label>] BITT <dma>
Indirect Addressing: [<label>] BITT {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(dma bit at bit address (15-T register(3-0))) → TC

Affects TC.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	1	0	1	1	1	0	Data Memory Address						
Indirect	0	1	0	1	0	1	1	1	1	See Section 4.1						

Description

The BITT instruction copies the specified bit of the data memory value to the TC bit of status register ST1. Note that the BIT, CMPR, LST1, and NORM instructions also affect the TC bit in status register ST1. The bit address is specified by a bit code value contained in the LSBs of the T register, as given in the following table:

Bit Address	Bit Code 3 2 1 0
(LSB) 0	1 1 1 1
1	1 1 1 0
2	1 1 0 1
3	1 1 0 0
4	1 0 1 1
5	1 0 1 0
6	1 0 0 1
7	1 0 0 0
8	0 1 1 1
9	0 1 1 0
10	0 1 0 1
11	0 1 0 0
12	0 0 1 1
13	0 0 1 0
14	0 0 0 1
(MSB) 15	0 0 0 0

Words 1
Cycles Class I (1)
Repeatability Category C

Example

```
BITT    >0      Value in T register points to bit 14 of
                data word (DP = 240).
or
BITT    *       If current auxiliary register contains >7800.
```

	Before Instruction		After Instruction	
Data Memory >7800	>4 D C 8		Data Memory >7800	>4 D C 8
TR	>1		TR	>1
TC	>0		TC	>1

Assembler Syntax [<label>] BLEZ <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

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

Execution
If (ACC) ≤ 0:
Then pma → PC;
Else (PC) + 2 → PC.
Modify AR(ARP) and ARP as specified.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified. Control then passes to the designated program memory address if the contents of the accumulator are less than or equal to zero. Otherwise, control passes to the next instruction. Note that no AR or ARP modification occurs if nothing is specified in those fields. Pma can be either a symbolic or a numeric address.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example

```
BLEZ    PRG63    63 is loaded into the program counter if the
                 accumulator is less than or equal to zero.
```

Assembler Syntax

Direct Addressing: [<label>] BLKD <dma1>,<dma2>
Indirect Addressing: [<label>] BLKD <dma1>,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

$0 \leq$ dma1 ≤ 65535
$0 \leq$ dma2 ≤ 127
$0 \leq$ next ARP ≤ 7

Execution

(PC) + 2 → PC
(PFC) → MCS
(dma1) → PFC

While (repeat counter) $\neq 0$:
(dma1, addressed by PFC) → dma2,
Modify AR(ARP) and ARP as specified,
(PFC) + 1 → PFC,
(repeat counter) - 1 → repeat counter.

(dma1, addressed by PFC) → dma2
Modify AR(ARP) and ARP as specified.
(MCS) → PFC

Encoding

Direct

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

Indirect

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	1	1	See Section 4.1						
Data Memory Address 1															

Description

Consecutive memory words are moved from a source data memory block to a destination data memory block. The starting address (lowest) of the source block is defined by the second word of the instruction. The starting address of the destination block is defined by either the dma contained in the opcode (for direct addressing) or the current AR (for indirect addressing). In the indirect addressing mode, both the current AR and ARP may be modified in the usual manner. In the direct addressing mode, dma2 is used as the destination address for the block move but is not modified upon repeated executions of the instruction. Thus, the contents of memory at the dma2 address will be the same as the contents of memory at the last dma1 address in a repeat sequence.

RPT or RPTK must be used with this instruction, in the indirect addressing mode, 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. At the end of this instruction, the RPTC contains zero and, if using indirect addressing, AR(ARP) will be modified to contain the address after the end of the destination block. Note that the source and destination blocks do NOT have to be entirely on-chip or off-chip. However, BLKD cannot be used to transfer data from a memory-mapped register to any other location in data memory.

The PC points to the instruction following BLKD after execution. Interrupts are inhibited during a BLKD operation used with RPT or RPTK.

Words 2
Cycles Class XIII (4)
Repeatability Category A

Example

```
RPTK    2
BLKD    >F400,*+    If current auxiliary register contains 1030.
```

dma1

	Before Instruction		After Instruction
Data Memory 62464	>7 F 9 8	Data Memory 62464	>7 F 9 8
Data Memory 62465	>F F E 6	Data Memory 62465	>F F E 6
Data Memory 62466	>9 5 2 2	Data Memory 62466	>9 5 2 2

dma2

	Before Instruction		After Instruction
Data Memory 1030	>8 D E E	Data Memory 1030	>7 F 9 8
Data Memory 1031	>9 3 1 5	Data Memory 1031	>F F E 6
Data Memory 1032	>2 5 3 1	Data Memory 1032	>9 5 2 2

Assembler Syntax

Direct Addressing: [<label>] BLKP <pma>,<dma>
Indirect Addressing: [<label>] BLKP <pma>,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 2 → PC
(PFC) → MCS
(pma) → PFC

While (repeat counter) $\neq 0$:
(pma, addressed by PFC) → dma,
Modify AR(ARP) and ARP as specified,
(PFC) + 1 → PFC,
(repeat counter) - 1 → repeat counter.

(pma, addressed by PFC) → dma
Modify AR(ARP) and ARP as specified.
(MCS) → PFC

Encoding

Direct

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

Indirect

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	1	See Section 4.1						
Program Memory Address															

Description

Consecutive memory words are moved from a source program memory block to a destination data memory block. The starting address (lowest) of the source block is defined by the second word of the instruction. The starting address of the destination block is defined by either the dma contained in the opcode (for direct addressing) or the current AR (for indirect addressing). In the indirect addressing mode, both the ARP and the current AR may be modified in the usual manner. In the direct addressing mode, dma is used as the destination address for the block move but is not modified by repeated executions of the instruction. Thus, the contents of memory at the dma address will be the same as the contents of memory at the last pma address in a repeat sequence.

RPT or RPTK must be used with this 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. At the end of this instruction, the RPTC contains zero and, if using indirect addressing, AR(ARP) will be modified to contain the address after the end of the destination block. Note that source and destination blocks do NOT have to be entirely on-chip or off-chip.

The PC points to the instruction following BLKP after execution. Interrupts are inhibited during a BLKP operation.

If the MP/$\overline{\text{MC}}$ pin is low at the time of execution of this instruction and the program memory address used is less than 4096, an on-chip ROM location will be read.

Words 2
Cycles Class XIV (4)
Repeatability Category A

Example

```
RPTK    2
BLKP    65120,*+   If current auxiliary register contains 2048.
```

pma

	Before Instruction		After Instruction
Program Memory 65120	>A 0 8 9	Program Memory 65120	>A 0 8 9
Program Memory 65121	>2 D C E	Program Memory 65121	>2 D C E
Program Memory 65122	>3 A 9 F	Program Memory 65122	>3 A 9 F

dma

	Before Instruction		After Instruction
Data Memory 2048	>1 2 3 4	Data Memory 2048	>A 0 8 9
Data Memory 2049	>2 0 0 5	Data Memory 2049	>2 D C E
Data Memory 2050	>E 9 8 C	Data Memory 2050	>3 A 9 F

Assembler Syntax [<label>] BLZ <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

Operands
$0 \leq$ pma ≤ 65535
$0 \leq$ next ARP ≤ 7

Execution
If (ACC) < 0:
Then pma → PC;
Else (PC) + 2 → PC.
Modify AR(ARP) and ARP as specified.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified. Control then passes to the designated program memory address if the contents of the accumulator are less than zero. Otherwise, control passes to the next instruction. Note that no AR or ARP modification occurs when nothing is specified in those fields Pma can be either a symbolic or a numeric address.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example

```
BLZ     PRG481    481 is loaded into the program counter if the
                  accumulator is less than zero.
```

Assembler Syntax [<label>] BNC <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

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

Execution
If carry bit C = 0:
Then pma → PC;
Else (PC) + 2 → PC.
Modify AR(ARP) and ARP as specified.

Affected by C.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified. Control then passes to the designated program memory address if the carry bit C = 0. Otherwise, control passes to the next instruction. Note that no AR or ARP modification occurs when nothing is specified in those fields. Pma can be either a symbolic or a numeric address.

Note that the carry bit C is affected by all add, subtract, and accumulate instructions as well as the ABS, LST1, NEG, RC, SC, rotate, and shift instructions. The carry bit is not affected by execution of the BC, BNC, or nonarithmetic instructions.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example

```
BNC     PRG325      If the carry bit C = 0, 325 is loaded into
                    the program counter. Otherwise, the program
                    counter is incremented by 2.
```

Assembler Syntax [<label>] BNV <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

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

Execution
If overflow OV status bit = 0:
Then pma → PC;
Else (PC) + 2 → PC and 0 → OV.
Modify AR(ARP) and ARP as specified.

Affects OV; affected by OV.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified. Control then passes to the designated program address if the OV (overflow flag) is clear. Otherwise, the OV is cleared, and control passes to the next instruction. Note that no AR or ARP modification occurs if nothing is specified in those fields. Pma can be either a symbolic or a numeric address.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example

```
BNV     PRG315    315 is loaded into the program counter if
                  the overflow flag is clear.  OV is cleared.
```

Assembler Syntax [<label>] BNZ <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

Operands
$0 \leq$ pma ≤ 65535
$0 \leq$ next ARP ≤ 7

Execution
If (ACC) $\neq 0$:
Then pma → PC;
Else (PC) + 2 → PC.
Modify AR(ARP) and ARP as specified.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified. Control then passes to the designated program memory address if the contents of the accumulator are not equal to zero. Otherwise, control passes to the next instruction. Note that no AR or ARP modification occurs if nothing is specified in those fields. Pma can be either a symbolic or a numeric address.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example

```
BNZ     PRG320    320 is loaded into the program counter if
                  the accumulator does not equal zero.
```

Assembler Syntax [<label>] BV <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

Operands
$0 \leq$ pma ≤ 65535
$0 \leq$ next ARP ≤ 7

Execution
If overflow (OV) status bit = 1:
Then pma → PC and 0 → OV;
Else (PC) + 2 → PC.
Modify AR(ARP) and ARP as specified.

Affects OV; affected by OV.

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified, and the overflow flag is cleared. Control passes to the designated program memory address if the OV (overflow flag) is set. Otherwise, control passes to the next instruction. Note that no AR or ARP modification occurs if nothing is specified in those fields. Pma can be either a symbolic or a numeric address.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example

```
BV      PRG610    If an overflow has occurred since the over-
                  flow flag was last cleared, then 610 is
                  loaded in the program counter.
                  OV is cleared.
```

Assembler Syntax [<label>] BZ <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

Operands
$0 \leq$ pma ≤ 65535
$0 \leq$ next ARP ≤ 7

Execution
If (ACC) = 0:
Then pma → PC;
Else (PC) + 2 → PC.
Modify AR(ARP) and ARP as specified.

Encoding

15	14	13	12	11	10	9	8	7	6 5 4 3 2 1 0
1	1	1	1	0	1	1	0	1	See Section 4.1
Program Memory Address									

Description
The current auxiliary register and ARP are modified as specified. Control then passes to the designated program memory address if the contents of the accumulator are equal to zero. Otherwise, control passes to the next instruction. Note that no AR or ARP modification occurs if nothing is specified in those fields. Pma can be either a symbolic or a numeric address.

Words 2
Cycles Class VII (3)
Repeatability Category X

Example

```
BZ      PRG102    102 is loaded into the program counter if
                  the accumulator is equal to zero.
```

Assembler Syntax [<label>] CALA

Operands None

Execution
(PC) + 1 → TOS
(ACC(15-0)) → PC

Encoding

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

Description
The current program counter is incremented and pushed onto the top of the stack. Then, the contents of the lower half of the accumulator are loaded into the PC. The carry bit is unaffected by this instruction.

The CALA instruction is used to perform computed subroutine calls.

Words 1
Cycles Class VIII (3)
Repeatability Category X

Example

```
CALA
```

	Before Instruction		After Instruction
PC	>2 5	PC	>8 3
ACC	>8 3	ACC	>8 3
STACK	>3 2 >7 5 >8 4 >4 9 >0 >0 >0 >0	STACK	>2 6 >3 2 >7 5 >8 4 >4 9 >0 >0 >0

Assembler Syntax [<label>] CALL <pma>[,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]]

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

Execution
(PC) + 2 → TOS
pma → PC

Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	See Section 4.1						
Program Memory Address															

Description
The current auxiliary register and ARP are modified as specified, and the PC (program counter) is incremented by two and pushed onto the top of the stack. The specified program memory address is then loaded into the PC. Note that no AR or ARP modification occurs if nothing is specified in those fields. Pma can be either a symbolic or a numeric address.

Words 2
Cycles Class VIII (3)
Repeatability Category X

Example

```
CALL    PRG109
```

	Before Instruction		After Instruction
PC	>3 3	PC	>6 D
STACK	>7 1	STACK	>3 5
	>4 8		>7 1
	>1 6		>4 8
	>8 0		>1 6
	>0		>8 0
	>0		>0
	>0		>0
	>0		>0

Assembler Syntax [<label>] CMPL

Operands None

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

Encoding

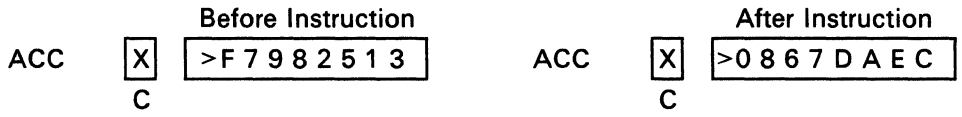
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	0	1	0	0	1	1	1

Description The contents of the accumulator are replaced with its logical inversion (one's complement).

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
CMPL
```



Assembler Syntax [<label>] CMPR <CM>

Operands $0 \leq CM \leq 3$

Execution (PC) + 1 → PC
Compare AR(ARP) to AR0, placing result in TC bit of status register ST1.

Affects TC.
Not affected by SXM; does not affect SXM.

Encoding

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

Description The CMPR instruction performs the following comparisons dependent on the value of CM:

If CM = 00, test if AR(ARP) = AR0
If CM = 01, test if AR(ARP) < AR0
If CM = 10, test if AR(ARP) > AR0
If CM = 11, test if AR(ARP) ≠ AR0

If the result of a test is true, a one is loaded into the TC status bit. Otherwise, TC is loaded with a zero. The auxiliary registers are treated as unsigned integers in the comparison.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
CMPR    2       (ARP = 4)
```

	Before Instruction		After Instruction
AR0	>F F F F	AR0	>F F F F
AR4	>7 F F F	AR4	>7 F F F
TC	>1	TC	>0

Assembler Syntax [<label>] CNFD

Operands None

Execution (PC) + 1 → PC
0 → RAM configuration control (CNF) status bit

Affects CNF.

Encoding

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

Description On-chip RAM block 0 is configured as data memory. The block is mapped to locations 512 through 767 in data memory. This instruction is the complement of the CNFP instruction and sets the CNF bit in status register ST1 to a zero. CNF is also loaded by the CNFP and LST1 instructions.

The next two instruction fetches immediately following a CNFD or CNFP instruction use the old value of CNF.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
CNFD        A zero is loaded into the CNF status bit, thus
            configuring block B0 as data memory (see
            memory maps in Section 3.2).
```

Assembler Syntax [<label>] CNFP

Operands None

Execution (PC) + 1 → PC
1 → RAM configuration control (CNF) status bit

Affects CNF.

Encoding

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

Description On-chip RAM block 0 is configured as program memory. The block is mapped to locations 65280 through 65535 in program memory space. This instruction is the complement of the CNFD instruction and sets the CNF bit in status register ST1 to a one. CNF is also loaded by the CNFD and LST1 instruction.

Configuring this block as program memory allows the use of the program counter as an address generator to access data from on-chip RAM. Used in conjunction with the repeat instructions, this allows two data memory locations to be addressed simultaneously, one from the auxiliary registers and one from the program counter. Instructions that take advantage of this feature are the MAC, MACD, BLKD, and BLKP instructions.

The next two instruction fetches immediately following a CNFD or CNFP instruction use the old value of CNF.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
CNFP        The CNF bit is set to a logic 1, thus config-
            uring block B0 as program memory (see memory
            maps in Section 3.2)
```

Assembler Syntax [<label>] DINT

Operands None

Execution (PC) + 1 → PC
1 → interrupt mode (INTM) status bit

Affects INTM.

Encoding

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

Description The interrupt mode (INTM) status bit is set to logic 1. Maskable interrupts are disabled immediately after the DINT instruction executes. Note that the LST instruction does not affect INTM.

The unmaskable interrupt, $\overline{RS}$, is not disabled by this instruction, and the interrupt mask register (IMR) is unaffected. Interrupts are also disabled by a reset.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
DINT            Maskable interrupts are disabled, and INTM is
                set to one.
```

Assembler Syntax

Direct Addressing: [<label>] DMOV <dma>
Indirect Addressing: [<label>] DMOV {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(dma) → dma + 1

Affected by CNF.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	1	0	1	1	0	0	Data Memory Address						
Indirect	0	1	0	1	0	1	1	0	1	See Section 4.1						

Description

The contents of the specified data memory address are copied into the contents of the next higher address. DMOV works only within the on-chip data RAM blocks B0, B1, and B2. It works within block B0 if it is configured as data memory, and the data move function is continuous across the boundaries of blocks B0 and B1; ie., it works for locations 512 to 1023. The data move function cannot be used on external data memory. If used on external data memory or memory-mapped registers, DMOV will read the specified memory location but will perform no other 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 and MACD instructions (see the LTD and MACD instructions for more information).

Words 1
Cycles Class I (1)
Repeatability Category A

Example

```
DMOV    DAT8
or
DMOV    *       If current auxiliary register contains 520.
```

	Before Instruction	After Instruction
Data Memory 520	>4 3	>4 3
Data Memory 521	>2	>4 3

Assembler Syntax [<label>] EINT

Operands None

Execution (PC) + 1 → PC
0 → interrupt-mode (INTM) status bit

Affects INTM.

Encoding

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

Description The interrupt-mode flag (INTM) in the status register is cleared to logic 0. Maskable interrupts are enabled after the instruction following EINT executes. This allows an interrupt service routine to re-enable interrupts and execute a RET instruction before any other pending interrupts are processed. Note that the LST instruction does not affect INTM. (See the DINT instruction for further information.)

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
EINT            Unmasked interrupts are enabled, and INTM is
                set to zero.
```

Assembler Syntax [<label>] FORT [<constant>]

Operands Constant = 0 or 1

Execution (PC) + 1 → PC
Constant → format (FO) status bit

Affects FO.

Encoding

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

Description The format (FO) status bit is loaded by the instruction with the LSB specified in the instruction. The FO bit is used to control the formatting of the transmit and receive shift registers of the serial port. If FO = 0, the registers are configured to receive/transmit 16-bit words. If FO = 1, the registers are configured to receive/transmit 8-bit bytes. FO is set to zero on a reset.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
FORT    1       The FO status bit is loaded with 1, making
                the bit length of the serial port 8 bits.
```

Assembler Syntax [<label>] IDLE

Operands None

Execution (PC) + 1 → PC
0 → interrupt mode (INTM) status bit

Affects INTM.

Encoding

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

Description The IDLE instruction forces the program being executed to wait until an interrupt or reset occurs. The PC is incremented only once, and the device remains in an idle state until interrupted. Note that INTM is set to zero in order for the maskable interrupts to be recognized. Execution of the IDLE instruction causes the device to enter the powerdown mode (see Section 3.4.6).

Words 1
Cycles Class XV (3)
Repeatability Category X

Example

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

Assembler Syntax

Direct Addressing: [<label>] IN <dma>,<PA>
Indirect Addressing: [<label>] IN {*|*+|*-|*0+|*0-|*BR0+|*BR0-},<PA>[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
0 ≤ port address PA ≤ 15

Execution

(PC) + 1 → PC
Port address → address bus A3-A0
0 → address bus A15-A4
Data bus D15-D0 → dma

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	1	0	0	0	Port Address				0	Data Memory Address						
Indirect	1	0	0	0	Port Address				1	See Section 4.1						

Description

The IN instruction reads a 16-bit value from one of the external I/O ports into the specified data memory location. 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 read.

Words 1
Cycles Class IX (2)
Repeatability Category A

Example

```
IN      STAT,PA5    Read in word from peripheral on port
                    address 5.  Store in data memory
                    location STAT.
```

or

```
LRLK    1,520       Load AR1 with decimal 520.
LARP    1           Load ARP with decimal 520.
IN      *-,PA1,0    Read in word from peripheral on port
                    address 1.  Store in data memory
                    location 520.  Decrement AR1 to 519.
                    Load the ARP with 0.
```

Assembler Syntax

Direct Addressing: [<label>] LAC <dma>,[<shift>]
Indirect Addressing: [<label>] LAC {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<shift>[,<next ARP>]]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
0 ≤ shift ≤ 15 (defaults to 0)

Execution

(PC) + 1 → PC
(dma) x 2^{shift} → ACC

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

Affected by SXM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	0	1	0	Shift				0	Data Memory Address						
Indirect	0	0	1	0	Shift				1	See Section 4.1						

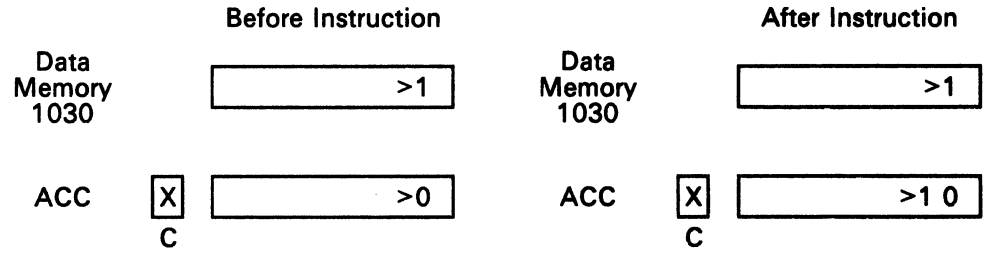
Description

The contents of the specified data memory address 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
Cycles Class I (1)
Repeatability Category C

Example

```
LAC     DAT6,4     (DP = 8)
or
LAC     *,4        If current auxiliary register contains 1030.
```



Assembler Syntax [<label>] LACK <constant>

Operands 0 ≤ constant ≤ 255

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

Not affected by SXM.

Encoding

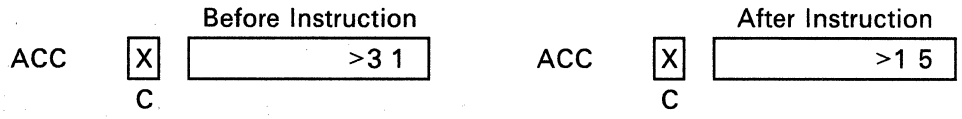
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	0	1	0	8-Bit Constant							

Description The 8-bit constant is loaded into the accumulator right-justified. The upper 24 bits of the accumulator are zeroed (i.e., sign extension is suppressed).

Words 1
Cycles Class IV (1)
Repeatability Category X

Example

```
LACK   >15
```



Assembler Syntax

Direct Addressing: [<label>] LACT <dma>
Indirect Addressing: [<label>] LACT {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 1 → PC
(dma) x $2^{T\ register(3-0)}$ → ACC

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

Affected by SXM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	0	0	1	0	0	Data Memory Address						
Indirect	0	1	0	0	0	0	1	0	1	See Section 4.1						

Description

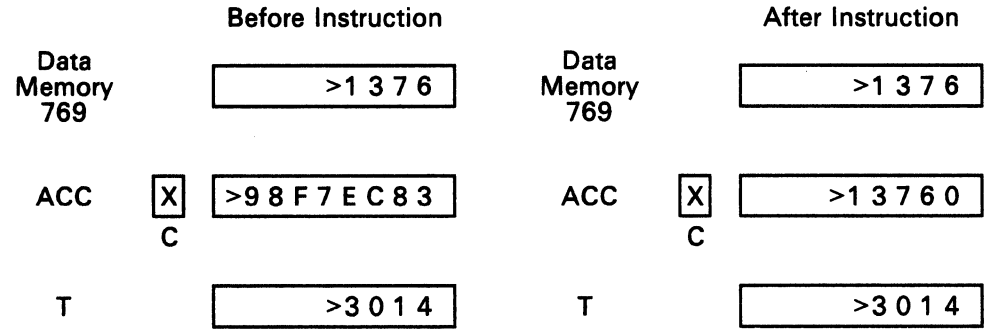
The LACT instruction loads the accumulator with a data memory value shifted left the number of places specified by the four LSBs of the T register. Using the T register's contents as a shift code provides a variable shift mechanism.

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 only be used when the magnitude of the exponent is four bits or less.

Words 1
Cycles Class I (1)
Repeatability Category C

Example

```
LACT    DAT1    (DP = 6)
or
LACT    *       If current auxiliary register contains 769.
```



Assembler Syntax [<label>] LALK <constant>[,<shift>]

Operands
16-bit constant
0 ≤ shift ≤ 15 (defaults to 0)

Execution
(PC) + 2 → PC
Constant x 2^{shift} → ACC

If SXM = 1:
Then -32768 ≤ constant ≤ 32767.
If SXM = 0:
Then 0 ≤ constant ≤ 65535.

Affected by SXM.

Encoding

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

Description
The left-shifted 16-bit immediate value is loaded into the accumulator. The shifted 16-bit constant is sign-extended if SXM = 1; otherwise, the high-order bits of the accumulator (past the shift) are set to zero. Note that the MSB of the accumulator can only be set if SXM = 1 and a negative number is loaded. The shift count is optional and defaults to zero.

Words 2
Cycles Class V (2)
Repeatability Category X

Example 1

```
LALK    >F794,8        (SXM=1)
```

	Before Instruction		After Instruction	
ACC	X: >1 2 3 4 5 6 7 8	C	ACC	X: >F F F 7 9 4 0 0	C

Example 2

```
LALK    >F794,8        (SXM=0)
```

	Before Instruction		After Instruction	
ACC	X: >1 2 3 4 5 6 7 8	C	ACC	X: >F 7 9 4 0 0	C

Assembler Syntax

Direct Addressing: [<label>] LAR <AR>,<dma>
Indirect Addressing: [<label>] LAR <AR>,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ auxiliary register AR ≤ 7
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(dma) → auxiliary register AR

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	0	1	1	0	Auxiliary Register			0	Data Memory Address						
Indirect	0	0	1	1	0	Auxiliary Register			1	See Section 4.1						

Description

The contents of the specified data memory address are loaded into the designated auxiliary register.

The LAR and SAR (store auxiliary register) instructions can be used to load and store the auxiliary registers during subroutine calls and interrupts. If an auxiliary register is not being used for indirect addressing, LAR and SAR enable the register to be used as an additional storage register, especially for swapping values between data memory locations without affecting the contents of the accumulator.

Words 1
Cycles Class II (1)
Repeatability Category C

Example 1

```
LAR    AR0,DAT10    (DP = 4)
```

	Before Instruction	After Instruction
Data Memory 522	>1 8	>1 8
AR0	>6	>1 8

Example 2

```
LARP  AR6
LAR   AR6,*-
```

	Before Instruction	After Instruction
Data Memory 617	>3 2	>3 2
AR6	>6 1 7	>3 2

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, AR6 is not decremented after the LAR instruction.

Assembler Syntax [<label>] LARK <AR>,<constant>

Operands
0 ≤ constant ≤ 255
0 ≤ auxiliary register AR ≤ 7

Execution
(PC) + 1 → PC
8-bit constant → auxiliary register AR

Encoding

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

Description
The 8-bit positive constant is loaded into the designated auxiliary register right-justified and zero-filled (i.e., sign-extension suppressed).

LARK is useful for loading an initial loop counter value into an auxiliary register for use with the BANZ instruction.

Words 1
Cycles Class IV (1)
Repeatability Category X

Example

```
LARK    AR0,>15
```

	Before Instruction		After Instruction
AR0	>0	AR0	>15

Assembler Syntax [<label>] LARP <constant>

Operands 0 ≤ constant ≤ 7

Execution

(PC) + 1 → PC
(ARP) → ARB
Constant → ARP

Affects ARP and ARB.

Encoding

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

Description

The auxiliary register pointer is loaded with the contents of the three LSBs of the instruction (a 3-bit constant identifying the desired auxiliary register). The old ARP is copied to the ARB field of status register ST1. ARP can also be modified by the LST, LST1, and MAR instructions, as well as any instruction that is used in the indirect addressing mode.

The LARP instruction is a subset of MAR; i.e., the opcode is the same as MAR in the indirect addressing mode. The following instruction has the same effect as LARP:

```
MAR  *,<constant>
```

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
LARP    1       Any succeeding instructions will use
                auxiliary register AR1 for indirect
                addressing.
```

Assembler Syntax

Direct Addressing: [<label>] LDP <dma>
Indirect Addressing: [<label>] LDP {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
Nine LSBs of (dma) → data page pointer register (DP) status bits

Affects DP.

Encoding

	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 Section 4.1						

Description

The nine LSBs of the contents of the addressed data memory location are loaded into the DP (data memory page pointer) register. The DP and 7-bit data memory address are concatenated to form 16-bit data memory addresses. The DP may also be loaded by the LST and LDPK instructions.

Words 1
Cycles Class II (1)
Repeatability Category C

Example

```
LDP     DAT127      (DP = 511)
or
LDP     *           If current auxiliary register contains 65535.
```

	Before Instruction		After Instruction	
Data Memory 65535	>F E D C		Data Memory 65535	>F E D C
DP	>1 F F		DP	>D C

Assembler Syntax [<label>] LDPK <constant>

Operands 0 ≤ constant ≤ 511

Execution (PC) + 1 → PC
Constant → data memory page pointer (DP) status bits

Affects DP.

Encoding

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

Description The DP (data memory page pointer) register is loaded with a 9-bit constant. The DP and 7-bit data memory address are concatenated to form 16-bit direct data memory addresses. DP ≥ 8 specifies external data memory. DP = 4 through 7 specifies on-chip RAM blocks B0 or B1. Block B2 is located in the upper 32 words of page 0. DP may also be loaded by the LST and LDP instructions.

Words 1
Cycles Class IV (1)
Repeatability Category X

Example

```
LDPK    64    The data page pointer is set to 64.
```

Assembler Syntax

Direct Addressing: [<label>] LPH <dma>
Indirect Addressing: [<label>] LPH {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(dma) → P register(31-16)

Encoding

	15	14	13	12	11	10	9	8	7	6 5 4 3 2 1 0
Direct	0	1	0	1	0	0	1	1	0	Data Memory Address
Indirect	0	1	0	1	0	0	1	1	1	See Section 4.1

Description

The P register high-order bits are loaded with the contents of data memory. The low-order P register bits are unaffected.

The LPH instruction is particularly useful for restoring the high-order bits of the P register after subroutine calls or interrupts.

Words 1
Cycles Class I (1)
Repeatability Category C

Example

```
LPH     DAT0     (DP = 4)
or
LPH     *        If current auxiliary register contains 512.
```

	Before Instruction		After Instruction
Data Memory 512	>F79C	Data Memory 512	>F79C
P	>30079844	P	>F79C9844

Assembler Syntax [<label>] LRLK <AR>,<constant>

Operands
0 ≤ auxiliary register ≤ 7
0 ≤ constant ≤ 65535

Execution
(PC) + 2 → PC
Constant → AR

Not affected by SXM; does not affect SXM.

Encoding

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

Description
The 16-bit immediate value is loaded into the auxiliary register specified by the AR field. The specified constant must be an unsigned integer, and its value is not affected by SXM.

Words 2
Cycles Class V (2)
Repeatability Category X

Example

```
LRLK    AR3,>3080
```

	Before Instruction		After Instruction
AR3	>7F80	AR3	>3080

Assembler Syntax

Direct Addressing: [<label>] LST <dma>
Indirect Addressing: [<label>] LST {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(dma) → status register ST0

Affects ARP, OV, OVM, and DP.
Does not affect INTM or ARB.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	1	0	0	0	0	0	Data Memory Address						
Indirect	0	1	0	1	0	0	0	0	1	See Section 4.1						

Description

Status register ST0 is loaded with the addressed data memory value. Note that the INTM (interrupt mode) bit is unaffected by LST. ARB is also unaffected 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.

The LST instruction is used to load status register ST0 after interrupts and subroutine calls. The ST0 contains the status bits: OV (overflow flag) bit, OVM (overflow mode) bit, INTM (interrupt mode) bit, ARP (auxiliary register pointer), and DP (data memory page pointer). These bits were stored (by the SST instruction) in the data memory word as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARP			OV	OVM	1	INTM	DP								

Words 1
Cycles Class II (1)
Repeatability Category C

Example 1

```
LARP    0
LST     *,1     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 even though a
                new ARP is loaded, the old ARP is not
                loaded into ARB.
```

Example 2

```
LST     >60     (DP = 0)
```

	Before Instruction	After Instruction
Data Memory 96	>2404	>2404
ST0	>6E00	>2604
ST1	>0580	>0580

Example 3

```
LARP    AR7     (AR7 = >3FF)
LST     *-
```

	Before Instruction	After Instruction
AR7	>3FF	>3FE
Data Memory 1023	>CE06	>CE06
ST0	>FC04	>CC06
ST1	>E780	>E780

Example 4

```
LARP    AR7     (AR7 = >3FF)
LST     *-,1
```

	Before Instruction	After Instruction
AR7	>3FF	>3FE
Data Memory 1023	>EE04	>EE04
ST0	>EE00	>EE04
ST1	>F780	>F780

Assembler Syntax

Direct Addressing: [<label>] LST1 <dma>
Indirect Addressing: [<label>] LST1 {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(dma) → status register ST1
(ARB) → ARP

Affects ARP, ARB, CNF, TC, SXM, C, HM, FSM, XF, FO, TXM, and PM.

Encoding

	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 Section 4.1						

Description

Status register ST1 is loaded with the data memory value. The bits of the data memory value, which are loaded into ARB, are also loaded into ARP to facilitate context switching. Note that if a next ARP value is specified via the indirect addressing mode, the specified value is ignored.

LST1 is used to load status bits after interrupts and subroutine calls. ST1 contains the status bits: ARB (auxiliary register pointer buffer), CNF (RAM configuration control) bit, TC (test/control) bit, SXM (sign-extension mode) bit, C (carry) bit, HM (hold mode) bit, FSM (frame synchronization mode) bit, XF (external flag) bit, FO (serial port format) bit, TXM (transmit mode) bit, and the PM (product register shift mode) bit. These bits are loaded into the status register from the data memory word as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARB			CNF	TC	SXM	C	1	1	HM	FSM	XF	FO	TXM	PM	

Words 1
Cycles Class II (1)
Repeatability Category C

Example 1

```
LARP    3
LST1    *-      The data memory word addressed by the
                contents of auxiliary register AR3
                replaces the status bits of status
                register ST1, and AR3 is decremented.
```

Example 2

```
LST1    >61     (DP = 0)
```

	Before Instruction	After Instruction
Data Memory 97	>0580	>0580
ST0	>AC00	>0C00
ST1	>0581	>0580

Example 3

```
LARP    AR7     (AR7 = >3FE)
LST1    *-
```

	Before Instruction		After Instruction
AR7	>3 F E	AR7	>3 F D
Data Memory 1022	>4 F 9 0	Data Memory 1022	>4 F 9 0
ST0	>F C 0 4	ST0	>5 C 0 4
ST1	>E 7 8 0	ST1	>4 F 9 0

Example 4

```
LARP    AR7     (AR7 = >3FE)
LST1    *-,1
```

	Before Instruction		After Instruction
AR7	>3 F E	AR7	>3 F D
Data Memory 1022	>6 1 9 0	Data Memory 1022	>6 1 9 0
ST0	>F E 0 4	ST0	>7 E 0 4
ST1	>0 5 9 3	ST1	>6 1 9 0

Assembler Syntax

Direct Addressing: [<label>] LT <dma>
Indirect Addressing: [<label>] LT {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(dma) → T register

Encoding

	15	14	13	12	11	10	9	8	7	6 5 4 3 2 1 0
Direct	0	0	1	1	1	1	0	0	0	Data Memory Address
Indirect	0	0	1	1	1	1	0	0	1	See Section 4.1

Description

The T register is loaded with the contents of the specified data memory location. The LT instruction may be used to load the T register in preparation for multiplication. See also the LTA, LTD, LTP, LTS, MPY, MPYA, MPYK, MPYS, and MPYU instructions.

Words 1
Cycles Class I (1)
Repeatability Category C

Example

```
LT    DAT24    (DP = 8)
or
LT    *        If current auxiliary register contains 1048.
```

	Before Instruction	After Instruction
Data Memory 1048	>6 2	>6 2
T	>3	>6 2

Assembler Syntax

Direct Addressing: [<label>] LTA <dma>
Indirect Addressing: [<label>] LTA {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(dma) → T register
(ACC) + (shifted P register) → ACC

Affects C and OV; affected by OVM and PM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	0	1	1	1	1	0	1	0	Data Memory Address						
Indirect	0	0	1	1	1	1	0	1	1	See Section 4.1						

Description

The T register is loaded with the contents of the specified data memory address. The contents of the product register, shifted as defined by the PM status bits, are added to the accumulator, with the result left in the accumulator.

The function of the LTA instruction is included in the LTD instruction.

Words 1
Cycles Class I (1)
Repeatability Category B

Example

```
LTA     DAT36     (DP = 6, PM = 0)
or
LTA     *         If current auxiliary register contains 804.
```

	Before Instruction		After Instruction	
Data Memory 804	>6 2		>6 2	
T	>3		>6 2	
P	>F		>F	
ACC	>5	X (C)	>1 4	0 (C)

Assembler Syntax

Direct Addressing: [<label>] LTD <dma>
Indirect Addressing: [<label>] LTD {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(dma) → T register
(dma) → dma + 1
(ACC) + (shifted P register) → ACC

Affects C and OV; affected by OVM and PM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	0	1	1	1	1	1	1	0	Data Memory Address						
Indirect	0	0	1	1	1	1	1	1	1	See Section 4.1						

Description

The T register is loaded with the contents of the specified data memory address. The contents of the P register, shifted as defined by the PM status bits, are added to the accumulator, and the result is placed in the accumulator. The contents of the specified data memory address are also copied to the next higher data memory address. This instruction is valid for blocks B1 and B2, and is also valid for block B0 if block B0 is configured as data memory. The data move function is continuous across the boundary of blocks B0 and B1, but cannot be used with external data memory or memory-mapped registers. This function is described under the instruction DMOV. Note that if used with external data memory, the function of LTD is identical to that of LTA.

Words 1
Cycles Class I (1)
Repeatability Category B

Example

```
LTD     DAT126      (DP = 7, PM = 0)
or
LTD     *           If current auxiliary register contains 1022.
```

	Before Instruction		After Instruction	
Data Memory 1022		>6 2	Data Memory 1022	>6 2
Data Memory 1023		>0	Data Memory 1023	>6 2
T		>3	T	>6 2
P		>F	P	>F
ACC	X	>5	ACC	0 >1 4
	C			C

Assembler Syntax

Direct Addressing: [<label>] LTP <dma>
Indirect Addressing: [<label>] LTP {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(dma) → T register
(shifted P register) → ACC

Affected by PM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	0	1	1	1	1	1	0	0	Data Memory Address						
Indirect	0	0	1	1	1	1	1	0	1	See Section 4.1						

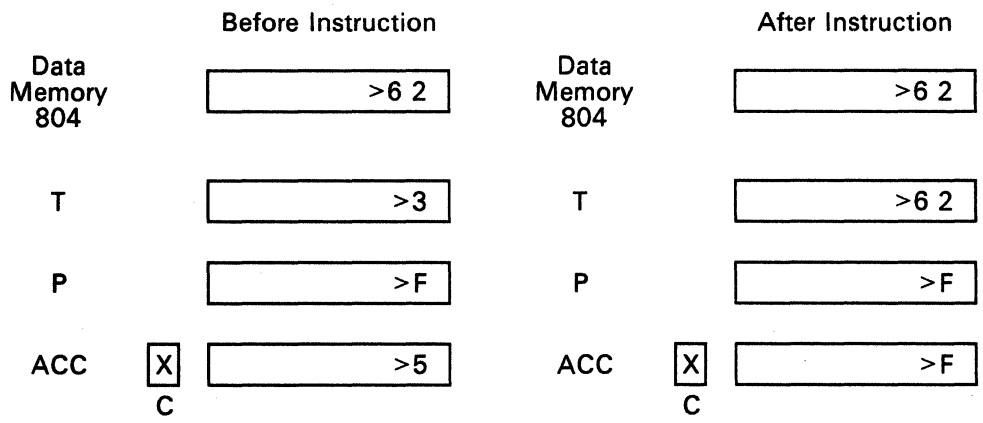
Description

The T register is loaded with the contents of the addressed data memory location, and the product register is stored in the accumulator. The shift at the output of the product register is controlled by the PM status bits.

Words 1
Cycles Class I (1)
Repeatability Category C

Example

```
LTP     DAT36     (DP = 6, PM = 0)
or
LTP     *         If current auxiliary register contains 804.
```



Assembler Syntax

Direct Addressing: [<label>] LTS <dma>
Indirect Addressing: [<label>] LTS {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(dma) → T register
(ACC) - (shifted P register) → ACC

Affects C and OV; affected by PM and OVM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	1	1	0	1	1	0	Data Memory Address						
Indirect	0	1	0	1	1	0	1	1	1	See Section 4.1						

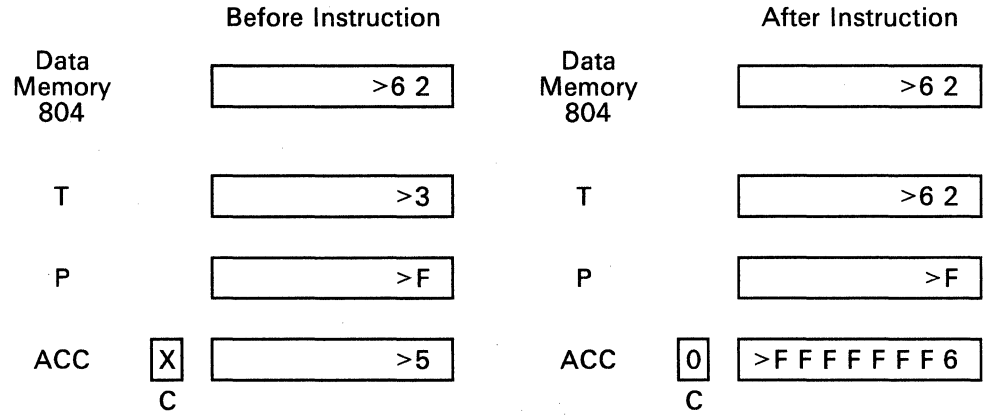
Description

The T register is loaded with the contents of the addressed data memory location. The contents of the product register, shifted as defined by the contents of the PM status bits, are subtracted from the accumulator. The result is left in the accumulator.

Words 1
Cycles Class I (1)
Repeatability Category B

Example

```
LTS     DAT36     (DP = 6, PM = 0)
or
LTS     *         If current auxiliary register contains 804.
```



Assembler Syntax

Direct Addressing: [<label>] MAC <pma>,<dma>
Indirect Addressing: [<label>] MAC <pma>,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 2 → PC
(PFC) → MCS
(pma) → PFC

While (repeat counter) ≠ 0:
(ACC) + (shifted P register) → ACC,
(dma) → T register,
(dma) x (pma, addressed by PFC) → P register,
Modify AR(ARP) and ARP as specified,
(PFC) + 1 → PFC,
(repeat counter) - 1 → repeat counter.

(ACC) + (shifted P register) → ACC
(dma) → T register
(dma) x (pma, addressed by PFC) → P register
Modify AR(ARP) and ARP as specified.
(MCS) → PFC

Affects C and OV; affected by OVM and PM.

Encoding

Direct

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

Indirect

15	14	13	12	11	10	9	8	7	6 5 4 3 2 1 0
0	1	0	1	1	1	0	1	1	See Section 4.1
Program Memory Address									

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 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. Note that the upper eight bits of the program memory address should be set to >FF to address B0 program RAM, and the upper six bits of dma should be set to 0 to address a location below 1024. When used in the direct addressing mode, the dma cannot be modified during repetition of the instruction.

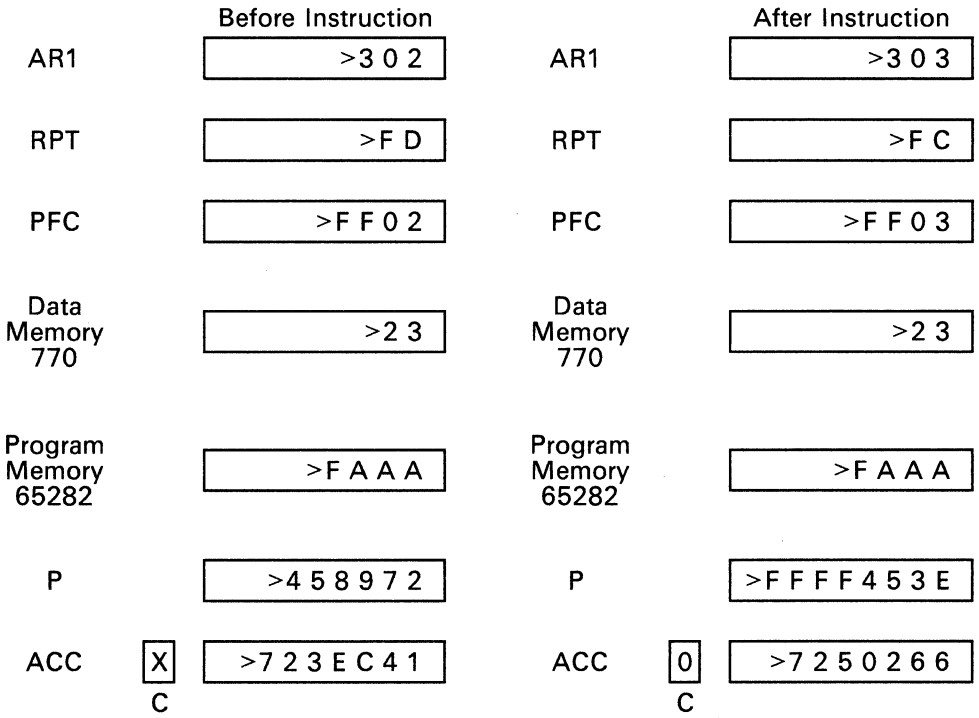
When the MAC instruction is repeated, the program memory address contained in the PFC is incremented by one during its operation. This enables accessing a series of operands in memory. MAC is useful for long sum-of-products operations, since MAC becomes a single-cycle instruction once the RPT pipeline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

Words 2
Cycles Class VI (4)
Repeatability Category A

Example

```
SPM   3          Select a "shift-right-by-6" mode on PR output.
CNFP             Configure block B0 as program memory (>FFXX).
LARP  1          Use AR1 to address block B1.
LRLK  1,768      Point to lowest location in RAM block B1.
RPTK  255        Compute 256 sum-of-product operations.
MAC   >FF00,*+   Multiply/accumulate and increment AR1.
```

The following example shows register and memory contents before and after the third step repeat loop:



Assembler Syntax

Direct Addressing: [<label>] MACD <pma>,<dma>
Indirect Addressing: [<label>] MACD <pma>,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 2 → PC
(PFC) → MCS
(pma) → PFC

While (repeat counter) ≠ 0:
(ACC) + (shifted P register) → ACC,
(dma) → T register,
(dma) x (pma, addressed by PFC) → P register,
(dma) → dma + 1,
Modify AR(ARP) and ARP as specified,
(PFC) + 1 → PFC,
(repeat counter) - 1 → repeat counter.

(ACC) + (shifted P register) → ACC
(dma) → T register
(dma) x (pma, addressed by PFC) → P register
(dma) → dma + 1
Modify AR(ARP) and ARP as specified.
(MCS) → PFC

Affects C and OV; affected by OVM and PM.

Encoding

Direct

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

Indirect

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	0	1	See Section 4.1						
Program Memory Address															

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 may be any nonreserved, on-chip or off-chip memory locations. 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).

If the program memory is block B0 of on-chip RAM, then the CNF must be set to one. Note that the upper eight bits of the program memory address should be set to >FF to address B0 program RAM, and the upper six bits of the effective 16-bit dma should be set to 0 to address a location below 1024. When used in the direct addressing mode, the dma cannot be modified during repetition of the instruction.

MACD functions in the same manner as MAC, with the addition of data move for block B0, B1, or B2. Otherwise, the effects are the same as for MAC. This feature makes MACD useful for applications such as convolution and transversal filtering.

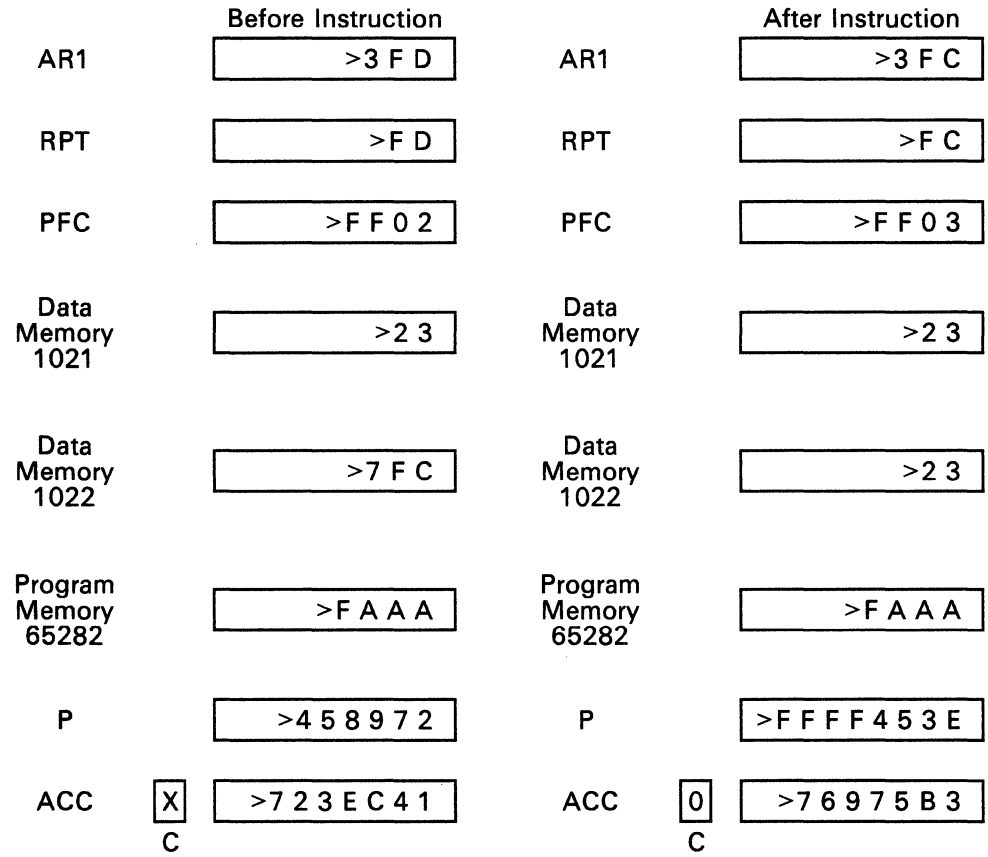
When the MACD instruction is repeated, the program memory address contained in the PFC is incremented by one during its operation. This enables accessing a series of operands in memory. When used with RPT or RPTK, MACD becomes a single-cycle instruction once the RPT pipeline is started.

Words 2
Cycles Class VI (4)
Repeatability Category A

Example

```
SPM     0         Select no shift mode on PR output.
SOVM              Set overflow mode.
CNFP              Configure block B0 as program memory (>FFXX).
LARP    3         Use AR3 to address block B1.
LRLK    3,1023    Point to highest location in RAM block B1.
RPTK    255       Compute 1 sample of a length-256 convolution.
MACD    >FF00,*-  Multiply/accumulate, shift data word in block
                  B1, and decrement AR3.
```

The following example shows register and memory contents before and after the third step repeat loop:



Note:

The data move function for MACD can only occur within on-chip data RAM blocks B0, B1, and B2.

Assembler Syntax

Direct Addressing: [<label>] MAR <dma>
Indirect Addressing: [<label>] MAR {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
Modifies ARP, AR(ARP) as specified by the indirect addressing field (acts as a NOP in direct addressing).

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	1	0	1	0	1	0	Data Memory Address						
Indirect	0	1	0	1	0	1	0	1	1	See Section 4.1						

Description

The MAR instruction acts as a no-operation instruction in the direct addressing mode. 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 only to modify the auxiliary registers or the ARP. The old ARP is copied to the ARB field of status register ST1. Note that any operation that MAR performs can also be performed with any instruction that supports indirect addressing. ARP may also be loaded by an LST instruction.

In the direct addressing mode, MAR is a NOP. Also, the instruction LARP is a subset of MAR (i.e., MAR *,4 performs the same function as LARP 4).

Words 1
Cycles Class IV (1)
Repeatability Category C

Example 1

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

	Before Instruction		After Instruction
ARP	0	ARP	1

Example 2

```
MAR   *-    Decrement current auxiliary register (in this
            case, AR1)
```

	Before Instruction		After Instruction
AR1	>3 5	AR1	>3 4

Example 3

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

	Before Instruction		After Instruction
AR1	>3 4	AR1	>3 5
ARP	1	ARP	5

Assembler Syntax

Direct Addressing: [<label>] MPY <dma>
Indirect Addressing: [<label>] MPY {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(T register) x (dma) → P register

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	0	1	1	1	0	0	0	0	Data Memory Address						
Indirect	0	0	1	1	1	0	0	0	1	See Section 4.1						

Description

The contents of the T register are multiplied by the contents of the addressed data memory location. The result is placed in the P register.

Words 1
Cycles Class I (1)
Repeatability Category C

Example

```
MPY     DAT13   (DP = 8)
or
MPY     *       If current auxiliary register contains 1037.
```

	Before Instruction	After Instruction
Data Memory 1037	>7	>7
T	>6	>6
P	>3 6	>2 A

Assembler Syntax

Direct Addressing: [<label>] MPYA <dma>
Indirect Addressing: [<label>] MPYA {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

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

Affects C and OV; affected by OVM and PM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	0	1	1	1	0	1	0	0	Data Memory Address						
Indirect	0	0	1	1	1	0	1	0	1	See Section 4.1						

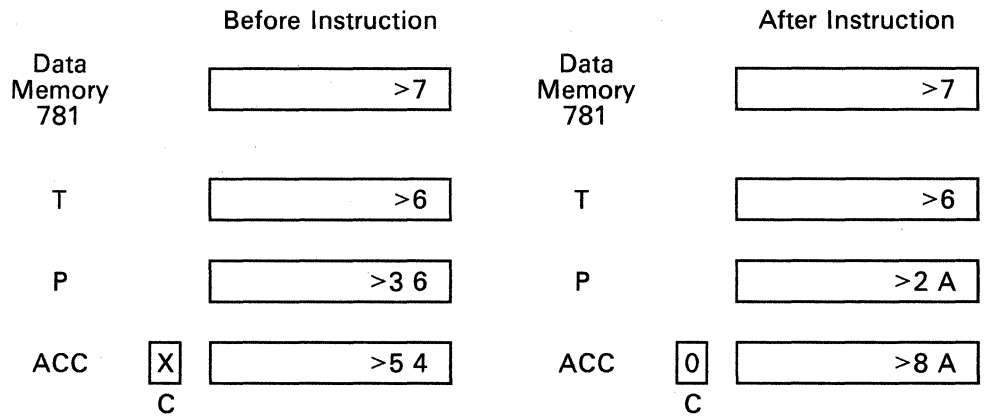
Description

The contents of the T register 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 added to the accumulator.

Words 1
Cycles Class I (1)
Repeatability Category A

Example

```
MPYA   DAT13   (DP = 6, PM = 0)
or
MPYA   *       If current auxiliary register contains 781.
```



Assembler Syntax [<label>] MPYK <constant>

Operands
$-4096 \leq$ constant ≤ 4095
$-2^{12} \leq$ constant $\leq 2^{12} - 1$

Execution
(PC) + 1 → PC
(T register) x constant → P register

Not affected by SXM.

Encoding

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

Description The contents of the T register are multiplied by the signed, 13-bit constant. The result is loaded into the P register. The immediate field is right-justified and sign-extended before multiplication, regardless of SXM.

Words 1
Cycles Class IV (1)
Repeatability Category X

Example

```
MPYK    -9
```

	Before Instruction		After Instruction
T	>7	T	>7
P	>2 A	P	>F F F F F F C 1

Assembler Syntax

Direct Addressing: [<label>] MPYS <dma>
Indirect Addressing: [<label>] MPYS {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(ACC) - (shifted P register) → ACC
(T register) x (dma) → P register

Affects C and OV; affected by OVM and PM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	0	1	1	1	0	1	1	0	Data Memory Address						
Indirect	0	0	1	1	1	0	1	1	1	See Section 4.1						

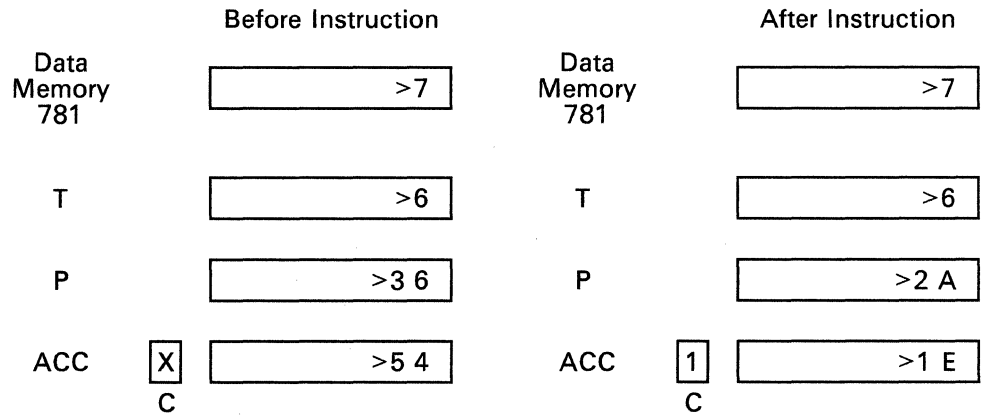
Description

The contents of the T register 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.

Words 1
Cycles Class I (1)
Repeatability Category A

Example

```
MPYS   DAT13   (DP = 6, PM = 0)
or
MPYS   *       If current auxiliary register contains 781.
```



Assembler Syntax

Direct Addressing: [<label>] MPYU <dma>
Indirect Addressing: [<label>] MPYU {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
Unsigned (T register) x unsigned (dma) → P register

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	1	1	0	0	1	1	1	1	0	Data Memory Address						
Indirect	1	1	0	0	1	1	1	1	1	See Section 4.1						

Description

The unsigned contents of the T register are multiplied by the unsigned contents of the addressed data memory location. The result is placed in the P register. Note that the multiplier acts as a 17 x 17-bit signed multiplier for this instruction, with the MSB of both operands forced to zero.

The shifter at the output of the P register will always invoke sign-extension on the P register when PM = 3 (right-shift by 6 mode). Therefore, this shift mode should not be used if unsigned products are desired.

The MPYU instruction is particularly useful for computing multiple-precision products, such as when multiplying two 32-bit numbers to yield a 64-bit product.

Words 1
Cycles Class I (1)
Repeatability Category C

Example

```
MPYU   DAT16   (DP = 4)
or
MPYU   *       If current auxiliary register contains 528.
```

	Before Instruction	After Instruction
Data Memory 528	>F F F F	>F F F F
T	>F F F F	>F F F F
P	>1	>F F F E 0 0 0 1

Assembler Syntax [<label>] NEG

Operands None

Execution
(PC) + 1 → PC
(ACC) x -1 → ACC

Affects C and OV; affected by OVM.

Encoding

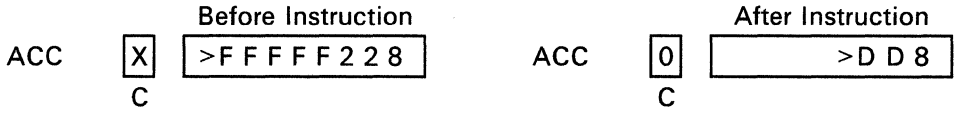
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	0	1	0	0	0	1	1

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

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
NEG
```



Assembler Syntax [<label>] NOP

Operands None

Execution (PC) + 1 → PC

Encoding

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

Description No operation is performed. The NOP instruction affects only the PC. NOP functions in the same manner as the MAR instruction in the direct addressing mode; NOP has the same opcode as MAR in the direct addressing mode with dma = 0.

The NOP instruction is useful as a pad or temporary instruction during program development.

Words 1
Cycles Class IV (1)
Repeatability Category B

Example

```
NOP
```

Assembler Syntax [<label>] NORM {*|*+|*-|*0+|*0-|*BR0+|*BR0-}

Operands None

Execution (PC) + 1 → PC

```
If (ACC) = 0:
 Then TC → 1;
 Else, if (ACC(31)).XOR.(ACC(30)) = 0:
  Then TC → 0,
   (ACC) x 2 → ACC,
   Modify AR(ARP) as specified;
  Else TC → 1.
```

Affects TC; affected by TC.

Encoding

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

Description The NORM instruction is provided for normalizing a signed number that is contained in the accumulator. Normalizing a fixed-point number separates it into a mantissa and an exponent. To do this, the magnitude of a sign-extended number must be found. ACC bit 31 is exclusive-ORed with ACC bit 30 to determine if bit 30 is part of the magnitude or part of the sign extension. If they are the same, they are both sign bits, and the accumulator is left-shifted to eliminate the extra sign bit.

The AR(ARP) is modified as specified to generate the magnitude of the exponent. It is assumed that AR(ARP) is initialized before the normalization begins. The default modification of the AR(ARP) is an increment, making it compatible with the TMS32020.

Multiple executions of the NORM instruction may be required to completely normalize a 32-bit number in the accumulator. Although using NORM with RPT or RPTK does not cause execution of NORM to "fall out" of the repeat loop automatically when the normalization is complete, no operation is performed for the remainder of the repeat loop. Note that NORM functions on both positive and negative two's-complement numbers.

Words 1
Cycles Class IV (1)
Repeatability Category A

Example 1 31-Bit Normalization:

```
        LARP    1       Use AR1 for exponent storage.
        LARK    1,0     Clear out exponent counter.
LOOP    NORM    *+      One bit is normalized.
        BBZ     LOOP    If TC = 0, magnitude is not found yet.
```

Example 2 15-Bit Normalization:

```
LARP    1       Use AR1 to store the exponent.
LARK    1,15    Initialize exponent counter.
RPTK    14      15-bit normalization is specified
                (yielding a 4-bit exponent and a 16-bit
                mantissa).
NORM    *-      NORM automatically stops shifting
                when the first significant magnitude
                bit is found, performing NOPs for the
                remainder of the repeat loop.
```

The first method is used to normalize a 32-bit number and yields a 5-bit exponent magnitude. The second method is used to normalize a 16-bit number and yields a 4-bit exponent magnitude. If it is known that the number requires only a small amount of normalization, the first method may be preferable to the second. This results because Example 1 runs only until normalization is complete. Example 2 always executes all 15 cycles of the repeat loop. Specifically, Example 1 is more efficient if the number requires five or less shifts. If the number requires six or more shifts, Example 2 is more efficient.

Note:

Source code compatibility with the TMS32020 allows the NORM instruction to 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. The user is therefore advised to replace the NORM instructions with NORM *+ when the default modification of increment is desired.

The resulting value in the auxiliary register will not be the real exponent of the number for all modification options. However, it can always be used to obtain the exponent.

Assembler Syntax

Direct Addressing: [<label>] OR <dma>
Indirect Addressing: [<label>] OR {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(ACC(15-0)) .OR.dma → ACC(15-0)
(ACC(31-16)) → ACC(31-16)

Not affected by SXM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	1	1	0	1	0	Data Memory Address						
Indirect	0	1	0	0	1	1	0	1	1	See Section 4.1						

Description

The low-order bits of the accumulator are ORed with the contents of the addressed data memory location. The high-order bits of the accumulator are ORed with all zeroes. Therefore, the upper half of the accumulator is unaffected by this instruction.

Words 1
Cycles Class I (1)
Repeatability Category B

Example

```
OR    DAT8    (DP = 8)
or
OR    *       Where current auxiliary register contains 1032.
```

	Before Instruction		After Instruction	
Data Memory 1032	>F000		>F000	
ACC	>00100002	X (C)	>0010F002	X (C)

Assembler Syntax [<label>] ORK <constant>[,<shift>]

Operands
16-bit constant
0 ≤ shift ≤ 15 (defaults to 0)

Execution
(PC) + 2 → PC
(ACC(30-0)).OR.[constant x 2^{shift}] → ACC(30-0)
(ACC(31)) → ACC(31)

Not affected by SXM.

Encoding

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

Description
The left-shifted 16-bit immediate constant is ORed with the accumulator. The result is left in the accumulator. Low-order bits below and high-order bits above the shifted value are treated as zeroes. The corresponding bits of the accumulator are unaffected. Note that the most-significant bit of the accumulator is not affected, regardless of the shift code value.

Words 2
Cycles Class V (2)
Repeatability Category X

Example

```
ORK     >FFFF,8
```

	Before Instruction		After Instruction	
ACC	X	>1 2 3 4 5 6 7 8	X	>1 2 F F F F 7 8
	C		C	

Assembler Syntax

Direct Addressing: [<label>] OUT <dma>,<PA>
Indirect Addressing: [<label>] OUT {*|*+|*-|*0+|*0-|*BR0+|*BR0-},<PA>[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
0 ≤ port address PA ≤ 15

Execution

(PC) + 1 → PC
Port address PA → address bus A3-A0
0 → address bus A15-A4
(dma) → data bus D15-D0

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	1	1	1	0	Port Address				0	Data Memory Address						
Indirect	1	1	1	0	Port Address				1	See Section 4.1						

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. OUT is a single-cycle instruction when in the PI/DI memory configuration (see Appendix E).

Words 1
Cycles Class X (1)
Repeatability Category A

Example

```
OUT     >78,7     (DP = 4) Output data word stored in data
                  memory location >78 to peripheral on
                  port address 7.
OUT     *,>F      Output data word referenced by current
                  auxiliary register to peripheral on port
                  address >F.
```

Assembler Syntax [<label>] PAC

Operands None

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

Affected by PM.

Encoding

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

Description The contents of the P register are loaded into the accumulator, shifted as specified by the PM status bits.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
PAC                (PM = 0)
```

	Before Instruction		After Instruction	
P	>1 4 4		P	>1 4 4
ACC	>2 3	X (C)	ACC	>1 4 4 X (C)

Assembler Syntax [<label>] POP

Operands None

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

Encoding

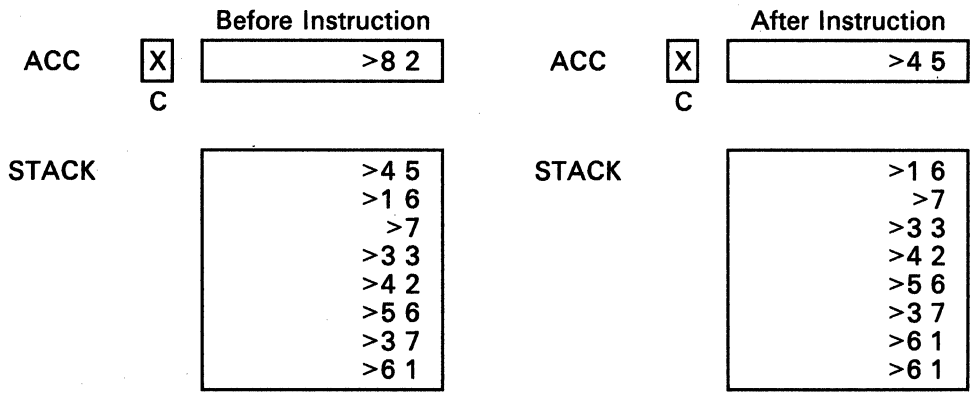
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	0	0	1	1	1	0	1

Description The contents of the top of the stack (TOS) are copied to the low accumulator, and the stack popped after the contents are copied. The upper half of the accumulator is set to all zeros. The hardware stack is a last-in, first-out stack 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 pops (due to POP, POPD, 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 Class IV (1)
Repeatability Category C

Example

```
POP
```



Assembler Syntax

Direct Addressing: [<label>] POPD <dma>
Indirect Addressing: [<label>] POPD {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(TOS) → dma
POP stack one level.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	1	1	1	0	1	0	0	Data Memory Address						
Indirect	0	1	1	1	1	0	1	0	1	See Section 4.1						

Description

The value from the top of the stack is transferred into the data memory location specified by the instruction. The values are also popped in the lower seven locations of the stack. The hardware stack is described in the previous instruction POP. The lowest stack location remains unaffected. No provision exists to check stack underflow.

Words 1
Cycles Class III (1)
Repeatability Category A

Example

```
POPD    DAT100    (DP = 8)
or
POPD    *         If current auxiliary register contains 1124.
```

	Before Instruction	After Instruction
Data Memory 1124	>5 5	>9 2
Stack	>9 2 >7 2 >8 >4 4 >8 1 >7 5 >3 2 >A A	>7 2 >8 >4 4 >8 1 >7 5 >3 2 >A A >A A

Assembler Syntax

Direct Addressing: [<label>] PSHD <dma>
Indirect Addressing: [<label>] PSHD {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(dma) → TOS
(PC) + 1 → PC
Push all stack locations down one level.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	1	0	1	0	0	0	Data Memory Address						
Indirect	0	1	0	1	0	1	0	0	1	See Section 4.1						

Description

The value from the data memory location specified by the instruction is transferred ~~to the top of the stack. The values are also pushed down in the lower seven~~ locations of the stack, as described in the next instruction PUSH. The lowest stack location is lost.

Words 1
Cycles Class I (1)
Repeatability Category A

Example

```
PSHD    DAT127      (DP = 3)
or
PSHD    *           If current auxiliary register contains 511.
```

	Before Instruction	After Instruction
Data Memory 511	>6 5	>6 5
Stack	>2	>6 5
	>3 3	>2
	>7 8	>3 3
	>9 9	>7 8
	>4 2	>9 9
	>5 0	>4 2
	>0	>5 0
	>0	>0

Assembler Syntax [<label>] PUSH

Operands None

Execution
(PC) + 1 → PC
Push all stack locations down one level.
(ACC(15-0)) → TOS

Encoding

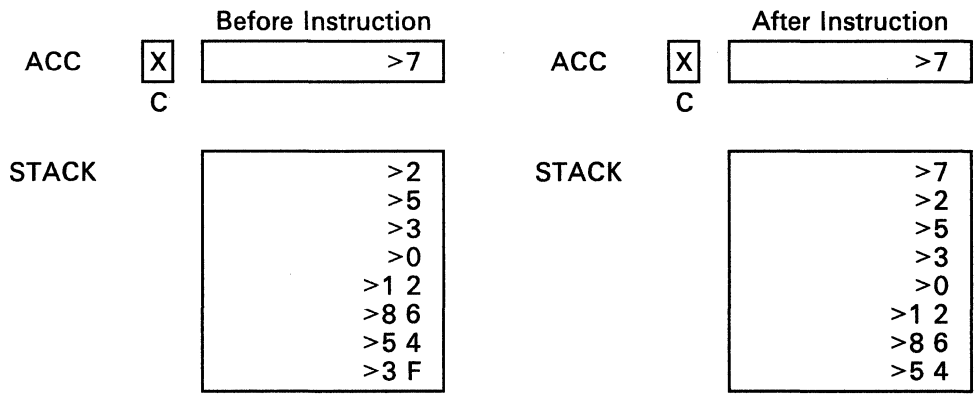
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	0	0	1	1	1	0	0

Description The contents of the lower half of the accumulator are copied onto the top of the hardware stack. The stack is pushed down before the accumulator value is copied. The hardware stack is a last-in, first-out stack with eight locations. If more than eight pushes (due to CALA, CALL, PSHD, PUSH, or TRAP instructions) occur before a pop, the first data values written will be lost with each succeeding push.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
PUSH
```



Assembler Syntax [<label>] RC

Operands None

Execution (PC) + 1 → PC
0 → carry bit C in status register ST1

Affects C.

Encoding

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

Description The carry bit C in status register ST1 is reset to logic zero. The carry bit may also be loaded directly by the LST1 and SC instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
RC              The carry bit C is reset to logic zero.
```

Assembler Syntax [<label>] RET

Operands None

Execution (TOS) → PC
Pop stack one level.

Encoding

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

Description The contents of the top stack register are copied into the program counter. The stack is then popped one level. RET is used with CALA and CALL for subroutines.

Words 1
Cycles Class VIII (3)
Repeatability Category X

Example

```
RET
```

	Before Instruction		After Instruction
PC	>9 6	PC	>3 7
STACK	>3 7	STACK	>4 5
	>4 5		>7 5
	>7 5		>2 1
	>2 1		>3 F
	>3 F		>4 5
	>4 5		>6 E
	>6 E		>6 E
	>6 E		>6 E

Assembler Syntax [<label>] RFSM

Operands None

Execution (PC) + 1 → PC
0 → FSM status bit in status register ST1

Affects FSM.

Encoding

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

Description The RFSM status bit resets the FSM status bit to logic zero. In this mode, external FSR pulses are not required to initiate the receive operation for each word received, but rather only one FSR pulse is required to initiate a "continuous mode" of operation. The same holds true for FSX when TXM = 0. After the first FSR/FSX pulse, these inputs are then in a "don't care" state. If TXM = 1, FSX is pulsed the first time DXR is loaded, but remains low thereafter. See Section 3.7 for further details on the operation of the serial port. FSM may also be loaded by the LST1 and SFSM instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
RFSM        FSM is reset, putting the serial port
            in a mode of operation where frame
            synchronization pulses are not required.
            This allows a continuous bit stream to
            be transmitted or received without FSX/FSR
            pulses every 8/16 bits.
```

Assembler Syntax [<label>] RHM

Operands None

Execution (PC) + 1 → PC
0 → HM status bit in status register ST1

Affects HM.

Encoding

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

Description The RHM instruction resets the HM status bit to logic zero. In this mode, the TMS320C25 is not halted during the assertion of $\overline{HOLD}$ when executing from on-chip program memory (either RAM or ROM), but instead places its external buses in the high-impedance state and continues execution until an external access must be made. External access can mean (in addition to the normal connotation) the following conditions:

MP/$\overline{MC}$	CNF	PC
0	0	PC 4096
0	1	4096 ≤ PC ≤ 65279
1	0	Any PC value (normal TMS32020-type hold mode)
1	1	PC ≤ 65279

HM can also be loaded by the LST1 and SHM instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
RHM             HM is reset, implementing the new
                TMS320C25 hold mode for on-chip program
                execution.
```

Assembler Syntax [<label>] ROL

Operands None

Execution
(PC) + 1 → PC
(ACC(31)) → C
(ACC(30-0)) → ACC(31-1)
(C, before ROL) → ACC(0)

Affects C.
Not affected by SXM.

Encoding

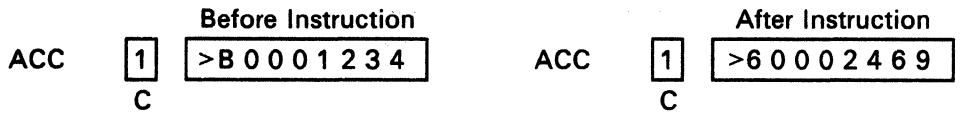
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	0	1	1	0	1	0	0

Description The ROL instruction rotates the accumulator left one bit. The MSB is shifted into the carry bit, and the value of the carry bit from before the execution of the instruction is shifted into the LSB.

Words 1
Cycles Class IV (1)
Repeatability Category A

Example

```
ROL
```



Assembler Syntax [<label>] ROR

Operands None

Execution

(PC) + 1 → PC
(ACC(0)) → C
(ACC(31-1)) → ACC(30-0)
(C, before ROR) → ACC(31)

Affects C.
Not affected by SXM.

Encoding

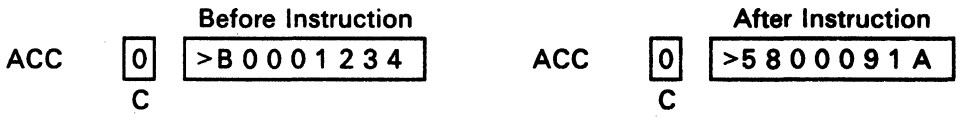
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	0	1	1	0	1	0	1

Description The ROR instruction rotates the accumulator right one bit. The LSB is shifted into the carry bit, and the value of the carry bit from before the execution of the instruction is shifted into the MSB.

Words 1
Cycles Class IV (1)
Repeatability Category A

Example

```
ROR
```



Assembler Syntax [<label>] ROVM

Operands None

Execution (PC) + 1 → PC
0 → OVM status bit in status register ST0

Affects OVM

Encoding

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

Description The OVM status bit is reset to logic zero, which disables the overflow mode. If an overflow occurs with OVM reset, the OV (overflow flag) is set, and the overflowed result is placed in the accumulator. OVM may also be loaded by the LST and SOVM instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
ROVM            The overflow mode bit OVM is reset,
                disabling the overflow mode on any
                subsequent arithmetic operations.
```

Assembler Syntax

Direct Addressing: [<label>] RPT <dma>
Indirect Addressing: [<label>] RPT {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 1 → PC
(dma(7-0)) → RPTC

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	1	0	1	1	0	Data Memory Address						
Indirect	0	1	0	0	1	0	1	1	1	See Section 4.1						

Description

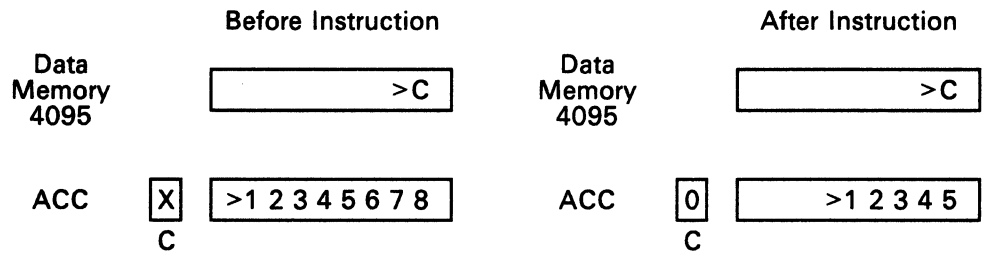
The eight LSBs of the addressed data memory value are loaded into the repeat counter (RPTC). This causes the following instruction to be executed one time more than the number loaded into the RPTC (provided that it is a repeatable instruction). Interrupts are masked out until the next instruction has been executed the specified number of times. (Interrupts cannot be allowed during the RPT/next instruction sequence, because the RPTC cannot be saved during a context switch.) The RPTC counter is cleared on a $\overline{RS}$.

RPT and RPTK are especially useful for repeating instructions, such as BLKP, BLKD, IN, MAC, MACD, NORM, OUT, TBLR, TBLW, and others.

Words 1
Cycles Class I (1)
Repeatability Category X

Example

```
RPT     DAT127      (DP = 31)
SFR
or
RPT      *          If current auxiliary register contains 4095.
SFR
```



Assembler Syntax [<label>] RPTK <constant>

Operands 0 ≤ constant ≤ 255

Execution
(PC) + 1 → PC
Constant → RPTC

Encoding

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

Description

The 8-bit immediate value is loaded into the RPTC (repeat counter). This causes the following instruction to be executed one time more than the number loaded into the RPTC (provided that it is a repeatable instruction). Interrupts are masked out until the next instruction has been executed the specified number of times. (Interrupts cannot be allowed during the RPT/next instruction sequence because the RPTC cannot be saved during a context switch.) The RPTC is cleared on a RS.

RPT and RPTK are especially useful for repeating instructions, such as BLKP, BLKD, IN, MAC, MACD, NORM, OUT, TBLR, TBLW, and others.

Words 1
Cycles Class IV (1)
Repeatability Category X

Example

```
LRLK    AR2,>200    Load AR2 with the address of X.
LARP    2
ZAC                 Clear the accumulator.
MPYK    0           Clear the P register.
RPTK    2           Repeat next instruction 3 times.
SQRA    *+          Compute X**2 + Y**2 + Z**2.
APAC
```

Assembler Syntax [<label>] RSXM

Operands None

Execution (PC) + 1 → PC
0 → SXM sign-extension mode status bit

Affects SXM.

Encoding

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

Description The RSXM instruction resets the SXM status bit to logic zero, which suppresses sign-extension on shifted data memory values for the following arithmetic instructions: ADD, ADDT, ADLK, LAC, LACT, LALK, SBLK, SUB, and SUBT.

The RSXM instruction affects the definition of the SFR instruction. SXM may also be loaded by the LST1 and SSXM instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
RSXM            SXM is reset, disabling sign-extension on
                subsequent instructions.
```

Assembler Syntax [<label>] RTC

Operands None

Execution (PC) + 1 → PC
0 → TC test/control flag in status register ST1

Affects TC.

Encoding

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

Description The TC (test/control) flag in status register ST1 is reset to logic zero. TC may also be loaded by the LST1 and STC instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
RTC         TC (test/control) flag is reset to logic zero.
```

Assembler Syntax [<label>] RTXM

Operands None

Execution
(PC) + 1 → PC
0 → TXM transmit mode status bit

Affects TXM mode bit.

Encoding

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

Description The RTXM instruction resets the TXM status bit, which configures the serial port transmit section in a mode where it is controlled by an FSX (external framing pulse). The transmit operation is started when an external FSX pulse is applied. TXM may also be loaded by the LST1 and STXM instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
RTXM          TXM is reset, configuring FSX as an input.
```

Assembler Syntax [<label>] RXF

Operands None

Execution (PC) + 1 → PC
0 → XF external flag pin and status bit

Affects XF.

Encoding

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

Description The XF pin and XF status bit in status register ST1 are reset to logic zero. XF may also be loaded by the LST1 and SXF instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
RXF             XF pin and status bit are reset to logic zero.
```

Assembler Syntax

Direct Addressing: [<label>] SACH <dma>,[<shift>]
Indirect Addressing: [<label>] SACH {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<shift>[,<next ARP>]]

Operands

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

Execution

(PC) + 1 → PC
16 MSBs of (ACC) x 2^{shift} → dma

Not affected by SXM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	1	0	1	Shift			0	Data Memory Address						
Indirect	0	1	1	0	1	Shift			1	See Section 4.1						

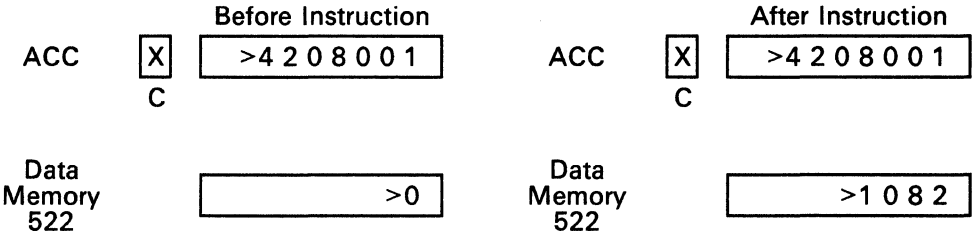
Description

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

Words 1
Cycles Class III (1)
Repeatability Category A

Example

```
SACH    DAT10,2    (DP = 4)
or
SACH    *,2        If current auxiliary register contains 522.
```



Assembler Syntax

Direct Addressing: [<label>] SACL <dma>,[<shift>]
Indirect Addressing: [<label>] SACL {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<shift>[,<next ARP>]]

Operands

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

Execution

(PC) + 1 → PC
16 LSBs of (ACC) x 2^{shift} → dma

Not affected by SXM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	1	0	0	Shift			0	Data Memory Address						
Indirect	0	1	1	0	0	Shift			1	See Section 4.1						

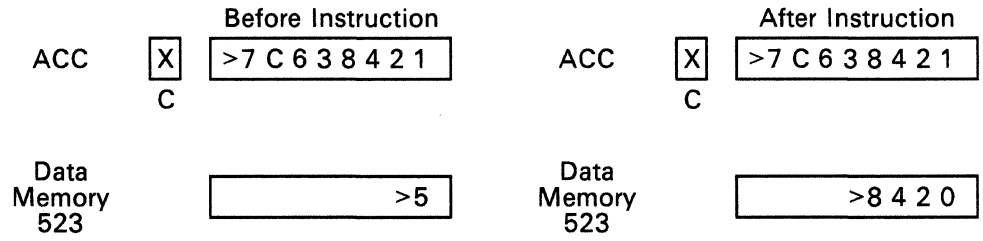
Description

The low-order bits of the accumulator, shifted left anywhere from 0 to 7 bits as specified by the shift code, are stored in data memory. The low-order bits are filled with zeros, and the high-order bits are lost. The accumulator itself is unaffected.

Words 1
Cycles Class III (1)
Repeatability Category A

Example

```
SACL    DAT11,5    (DP = 4)
or
SACL    *,5        If current auxiliary register contains 523.
```



Assembler Syntax

Direct Addressing: [<label>] SAR <AR>,<dma>
Indirect Addressing: [<label>] SAR <AR>,{*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ auxiliary register AR ≤ 7
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(AR) → dma

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	1	1	0	Auxiliary Register			0	Data Memory Address						
Indirect	0	1	1	1	0	Auxiliary Register			1	See Section 4.1						

Description

The contents of the designated auxiliary register are stored in the addressed data memory location.

When modifying the contents of the current auxiliary register in the indirect addressing mode, SAR ARn (when n = ARP) stores the value of the auxiliary register contents before it is incremented, decremented, or indexed by AR0.

Words 1
Cycles Class III (1)
Repeatability Category B

Example 1

```
SAR    AR0,DAT30    (DP = 6)
or
SAR    AR0,*        If current auxiliary register contains 798.
```

	Before Instruction		After Instruction
AR0	>3 7	AR0	>3 7
Data Memory 798	>1 8	Data Memory 798	>3 7

Example 2

```
LARP   AR0
SAR    AR0,*0+
```

AR0	>4 0 1	AR0	>8 0 2
Data Memory 1025	>0	Data Memory 1025	>4 0 1

Assembler Syntax [<label>] SBLK <constant>[,<shift>]

Operands
16-bit constant
0 ≤ shift ≤ 15 (defaults to 0)

Execution
(PC) + 2 → PC
(ACC) - [constant x 2^{shift}] → ACC

If SXM = 1:
Then -32768 ≤ constant ≤ 32767.
If SXM = 0:
Then 0 ≤ constant ≤ 65535.

Affects C and OV; affected by OVM and SXM.

Encoding

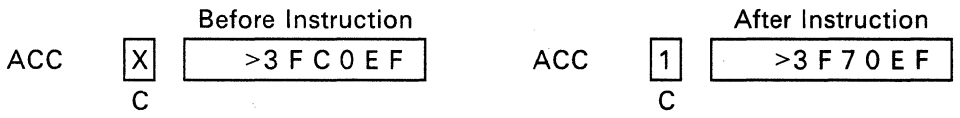
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Shift				0	0	0	0	0	0	1	1
16-Bit Constant															

Description
The immediate field of the instruction is subtracted from the accumulator. The result replaces the accumulator contents. SXM determines whether the constant is treated as a signed two's-complement number or as an unsigned number. The shift count is optional and defaults to zero.

Words 2
Cycles Class V (2)
Repeatability Category X

Example

```
SBLK    5,12
```



Assembler Syntax [<label>] SBRK <constant>

Operands 0 ≤ constant ≤ 255

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

Encoding

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

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 an 8-bit positive integer.

Words 1
Cycles Class IV (1)
Repeatability Category X

Example

```
SBRK   >FF     (ARP = 7)
```

	Before Instruction		After Instruction
AR7	>0	AR7	>F F 0 1

Assembler Syntax [<label>] SC

Operands None

Execution (PC) + 1 → PC
1 → carry bit C in status register ST1

Affects C.

Encoding

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

Description The carry bit C in status register ST1 is set to logic one. The carry bit may also be loaded directly by the LST1 and RC instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
SC              Carry bit C is set to logic one.
```

Assembler Syntax [<label>] SFL

Operands None

Execution

(PC) + 1 → PC
(ACC(31)) → C
(ACC(30-0)) → ACC(31-1)
0 → ACC(0)

Affects C.
Not affected by SXM bit.

Encoding

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

Description

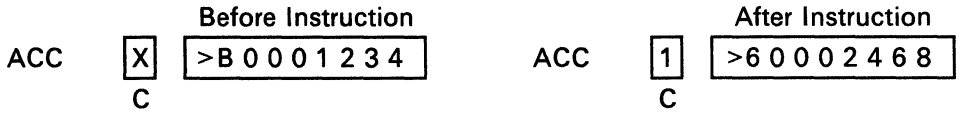
The SFL instruction shifts the entire accumulator left one bit.

The least-significant bit is filled with a zero, and the most-significant bit is shifted into the carry bit. Note that SFL, unlike SFR, is unaffected by SXM.

Words 1
Cycles Class IV (1)
Repeatability Category A

Example

```
SFL
```



Assembler Syntax [<label>] SFR

Operands None

Execution (PC) + 1 → PC

If SXM = 0:
Then (ACC(0)) → C
(ACC(31-1)) → ACC (30-0) and 0 → ACC(31).
If SXM = 1:
Then (ACC(0)) → C
(ACC(31-1)) → ACC(30-0) and (ACC(31)) → ACC(31).

Affects C.
Affected by SXM bit.

Encoding

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

Description The SFR instruction shifts the accumulator right one bit.

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

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

Words 1
Cycles Class IV (1)
Repeatability Category A

Example 1

```
SFR          (SXM=0)
```

	Before Instruction			After Instruction	
ACC	X	>B0001234	ACC	0	>5800091A
	C			C	

Example 2

```
SFR          (SXM=1)
```

ACC	X	>B0001234	ACC	0	>D800091A
	C			C	

Assembler Syntax [<label>] SFSM

Operands None

Execution (PC) + 1 → PC
1 → FSM status bit in status register ST1

Affects FSM.

Encoding

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

Description The SFSM instruction sets the FSM status bit to logic one. In this mode, an external FSR pulse is required for a receive operation, and an external FSX pulse is required if TXM = 0. If TXM = 1, FSX pulses are generated in the normal manner every time the transmit shift register XSR is loaded. See Section 3.7 for details on the operation of the serial port. FSM may also be loaded by the LST1 and RFSM instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
SFSM            FSM is set, putting the serial port in a
                mode of operation where frame synchronization
                pulses are required for each word to be
                transmitted or received.
```

Assembler Syntax [<label>] SHM

Operands None

Execution (PC) + 1 → PC
1 → HM status bit in status register ST1

Affects HM.

Encoding

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

Description The SHM instruction sets the HM status bit to logic one. In this mode, the TMS320C25 is halted in the normal manner whenever $\overline{\text{HOLD}}$ is asserted, regardless of the PC value or the state of the MP/$\overline{\text{MC}}$ pin. HM may also be loaded by the LST1 and RHM instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
SHM             HM is set, implementing the normal
                (TMS32020-type) hold mode of operation.
```

Assembler Syntax [<label>] SOVM

Operands None

Execution
(PC) + 1 → PC
1 → overflow mode (OVM) status bit

Affects OVM.

Encoding

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

Description The OVM status bit is set to logic one, which enables the overflow (saturation) mode. If an overflow occurs with OVM set, the overflow flag OV is set, and the accumulator is set to the largest representable 32-bit positive (>7FFFFFFF) or negative (>80000000) number according to the direction of overflow. OVM may also be loaded by the LST and ROVM instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
SOVM            The overflow mode bit OVM is set, enabling
                the overflow mode on any subsequent
                arithmetic operations.
```

Assembler Syntax [<label>] SPAC

Operands None

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

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

Encoding

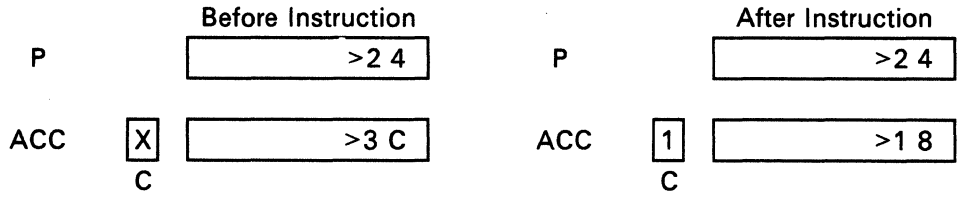
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	0	0	1	0	1	1	0

Description The contents of the P register, shifted as defined by the PM status bits, are subtracted from the contents of the accumulator. The result is stored in the accumulator. Note that SPAC is unaffected by SXM; the P register is always sign-extended. SPAC is a subset of LTS, MPYS, and SQRS.

Words 1
Cycles Class IV (1)
Repeatability Category B

Example

```
SPAC          (PM = 0)
```



A = A - P

Assembler Syntax

Direct Addressing: [<label>] SPH <dma>
Indirect Addressing: [<label>] SPH {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(PR shifter output (31-16)) → dma

Affected by PM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	1	1	1	1	0	1	0	Data Memory Address						
Indirect	0	1	1	1	1	1	0	1	1	See Section 4.1						

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 are 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 Class III (1)
Repeatability Category B

Example

```
SPH    DAT3      (DP = 4, PM = 2)
or
SPH    *         If current auxiliary register contains 515.
```

	Before Instruction		After Instruction
P	>FE079844	P	>FE079844
Data Memory 515	>4567	Data Memory 515	>E079

Assembler Syntax

Direct Addressing: [<label>] SPL <dma>
Indirect Addressing: [<label>] SPL {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(PR shifter output (15-0)) → dma

Affected by PM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	1	1	1	1	0	0	0	Data Memory Address						
Indirect	0	1	1	1	1	1	0	0	1	See Section 4.1						

Description

The low-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 are affected by this instruction. High-order bits are taken from the high P register when the right-shift by 6 mode is selected. Low-order bits are zero-filled when left-shifts are selected.

Words 1
Cycles Class III (1)
Repeatability Category B

Example

```
SPL     DAT3      (DP = 4, PM = 2)
or
SPL     *         If current auxiliary register contains 515.
```

	Before Instruction		After Instruction
P	>FE079844	P	>FE079844
Data Memory 515	>4567	Data Memory 515	>8440

Assembler Syntax [<label>] SPM <constant>

Operands 0 ≤ constant ≤ 3

Execution
(PC) + 1 → PC
Constant → product register shift mode (PM) status bits

Affects PM.

Encoding

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

Description The two low-order bits of the instruction word are copied into the PM field of status register ST1. The PM status bits control the P register output shifter. This shifter has the ability to shift the P register output either one or four bits to the left or six bits to the right, or to perform no shift. The bit combinations and their meanings are shown below.

PM	ACTION
00	No shift of multiplier output
01	Output left-shifted 1 place and zero-filled
10	Output left-shifted 4 places and zero-filled
11	Output right-shifted 6 places, sign-extended, and LSB bits lost.

The left-shifts allow the product to be justified for fractional arithmetic. The right-shift by six bits has been incorporated to implement up to 128 multiply-accumulate processes without the possibility of overflow occurring. PM may also be loaded by an LST1 instruction.

Words 1
Cycles Class IV (1)
Repeatability Category X

Example

```
SPM     3       Product register shift mode 3 is selected,
                causing all subsequent transfers from the
                product register to the ALU to be shifted
                to the right six places.
```

Assembler Syntax

Direct Addressing: [<label>] SQRA <dma>
Indirect Addressing: [<label>] SQRA {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

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

Affects C and OV; affected by PM and OVM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	0	1	1	1	0	0	1	0	Data Memory Address						
Indirect	0	0	1	1	1	0	0	1	1	See Section 4.1						

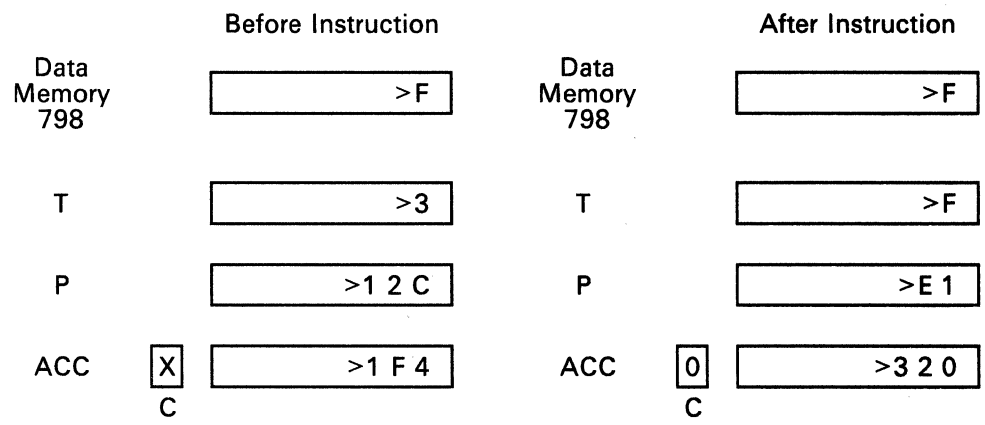
Description

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 the T register, squared, and stored in the P register.

Words 1
Cycles Class I (1)
Repeatability Category A

Example

```
SQRA    DAT30     (DP = 6, PM = 0)
or
SQRA    *         If current auxiliary register contains 798.
```



Assembler Syntax

Direct Addressing: [<label>] SQRS <dma>
Indirect Addressing: [<label>] SQRS {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(ACC) - (shifted P register) → ACC
(dma) → T register
(dma) x (dma) → P register

Affects C and OV; affected by PM and OVM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	1	1	0	1	0	0	Data Memory Address						
Indirect	0	1	0	1	1	0	1	0	1	See Section 4.1						

Description

The contents of the P register, shifted as defined by the PM status bits, are subtracted from the accumulator. The addressed data memory value is then loaded into the T register, squared, and stored into the P register.

Words 1
Cycles Class I (1)
Repeatability Category A

Example

```
SQRS    DAT9    (DP = 6, PM = 0)
or
SQRS    *       If current auxiliary register contains 777.
```

	Before Instruction		After Instruction	
Data Memory 777	>8		>8	
T	>1124		>8	
P	>190		>40	
ACC	>1450	X (C)	>12C0	1 (C)

Assembler Syntax

Direct Addressing: [<label>] SST <dma>
Indirect Addressing: [<label>] SST {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(status register ST0) → dma

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	1	1	1	0	0	0	0	Data Memory Address						
Indirect	0	1	1	1	1	0	0	0	1	See Section 4.1						

Description

Status register ST0 is stored in data memory.

In the direct addressing mode, status register ST0 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 specific 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.)

The SST instruction can be used to store status register ST0 after interrupts and subroutine calls. The ST0 contains the status bits: OV (overflow flag) bit, OVM (overflow mode) bit, INTM (interrupt mode) bit, ARP (auxiliary register pointer) bit, and DP (data memory page pointer) bit. The status bits are stored in the data memory word as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARP			OV	OVM	1	INTM	DP								

Note that SST * may be used to store status register ST0 anywhere in data memory, while SST in direct mode is forced to page 0.

Words 1
Cycles Class III (1)
Repeatability Category C

Example

```
SST     DAT96     (DP = don't care)
or
SST     *         If current auxiliary register contains 96.
```

	Before Instruction	After Instruction
Status Register ST0	>A408	>A408
Data Memory 96	>A	>A408

Assembler Syntax

Direct Addressing: [<label>] SST1 <dma>
Indirect Addressing: [<label>] SST1 {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 1 → PC
(status register ST1) → dma

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	1	1	1	0	0	1	0	Data Memory Address						
Indirect	0	1	1	1	1	0	0	1	1	See Section 4.1						

Description

Status register ST1 is stored in data memory.

In the direct addressing mode, status register ST1 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 specific location within that page is defined in the instruction. Note that the DP register is not physically modified. This allows the storage of the DP 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 LST1 instruction for more information.)

This instruction can be used to store status register ST1 after interrupts and subroutine calls. The ST1 contains the status bits: CNF (RAM configuration mode) bit, TC (test/control) bit, SXM (sign-extension mode) bit, C (carry) bit, HM (hold mode) bit, FSM (frame synchronization mode) bit, XF (external flag) bit, FO (serial port format), TXM (transmit mode) bit, ARB (auxiliary register pointer buffer), and PM (product register shift mode) bit. The status bits are stored in the data memory word as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARB			CNF	TC	SXM	C	1	1	HM	FSM	XF	FO	TXM	PM	

Note that SST1 * may be used to store status register ST1 anywhere in data memory, while SST1 in direct mode is forced to page 0.

Words 1
Cycles Class III (1)
Repeatability Category C

Example

```
SST1    DAT97     (DP = don't care)
SST1    *         If current auxiliary register contains 97.
```

	Before Instruction	After Instruction
Status Register ST1	>A 7 E 0	>A 7 E 0
Data Memory 97	>B	>A 7 E 0

Assembler Syntax [<label>] SSXM

Operands None

Execution (PC) + 1 → PC
1 → SXM status bit in status register ST1

Affects SXM.

Encoding

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

Description The SSXM instruction sets the SXM status bit to logic 1, which enables sign-extension on shifted data memory values for the following arithmetic instructions: ADD, ADDT, ADLK, LAC, LACT, LALK, SBLK, SUB, and SUBT.

SSXM also affects the definition of the SFR instruction. SXM may also be loaded by the LST1 and RSXM instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
SSXM            SXM is set, enabling sign extension on
                subsequent instructions.
```

Assembler Syntax [<label>] STC

Operands None

Execution (PC) + 1 → PC
1 → TC test/control flag in status register ST1

Affects TC.

Encoding

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

Description The TC (test/control) flag in status register ST1 is set to logic one. TC may also be loaded by the LST1 and RTC instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
STC             TC (test/control) flag is set to logic one.
```

Assembler Syntax [<label>] STXM

Operands None

Execution (PC) + 1 → PC
1 → TXM status bit in status register ST1

Affects TXM.

Encoding

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

Description The STXM instruction sets the TXM status bit to logic 1, which configures the serial port transmit section to a mode where the FSX pin behaves as an output. A pulse is produced on the FSX pin each time the DXR register is loaded internally. The transmission is initiated by the negative edge of this pulse. TXM may also be loaded by the LST1 and RTXM instructions. If the FSM status bit is a logic zero and serial port operation has already started, the FSX pin will be driven low if TXM = 1.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
STXM            TXM is set, configuring FSX as an output.
```

Assembler Syntax

Direct Addressing: [<label>] SUB <dma>,[<shift>]
Indirect Addressing: [<label>] SUB {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<shift>[,<next ARP>]]

Operands

$0 \leq$ dma ≤ 127
$0 \leq$ next ARP ≤ 7
$0 \leq$ shift ≤ 15 (defaults to 0)

Execution

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

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

Affects C and OV; affected by OVM and SXM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	0	0	1	Shift				0	Data Memory Address						
Indirect	0	0	0	1	Shift				1	See Section 4.1						

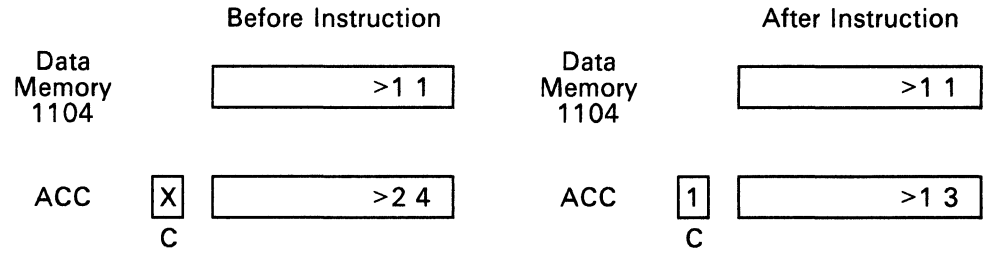
Description

The contents of the addressed data memory location are left-shifted and subtracted from 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.

Words 1
Cycles Class I (1)
Repeatability Category A

Example

```
SUB     DAT80     (DP = 8)
or
SUB     *         If current auxiliary register contains 1104.
```



Assembler Syntax

Direct Addressing: [<label>] SUBB <dma>
Indirect Addressing: [<label>] SUBB {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(ACC) - (dma) - ($\overline{C}$) → ACC

Affects C and OV; affected by OVM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	1	1	1	1	0	Data Memory Address						
Indirect	0	1	0	0	1	1	1	1	1	See Section 4.1						

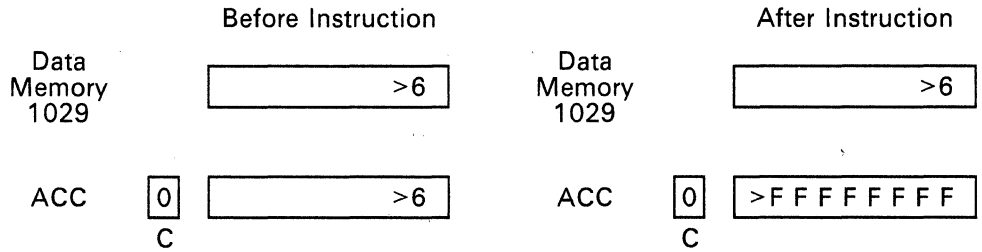
Description

The contents of the addressed data memory location and the value of the carry bit are subtracted from the accumulator. The carry bit is then affected in the normal manner (see Section 3.4.3).

Words 1
Cycles Class I (1)
Repeatability Category B

Example

```
SUBB    DAT5    (DP = 8)
or
SUBB    *       If current auxiliary register contains 1029.
```



In the above example, C is originally zeroed, presumably from the result of a previous subtract instruction that performed a borrow. Thus, 6 - 6 - ($\overline{0}$) = -1 was the effective operation performed, generating another borrow (and resetting carry again) in the process.

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

Assembler Syntax

Direct Addressing: [<label>] SUBC <dma>
Indirect Addressing: [<label>] SUBC {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

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

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	0	1	1	1	0	Data Memory Address						
Indirect	0	1	0	0	0	1	1	1	1	See Section 4.1						

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 high-order 16 bits of the accumulator. SUBC assumes the divisor and the dividend are both positive.

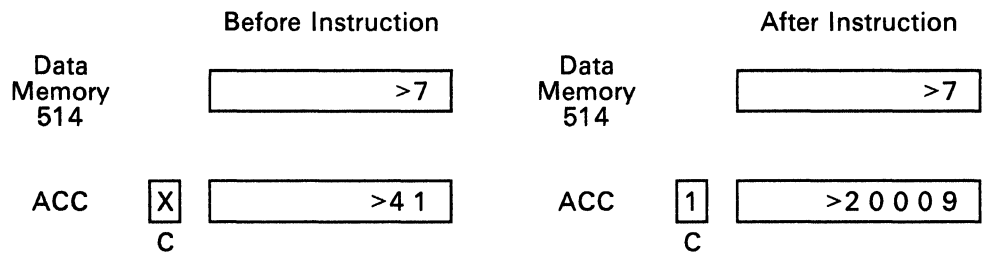
If the 16-bit dividend contains less than 16 significant bits, the dividend may be placed in the accumulator left-shifted by the number of leading non-significant zeroes. The number of executions of SUBC is reduced from 16 by that number. One leading zero is always significant.

Note that SUBC affects OV but is <u>not</u> affected by OVM, and therefore the accumulator does not saturate upon positive or negative overflows when executing this instruction.

Words 1
Cycles Class I (1)
Repeatability Category A

Example

```
RPTK    15
SUBC    DAT2    (DP = 4)
or
RPTK    15
SUBC    *       If current auxiliary register contains 514.
```



Assembler Syntax

Direct Addressing: [<label>] SUBH <dma>
Indirect Addressing: [<label>] SUBH {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 1 → PC
(ACC) - [(dma) x 2^{16}] → ACC

Affects C and OV; affected by OVM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	0	1	0	0	0	Data Memory Address						
Indirect	0	1	0	0	0	1	0	0	1	See Section 4.1						

Description

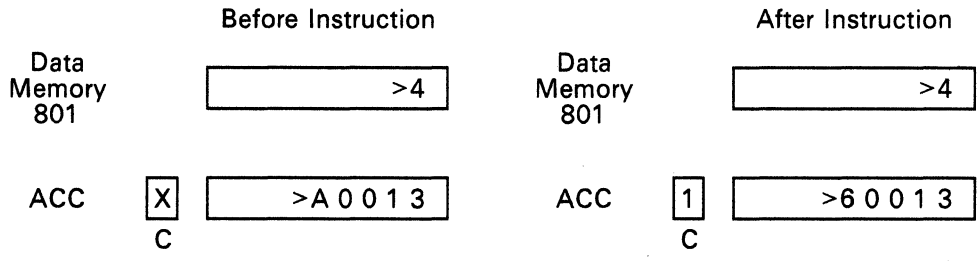
The contents of the addressed data memory location are subtracted from the upper 16 bits of the accumulator. The 16 low-order bits of the accumulator are unaffected. The result is stored in the accumulator. The carry bit C is reset if the result of the subtraction generates a borrow; otherwise, C is unaffected.

The SUBH instruction can be used for performing 32-bit arithmetic.

Words 1
Cycles Class I (1)
Repeatability Category B

Example

```
SUBH    DAT33     (DP = 6)
or
SUBH    *         If current auxiliary register contains 801.
```



Assembler Syntax [<label>] SUBK <constant>

Operands 0 ≤ constant ≤ 255

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

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

Encoding

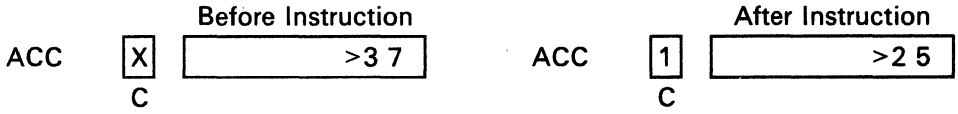
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	0	1	8-Bit Constant							

Description The 8-bit immediate value is subtracted, right-justified, from the accumulator with the result replacing the accumulator contents. The immediate value is treated as an 8-bit positive number, regardless of the value of SXM.

Words 1
Cycles Class IV (1)
Repeatability Category X

Example

```
SUBK    >12
```



Subtract from Low Accumulator with Sign-Extension Suppressed

Assembler Syntax

Direct Addressing: [<label>] SUBS <dma>
Indirect Addressing: [<label>] SUBS {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(ACC) - (dma) → ACC

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

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	0	1	0	1	0	Data Memory Address						
Indirect	0	1	0	0	0	1	0	1	1	See Section 4.1						

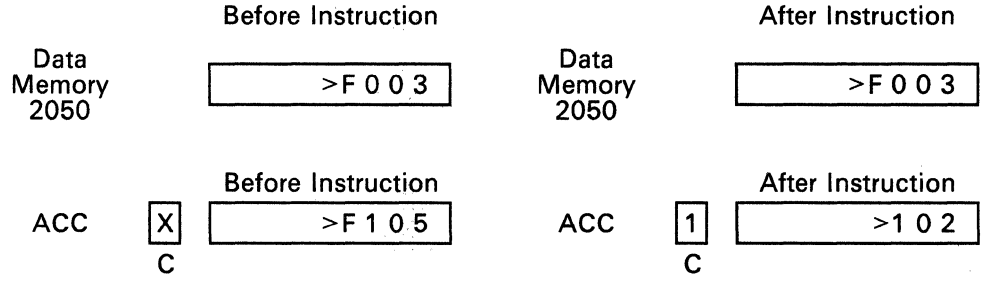
Description

The contents of the addressed 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 result as a SUB instruction with SXM = 0 and a shift count of 0.

Words 1
Cycles Class I (1)
Repeatability Category B

Example

```
SUBS    DAT2    (DP = 16)
or
SUBS    *       If current auxiliary register contains 2050.
```



Subtract from Accumulator with Shift Specified by T Register

Assembler Syntax

Direct Addressing: [<label>] SUBT <dma>
Indirect Addressing: [<label>] SUBT {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 1 → PC
(ACC) - [(dma) x $2^{T\ register(3-0)}$] → (ACC)

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

Affects C and OV; affected by SXM and OVM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	0	1	1	0	0	Data Memory Address						
Indirect	0	1	0	0	0	1	1	0	1	See Section 4.1						

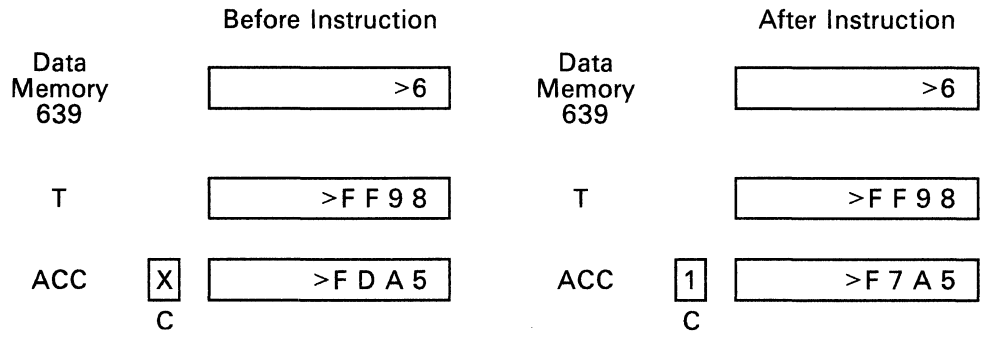
Description

The data memory value, left-shifted as defined by the four LSBs of the T register, is subtracted from the accumulator. The result replaces the accumulator contents. Sign-extension on the data memory value is controlled by the SXM status bit.

Words 1
Cycles Class I (1)
Repeatability Category A

Example

```
SUBT    DAT127      (DP = 4)
or
SUBT    *           If current auxiliary register contains 639.
```



Assembler Syntax [<label>] SXF

Operands None

Execution (PC) + 1 → PC
1 → external flag (XF) pin and status bit

Affects XF.

Encoding

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

Description The XF pin and the XF status bit in status register ST1 are set to logic 1. XF may also be loaded by the LST1 and RXF instructions.

Words 1
Cycles Class IV (1)
Repeatability Category C

Example

```
SXF             The XF pin and status bit are set to logic 1.
```

Assembler Syntax

Direct Addressing: [<label>] TBLR <dma>
Indirect Addressing: [<label>] TBLR {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 1 → PC
(PFC) → MCS
(ACC(15-0)) → PFC

While (repeat counter) ≠ 0:
(pma, addressed by PFC) → dma,
Modify AR(ARP) and ARP as specified,
(PFC) + 1 → PFC,
(repeat counter) - 1 → repeat counter.

(pma, addressed by PFC) → dma
Modify AR(ARP) and ARP as specified.
(MCS) → PFC

Encoding

	15	14	13	12	11	10	9	8	7	6 5 4 3 2 1 0
Direct	0	1	0	1	1	0	0	0	0	Data Memory Address
Indirect	0	1	0	1	1	0	0	0	1	See Section 4.1

Description

The TBLR instruction transfers a word from a location in program memory to a data memory location specified by the instruction. The program memory address is defined by the low-order 16 bits of the accumulator. For this operation, a read from program memory is performed, followed by a write to data memory. When in the repeat mode, TBLR effectively becomes a single-cycle instruction, and the program counter that contains the ACCL is incremented once each cycle. If the MP/$\overline{\text{MC}}$ pin is low at the time of execution of this instruction and the program memory address used is less than 4096, an on-chip ROM location will be read.

Words 1
Cycles Class XI (4)
Repeatability Category A

Example

```
TBLR    DAT6    (DP = 4)
TBLR    *       If current auxiliary register contains 518.
```

	Before Instruction	After Instruction
ACC	>2 3	>2 3
Program Memory 35	>3 0 6	>3 0 6
Data Memory 518	>7 5	>3 0 6

Assembler Syntax

Direct Addressing: [<label>] TBLW <dma>
Indirect Addressing: [<label>] TBLW {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 1 → PC
(PFC) → MCS
(ACC(15-0)) → PFC

While (repeat counter) ≠ 0:
(dma) → pma, addressed by PFC,
Modify AR(ARP) and ARP as specified,
(PFC) + 1 → PFC,
(repeat counter) - 1 → repeat counter.

(dma) → pma, addressed by PFC,
Modify AR(ARP) and ARP as specified.
~~(MCS) → PFC~~

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	1	1	0	0	1	0	Data Memory Address						
Indirect	0	1	0	1	1	0	0	1	1	See Section 4.1						

Description

The TBLW instruction transfers a word in data memory to program memory. The data memory address is specified by the instruction, and the program memory address is specified by the lower half of the accumulator. A read from data memory is followed by a write to program memory to complete the instruction. When in the repeat mode, TBLW effectively becomes a single-cycle instruction, and the program counter that contains the ACCL is incremented once each cycle. If the MP/$\overline{\text{MC}}$ pin is low at the time of execution of this instruction and the program memory address used is less than 4096, an on-chip ROM location will be addressed but not written to.

Words 1
Cycles Class XII (3)
Repeatability Category A

Example

```
TBLW    DAT5    (DP = 32)
TBLW    *       If current auxiliary register contains 4101.
```

	Before Instruction	After Instruction
ACC	>2 5 7	>2 5 7
Data Memory 4101	>4 3 3 9	>4 3 3 9
Program Memory 599	>3 0 6	>4 3 3 9

Assembler Syntax [<label>] TRAP

Operands None

Execution (PC) + 1 → stack
30 → PC

Not affected by INTM; does not affect INTM.

Encoding

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

Description The TRAP instruction is a software interrupt that transfers program control to program memory location 30 and pushes the program counter plus one onto the hardware stack. The instruction at location 30 may contain a branch instruction to transfer control to the TRAP routine. Putting the PC + 1 onto the stack enables an RET instruction to pop the return PC (points to instruction after the TRAP) from the stack.

Words 1
Cycles Class VIII (3)
Repeatability Category X

Example

```
TRAP            Control is passed to program memory
                location 30.  PC + 1 is pushed onto
                the stack.
```

Assembler Syntax

Direct Addressing: [<label>] XOR <dma>
Indirect Addressing: [<label>] XOR {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
(ACC(15-0)).XOR.dma → ACC(15-0)
(ACC(31-16)) → ACC(31-16)

Not affected by SXM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	1	1	0	0	0	Data Memory Address						
Indirect	0	1	0	0	1	1	0	0	1	See Section 4.1						

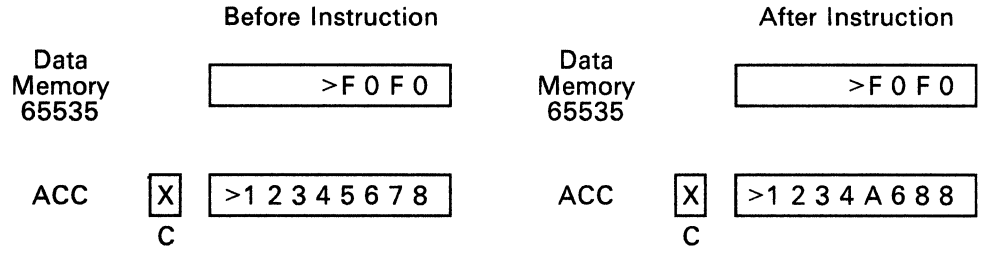
Description

The low half of the accumulator is exclusive-ORed with the contents of the addressed data memory location. The upper half of the accumulator is not affected by this instruction.

Words 1
Cycles Class I (1)
Repeatability Category B

Example

```
XOR     DAT127    (DP = 511)
or
XOR     *         If current auxiliary register contains 65535.
```



Assembler Syntax [<label>] XORK <constant>[,<shift>]

Operands
16-bit constant
0 ≤ shift ≤ 15 (defaults to 0)

Execution
(PC) + 2 → PC
(ACC(30-0)).XOR.[constant x 2^{shift}] → ACC(30-0)
(ACC(31)) → ACC(31)

Not affected by SXM.

Encoding

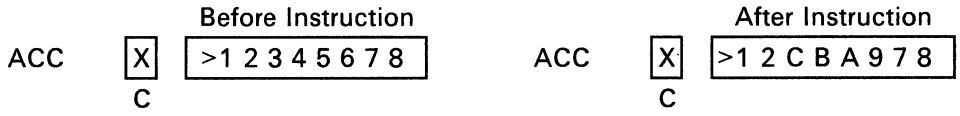
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Shift				0	0	0	0	0	1	1	0
16-Bit Constant															

Description
The left-shifted 16-bit immediate constant is exclusive-ORed with the accumulator, leaving the result in the accumulator. Low-order bits below and high-order bits above the shifted value are treated as zeroes, thus not affecting the corresponding bits of the accumulator. Note that the most-significant bit of the accumulator is not affected, regardless of the shift code value.

Words 2
Cycles Class V (2)
Repeatability Category X

Example

```
XORK    >FFFF,8
```



Assembler Syntax [<label>] ZAC

Operands None

Execution
(PC) + 1 → PC
0 → ACC

Encoding

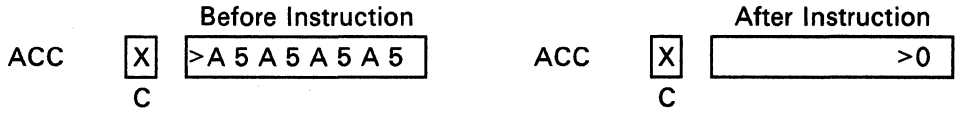
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0

Description The contents of the accumulator are replaced with zero. The ZAC instruction has been implemented as a special case of LACK. (ZAC assembles as LACK 0.)

Words 1
Cycles Class IV (1)
Repeatability Category X

Example

```
ZAC
```



Assembler Syntax

Direct Addressing: [<label>] ZALH <dma>
Indirect Addressing: [<label>] ZALH {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

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

Execution

(PC) + 1 → PC
0 → ACC(15-0)
(dma) → ACC(31-16)

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	0	0	0	0	0	Data Memory Address						
Indirect	0	1	0	0	0	0	0	0	1	See Section 4.1						

Description

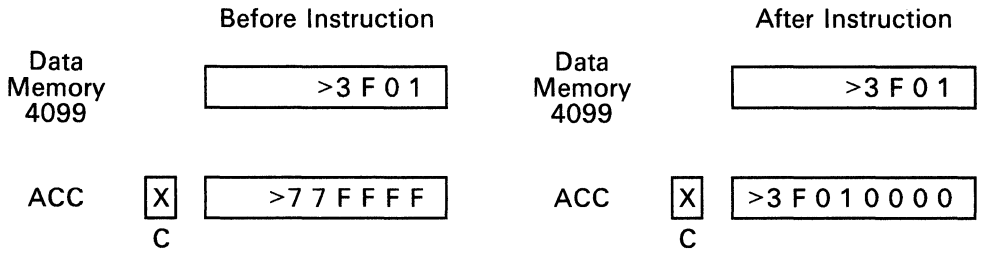
ZALH loads a data memory value into the high-order half of the accumulator. The low-order bits of the accumulator are zeroed.

ZALH is useful for 32-bit arithmetic operations.

Words 1
Cycles Class I (1)
Repeatability Category C

Example

```
ZALH    DAT3    (DP = 32)
or
ZALH    *       If current auxiliary register contains 4099.
```



Zero Low Accumulator, Load High Accumulator with Rounding

Assembler Syntax

Direct Addressing: [<label>] ZALR <dma>
Indirect Addressing: [<label>] ZALR {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
>8000 → ACC(15-0)
(dma) → ACC(31-16)

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	1	1	1	0	1	1	0	Data Memory Address						
Indirect	0	1	1	1	1	0	1	1	1	See Section 4.1						

Description

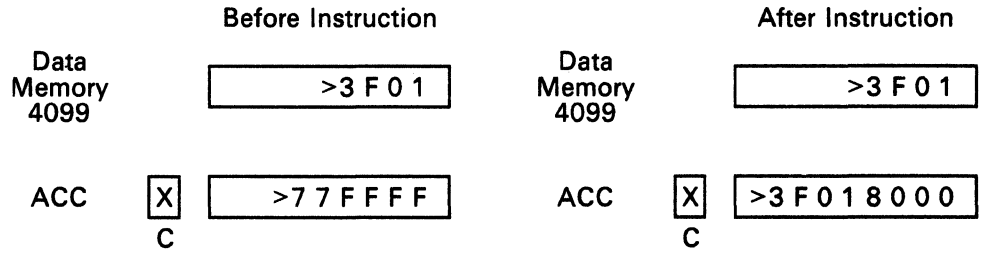
The ZALR instruction loads a data memory value into the high-order half of the accumulator with rounding the value by adding 1/2 LSB; i.e., the 15 low bits (bits 0-14) of the accumulator are set to zero and bit 15 of the accumulator is set to one.

ZALR is a derivative instruction from ZALH.

Words 1
Cycles Class I (1)
Repeatability Category C

Example

```
ZALR    DAT3    (DP = 32)
or
ZALR    *       If current auxiliary register contains 4099.
```



Assembler Syntax

Direct Addressing: [<label>] ZALS <dma>
Indirect Addressing: [<label>] ZALS {*|*+|*-|*0+|*0-|*BR0+|*BR0-}[,<next ARP>]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Execution

(PC) + 1 → PC
0 → ACC(31-16)
(dma) → ACC(15-0)

Not affected by SXM.

Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct	0	1	0	0	0	0	0	1	0	Data Memory Address						
Indirect	0	1	0	0	0	0	0	1	1	See Section 4.1						

Description

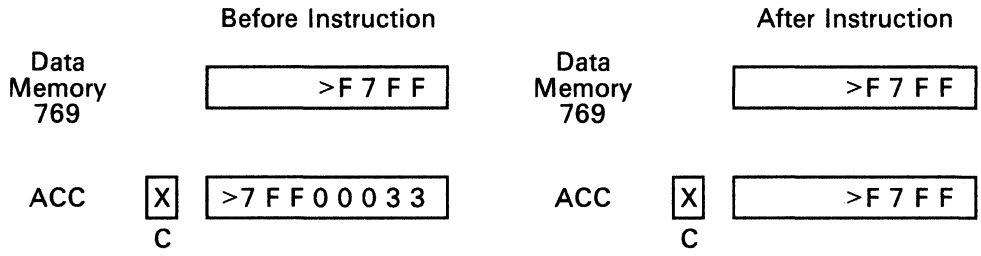
The contents of the addressed data memory location are loaded into the 16 low-order bits of the accumulator. The upper half of the accumulator is zeroed. The data is treated as a 16-bit unsigned number rather than a two's-complement number. Therefore, there is no sign-extension with this instruction, regardless of the state of SXM. (ZALS behaves the same as a LAC instruction with no shift and SXM = 0.)

ZALS is useful for 32-bit arithmetic operations.

Words 1
Cycles Class I (1)
Repeatability Category C

Example

```
ZALS    DAT1    (DP = 6)
or
ZALS    *       If current auxiliary register contains 769.
```



5. Software Applications

The TMS320C25 microprocessor/microcomputer design emphasizes overall speed, communication, and flexibility. Control signals and instructions provide block-memory transfers, communication with off-chip devices (both serial and parallel), and multiprocessing possibilities. The instructions are tailored to digital signal processing tasks, providing single-cycle multiply/accumulates, adaptive filtering support, and many other features. There is also instruction support for floating-point, extended-precision, and logical processing. Increased throughput for many digital signal processing (DSP) applications is accomplished by the single-cycle multiply/accumulate instructions, two large on-chip RAM blocks, eight auxiliary registers with a dedicated arithmetic unit, a serial port, hardware timer, and single-cycle I/O.

This section provides explanations of how to use the various TMS320C25 processor and instruction set features along with assembly language coding examples. More information about specific applications can be found in the book, *Digital Signal Processing Applications with the TMS320 Family*.

Major topics discussed in this section are listed below.

5.1 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.

When reset is activated by applying a low level to the $\overline{RS}$ (reset) input for at least three cycles, the TMS320C25 terminates execution and forces the program counter (PC) to zero. Program memory location 0 normally contains a B (branch) instruction in order to direct program execution to the system initialization routine. The hardware reset also initializes various registers and status bits.

After reset, the processor should be initialized to meet the requirements of the system. Instructions should be executed that set up operational modes, memory pointers, interrupts, and the remaining functions necessary to meet system requirements.

To configure the processor after reset, the following internal functions should be initialized:

- Memory-mapped registers
- Interrupt structure
- Mode control (OVM, SXM, HM, FSM, FO, TXM, PM)
- Memory control (CNF)
- Auxiliary registers and the auxiliary register pointer (ARP)
- Data memory page pointer (DP).

The OVM (overflow mode), TC (test/control flag), and IMR (interrupt mask register) bits are not initialized by reset. The auxiliary register pointer (ARP), auxiliary register pointer buffer (ARB), and data memory page pointer (DP) are also not initialized by reset.

Example 5-1 shows coding for initializing the TMS320C25 to the following machine state, in addition to the initialization performed during the hardware reset:

- All interrupts enabled
- Overflow mode (OVM) disabled
- Data memory page pointer (DP) set to zero
- Auxiliary register pointer (ARP) set to seven
- Internal memory filled with zero.

Example 5-1. Processor Initialization

```
        TITL    'PROCESSOR INITIALIZATION'
        IDT     'EXAMPLE'
        DEF     RESET,INT0,INT1,INT2
        DEF     TINT,RINT,XINT,USER
        REF     ISR0,ISR1,ISR2
        REF     TIME,RCV,XMT,PROC
*
* PROCESSOR INITIALIZATION
* RESET AND INTERRUPT VECTOR SPECIFICATION
* BRANCHES FOR EXTERNAL AND INTERNAL INTERRUPTS
*
        AORG    >0000
RESET   B       INIT            ; RS- BEGINS PROCESSING HERE.
*
INT0    B       ISR0            ; INT0- BEGINS PROCESSING HERE.
INT1    B       ISR1            ; INT1- BEGINS PROCESSING HERE.
INT2    B       ISR2            ; INT2- BEGINS PROCESSING HERE.
*
        AORG    >0018
TINT    B       TIME            ; TIMER INTERRUPT PROCESSING.
RINT    B       RCV             ; SERIAL PORT RECEIVE PROCESSING.
XINT    B       XMT             ; SERIAL PORT TRANSMIT PROCESSING.
*
USER    B       PROC            ; TRAP VECTOR PROCESSING BEGINS.
*
* THE BRANCH INSTRUCTION AT PROGRAM MEMORY LOCATION 0 DIRECTS
* EXECUTION TO BEGIN HERE FOR RESET PROCESSING THAT INITIALIZES
* THE PROCESSOR. WHEN RESET IS APPLIED, THE FOLLOWING CONDITIONS
* ARE ESTABLISHED FOR THE STATUS AND OTHER INTERNAL REGISTERS:
*
*         ARP   OV   OVM  1  INTM      DP
* ST0:    XXX    0    X   1    1    XXXXXXXXX
*
*         ARB   CNF  TC  SXM  C   11  HM  FSM XF  FO  TXM  PM
* ST1:    XXX    0    X   1   1   11   1   1   1   0   0   00
*
* REGISTER    ADDRESS          DATA
*   DRR         >0000     XXXX XXXX XXXX XXXX
*   DXR         >0001     XXXX XXXX XXXX XXXX
*   TIM         >0002     1111 1111 1111 1111
*   PRD         >0003     1111 1111 1111 1111
*   IMR         >0004     1111 1111 11XX XXXX
*   GREG        >0005     1111 1111 0000 0000
*
*          RESERVED   XINT   RINT   TINT   INT2   INT1   INT0
* IMR:    1111111111    X      X      X      X      X      X
*
INIT    ROVM                    ; DISABLE OVERFLOW MODE.
        LDPK    0               ; POINT DP REGISTER TO DATA PAGE 0.
        LARP    7               ; POINT TO AUXILIARY REGISTER 7.
        LACK    >3F             ; LOAD ACCUMULATOR WITH >3F.
        SACL    4               ; ENABLE ALL INTERRUPTS VIA IMR.
```

```
*
* INTERNAL DATA MEMORY INITIALIZATION.
*
        ZAC                 ; ZERO THE ACCUMULATOR.
        LARK    AR7,>60     ; POINT TO BLOCK B2.
        RPTK    31
        SACL    *+          ; STORE ZERO IN ALL 32 LOCATIONS.
*
        LRLK    AR7,>200    ; POINT TO BLOCK B0.
        RPTK    255
        SACL    *+          ; ZERO ALL OF PAGE 4.
        RPTK    255
        SACL    *+          ; ZERO ALL OF PAGE 5.
*
*                           ; POINT TO BLOCK B1.
        RPTK    255
        SACL    *+          ; ZERO ALL OF PAGE 6.
        RPTK    255
        SACL    *+          ; ZERO ALL OF PAGE 7.
*
* THE PROCESSOR IS INITIALIZED. THE REMAINING APPLICATION-
* DEPENDENT PART OF THE SYSTEM (BOTH ON- AND OFF-CHIP) SHOULD
* NOW BE INITIALIZED.
*
        EINT                ; ENABLE ALL INTERRUPTS.
```

5.2 Program Control

To facilitate the TMS320C25's use in general-purpose high-speed processing, a variety of instructions are provided for software stack expansion, subroutine calls, timer operation, single-instruction loops, and external branch control. Descriptions and examples of how to use these features of the TMS320C25 are given in this section.

5.2.1 Subroutines

The TMS320C25 has a 16-bit Program Counter (PC) and an eight-level hardware stack for PC storage. The CALL and CALA subroutine calls store the current contents of the program counter on the top of the stack. The RET (return from subroutine) instruction then pops the top of the stack to the program counter.

Example 5-2 illustrates the use of a subroutine to determine the square root of a 16-bit number. Processing proceeds in the main routine to the point where the square root of a number should be taken. At this point a CALL 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 RET instruction when execution has completed.

Example 5-2. Subroutines

```
* AUTOCORRELATION
*
* THIS ROUTINE PERFORMS A CORRELATION OF TWO VECTORS AND THEN
* CALLS A SQUARE ROOT SUBROUTINE THAT WILL DETERMINE THE RMS
* AMPLITUDE OF THE WAVEFORM.
*
AUTOC
        .
        .
        .
       LAC   ENERGY
       CALL  SQRT
       SACL  ENERGY
        .
        .
        .
*
* SQUARE ROOT
*
* THIS SUBROUTINE DETERMINES THE SQUARE ROOT OF A NUMBER X THAT
* IS LOCATED IN THE LOW HALF OF THE ACCUMULATOR WHEN THE ROUTINE
* IS CALLED. THE FRACTIONAL SQUARE ROOT OF X IS TAKEN, WHERE
* 0 < X < 1 AND WHERE 1 IS REPRESENTED BY >7FFF. THE RESULT IS
* RETURNED TO THE CALLING ROUTINE IN THE ACCUMULATOR.
*
ST0     EQU   >60      ; SAVED STATUS REGISTER ST0 ADDRESS
ST1     EQU   >61      ; SAVED STATUS REGISTER ST1 ADDRESS
NUMBER  EQU   >62      ; NUMBER X WHOSE SQUARE ROOT IS TAKEN
TEMPR   EQU   >63      ; INTERMEDIATE ROOTS
GUESS   EQU   >64      ; SQUARE ROOT OF X
*
SQRT    SST   ST0      ; SAVE STATUS REGISTER ST0.
        SST1  ST1      ; SAVE STATUS REGISTER ST1.
        LDPK  0        ; LOAD DATA PAGE POINTER = 0.
        SSXM           ; SET SIGN-EXTENSION MODE.
        SPM   1        ; LEFT-SHIFT PR OUTPUT TO ACCUMULATOR.
        SACL  NUMBER   ; SAVE X.
        LARP  AR1      ; INITIALIZE VARIABLES FOR SQUARE ROOT.
        LARK  AR1,11   ; 12 ITERATIONS
        LALK  >800     ; ASSUME X IS LESS THAN >200.
        SACL  GUESS    ; SET INITIAL GUESS TO >800.
        SACL  TEMPR    ; SET FIRST INTERMEDIATE ROOT TO >800.
        SACH  ROOT     ; SET SQUARE ROOT VALUE TO 0.
        LAC   NUMBER   ; LOAD X INTO THE ACCUMULATOR.
        SBLK  >200     ; TEST IF X IS LESS THAN >200.
        BLZ   SQRTLP   ; IF YES, TAKE THE ROOT;
        LAC   GUESS,3  ; IF NO, THEN  REINITIALIZE.
        SACL  GUESS    ; SET INITIAL GUESS TO >4000.
        SACL  TEMPR    ; SET FIRST INTERMEDIATE ROOT TO >4000.
        LARK  AR1,14   ; 15 ITERATIONS
*
* SQUARE ROOT LOOP
*
SQRTLP  SQRA  TEMPR    ; SQUARE TEMPORARY (INTERMEDIATE) ROOT.
        ZALH  NUMBER   ; CHECK IF RESULT IS LESS THAN X.
        SPAC
        BLZ   NEXTLP   ; IF IT'S NOT, SKIP ROOT UPDATE.
        ZALH  TEMPR    ; IF IT IS, SET ROOT EQUAL TEMPR.
        SACH  ROOT
```

```
NEXTLP LAC   GUESS,15 ; SCALE DOWN GUESS BY 2 TO CONVERGE.
       SACH  GUESS
       ADDH  ROOT     ; ADD CURRENT ROOT ESTIMATE.
       SACH  TEMPR    ; UPDATE TEMPORARY ROOT VALUE.
       BANZ  SQRTLP   ; REPEAT SPECIFIED NUMBER OF ITERATIONS.
       LAC   ROOT     ; LOAD THE ROOT OF X.
       LST1  ST1      ; RESTORE STATUS REGISTER ST1.
       LST   ST0      ; RESTORE STATUS REGISTER ST0.
       RET
```

Hardware stack allocation involves its use in interrupts, subroutine calls, pipelined instructions, and the emulator (XDS). The TMS320C25 disables all interrupts when taking an interrupt trap. If interrupts are enabled more than one instruction before the return of the interrupt service routine, the routine can also be interrupted, thus using another level of the hardware stack. This condition should be considered when managing the use of the stack. When nesting subroutine calls, each call uses a level of the stack. The number of levels used by the interrupt must be remembered as well as the depth of the nesting of subroutines. The emulator (XDS) uses one level of the stack for breakpoint/single-step operations. Given these constraints, the following listings describe possible allocations of the hardware stack levels:

- 1 level suggested for emulator (XDS) stack
- 1 level reserved for TRAP (software interrupt) instruction
- 1 level reserved for interrupt service routines (ISR)
- 5 levels available for subroutine calls.

or:

- 1 level suggested for emulator (XDS) stack
- 1 level reserved for TRAP (software interrupt) instruction
- 2 levels reserved for interrupt service routines (ISR)
- 4 levels available for subroutine calls.

When two levels are allocated for ISRs, the individual ISRs can utilize one level of subroutine calls or one level of interrupt nesting.

5.2.2 Software Stack

Provisions have been made on the TMS320C25 for extending the hardware stack into data memory. This is useful for deep subroutine nesting or stack overflow protection.

The hardware stack is accessible via the accumulator using the PUSH and POP instructions. Two additional instructions, PSHD and POPD, are included in the instruction set so that the stack may be directly stored to and recovered from data memory.

A software stack can be implemented by using the 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 shown in Example 5-3. In this example, the main program stores the stack starting location in memory in AR2 and indicates to the subroutine whether to push data from memory onto the stack or pop data from the stack to memory. If a '0' is loaded into the accumulator before calling the subroutine, the

subroutine pushes data from memory to the stack. If a '1' is loaded into the accumulator, 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 (branch to address specified by accumulator) instruction to return to the main program. This prevents the program counter from being stored into a memory location. The subroutine in Example 5-3 uses the BANZ (branch on auxiliary register not zero) instruction to control all of its loops.

Example 5-3. Software Stack Expansion

```
* THIS ROUTINE EXPANDS THE STACK WHILE LETTING THE MAIN
* PROGRAM DETERMINE WHERE TO STORE THE STACK CONTENTS OR FROM
* WHERE TO RECOVER THEM.
*
STACK   LARP    2               ; USE AR2.
        LARK    AR1,6           ; LOAD COUNTER.
        BNZ     PO              ; IF POPD IS NEEDED, GOTO PO.
        POP                     ; ELSE, SAVE PROGRAM COUNTER.
P       PSHD    *+,AR1          ; PUT MEMORY IN STACK.
        BANZ    P,*-,AR2        ; BRANCH TO P UNTIL STACK IS FULL.
        BACC                    ; RETURN TO MAIN PROGRAM.
PO      POP                     ; SAVE PROGRAM COUNTER.
        MAR     *-              ; ALIGN STACK POINTER.
PO1     POPD    *-,AR1          ; PUT STACK IN MEMORY.
        BANZ    PO1,*-,AR2      ; BRANCH TO PO1 UNTIL SAVED.
        MAR     *+              ; REALIGN STACK POINTER.
        BACC                    ; RETURN TO MAIN PROGRAM.
```

5.2.3 Timer Operation

The TMS320C25 provides an on-chip timer and its associated interrupt to perform various functions at regular time intervals. By programming the period (PRD) register from 1 to 65,535 (>FFFF), a timer interrupt (TINT) can be generated every 2 to 65,536 cycles, respectively. (A period register value of zero is not allowed.)

Two memory-mapped registers are used to operate the timer. The timer (TIM) register, data memory location 2, holds the current count of the timer. At every CLKOUT1 cycle, the TIM register is decremented by one. The PRD register, data memory location 3, holds the starting count for the timer. When the TIM register decrements to zero, a timer interrupt (TINT) is generated. In the following cycle, the contents of the PRD register are loaded into the TIM register. In this way, a TINT is generated every (PRD + 1) cycles of CLKOUT1.

The timer and period registers can be read from or written to on any cycle. The count can be monitored by reading the TIM register. A new counter period can be written to the PRD register without disturbing the current timer count. The timer will then start the new period after the current count is complete. If both the PRD and TIM registers are loaded with a new period, the timer begins decrementing the new period without generating an interrupt. Thus, the programmer has complete control of the current and next periods of the timer.

The TIM and PRD registers are both set to the maximum value on reset (>FFFF). The TIM register begins decrementing only after $\overline{RS}$ is de-asserted. If the timer is not used, TINT should be masked. The PRD register can then be used as a gener-

al-purpose data memory location. If TINT is used, the PRD and TIM registers should be programmed before unmasking the TINT.

Example 5-4 shows the assembly code that implements the use of the timer to divide down the CLKOUT1 signal. To generate a 9600-Hz clock signal, the PRD register should be loaded with 520. In the timer interrupt service routine, the XF line is toggled. The XF output is also used as an input for $\overline{BIO}$ in this example. The output of XF will provide a 50-percent duty cycle clock signal as long as the main routine or other interrupt routines do not disable interrupts. Interrupts may be disabled by direct or implied use of DINT, or by executing instructions in the repeat mode. The value for the PRD register is calculated as follows:

CLKOUT1/(PRD + 1) = 2 × frequency of signal
10 MHz/(520 + 1) = 2 x 9600 Hz

Assuming a 10-MHz CLKOUT1 frequency, the frequency of the divided signal is 9597 Hz.

Example 5-4. Clock Divider Using Timer

```
* SETUP FOR INTERRUPT SERVICE ROUTINE.
*
        LALK    520
        SACL    DMA3            ; LOAD THE PERIOD REGISTER.
        LACK    8
        OR      DMA4
        SACL    DMA4            ; ENABLE THE TIMER INTERRUPT.
        EINT                    ; ENABLE INTERRUPTS.
          .
          .
          .
* I/O SERVICE ROUTINE.
*
TIME    BIOZ    SET1            ; CHECK THE CURRENT XF STATE.
        RXF                     ; XF WAS HIGH; SET IT LOW.
        EINT                    ; ENABLE INTERRUPTS.
        RET                     ; RETURN TO INTERRUPTED CODE.
SET1    SXF                     ; XF WAS LOW; SET IT HIGH.
        EINT                    ; ENABLE INTERRUPTS.
        RET                     ; RETURN TO INTERRUPTED CODE.
```

5.2.4 Single-Instruction Loops

When programming time-critical high-computational tasks, it is often necessary to repeat the same operation many times. For these cases, repeat instructions that allow the execution of the next single instruction N+1 times are provided. N is defined by an eight-bit repeat counter (RPTC), which is loaded by the RPT or RPTK instructions. The instruction immediately following is then executed, and the RPTC is decremented until it reaches zero.

When using the repeat feature, the instruction being repeated is fetched only once. As a result, many multicycle instructions become single-cycle when repeated. This is especially useful for I/O instructions, such as TBLR/TBLW, IN/OUT, or BLKD/BLKP.

Since the instruction is fetched and internally latched, the program bus can be used to fetch or write a second operand in parallel to operations using the data bus. With the instruction latched for repeated execution, the program counter can be loaded with a data address and incremented on succeeding executions to fetch data in

successive memory locations. As an example, the MAC instruction fetches the multiplicand from program memory via the program bus. Simultaneous with the program bus fetch, the second multiplicand is fetched from data memory via the data bus. In addition to these data fetches, preparation is made for accesses in the following cycles by incrementing the program counter and by indexing the auxiliary register. TBLR is another example of an instruction that benefits from simultaneous transfers of data on both the program and data buses. In this case, data values from a table in program memory may be read and transferred to data memory. When repeated, the program overhead of reading the instruction from program memory must be executed only once, thus allowing the rest of the executions to operate in a single cycle.

Programs, such as those implementing digital filters, require loops that execute in a minimum amount of time. Example 5-5 shows the use of the RPT or RPTK instructions.

Example 5-5. Instruction Repeating

```
* THIS ROUTINE USES THE RPT INSTRUCTION TO SET UP THE LOOP COUNTER
* IN ONE CYCLE. THE FOLLOWING EQUATION IS IMPLEMENTED IN THIS
* ROUTINE:
*
*       10
*     -----
*     \       X(I) x Y(I)
*     /
*     -----
*     I = 1
*
* THIS ROUTINE ASSUMES THAT THE X VALUES ARE LOCATED IN ON-CHIP
* ROM, AND THE Y VALUES IN BLOCK B1. WHEN REPLACING RPT NUM
* WITH RPTK 9, THE PROGRAM WILL EXECUTE THE SAME WAY.
*
SERIES  LARP  AR6
        LACK  9           ; SET COUNTER TO 9.
        SACL  NUM         ; (NUM) = 9.
        LRLK  AR6,>300    ; POINT AT BEGINNING OF DATA.
        MPYK  >0          ; CLEAR P REGISTER.
        ZAC               ; CLEAR ACCUMULATOR.
        RPT   NUM         ; EXECUTE FOLLOWING INSTRUCTION 10 TIMES.
        MAC   >600,*+     ; MULTIPLY AND ACCUMULATE; INCREMENT AR6.
        APAC
        RET               ; RETURN TO MAIN PROGRAM.
```

5.2.5 Computed GOTOs

Processing may be executed in a time- and process-dependent or selected way. Following a specific time or data processing path may then result in selecting one of several processing options.

A simple computed GOTO can be programmed in the TMS320C25 by using the CALA instruction. This instruction uses the contents of the accumulator as the direct address of the call. Thus, the call address can be computed in the ALU, as shown in Example 5-6.

Example 5-6. Computed GOTO

```
* TASK CONTROLLER
*
* THIS MAIN TASK ROUTINE CONTROLS THE ORDER OF EXECUTION
* AND SCHEDULING OF TASKS. WHEN AN INTERRUPT OCCURS, THE
* INTERRUPT SERVICE ROUTINE IS EXECUTED TO PROCESS THE INPUT
* AND OUTPUT DATA SAMPLES. AFTER THE INTERRUPT SERVICE
* ROUTINE HAS COMPLETED, THE PROCESSOR BEGINS EXECUTION WITH
* THE INSTRUCTION FOLLOWING THE IDLE INSTRUCTION. THIS
* ROUTINE SELECTS THE TASK APPROPRIATE FOR THE CURRENT
* SAMPLE CYCLE, CALLS THE TASK AS A SUBROUTINE, AND BRANCHES
* BACK TO THE IDLE TO WAIT FOR THE NEXT SAMPLE INTERRUPT
* WHEN THE SCHEDULED TASK HAS COMPLETED EXECUTION.
*
WAIT    IDLE            ; WAIT FOR SAMPLE INTERRUPT.
        LAC     SAMPLE  ; FETCH SAMPLE COUNT VALUE.
        SUB     ONE     ; DECREMENT THE SAMPLE COUNT.
        BGEZ    OVRSAM  ; TEST FOR END OF BAUD INTERVAL.
        LACK    15      ; INIT COUNT FOR NEW BAUD INTERVAL.
OVRSAM  SACL    SAMPLE  ; SAVE NEW COUNT VALUE.
        ADLK    TSKSEQ  ; ADD TASK TABLE BASE ADDRESS.
        TBLR    TEMP    ; READ SUBROUTINE TASK ADDRESS.
        LAC     TEMP    ; LOAD ACCUMULATOR FOR TASK CALL.
        CALA            ; EXECUTE APPROPRIATE TASK.
        B       WAIT
*
TSKSEQ  EQU     $
        DATA    DUMMY   ; 15 - UNUSED CYCLE
        DATA    DUMMY   ; 14 - UNUSED CYCLE
        DATA    DUMMY   ; 13 - UNUSED CYCLE
        DATA    DUMMY   ; 12 - UNUSED CYCLE
        DATA    BDCLK2  ; 11 - COMPUTE ENERGY E(11)
        DATA    DUMMY   ; 10 - UNUSED CYCLE
        DATA    OUT     ; 9 - COMMUNICATE WITH U-CONTROLLER
        DATA    DECODE  ; 8 - DECODE/GET SCRAMBLED DIBIT
        DATA    DEMODB  ; 7 - DEMODULATE IN MIDDLE OF BAUD
        DATA    DUMMY   ; 6 - UNUSED CYCLE
        DATA    AGCUPT  ; 5 - UPDATE AGC EVERY 3RD BAUD
        DATA    DUMMY   ; 4 - UNUSED CYCLE
        DATA    BDCLK1  ; 3 - COMPUTE ENERGY E(3)
        DATA    DUMMY   ; 2 - UNUSED CYCLE
        DATA    DUMMY   ; 1 - UNUSED CYCLE
        DATA    DUMMY   ; 0 - UNUSED CYCLE
```

5.3 Interrupt Service Routine

Interrupts on the TMS320C25 are prioritized and vectored. When an interrupt occurs, the corresponding flag is set in the Interrupt Flag Register (IFR). If the corresponding bit in the Interrupt Mask Register (IMR) is set and interrupts are enabled (INTM=0), then interrupt processing begins.

When the interrupt vector is loaded into the program counter, interrupts are disabled (INTM=1) and a branch is made to the appropriate routine via the branch instruction stored at the associated vector location. Since all interrupts are disabled, interrupt processing may proceed without further interruption unless the interrupt service routine (ISR) re-enables interrupts.

Unless the interrupt service routines are simple I/O handlers, the processing in each ISR generally must assure that the processor context is preserved during execution. The context must be saved before executing the routine itself and restored when the routine is finished. A common routine or routines individualized for each interrupt may be used to secure the context of the processor during interrupt processing. Context switching is also useful for subroutine calls, especially when extensive use is made of the stack or auxiliary registers. Code examples of context switching and an interrupt service routine are provided in this section.

5.3.1 Context Switching

Context switching, commonly required when processing a subroutine call or interrupt, may be quite extensive or simple, depending on the system requirements. On the TMS320C25, the program counter is stored automatically on the hardware stack. If there is any important information in the other TMS320C25 registers, such as the status or auxiliary registers, these must be saved by software command. A stack in data memory, identified by an auxiliary register, is useful for storing the machine state when processing interrupts.

Examples of saving and restoring the state of the TMS320C25 are given in Example 5-7 and Example 5-8. Auxiliary register 7 (AR7) is used in both examples as the stack pointer. As the stack grows, it expands into lower memory addresses. The registers saved are the status registers (ST0 and ST1), accumulator (ACCH and ACCL), product register (PR), temporary register (TR), all eight levels of the hardware stack, and the auxiliary registers (AR0 through AR6).

The routines in Example 5-7 and Example 5-8 are protected against interrupts, allowing context switches to be nested. This is accomplished by the use of the MAR *- and MAR *+ instructions at the beginning of the context save and context restore routines, respectively. Note that the last instruction of the context save decrements AR7 while the context restore is completed with an additional increment of AR7. This prevents the loss of data if a context save or restore routine is interrupted.

Example 5-7. Context Save

```
      TITL   'CONTEXT SAVE'
      DEF    SAVE
*
* CONTEXT SAVE ON SUBROUTINE CALL OR INTERRUPT.
*
* ASSUME AR7 IS THE STACK POINTER AND AR7 = 128.
*
SAVE  LARP   AR7      ; (ARP) --> ARB, 7 --> ARP,   AR7 = 128
      MAR    *-       ;                             AR7 = 127
*
* SAVE THE STATUS REGISTERS.
      SST1   *-       ; ST1 --> (127),              AR7 = 126
      SST    *-       ; ST0 --> (126),              AR7 = 125
*
* SAVE THE ACCUMULATOR.
      SACH   *-       ; ACCH --> (125),             AR7 = 124
      SACL   *-       ; ACCL--> (124),              AR7 = 123
*
* SAVE THE P REGISTER.
      SPM    0        ; NO SHIFT ON PR OUTPUT
      SPH    *-       ; PRH --> (123),              AR7 = 122
      SPL    *-       ; PRL --> (122),              AR7 = 121
*
* SAVE THE T REGISTER.
      MPYK   1        ; PR = TR
      SPL    *-       ; TR  --> (121),              AR7 = 120
*
* SAVE ALL EIGHT LEVELS OF THE HARDWARE STACK.
      RPTK   7
      POPD   *-       ; TOS  (8) --> (120),         AR7 = 119
*                     ; STACK(7) --> (119),         AR7 = 118
*                     ; STACK(6) --> (118),         AR7 = 117
*                     ; STACK(5) --> (117),         AR7 = 116
*                     ; STACK(4) --> (116),         AR7 = 115
*                     ; STACK(3) --> (115),         AR7 = 114
*                     ; STACK(2) --> (114),         AR7 = 113
*                     ; BOS  (1) --> (113),         AR7 = 112
*
* SAVE AUXILIARY REGISTERS AR0 THROUGH AR6.
      SAR    AR0,*-   ; AR0 --> (112),              AR7 = 111
      SAR    AR1,*-   ; AR0 --> (111),              AR7 = 110
      SAR    AR2,*-   ; AR0 --> (110),              AR7 = 109
      SAR    AR3,*-   ; AR0 --> (109),              AR7 = 108
      SAR    AR4,*-   ; AR0 --> (108),              AR7 = 107
      SAR    AR5,*-   ; AR0 --> (107),              AR7 = 106
      SAR    AR6,*-   ; AR0 --> (106),              AR7 = 105
*
* SAVE IS COMPLETE.
```

Example 5-8. Context Restore

```
        TITL    'CONTEXT RESTORE'
        DEF     RESTOR
*
* CONTEXT RESTORE AT THE END OF A SUBROUTINE OR INTERRUPT.
*
* ASSUME AR7 IS THE STACK POINTER AND AR7 = 105.
*
RESTOR LARP    AR7         ; (ARP) --> ARB, 7 --> ARP,  AR7 = 105
        MAR     *+          ;                            AR7 = 106
*
* RESTORE AUXILIARY REGISTERS AR0 THROUGH AR6.
        LAR     AR6,*+      ; (106) --> AR6,             AR7 = 107
        LAR     AR5,*+      ; (107) --> AR5,             AR7 = 108
        LAR     AR4,*+      ; (108) --> AR4,             AR7 = 109
        LAR     AR3,*+      ; (109) --> AR3,             AR7 = 110
        LAR     AR2,*+      ; (110) --> AR2,             AR7 = 111
        LAR     AR1,*+      ; (111) --> AR1,             AR7 = 112
        LAR     AR0,*+      ; (112) --> AR0,             AR7 = 113
*
* RESTORE ALL EIGHT LEVELS OF THE HARDWARE STACK.
        RPTK    7
        PSHD    *+          ; (113) --> BOS  (1),        AR7 = 114
*                           ; (114) --> STACK(2),        AR7 = 115
*                           ; (115) --> STACK(3),        AR7 = 116
*                           ; (116) --> STACK(4),        AR7 = 117
*                           ; (117) --> STACK(5),        AR7 = 118
*                           ; (118) --> STACK(6),        AR7 = 119
*                           ; (119) --> STACK(7),        AR7 = 120
*                           ; (120) --> TOS  (8),        AR7 = 121
*
* THE RETURN PC IS NOW ON THE TOP OF THE STACK FOR THE
* RET INSTRUCTION. NOTE THAT THE LOWER 16 BITS OF THE
* P REGISTER MUST BE LOADED VIA THE T REGISTER AND THAT
* THE STACK POINTER IS POINTING AT THE VALUE TO BE LOADED
* IN THE T REGISTER.
*
* RESTORE THE LOW P REGISTER.
        MAR     *+          ; SKIP T REGISTER,           AR7 = 122
        LT      *-          ; (122) --> TR,              AR7 = 121
        MPYK    1           ; (TR) --> PRL
*
* RESTORE THE T REGISTER.
        LT      *+          ; (121) --> TR,              AR7 = 122
        MAR     *+          ; SKIP P REGISTER LOW,       AR7 = 123



        LPH     *+          ; (123) --> PRH,             AR7 = 124
*
* RESTORE THE ACCUMULATOR.
        ZALS    *+          ; (124) --> ACCL,            AR7 = 125
        ADDH    *+          ; (125) --> ACCH,            AR7 = 126
*
* RESTORE THE STATUS REGISTERS.
        LST     *+          ; (126) -> ST0,              AR7 = 127
        LST1    *+          ; (127) -> ST1,              AR7 = 128
*
* RESTORE IS COMPLETE.
        EINT                ; ENABLE INTERRUPTS.
        RET                 ; RETURN TO INTERRUPTS OR
*                           ; CALLING ROUTINE.
```

5.3.2 Interrupt Priority

Interrupts on the TMS320C25 are prioritized in hardware. This allows interrupts that occur simultaneously to be serviced in a prioritized order. Sometimes priority may be determined by frequency or rate of occurrence. An infrequent, but lengthy, interrupt service routine (ISR) might need to be interrupted by a more frequently occurring interrupt. In the routine of Example 5-9, the ISR for $\overline{INT1}$ temporarily modifies the interrupt mask register (IMR) to permit interrupt processing when an interrupt on $\overline{INT0}$ (but no other interrupt) occurs. When the routine has finished processing, the IMR is restored to its original state.

Example 5-9. Interrupt Service Routine

```
        TITL   'INTERRUPT SERVICE ROUTINE'
        DEF    ISR1
        REF    IMR
*
* INTERRUPT PROCESSING FOR EXTERNAL INTERRUPT INT1-.
*
* THIS ROUTINE MAY BE INTERRUPTED BY AN INTERRUPT FROM THE
* EXTERNAL INTERRUPT INT0-, BUT NO OTHER.
*
ISR1    LARP   AR7          ; 7 --> ARP
        MAR    *-           ;                  AR7 = AR7 - 1
        SST1   *-           ; ST1  --> *AR7, AR7 = AR7 - 1
        SST    *-           ; ST0  --> *AR7, AR7 = AR7 - 1
        SACH   *-           ; ACCH --> *AR7, AR7 = AR7 - 1
        SACL   *-           ; ACCL --> *AR7, AR7 = AR7 - 1
        LDPK   0            ; DP = 0
        PSHD   IMR          ; IMR --> TOS
        LACK   >0001        ; MASK FOR INT0-
        AND    IMR          ; MASK CURRENT IMR CONTENTS.
        SACL   IMR          ; ACC --> IMR
        EINT                ; ENABLE INTERRUPTS.
*
* MAIN PROCESSING SECTION FOR ISR1.
         .
         .
         .
*
        DINT                ; DISABLE INTERRUPTS.
        LDPK   0            ; DP = 0
        POPD   IMR          ; TOS --> IMR
        LARP   AR7          ; 7 --> ARP
        MAR    *+           ;                  AR7 = AR7 + 1
        ZALS   *+           ; *AR7 --> ACCL, AR7 = AR7 + 1
        ADDH   *+           ; *AR7 --> ACCH, AR7 = AR7 + 1
        LST    *+           ; *AR7 --> ST0,  AR7 = AR7 + 1
        LST1   *+           ; *AR7 --> ST1,  AR7 = AR7 + 1
        EINT                ; ENABLE INTERRUPTS.
        RET
```

5.4 Memory Management

The structure of the TMS320C25's memory map is programmable and can vary for each application. Instructions are provided for moving blocks of data or program memory, configuring a block of on-chip data RAM as program memory, and defining part of external data memory as global. Explanations and examples of moving, configuring, and manipulating memory are provided in this section.

5.4.1 Block Moves

Since the TMS320C25 directly addresses a large amount of memory, blocks of data or program code can be stored off-chip in slow memories and then loaded on-chip for faster execution. Data can also be moved from on-chip to off-chip for storage or for multiprocessor data transfers.

The BLKD and BLKP instructions facilitate memory-to-memory block moves on the TMS320C25. The BLKD instruction moves a block within data memory as shown in Example 5-10. Data may also be transferred between data memory and program memory by means of the TBLR and TBLW instructions. The instructions IN and OUT are used to transfer data between the data memory and the I/O space.

Example 5-10. Moving External Data Memory to Internal Data Memory with BLKD

```
* THIS ROUTINE USES THE BLKD INSTRUCTION TO MOVE A BLOCK OF
* EXTERNAL DATA MEMORY (DATA PAGES 8 AND 9) TO INTERNAL BLOCK
* B1 (DATA PAGES 6 AND 7).
*
MOVED   LARP    AR2
        LRLK    AR2,>300    ; DESTINATION IS BLOCK B1 IN RAM.
        RPTK    255         ; REPEAT NEXT INSTRUCTION 256 TIMES.
        BLKD    >400,*+     ; MOVE EXTERNAL BLOCK TO BLOCK B1.
        RET                 ; RETURN TO MAIN PROGRAM.
```

For systems that have external program memory but no external data memory, BLKP can be used to move program memory blocks into data memory. Example 5-11 demonstrates how to use the BLKP instruction.

Example 5-11. Moving Program Memory to Data Memory with BLKP

```
* THIS ROUTINE USES THE BLKP INSTRUCTION TO MOVE DATA VALUES
* FROM PROGRAM MEMORY INTO DATA MEMORY. SPECIFICALLY, THE
* VALUES IN LOCATIONS 2, 3, 4, AND 5 IN PROGRAM MEMORY ARE
* MOVED TO LOCATIONS 512, 513, 514, AND 515 IN DATA MEMORY.
*
MOVEP   LARP    AR2         ; SET REFERENCE FOR INDIRECT ADDRESSING.
        LRLK    AR2,512     ; LOAD BEGINNING OF BLOCK B0 IN AR2.
        RPTK    3           ; SET UP LOOP.
        BLKP    >2,*+       ; PUT DATA INTO DATA RAM.
        RET                 ; RETURN TO MAIN PROGRAM.
```

Another method for transferring data from program memory into data memory makes use of the TBLR instruction. By using the TBLR instruction, a calculated, rather than predetermined, location of a block of data in program memory may be specified for transfer. A routine using this approach is shown in Example 5-12.

Example 5-12. Moving Program Memory to Data Memory with TBLR

```
* THIS ROUTINE USES THE TBLR INSTRUCTION TO MOVE DATA VALUES
* FROM PROGRAM MEMORY INTO DATA MEMORY. BY USING THIS ROUTINE,
* THE PROGRAM MEMORY LOCATION IN THE ACCUMULATOR FROM WHICH
* DATA IS TO BE MOVED TO A SPECIFIC DATA MEMORY LOCATION CAN
* BE SPECIFIED. ASSUME THAT THE ACCUMULATOR CONTAINS THE
* ADDRESS IN PROGRAM MEMORY FROM WHICH TO TRANSFER THE DATA.
*
TABLER LARP   AR5
       LRLK   AR5,380     ; DESTINATION ADDRESS = PAGE 7.
       RPTK   127         ; TRANSFER 128 VALUES.
       TBLR   *+          ; MOVE DATA INTO DATA RAM.
       RET                ; RETURN TO CALLING PROGRAM.
```

In cases where systems require that temporary storage be allocated in the program memory, TBLW can be used to transfer data from internal data memory to external program memory. The code in Example 5-13 demonstrates how this may be accomplished.

Example 5-13. Moving Internal Data Memory to Program Memory with TBLW

```
* THIS ROUTINE USES THE TBLW INSTRUCTION TO MOVE DATA VALUES
* FROM INTERNAL DATA MEMORY TO EXTERNAL PROGRAM MEMORY. THE
* CALLING ROUTINE MUST SPECIFY THE DESTINATION PROGRAM MEMORY
* ADDRESS IN THE ACCUMULATOR. ASSUME THAT THE ACCUMULATOR
* CONTAINS THE ADDRESS IN PROGRAM MEMORY INTO WHICH THE DATA
* IS TRANSFERRED.
*
TABLEW LARP   AR6
       LRLK   AR6,380      ; SOURCE ADDRESS = PAGE 7.
       RPTK   127          ; TRANSFER 128 VALUES.
       TBLW   *+           ; MOVE DATA TO EXTERNAL PROGRAM RAM.
       RET                 ; RETURN TO CALLING PROGRAM.
```

The IN and OUT instructions are used to transfer data between the data memory and the I/O space, as shown in Example 5-14 and Example 5-15.

Example 5-14. Moving Data from I/O Space into Data Memory with IN

```
* THIS ROUTINE USES THE IN INSTRUCTION TO MOVE DATA VALUES
* FROM THE I/O SPACE INTO DATA MEMORY. DATA ACCESSED FROM
* I/O PORT 15 IS TRANSFERRED TO SUCCESSIVE MEMORY LOCATIONS
* ON DATA PAGE 5.
*
INPUT  LARP   AR2
       LRLK   AR2,>2C0     ; DESTINATION ADDRESS = PAGE 5.
       RPTK   63           ; TRANSFER 64 VALUES.
       IN     PA15,*+      ; MOVE DATA INTO DATA RAM.
       RET                 ; RETURN TO CALLING PROGRAM.
```

Example 5-15. Moving Data from Data Memory to I/O Space with OUT

```
* THIS ROUTINE USES THE OUT INSTRUCTION TO MOVE DATA VALUES
* FROM THE DATA MEMORY TO THE I/O SPACE. DATA IS TRANSFERRED
* TO I/O PORT 8 FROM SUCCESSIVE MEMORY LOCATIONS ON DATA
* PAGE 4.
*
OUTPUT LARP   AR4
       LRLK   AR4,>200   ; SOURCE ADDRESS = PAGE 4.
       RPTK   63         ; TRANSFER 64 VALUES.
       OUT    PA8,*+     ; MOVE DATA FROM DATA RAM.
       RET               ; RETURN TO CALLING PROGRAM.
```

5.4.2 Configuring On-Chip RAM

The large amount of external memory and the configurability of on-chip RAM simplify the downloading of data or program memory into the TMS320C25. Also, since data in the RAM is preserved when redefining on-chip RAM, block B0 can be configured dynamically as either data or program memory. Figure 5-1 illustrates the changes in on-chip RAM when switching configurations.

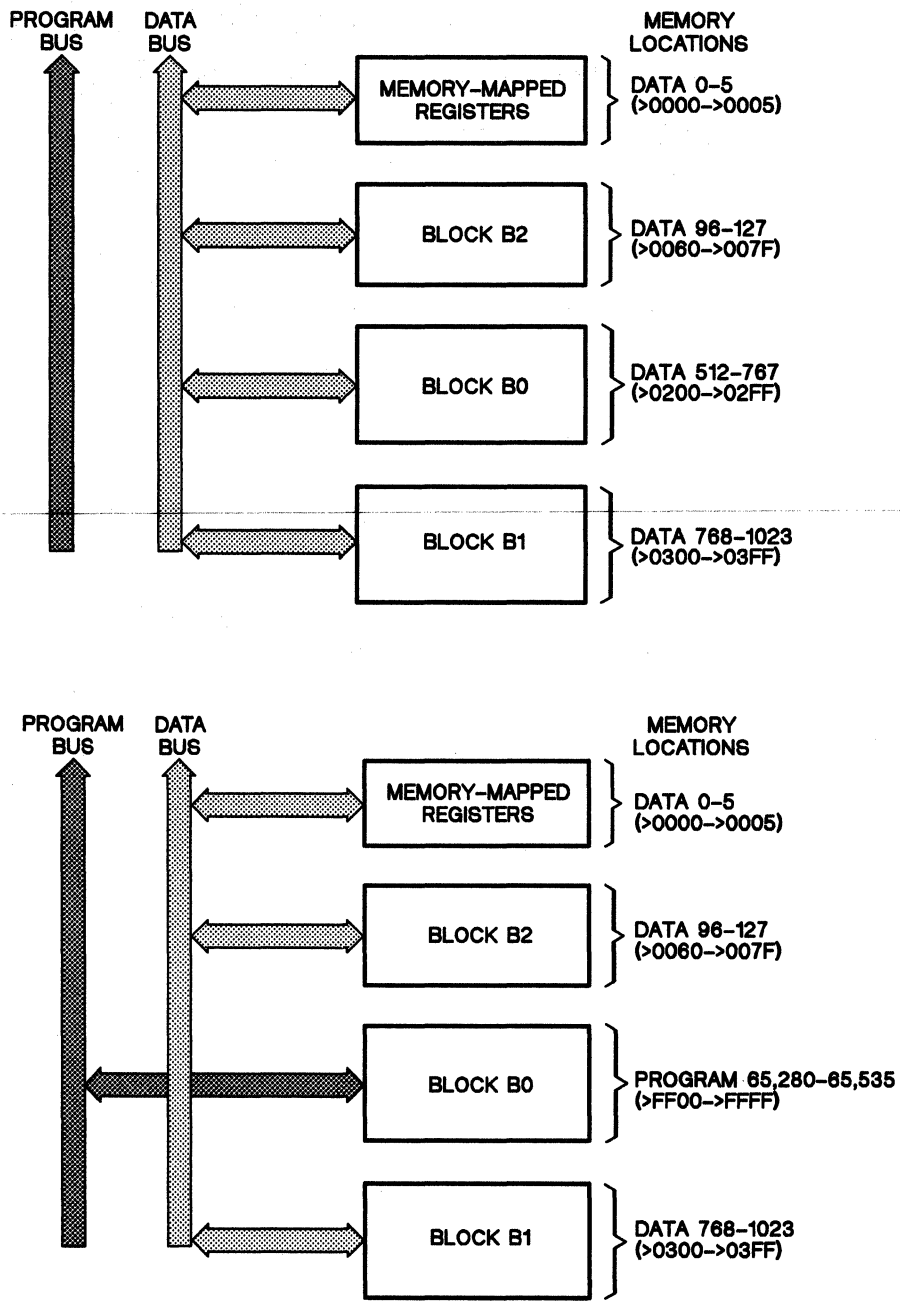


Figure 5-1. On-Chip RAM Configurations

On-chip memory is configured by a reset or by the CNFD and CNFP instructions. Block B0 is configured as data memory by executing CNFD or reset. A CNFP instruction configures block B0 as program memory.

Configuring block B0 as program memory is useful for implementing adaptive filters or other similar applications at full speed with only on-chip memories. Example 5-16

illustrates the use of the configuration modes to utilize block B0 as data and program memory while executing from on-chip program ROM.

Example 5-16. Configuring and Using On-Chip RAM

```
        TITL  'ADAPTIVE FILTER'
        DEF   ADPFIR
        DEF   X,Y
*
* THIS 128-TAP ADAPTIVE FIR FILTER USES ON-CHIP MEMORY BLOCK
* B0 FOR COEFFICIENTS AND BLOCK B1 FOR DATA SAMPLES. THE
* NEWEST INPUT SHOULD BE IN MEMORY LOCATION X WHEN CALLED.
* THE OUTPUT WILL BE IN MEMORY LOCATION Y WHEN RETURNED.
*
COEFFP EQU   >FF00          ; B0 PROGRAM MEMORY ADDRESS
COEFFD EQU   >0200          ; B0 DATA MEMORY ADDRESS
*
ONE    EQU   >7A            ; CONSTANT ONE          (DP=6)
BETA   EQU   >7B            ; ADAPTATION CONSTANT   (DP=6)
ERR    EQU   >7C            ; SIGNAL ERROR          (DP=6)
ERRF   EQU   >7D            ; ERROR FUNCTION        (DP=6)
Y      EQU   >7E            ; FILTER OUTPUT         (DP=6)
X      EQU   >7F            ; NEWEST DATA SAMPLE    (DP=6)
FRSTAP EQU   >0380          ; NEXT NEWEST DATA SAMPLE
LASTAP EQU   >03FF          ; OLDEST DATA SAMPLE
*
* FINITE IMPULSE  RESPONSE (FIR) FILTER.
*
ADPFIR CNFP                 ; CONFIGURE B0 AS PROGRAM:
       MPYK  0              ; Clear the P register.
       LAC   ONE,14         ; Load output rounding bit.
       LARP  AR3
       LRLK  AR3,LASTAP     ; Point to the oldest sample.
FIR    RPTK  127
       MACD  COEFFP,*-      ; 128-tap FIR filter.
       CNFD                 ; CONFIGURE B0 AS DATA:
       APAC
       SACH  Y,1            ; Store the filter output.
       NEG
       ADD   X,15           ; Add the newest input.
       SACH  ERR,1          ; err(n) = x(n) - y(n)
*
* LMS ADAPTATION OF FILTER COEFFICIENTS.
*
       LT    ERR
       MPY   BETA           ; 128-TAP FIR FILTER.
       PAC                  ; errf(n) = beta * err(n)
       ADD   ONE,14         ; ROUND THE RESULT.
       SACH  ERRF,1
*
       LARP  AR3
       LARK  AR1,127        ; 128 COEFFICIENTS TO UPDATE.
       LRLK  AR2,COEFFD     ; POINT TO THE COEFFICIENTS.
       LRLK  AR3,LASTAP     ; POINT TO THE DATA SAMPLES.
       DMOV  X              ; INCLUDE NEWEST SAMPLE.
       LT    ERRF
       MPY   *-,AR2         ; P = 2*beta*err(n)*x(n-k)
```

```
*
ADAPT   ZALR  *,AR3          ; LOAD ACCH WITH ak(n) & ROUND.
        MPYA  *-,AR2         ; ak(n+1) = ak(n) + P
*                            ; P = 2*beta*err(n)*x(n-k)
        SACH  *+,0,AR1       ; STORE ak(n+1).
        BANZ  ADAPT,*-,AR2   ; END OF LOOP TEST.
*
        RET                  ; RETURN TO CALLING ROUTINE.
```

5.4.3 Using On-Chip RAM for Program Execution

In using on-chip memory (block B0) for program execution, this memory must first be loaded with executable code from external memories while configured as data memory. On-chip execution is initiated by using the CNFP instruction to reconfigure block B0 as program memory and performing a branch or call to an on-chip RAM address. By configuring block B0 as program memory and executing from this internal memory, full-speed execution can be achieved in systems using slower external memory. Example 5-17 illustrates how a program may be written to be loaded into and executed from on-chip memory.

One group of instructions, the branch/call instructions, are impacted by the location of execution. Normally, by using labels, the assembler properly determines the location to which a branch is taken. Since the code is relocated prior to execution from on-chip memory, it is necessary to alter the address determined by the assembler for branch instructions. This alteration is necessary so that the branch address that is determined can be consistent with the address space used during execution. In Example 5-17, this is accomplished by adding an offset value (OFFSET) to the branch label representing the destination address in the operand field for each branch instruction. The offset address is determined by use of an EQU (equate) directive that subtracts the assembler location of the code to be relocated (equivalent to base-0 addressing) from the base address of the relocation address (internal block B0 address in this case).

Example 5-17. Program Execution from On-Chip Memory

```
        AORG  0
RESET   B     INIT
*
* BRANCHES FOR EXTERNAL OR INTERNAL INTERRUPTS FOLLOW HERE AT
* THE DESIGNATED LOCATIONS AS REQUIRED.
*
        AORG  >20
*
* A BRANCH INSTRUCTION AT PROGRAM MEMORY LOCATION 0 DIRECTS
* PROCESSOR EXECUTION HERE.
*
* INITIALIZE THE PROCESSOR.
*
INIT    ROVM              ; DISABLE OVERFLOW MODE.
        SSXM              ; SET SIGN EXTENSION.
        LDPK  0           ; POINT DP REGISTER TO DATA MEMORY PAGE 0.
        SPM   0           ; NO SHIFT ON PRODUCT REGISTER OUTPUT.
        LARP  AR4         ; USE AUXILIARY REGISTER 4 (SET ARP = 4).
        LARK  AR4,PRD     ; POINT AR4 TO PERIOD REGISTER.
        LALK  >FFFF       ; SET ACCUMULATOR TO >FFFF.
        SACL  *+          ; LOAD PERIOD REGISTER WITH MAXIMUM VALUE.
        SACL  *+          ; ENABLE ALL INTERRUPTS VIA IMR.
        ZAC               ; CLEAR ACCUMULATOR.
        SACH  *           ; CLEAR GREG TO MAKE ALL MEMORY LOCAL.
*
* LOAD TIME-CRITICAL CODE FROM EXTERNAL SLOW MEMORY TO INTERNAL RAM.
*
        LARP  AR1         ; USE AUXILIARY REGISTER 1 (SET ARP = 1).
        LRLK  AR1,BLK0    ; POINT AR1 TO RECONFIGURABLE BLOCK B0.
        RPTK  PROGL-1     ; LOAD REPEAT COUNTER WITH BLOCK LENGTH.
        BLKP  PROG,*+     ; MOVE CODE FROM PROG MEMORY TO ON-CHIP RAM.
*
* INITIALIZE PARAMETERS FOR EXECUTION.
*
        LDPK  6             ; POINT DP REGISTER TO DATA MEMORY PAGE 6.
        LACK  1             ; SET ACCUMULATOR TO >0001.
        SACL  ONE           ; STORE VALUE OF 1.
        LRLK  AR1,BLK0+PRGL ; POINT AR1 TO INTERNAL MEMORY ADDRESS.
        RPTK  COEFL-1       ; LOAD REPEAT COUNTER WITH BLOCK LENGTH.
        BLKP  COEF,*+       ; MOVE DATA FROM PROG MEMORY TO ON-CHIP RAM.
        CNFP                ; CONFIGURE BLOCK B0 AS PROGRAM MEMORY.
        LALK  >FF00         ; LOAD ACC WITH PROG ADDR IN INTERNAL RAM.
        BACC                ; BRANCH TO ON-CHIP EXECUTION ADDRESS.
```

```
*
* SIGNAL PROCESSING CODE TO BE EXECUTED FROM ON-CHIP RAM.
*
PROG    EQU   $
LPTS    BIOZ  GET+OFFSET      ; WAIT FOR INPUT SIGNAL.
        B     LPTS+OFFSET     ; BRANCH IF NO SIGNAL.
GET     OUT   FILOUT,PA2      ; OUTPUT LAST FILTER OUTPUT.
        IN    FILIN,PA2       ; INPUT NEW SIGNAL SAMPLE.
        LRLK  AR1,BLK1+SIGNAL; POINT AR1 TO SIGNAL DATA TO PROCESS.
        ZAC                   ; CLEAR THE ACCUMULATOR.
        MPYK  0               ; CLEAR THE P REGISTER.
        RPTK  15              ; REPEAT MACD INSTRUCTION FOR 16 TAPS.
        MACD  >FF00+COEFF,*-  ; MULTIPLY/ACCUMULATE, SAMPLE DELAY.
        APAC                  ; ACCUMULATE THE LAST PRODUCT.
        SACH  FILOUT,1        ; SAVE THE RESULT.
        B     LPTS+OFFSET     ; LOOP TO WAIT FOR NEXT SAMPLE.
PROGE   EQU   $
PROGL   EQU   PROGE-PROG      ; PROGRAM CODE LENGTH.
OFFSET  EQU   >FF00-PROG      ; BASE ADDRESS OFFSET.
*
* COEFFICIENT DATA TO BE LOADED INTO ON-CHIP RAM.
*
COEF    DATA  385,-1196,1839,-2009
        DATA  1390,407,-4403,19958
        DATA  19958,-4403,407,1390
        DATA  -2009,1839,-1196,385
COEFE   EQU   $
COEFL   EQU   COEFE-COEF      ; COEFFICIENT DATA LENGTH.
*
* INTERNAL MEMORY CONSTANTS.
*
BLK0    EQU   >200
BLK1    EQU   >300
*
* DATA PAGE 0 (BLOCK B2) - DATA MEMORY LABELS.
*
        DORG  0
DRR     BSS   1               ; SERIAL PORT DATA RECEIVE REGISTER.
DXR     BSS   1               ; SERIAL PORT DATA TRANSMIT REGISTER.
TIM     BSS   1               ; TIMER REGISTER.
PRD     BSS   1               ; PERIOD REGISTER.
IMR     BSS   1               ; INTERRUPT MASK REGISTER.
GREG    BSS   1               ; GLOBAL MEMORY ALLOCATION REGISTER.
*
* DATA PAGE 4 (BLOCK B0) - DATA MEMORY LABELS.
*
        DORG  0
B0      BSS   PROGL           ; LOCATIONS FOR INTERNAL PROGRAM CODE.
COEFF   BSS   COEFL           ; LOCATIONS FOR COEFFICIENT MEMORY.
*
* DATA PAGE 6 (BLOCK B1) - DATA MEMORY LABELS.
*
        DORG  0
ONE     BSS   1               ; RESERVED FOR DATA VALUE OF 1.
FILOUT  BSS   1               ; FILTER OUTPUT SIGNAL VALUE.
FILIN   BSS   1               ; FILTER INPUT SIGNAL VALUE.
SIGNAL  BES   14              ; LAST SIGNAL DELAY VALUE.
        END
```

5.5 Fundamental Logical and Arithmetic Operations

Although the TMS320C25 instruction set is oriented toward digital signal processing, the same fundamental operations of a general-purpose processor are included. This section explains basic operations of the TMS320C25's Central Arithmetic Logic Unit (CALU), particularly accumulator operations, the status register effect on data processing, and bit manipulation.

The TMS320C25 provides a complete set of logical operations, including AND, OR, XOR, and CMPL (complement) instructions. This enables the device to perform any logical function. These instructions may be used to perform sign magnitude to two's complement or the reverse conversions.

The contents of the accumulator may be stored in data memory using the SACH and SACL instructions or stored in the stack by using the PUSH instruction. The accumulator may be loaded from data memory using the ZALH and ZALS instructions, which zero the accumulator before loading the data value. The ZAC instruction zeroes the accumulator. POP can be used to restore the accumulator contents from the stack.

The accumulator is also affected by the ABS and NEG instructions. ABS replaces the contents of the accumulator with the absolute value of its contents. NEG generates the arithmetic complement of the accumulator in two's-complement form.

5.5.1 Status Register Effect on Data Processing

Three data processing options allow the ALU to automatically suppress sign extension, manage overflow, or scale product accumulations. These options are enabled or disabled through bits in the status registers. These options function in parallel with normal execution of the instructions and cause no additional machine cycles, therefore no performance overhead.

The sign-extension mode option is used to determine whether or not the shifted data values fetched for ALU operations should be sign-extended. The SXM status bit controls this operation. This bit is set to '1' for enabling sign extension using the SSXM instruction, and set to '0' for suppressing sign extension using the RSXM instruction. This operation affects all the instructions that include a shift of the incoming data value (i.e., ADD, ADDT, ADLK, LAC, LACT, LALK, SBLK, SFR, SUB, and SUBT).

The overflow mode option is used to minimize the effects of an arithmetic overflow by forcing the accumulator to saturate at the largest positive value (or in the case of underflow, the largest negative value). The OVM status bit controls this operation. The overflow mode is enabled by setting the OVM bit to a '1' using the SOVM instruction, and reset using the ROVM instruction. This feature affects all arithmetic operations in the ALU.

The product register shift mode option forces all products to be shifted before they are accumulated. The products can be left-shifted one bit to delete the extra sign bit in the multiply of two 16-bit signed numbers. The products can be left-shifted four bits to delete the extra sign bits in multiplying a 16-bit data value by a 13-bit constant. The product shifter can also be used to shift all products six bits to the right to allow up to 128 product accumulations without the threat of an arithmetic overflow, thereby avoiding the overhead of overflow management. The shifter can be disabled to cause no shift in the product when working with integer or 32-bit precision operations. This also maintains compatibility with TMS32010 code. These operations are controlled by the value contained in the PM bits of status register ST1. The PM bits are set using the SPM instruction. This feature affects all the instructions

that use the product of the multiplier (i.e., APAC, LTA, LTD, LTP, LTS, MAC, MACD, MPYA, MPYS, PAC, SPAC, SPH, SPL, SQRA, and SQRS).

5.5.2 Bit Manipulation

The BIT instruction tests any of the 16 bits of the addressed data word. The specified bit is copied into the TC of the status register. The bit tested is specified by a bit code in the opcode of the instruction. Either the BBZ (branch on TC bit = 0) or BBNZ (branch on TC bit = 1) instructions check the bit and allow branching to a service routine.

Bit testing is useful in control applications where a number of states or conditions may be latched externally and read into the TMS320C25 via an IN instruction. At this point, individual bits can be tested and branches taken for appropriate processing.

Since the BIT instruction requires the bit code to be specified with the instruction, it cannot be placed in a loop to test several different bits of a data word or bits determined by prior processing for efficient use. The TMS320C25 also has a BITT instruction in which the bit code is specified in the T register. Since the T register can easily be modified, BITT may be used to test all bits of a data word if placed within a loop or to test a bit location determined by past processing.

Example 5-18. Using BIT and BBZ

```
* THIS ROUTINE USES THE BIT INSTRUCTION TO TEST THE CONDITION
* OF AN EXTERNAL MUX. BIT 4 DETERMINES THE UTILITY OF THE
* REMAINING DATA. IF ZERO, A COUNTER IS INCREMENTED. IF ONE,
* ADDITIONAL PROCESSING OCCURS AND THE COUNTER IS CLEARED.
* THE ROUTINE IS INVOKED WHENEVER A TIMER INTERRUPT OCCURS.
*
TIME    SST    ST0        ; SAVE STATUS REGISTER ST0.
        LDPK   0
        LARP   AR6
        IN     DAT,PA8    ; READ IN VALUE.
        BIT    DAT,>B     ; TEST BIT 4.
        BBZ    INCR       ; BRANCH AND INCREMENT IF POSITIVE.
         .
         .
         .
        LARK   AR6,0      ; CLEAR THE COUNTER.
        LST    ST0        ; RELOAD THE STATUS REGISTER.
        EINT              ; ENABLE INTERRUPTS.
        RET               ; RETURN TO INTERRUPTED ROUTINE.
*
INCR    MAR    *+         ; INCREMENT THE COUNTER.
        LST    ST0        ; RELOAD THE STATUS REGISTER.
        EINT              ; ENABLE INTERRUPTS.
        RET               ; RETURN TO INTERRUPTED ROUTINE.
```

Example 5-19. Using BITT and BBNZ

```
* THIS ROUTINE USES THE BITT INSTRUCTION TO TEST THE CONDITION
* OF AN EXTERNAL MUX. A BIT IN THE MUX IS SIGNIFICANT ONLY
* WHEN PRIOR PROCESSING HAS DESIGNATED THE BIT TO BE ACTIVE.
* INDIVIDUAL PROCESSING WILL TAKE PLACE BASED UPON THE STATE
* OF THE TESTED BIT. THE BITS ARE TESTED EACH TIME A TIMER
* INTERRUPT OCCURS.
*
TIME    SST   ST0           ; SAVE STATUS REGISTER ST0.
        LDPK  0
        LARP  AR5
        LAR   AR5,BCNT      ; LOAD COUNT OF ACTIVE BITS.
        LRLK  AR6,BTBL      ; LOAD THE BIT TABLE ADDRESS.
        IN    DAT,PA8       ; READ IN VALUE.
        B     LTEST,*-,6
TMLOOP  LT    *+,5          ; LOAD BIT CODE.
        BITT  DAT           ; TEST SPECIFIED BIT.
        BBNZ  LTEST         ; BRANCH IF BIT IS ONE.
          .
          .
          .
LTEST   BANZ  TMLOOP,*-,6
        LST   ST0           ; RELOAD THE STATUS REGISTER.
        EINT                ; ENABLE INTERRUPTS.
        RET                 ; RETURN TO INTERRUPTED ROUTINE.
```

5.6 Advanced Arithmetic Operations

The TMS320C25 provides special instructions that facilitate efficient execution of arithmetic-intensive DSP algorithms, such as MACD, SQRA, SUBC, and NORM. Explanations and examples of how to use these instructions with overflow management, and for data moves, multiplications, division, floating-point arithmetic, indexed addressing, and extended-precision arithmetic are included in this section.

5.6.1 Overflow Management

The TMS320C25 has four features that can be used to handle overflow management. These include the branch on overflow conditions, accumulator saturation (overflow mode), product register right shift, and accumulator right shift. These features provide several options for overflow protection within an algorithm.

A program can branch to an error handler routine on an overflow of the accumulator by using the BV (branch on overflow) instruction or bypass an error handler by using the BNV (branch if no overflow) instruction. These instructions can be performed after any ALU operation that may cause an accumulator overflow.

The overflow mode is a feature useful for DSP applications. This mode simulates the saturation effect characteristic of analog systems. When enabled, any overflow in the accumulator results in the accumulator contents being replaced with the largest positive value (>7FFFFFFF) if the overflowed number is positive, or the largest negative value (>80000000) if negative. The overflow mode is controlled by the OVM bit of status register ST0 and can be changed by the SOVM (set overflow mode), ROVM (reset overflow mode), or LST (load status register) instructions. Overflows can be detected in software by testing the OV (overflow) bit in status register ST0. When a branch is used to test the overflow bit, OV is automatically reset. Note that the OV bit does not function as a carry bit. It is set only when the absolute

value of a number is too large to be represented in the accumulator, and it is not reset except by specific instructions.

Another method of overflow management, which applies to multiply-accumulate operations, is the use of the right shifter of the product register. The right shifter, which operates with no cycle overhead, allows up to 128 accumulations without the possibility of an overflow. The least-significant six bits of the product are lost, and the MSBs are filled with sign bits. This feature is initiated by setting the PM bits of status register ST1 to '11' using the SPM or LST1 instructions.

The TMS320C25 also has a right shift of the accumulator (using the SFR instruction) to scale down the accumulator when it nears overflow.

5.6.2 Scaling

Scaling the data coming into the accumulator or already in the accumulator is useful in signal processing algorithms. This is frequently necessary in adaptation or other algorithms that must compute and apply correction factors or normalize intermediate results. Scaling and normalizing are implemented on the TMS320C25 via right and left shifts in the accumulator and shifts of data on the incoming path to the accumulator.

Right and left shifts of the accumulator can be performed using the SFL and SFR instructions. SFL performs a logical left shift. SFR performs logical or arithmetic right shifts depending on the state of the SXM bit in the status register. A '1' in the SXM bit, corresponding to sign-extension enabled, causes an arithmetic shift to be performed.

In addition to the shift instructions, data can be left-shifted 0 to 15 bits when the accumulator is loaded using a LAC instruction, and left-shifted 0 to 7 bits when storing from the accumulator using SACH or SACL instructions. These shifts can be used for loading numbers into the high 16 bits of the accumulator and renormalizing the result of a multiply. The incoming left shift of 0 to 15 bits can be supplied in the instruction itself or can be taken from the lowest four bits of the T register. Left shifts of data fetched from data memory are available for loading the accumulator (LAC/LACT), adding to the accumulator (ADD/ADDT), and subtracting from the accumulator (SUB/SUBT). The contents of the P register may also be shifted prior to accumulation.

5.6.3 Moving Data

Many DSP applications must perform convolution operations or other operations similar in form. These operations require data to be shifted or delayed. The DMOV, LTD, and MACD instructions can perform the needed data moves for convolution.

The data move function allows a word to be copied from the currently addressed data memory location in on-chip RAM to the next higher location while the data from the addressed location is being operated upon (e.g., by the CALU). The data move and the CALU operation are performed in the same cycle. In addition, an ARAU operation may also be performed in the same cycle when using the indirect addressing mode. The data move function is useful in implementing algorithms, such as convolutions and digital filtering, where data is being passed through a time window. It models the z^{-1} delay operation encountered in those applications. The data move function is continuous across the boundary of the on-chip data memory blocks B0, B1, and B2. However, the data move function cannot be used if off-chip memory is referenced.

In Example 5-20, the following equation is implemented:

$$Y(n) = \sum_{k=0}^{2} H(k)\, X(n-k)$$

where the H values stay the same, and the X values are shifted each time the microprocessor performs one of the following series of multiplications (similar to operations performed in FIR filters):

First Series: $Y(2) = (H0)(X2)+(H1)(X1)+(H2)(X0)$
Second Series: $Y(3) = (H0)(X3)+(H1)(X2)+(H2)(X1)$
Third Series: $Y(4) = (H0)(X4)+(H1)(X3)+(H2)(X2)$

The MACD instruction, which combines accumulate and multiply operations with a data move, is tailored to the type of calculation shown in the summation equation above. In order to use MACD, the H values have been stored in block B0, configured as program RAM, and the X values have been read into block B1 of data RAM as shown in Figure 5-2.

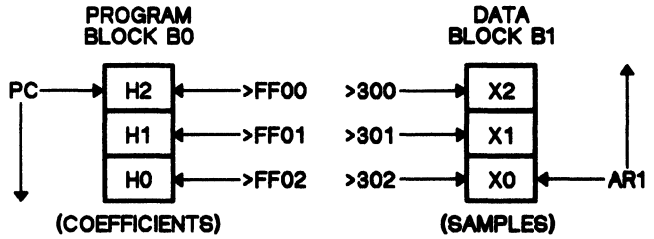


Figure 5-2. MACD Operation

Also in Example 5-20, the summation in the above equation is performed in the reverse order, i.e., from K = 2 to 0, due to the operation of the data move function. This results in the oldest X value being used and discarded first.

If the MACD instruction is replaced with the following two instructions, then the MAC instruction can be utilized with the same results.

```
MAC     *
DMOV    *-
```

In cases where many more than three MACD instructions are required, the RPT or RPTK instructions may be used with MACD, yielding the same computational results but using less assembly code.

Example 5-20. Using MACD for Moving Data

```
* THIS ROUTINE IMPLEMENTS A SINGLE PASS OF A THIRD-ORDER FIR
* FILTER. IT IS ASSUMED THAT THE H AND X VALUES HAVE ALREADY
* BEEN LOADED INTO THEIR RESPECTIVE MEMORY LOCATIONS, THAT
* THE ACCUMULATOR AND P REGISTER ARE BOTH RESET TO ZERO, AND
* THAT AR1 IS POINTING AT X0. NOTE THAT THE MACD INSTRUCTION
* MAY BE USED IN THE REPEAT MODE, BUT IT IS NOT IMPLEMENTED
* HERE.
*
FIR    CNFP              ; CONFIGURE BLOCK B0 AS PROGRAM MEMORY.
       LARP   1          ; AR1 SHOULD POINT AT THE X VALUES.
       MAC    >FF00,*-   ; P = (X0)(H2)
       MACD   >FF01,*-   ; ACC = (X0)(H2)
       MACD   >FF02,*    ; ACC = (X0)(H2) + (X1)(H1)
       APAC              ; ACC = (X0)(H2) + (X1)(H1) + (X2)(H0)
       CNFD              ; CONFIGURE BLOCK B0 AS DATA MEMORY.
       RET               ; RETURN TO MAIN PROGRAM.
```

5.6.4 Multiplication

The TMS320C25 hardware multiplier normally performs two's-complement 16-bit by 16-bit multiplies and produces a 32-bit result in one processor cycle. A single instruction, MPYU, can be used to multiply two 16-bit unsigned numbers. To multiply two operands, one operand must be loaded into the T register (TR). The second operand is moved by the multiply instruction to the multiplier, which then produces the product in the P register (PR). Before another multiply can be performed, the contents of the PR must be moved to the accumulator. A single-multiply program is shown in Example 5-21. By pipelining multiplies and PR moves, most multiply operations can be performed with a single instruction.

A common operation in DSP algorithms is the summation of products. The MAC instruction, normally performed in four cycles, adds the contents of the PR to the accumulator and then simultaneously reads two values and multiplies them. When using the MAC instruction, a data memory value is multiplied by a program memory value. One of the operands can come from block B1 or B2 in on-chip data memory while the other operand may come from block B0. Block B0 must be configured as program memory when it supplies the second operand. Pipelining of the MAC instruction with a repeat instruction results in an execution time for each succeeding multiply-and-accumulate operation of only one cycle.

Example 5-21. Multiply

```
* THIS ROUTINE MULTIPLIES TWO VALUES IN DATA MEMORY LOCATIONS
* >200 AND >201 WITH THE RESULT STORED IN >202 AND >203.
*
MUL  LRLK    AR1,>200     ; POINT AT BLOCK B0.
     LARP    1
     LT      *+           ; GET FIRST VALUE AT >200.
     MPY     *+           ; MULTIPLY BY VALUE AT >201.
     PAC                  ; PUT RESULT IN ACCUMULATOR.
     SACL    *+           ; STORE LOW WORD AT >202.
     SACH    *            ; STORE HIGH WORD AT >203.
     RET                  ; RETURN TO MAIN PROGRAM.
```

The pipelining of the MAC and MACD instructions incurs a certain amount of overhead in execution. In those cases where speed is more critical than program

memory, it may be beneficial to use LTA or LTD and MPY instructions rather than MAC or MACD. Example 5-22 and Example 5-23 show an implementation of multiply-accumulates using the MAC instruction and the LTA-MPY instruction pair, respectively. Figure 5-3 and Figure 5-4 provide graphically the information necessary to determine the efficiency of use for each of the techniques.

Example 5-22. Multiply-Accumulate Using the MAC Instruction

```
*                          clock    total clock   program   total program
*                          cycles     cycles      memory       memory
*
      LARP   AR1              1                     1
      LRLK   AR1,>300         2                     2
      CNFP                    1                     1
      ZAC                     1                     1
      MPYK   0                1                     1
      RPTK   N-1              1                     1
      MAC    >FF00,*+     3 + N                    2
      APAC                    1        11 + N       1           10
```

Example 5-23. Multiply-Accumulate Using the LTA-MPY Instruction Pair

```
*                          clock    total clock   program   total program
*                          cycles     cycles      memory       memory
*
      ZAC                     1                     1
      LT     D1               1                     1
      MPY    C1               1                     1
      LTA    D2               1                     1
      MPY    C2               1                     1
       .
       .                     2 x N                   2 x N
       .
      LTA    DN               1                     1
      MPY    CN               1                     1
      APAC                    1        2 + 2N       1           2 + 2N
```

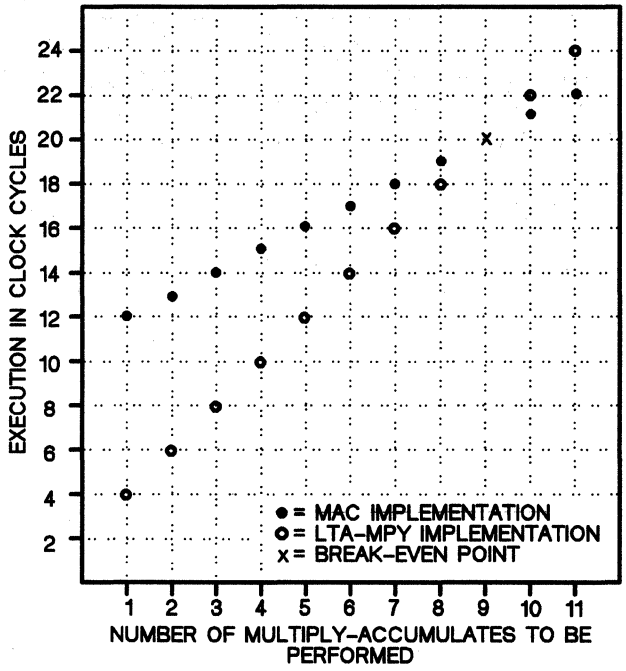


Figure 5-3. Execution Time vs. Number of Multiply-Accumulates

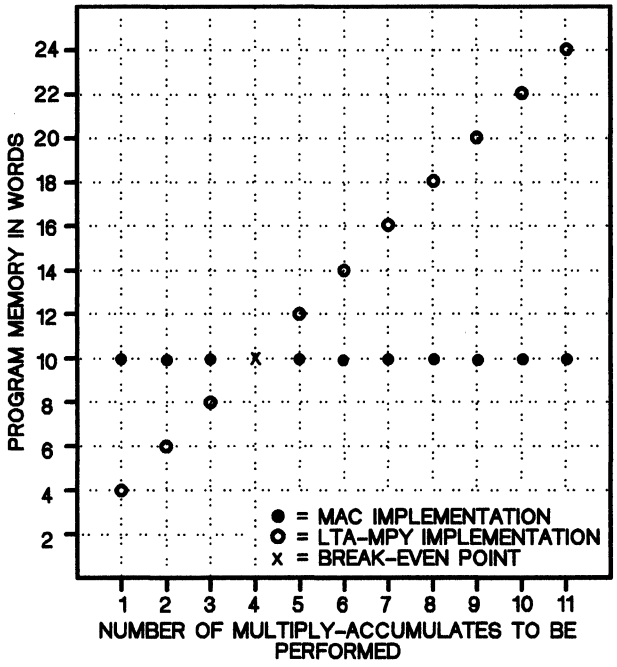


Figure 5-4. Program Memory vs. Number of Multiply-Accumulates

In numerical analysis, it is often necessary to square numbers along with adding or subtracting. The TMS320C25 has two instructions, SQRA and SQRS, that accomplish this in a single machine cycle. The result of the previous operation in the PR is first added to the accumulator if SQRA is used, or subtracted from the accumulator if SQRS is used. Then the data value addressed is squared, and the result is stored in the PR. Example 5-24 uses the SQRA instruction to perform the computation.

Example 5-24. Using SQRA

```
* THIS ROUTINE USES THE SQRA INSTRUCTION TO COMPUTE THE
* SQUARE OF THE DISTANCE BETWEEN TWO POINTS WHERE D**2
* IS DEFINED AS FOLLOWS:
*
*   D**2 = (XA - XB)**2 + (YA - YB)**2
*
DIST    LAC    XA
        SUB    XB
        SACL   XT          ; XT = XA - XB
*
        LAC    YA
        SUB    YB
        SACL   YT          ; YT = YA - YB
*
        SQRA   XT          ; (P) = XT**2
        ZAC                ; (ACC) = 0
        SQRA   YT          ; (P) = YT**2, (ACC) = XT**2
        APAC               ; (ACC) = XT**2 + YT**2 = D**2
*
        RET                ; RETURN TO MAIN PROGRAM.
```

When performing multiply-and-accumulate operations, it may be desirable to shift the product before adding it to the accumulator. This can be accomplished simultaneously with the MAC instruction by using the product shift mode. This mode, controlled by two bits in the PM field of status register ST1, shifts the value from the PR while it is transferred to the accumulator. The contents of the PR are not shifted.

5.6.5 Division

Division is implemented on the TMS320C25 by repeated subtractions using SUBC, a special conditional subtract instruction. Given a 16-bit positive dividend and divisor, the repetition of the SUBC command 16 times produces a 16-bit quotient in the low accumulator and a 16-bit remainder in the high accumulator.

SUBC implements binary division in the same manner as is commonly done in long division. The dividend is shifted until subtracting the divisor no longer produces a negative result. For each subtract that does not produce a negative answer, '1' is put in the LSB of the quotient and then shifted. The shifting of the remainder and quotient after each subtract produces the separation of the quotient and remainder in the low and high halves of the accumulator.

There are similarities between long division and the SUBC method of division. Both methods are used to divide 33 by 5.

LONG DIVISION:

```
                    0000000000000110      Quotient
0000000000000101 )0000000000100001
                      -101
                       110
                      -101
                        11      Remainder
```

SUBC METHOD:

32 HIGH ACC	LOW ACC 0		COMMENT
0000000000000000	0000000000100001	(1)	Dividend is loaded into ACC. The divisor is left-shifted 15 and subtracted from ACC. The subtraction is negative, so discard the result and shift left the ACC one bit.
0000000000000000	0000000001000010	(2)	2nd subtract produces negative answer, so discard result and shift ACC (dividend) left.
0000000000000100	0010000000000000	(14)	14th SUBC command. The result is positive. Shift result left and replace LSB with '1'.
0000000000000011	0100000000000001	(15)	Result is again positive. Shift result left and replace LSB with '1'.
0000000000000001	1000000000000011	(16)	Last subtract. Negative answer, so discard result and shift ACC left.
0000000000000011	0000000000000110		Answer reached after 16 SUBC instructions.
REMAINDER	QUOTIENT		

Note that since the condition of the divisor being less than the shifted dividend is determined by the sign of the result, both the dividend and divisor must be positive when using the SUBC command. Thus, the sign of the quotient must be determined and the quotient computed using the absolute value of the dividend and divisor.

Integer and fractional division can be implemented with the SUBC instruction as shown in Example 5-25 and Example 5-26, 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.

Example 5-25. Using SUBC for Integer Division

```
* THIS ROUTINE IMPLEMENTS INTEGER DIVISION.
*
DN1     LT    NUMERA      ; GET SIGN OF QUOTIENT.
        MPY   DENOM
        PAC
        SACH  TEMSGN      ; SAVE SIGN OF QUOTIENT.
        LAC   DENOM
        ABS
        SACL  DENOM       ; MAKE DENOMINATOR POSITIVE.
        ZALS  NUMERA      ; ALIGN NUMERATOR.
        ABS
*
* IF DIVISOR AND DIVIDEND ARE ALIGNED, DIVISION CAN START
* HERE.
*
        RPTK  15
        SUBC  DENOM       ; 16-CYCLE DIVIDE LOOP.
        SACL  QUOT
        LAC   TEMSGN
        BGEZ  DONE        ; DONE IF SIGN IS POSITIVE.
        ZAC
        SUB   QUOT
        SACL  QUOT        ; NEGATE QUOTIENT IF NEGATIVE.
DONE    LAC   QUOT
        RET               ; RETURN TO MAIN PROGRAM.
```

Example 5-26. Using SUBC for Fractional Division

```
* THIS ROUTINE IMPLEMENTS FRACTIONAL DIVISION.
*
DN1     LT    NUMERA      ; GET SIGN OF QUOTIENT.
        MPY   DENOM
        PAC
        SACH  TEMSGN      ; SAVE SIGN OF QUOTIENT.
        LAC   DENOM
        ABS
        SACL  DENOM       ; MAKE DENOMINATOR POSITIVE.
        ZALH  NUMERA      ; ALIGN NUMERATOR.
        ABS
*
* IF DIVISOR AND DIVIDEND ARE ALIGNED, DIVISION CAN START
* HERE.
*
        RPTK  14
        SUBC  DENOM       ; 15-CYCLE DIVIDE LOOP.
        SACL  QUOT
        LAC   TEMSGN
        BGEZ  DONE        ; DONE IF SIGN IS POSITIVE.
        ZAC
        SUB   QUOT
        SACL  QUOT        ; NEGATE QUOTIENT IF NEGATIVE.
DONE    LAC   QUOT
        RET               ; RETURN TO MAIN PROGRAM.
```

5.6.6 Floating-Point Arithmetic

Floating-point numbers are often represented on microprocessors in a two-word format of mantissa and exponent. The mantissa is stored in one word. The exponent, the second word, indicates how many bit positions from the left the decimal point is located. If the mantissa is 16 bits, a 4-bit exponent is sufficient to express the location of the decimal point. Because of its 16-bit word size, the 16/4-bit floating-point format functions most efficiently on the TMS320C25.

Operations in the TMS320C25's central ALU are performed in two's-complement fixed-point notation. To implement floating-point arithmetic, operands must be converted to fixed point for arithmetic operations, and then converted back to floating point.

Conversion to floating-point notation is performed by normalizing the input data (i.e., shifting the MSB of the data word into the MSB of the internal memory word). The exponent word then indicates how many shifts are required. To multiply two floating-point numbers, the mantissas are multiplied and the exponents added. The resulting mantissa must be renormalized. (Since the input operands are normalized, no more then one left shift is required to normalize the result.)

Floating-point addition or subtraction requires shifting the mantissa so that the exponents of the two operands match. The difference between the exponents is used to left-shift the lower power operand before adding. Then, the output of the add must be renormalized.

TMS320C25 instructions useful in floating-point operations are the NORM, LACT, ADDT, and SUBT instructions. NORM (see Example 5-7) may be used to convert fixed-point numbers to floating-point. LACT may be used to convert back to fixed-point numbers. Addition and subtraction can be computed in floating point using ADDT and SUBT.

Example 5-27 performs a floating-point multiply. The mantissas are assumed to be in Q15 format. Q15, one of the various types of Q format, is a number representation commonly used when performing operations on non-integer numbers. In Q format, the Q number (15 in Q15) denotes how many digits are located to the right of the decimal point. A 16-bit number in Q15 format, therefore, has an assumed decimal point immediately to the right of the most significant bit. Since the most significant bit constitutes the sign of the number, then numbers represented in Q15 may take on values from +1 (represented by +0.9999999...) to -1 (represented by -0.9999999...).

Example 5-27. Using NORM for Floating-Point Multiply

```
* THIS SUBROUTINE PERFORMS A FLOATING-POINT MULTIPLY USING
* THE NORM INSTRUCTION. THE INPUTS AND OUTPUTS ARE OF THE
* FORM:
*
*       C = MC * 2**EC
*
* SINCE THE MANTISSAS, MA AND MB, ARE NORMALIZED, MC CAN BE
* NORMALIZED WITH A LEFT SHIFT OF EITHER 0 OR 1 IN THE
* ACCUMULATOR. THE EXPONENT OF THE RESULT IS ADJUSTED
* APPROPRIATELY. FOR EXAMPLE, MULTIPLICATION OF THE TWO
* NUMBERS A AND B, WHERE A = 0.1 * 2**2 AND B = 0.1 * 2**4,
* PROCEEDS AS FOLLOWS:
*
*       1) A * B = 0.01 * 2**6
*       2) A * B = 0.1 * 2**5     (NORMALIZED RESULT)
*
MULT    LAC    EA
        ADD    EB           ; EC = EXPONENT OF RESULT BEFORE
        SACL   EC           ; NORMALIZATION.
        LT     MA
        MPY    MB
        PAC                 ; (ACC) = MA * MB
*
        SFL                 ; TAKES CARE OF REDUNDANT SIGN BIT.
        LARP   AR5
        LAR    AR5,EC       ; AR5 IS INITIALIZED WITH EC.
*
        NORM   *-           ; FINDS MSB AND MODIFIES AR5.
*
        SACH   MC           ; MC = MA * MB (NORMALIZED)
        SAR    AR5,EC
        RET                 ; RETURN TO MAIN PROGRAM.
```

Floating-point implementation programs often require denormalization as well as normalization to return results in a 16-bit format. Example 5-28 is tailored for denormalizing numbers that were normalized using the NORM instruction. This program assumes that the mantissa is in the accumulator and the exponent is in AR5, which is the format of the NORM instruction after execution.

Example 5-28. Using LACT for Denormalization

```
* THIS ROUTINE DENORMALIZES NUMBERS NORMALIZED BY THE NORM
* INSTRUCTION. THE DENORMALIZED NUMBER WILL BE IN THE
* ACCUMULATOR.
*
DENORM  LARP   1           ; USE AR1 TO POINT AT BLOCK B0.
        LRLK   AR1,>200
        SAR    AR5,*+      ; STORE EXPONENT AT >200.
        SACH   *-          ; STORE MANTISSA AT >201.
*
* SUBTRACT EXPONENT FROM 16 TO DETERMINE THE NUMBER OF SHIFTS
* REQUIRED TO DENORMALIZE.
*
        LAC    *           ; LOAD ACCUMULATOR WITH EXPONENT.
        BZ     OUT         ; CHECK FOR ZERO EXPONENT.
        LT     *+
        LACT   *           ; DENORMALIZE NUMBER.
        RET                ; RETURN TO MAIN PROGRAM.
OUT     MAR    *+          ; POINT TO MANTISSA.
        ZALH   *           ; LOAD ACCUMULATOR WITH RESULT.
        RET                ; RETURN TO MAIN PROGRAM.
```

5.6.7 Indexed Addressing

The Auxiliary Register Arithmetic Unit (ARAU) allows the the next indirect address to be calculated using increment/decrement calculations or indexed addressing in parallel to the current arithmetic operation. For example, in the multiplication of two matrices, the operation requires addressing across the rows (incrementing the address by one) or down the columns (incrementing by n). Example 5-29 gives the code for multiplying a row times a column of two 10 x 10 matrices. The first matrix resides in data RAM block B1, and the second matrix resides in block B0.

Example 5-29. Row Times Column

```
   LARK  0,>A       ; SET INDEX TO 10.
   LARP  1          ; USE AR1 FOR ADDRESSING THE COLUMN.
   LRLK  1,>300     ; POINT AR1 TO THE START OF BLOCK B1.
   CNFP             ; SET B0 TO PROGRAM ADDRESS FOR PIPELINING.
   ZAC              ; INITIALIZE THE ACCUMULATOR.
   MPYK  0          ; CLEAR THE PRODUCT REGISTER.
   RPTK  9          ; REPEAT 10 TIMES AS DIMENSION OF MATRIX.
   MAC   >F000,*0+  ; MULTIPLY ROW ELEMENT TIMES COLUMN ELEMENT.
   APAC             ; EXECUTE FINAL ACCUMULATION.
*                   ; ACCUMULATOR CONTAINS PRODUCT ELEMENT.
```

The algorithm in Example 5-29 executes in 22 machine cycles. The key to this performance is the parallel addressing of both multiplicands simultaneously. The operation is made possible by the use of the data bus to fetch one multiplicand and the program bus to fetch the other. The auxiliary register indexes down the column of one matrix while the PC generates incremental addressing of each row of the other matrix. Each cycle of the repeat loop performs the following operations:

1) Accumulates the previous product,
2) Multiplies the row element times the column element,
3) Increments the row address, and
4) Indexes the column address.

5.6.8 Extended-Precision Arithmetic

Numerical analysis, floating-point computations, or other operations may require arithmetic to be executed with more than 32 bits of precision. Two features of the TMS320C25 help to make extended-precision calculations more efficient. One of the features is the carry status bit. This bit is affected by all arithmetic operations of the accumulator (ABS, ADD, ADDH, ADDK, ADDS, ADDT, ADLK, APAC, LTA, LTD, LTS, MAC, MACD, MPYA, MPYS, NEG, SBLK, SPAC, SQRA, SQRS, SUB, SUBB, SUBC, SUBH, SUBK, SUBS, and SUBT). The carry bit is also affected by the rotate and shift accumulator instructions (ROL, ROR, SFL, and SFR) or may be explicitly modified by the load status register ST1 (LST1), reset carry (RC), and set carry (SC) 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. Note that this means that some additional code may be required if overflow of the most significant portion of the result is expected.

The carry bit is set whenever the addition of a value from the input scaling shifter or the P register to the accumulator contents generates a carry out of bit 31. Otherwise, the carry bit is reset since the carry out of bit 31 is a zero. One exception to this case is the ADDH instruction which can only set the carry bit. This allows the accumulation to generate the proper single carry when either the addition to the lower or upper half of the accumulator actually causes the carry. The following examples help to demonstrate the significance of the carry bit for additions:

```
C  MSB          LSB
X  FFFF FFFF  ACC
  +         1
1  0000 0000

X  7FFF FFFF  ACC
  +         1
0  8000 0000

X  8000 0000  ACC
  +         1
0  8000 0001

1  0000 0000  ACC
  +         0  (ADDC)
0  0000 0001

1  8000 FFFF  ACC
  +0000 0000  (ADDH)
1  8000 FFFF
```

```
C  MSB          LSB
X  FFFF FFFF  ACC
  +FFFF FFFF
1  FFFF FFFE

X  7FFF FFFF  ACC
  +FFFF FFFF
1  7FFF FFFE

X  8000 0000  ACC
  +FFFF FFFF
1  7FFF FFFF

1  FFFF FFFF  ACC
  +         0  (ADDC)
1  0000 0000

1  8000 FFFF  ACC
  +7FFF 0000  (ADDH)
1  FFFF FFFF
```

Example 5-30 shows an implementation of two 64-bit numbers added to each other to obtain a 64-bit result.

Example 5-30. 64-Bit Addition

```
* TWO 64-BIT NUMBERS ARE ADDED TO EACH OTHER PRODUCING A
* 64-BIT RESULT. THE NUMBERS X (X3,X2,X1,X0) AND Y
* (Y3,Y2,Y1,Y0) ARE ADDED RESULTING IN W (W3,W2,W1,W0).
*
*            X3 X2 X1 X0
*          + Y3 Y2 Y1 Y0
*            -----------
*            W3 W2 W1 W0
*
ADD64   ZALH   X1          ; ACC = X1 00
        ADDS   X0          ; ACC = X1 X0
        ADDS   Y0          ; ACC = X1 X0 + 00 Y0
        ADDH   Y1          ; ACC = X1 X0 + Y1 Y0 = W1 W0
        SACL   W0
        SACH   W1
        ZALH   X3          ; ACC = X3 00
        ADDC   X2          ; ACC = X3 X2 + C
        ADDS   Y2          ; ACC = X3 X2 + 00 Y2 + C
        ADDH   Y3          ; ACC = X3 X2 + Y3 Y2 + C = W3 W2
        SACL   W2
        SACH   W3
        RET
```

In a similar way, the carry bit is reset whenever the input scaling shifter or the P-register value subtracted from the accumulator contents generates a borrow into bit 31. Otherwise, the carry bit is set since no borrow into bit 31 is required. One exception to this case is the SUBH instruction which can only reset the carry bit. This allows the generation of the proper single carry when either the subtraction from the lower or upper half of the accumulator actually causes the borrow. The following examples help to demonstrate the significance of the carry bit for subtractions:

```
C  MSB          LSB
X  0000 0000 ACC
 - ---------- 1
0  FFFF FFFF

X  7FFF FFFF ACC
 - ---------- 1
1  7FFF FFFE

X  8000 0000 ACC
 - ---------- 1
1  7FFF FFFF

0  0000 0000 ACC
 - ---------- 0 (SUBB)
0  FFFF FFFF

0  8000 FFFF ACC
 - 0001 0000 (SUBH)
0  7FFF FFFF
```

```
C  MSB          LSB
X  0000 0000 ACC
 - FFFF FFFF
0  0000 0001

X  7FFF FFFF ACC
 - FFFF FFFF
0  8000 0000

X  8000 0000 ACC
 - FFFF FFFF
0  8000 0001

0  FFFF FFFF ACC
 - ---------- 0 (SUBB)
1  FFFF FFFE

0  8000 FFFF ACC
 - FFFF 0000 (SUBH)
0  8000 FFFF
```

Example 5-31 provides the code for the implementation of two 64-bit numbers subtracted to obtain a 64-bit number.

Example 5-31. 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).
*
*            X3 X2 X1 X0
*          - Y3 Y2 Y1 Y0
*            -----------
*            W3 W2 W1 W0
*
SUB64   ZALH   X1          ; ACC = X1 00
        ADDS   X0          ; ACC = X1 X0
        SUBS   Y0          ; ACC = X1 X0 - 00 Y0
        SUBH   Y1          ; ACC = X1 X0 - Y1 Y0 = W1 W0
        SACL   W0
        SACH   W1
        ZALS   X2          ; ACC = 00 X2
        SUBB   Y2          ; ACC = 00 X2 - 00 Y2 - C
        ADDH   X3          ; ACC = X3 X2 - 00 Y2 - C
        SUBH   Y3          ; ACC = X3 X2 - Y3 Y2 - C = W3 W2
        SACL   W2
        SACH   W3
        RET
```

The second feature of the TMS320C25 aiding in extended-precision calculations is the MPYU (unsigned multiply) instruction. The MPYU instruction allows two unsigned 16-bit numbers to be multiplied and the 32-bit result placed in the product register in a single cycle. Efficiency is gained by the ability to generate partial products from the 16-bit portions of a 32-bit or larger value instead of having to split the value into 15-bit or smaller parts. Example 5-32 shows an implementation of multiplying two 32-bit numbers to obtain a 64-bit result.

Example 5-32. 32 x 32-Bit Multiplication

```
* TWO 32-BIT NUMBERS ARE MULTIPLIED, PRODUCING A 64-BIT
* RESULT. THE NUMBERS X (X1,X0) AND Y (Y1,Y0) ARE
* MULTIPLIED RESULTING IN W (W3,W2,W1,W0).
*
*                  X1 X0
*           x      Y1 Y0
*              -----------
*                  X0*Y0
*               X1*Y0
*               X0*Y1
*            X1*Y1
*            -----------
*            W3 W2 W1 W0
*
* DETERMINE THE SIGN OF THE PRODUCT.
*
MPY32   ZALS   X1        ; ACCL = SXXX XXXX XXXX XXXX
        XOR    Y1        ; ACCL = S--- ---- ---- ----
        SACH   SIGN      ; SAVE THE PRODUCT SIGN 0=+, 1=-.
*
* TAKE THE ABSOLUTE VALUE OF BOTH X AND Y.
*
ABSX    ZALH   X1        ; ACC = X1 00
        ADDS   X0        ; ACC = X1 X0
        ABS
        SACH   X1        ; SAVE |X1|.
        SACL   X0        ; SAVE |X0|.
ABSY    ZALH   Y1        ; ACC = Y1 00
        ADDS   Y0        ; ACC = Y1 X0
        ABS
        SACH   Y1        ; SAVE |Y1|.
        SACL   Y0        ; SAVE |Y0|.
*
* MULTIPLY |X| AND |Y| TO PRODUCE |W|.
*
MULT    LT     X0        ; T = X0
        MPYU   Y0        ; T = X0, P = X0*Y0
        SPL    W1        ; SAVE |W0|.
        SPH    W0        ; SAVE PARTIAL |W1|.
        MPYU   Y1        ; T = X0, P = X0*Y1
        LTP    X1        ; T = X1, P = X0*Y1, ACC = X0*Y1
        MPYU   Y0        ; T = X1, P = X1*Y0, ACC = X0*Y1
        ADDS   W1        ; T = X1, P = X1*Y0,
*                        ; ACC = X0*Y1 + X0*Y0*2**-16
        MPYA   Y1        ; T = X1, P = X1*Y1,
*                        ; ACC = X1*Y0 + X0*Y1 + X0*Y0*2**-16
        SACL   W1        ; SAVE |W1|.
        SACH   W2        ; SAVE PARTIAL |W2|.
        ZALS   W2        ; P = X1*Y1,
*                        ; ACC = (X1*Y0 + X0*Y1)*2**-16
        BNC    SUM       ; TEST FOR CARRY FROM W2.
        ADDH   ONE
SUM     APAC             ; ACC = X1*Y1 + (X1*Y0 + X0*Y1)*2**-16
        SACL   W2        ; SAVE |W2|.
        SACH   W3        ; SAVE |W3|.
```

```
*
* TEST THE SIGN OF THE PRODUCT; NEGATE IF NEGATIVE.
*
        LAC     SIGN
        BZ      DONE        ; RETURN IF POSITIVE.
*
        ZALH    W1          ; ACC = |W1 00|
        ADDS    W0          ; ACC = |W1 W0|
        CMPL
        ADD     ONE         ; ACC = W1 W0 AND CARRY GENERATION
        SACL    W0          ; SAVE W0.
        SACH    W1          ; SAVE W1.
        ZALS    W2          ; ACC = |00 W2|
        ADDH    W3          ; ACC = |W3 W2|
        CMPL
        ADDC    ONE         ; ACC = W3 W2
        SACL    W2          ; SAVE W2.
        SACH    W3          ; SAVE W3.
*
DONE    RET
```

5.7 Application-Oriented Operations

The TMS320C25 has been designed to provide efficient implementations of many common digital signal processing algorithms. The architecture supporting these design features was discussed in Section 3. In general, the features provide efficient solutions to numerically intensive problems usually characterized by multiply/accumulates. Some device-specific features that aid in the implementation of specific algorithms are discussed in this section.

5.7.1 Companding

Applications implemented on the TMS320C25 include filtering, FFTs, and more complex processes comprised primarily of filtering and FFTs. These applications require I/O performed either in parallel or serial. Hardware requirements for I/O are discussed in Sections 3 and 6.

In the area of telecommunications, one of the primary concerns is the I/O bandwidth in the communications channel. One way to minimize this bandwidth is by companding. Two modes commonly used are A-law and μ-law companding. Detailed descriptions of companding are found in the application report available from Texas Instruments (see the book, *Digital Signal Processing Applications with the TMS320 Family*).

The technique of companding allows the digital sample information corresponding to a 13-bit dynamic range to be transmitted as 8-bit data. For processing in the TMS320C25, it is necessary to convert the 8-bit (logarithmic) sign-magnitude data to a 16-bit two's-complement (linear) format. Prior to output, the linear result must be converted to the compressed or companded format. Table lookup or conversion subroutines may be used to implement these functions.

In expanding from the 8-bit data to the 13-bit linear representation, table lookup is very effective since the table length is only 256 words. This is especially true for a microcomputer design since the TMS320C25 has 4K words of mask-programmable ROM. The table lookup technique requires three instructions (four words of program

memory), one data memory location, 256 words of table memory, and seven instruction cycles (program in on-chip ROM) to execute.

```
LAC    SAMPLE    ; LOAD 8-BIT DATA.
ADLK   MUTABL    ; ADD THE CONVERSION TABLE BASE ADDRESS.
TBLR   SAMPLE    ; READ THE CORRESPONDING LINEAR VALUE.
```

The above conversion could be programmed as a subroutine. This would eliminate the need for a table, but would increase execution time and require additional data memory locations.

When the output data has been determined in a system transmitting companded data, a compression of the data must be performed. The compression reduces the data back to the 8-bit format. Unless memory for a table of length 16384 is acceptable, the table lookup approach must be abandoned for conversion routines. Details of these implementations may be found in the application report on companding.

5.7.2 Filtering

Digital filters are a common requirement for digital signal processing systems. The filters fall into two basic categories: Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters. For either category of filter, the coefficients of the filter (weighting factors) may be fixed or adapted during the course of the signal processing. The theory and implementation of digital filters has been presented and discussed in an application report (see the book, *Digital Signal Processing Applications with the TMS320 Family*). The TMS320C25 reduces the execution time of all filters by virtue of its 100-ns instruction cycle time.

IIR filters benefit from the 100-ns instruction cycle time of the TMS320C25. IIR filters typically require fewer multiply/accumulates. Correspondingly, the amount of data memory for samples and coefficients is not usually the limiting factor. Because of sensitivity to quantization of the coefficients themselves, IIR filters are usually implemented in cascaded second-order sections. This translates to instruction code consisting of LTDs and MPYs rather than MACDs.

FIR filters also benefit from the faster instruction cycle time. In addition, an FIR filter requires many more multiply/accumulates than does the IIR filter with equivalent sharpness at the cutoff frequencies and distortion and attenuation in the passbands and stopbands. The TMS320C25 can help solve this problem by making longer filters feasible to implement. This is accomplished by allowing the coefficients to be fetched from program memory at the same time as a sample is being fetched from data memory. The simple implementation of this process uses the MACD instruction with the RPT/RPTK instruction.

```
RPTK   255
MACD   *-,COEFFP
```

The coefficients may be stored anywhere in program memory (reconfigurable on-chip RAM, on-chip ROM, or external memories). When the coefficients are stored in on-chip ROM or externally, the entire on-chip data RAM may be used to store the sample sequence. Ultimately, this allows filters of up to 512 taps to be implemented on the TMS320C25. Execution of the filter will be at full speed or 100 ns per tap as long as the memory supports full-speed execution.

Up to this point, it has been assumed that the filter coefficients are themselves fixed. If the coefficients are adapted or updated with time, then another factor impacts the computational capacity. The second factor is the requirement to adapt each of the coefficients, usually with each sample. New instructions (MPYA or MPYS and ZALR) on the TMS320C25 aid with this adaptation to reduce the execution time. A means

of adapting the coefficients is the Least-Mean-Square (LMS) algorithm given by the following equation:

$$b_k(i+1) = b_k(i) + 2B\ e(i)\ x(i-k)$$

$$\text{where } e(i) = x(i) - y(i)$$

$$\text{and } y(i) = \sum_{k=0}^{N-1} b_k\ x(i-k)$$

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. Thus, the computational requirement has become one multiply/accumulate plus rounding. Without the new instructions, the adaptation of each coefficient is five instructions corresponding to five clock cycles. This is shown in the following instruction sequence:

```
      LRLK  AR2,COEFFD  ; LOAD ADDRESS OF COEFFICIENTS.
      LRLK  AR3,LASTAP  ; LOAD ADDRESS OF DATA SAMPLES.
      LARP  AR2
      LT    ERRF        ; errf = 2*B*e(i)
        .
        .
        .
      ZALH  *,AR3       ; ACC = bk(i)*2**16
      ADD   ONE,15      ; ACC = bk(i)*2**16 + 2**15
      MPY   *-,AR2
      APAC              ; ACC = bk(i)*2**16 + errf*x(i-k) + 2**15
      SACH  *+          ; SAVE bk(i+1).
        .
        .
        .
```

When the MPYA and ZALR instructions are used, the adaptation reduces to three instructions corresponding to three clock cycles, as shown in the following instruction sequence. Note that the processing order has been slightly changed to incorporate the use of the MPYA instruction. This is due to the fact that the accumulation performed by the MPYA is the accumulation of the previous product.

```
      LRLK  AR2,COEFFD  ; LOAD ADDRESS OF COEFFICIENTS.
      LRLK  AR3,LASTAP  ; LOAD ADDRESS OF DATA SAMPLES.
      LARP  AR2
      LT    ERRF        ; errf = 2*B*e(i)
        .
        .
        .
      ZALR  *,AR3       ; ACC = bk(i)*2**16 + 2**15
      MPYA  *-,AR2      ; ACC = bk(i)*2**16 + errf*x(i-k) + 2**15
*                       ; PREG = errf*x(i-k+1)
      SACH  *+          ; SAVE bk(i+1).
        .
        .
        .
```

Example 5-33 shows a routine to filter a signal and update the coefficients. The total execution time of the routine is 33 + 4n where n is the filter length. Data and program memory requirements are 5 + 2n and 30 + 3n words, respectively. Note that for adaptive filters, the filter length is restricted both by execution time as well as memory. There is obviously more processing to be completed per sample due to the adaptation, and the adaptation itself dictates that the coefficients be stored in the reconfigurable block of on-chip RAM. Thus, the practical limit of an adaptive filter with no external data memory is 256 taps.

Example 5-33. 256-Tap Adaptive FIR Filter

```
        TITL  'ADAPTIVE FILTER'
        DEF   ADPFIR
        DEF   X,Y
*
* THIS 256-TAP ADAPTIVE FIR FILTER USES ON-CHIP MEMORY BLOCK
* B0 FOR COEFFICIENTS AND BLOCK B1 FOR DATA SAMPLES. THE
* NEWEST INPUT SHOULD BE IN MEMORY LOCATION X WHEN CALLED.
* THE OUTPUT WILL BE IN MEMORY LOCATION Y WHEN RETURNED.
* ASSUME THAT THE DATA PAGE IS 0 WHEN THE ROUTINE IS CALLED.
*
COEFFP EQU   >FF00          ; B0 PROGRAM MEMORY ADDRESS
COEFFD EQU   >0200          ; B0 DATA MEMORY ADDRESS
*
ONE    EQU   >7A            ; CONSTANT ONE          (DP=0)
BETA   EQU   >7B            ; ADAPTATION CONSTANT   (DP=0)
ERR    EQU   >7C            ; SIGNAL ERROR          (DP=0)
ERRF   EQU   >7D            ; ERROR FUNCTION        (DP=0)
Y      EQU   >7E            ; FILTER OUTPUT         (DP=0)
X      EQU   >7F            ; NEWEST DATA SAMPLE    (DP=0)
FRSTAP EQU   >0300          ; NEXT NEWEST DATA SAMPLE
LASTAP EQU   >03FF          ; OLDEST DATA SAMPLE
*
* FINITE IMPULSE  RESPONSE (FIR) FILTER.
*
ADPFIR CNFP                 ; CONFIGURE B0 AS PROGRAM:
       MPYK  0              ; Clear the P register.
       LAC   ONE,14         ; Load output rounding bit.
       LARP  AR3
       LRLK  AR3,LASTAP     ; Point to the oldest sample.
FIR    RPTK  255
       MACD  COEFFP,*-      ; 256-tap FIR filter.
       CNFD                 ; CONFIGURE B0 AS DATA:
       APAC
       SACH  Y,1            ; Store the filter output.
       NEG
       ADD   X,15           ; Add the newest input.
       SACH  ERR,1          ; err(i) = x(i) - y(i)
*
* LMS ADAPTATION OF FILTER COEFFICIENTS.
*
       LT    ERR
       MPY   BETA
       PAC                  ; errf(i) = beta * err(i)
       ADD   ONE,14         ; ROUND THE RESULT.
       SACH  ERRF,1
*
       MAR   *+
       LAC   X              ; INCLUDE NEWEST SAMPLE.
       SACL  *
*
       LRLK  AR2,COEFFD     ; POINT TO THE COEFFICIENTS.
       LRLK  AR3,LASTAP     ; POINT TO THE DATA SAMPLES.
       LT    ERRF
       MPY   *-,AR2         ; P = 2*beta*err(i)*x(i-255)
```

```
*
ADAPT   ZALR  *,AR3          ; LOAD ACCH WITH b255(i) & ROUND.
        MPYA  *-,AR2         ; b255(i+1) = b255(i) + P
*                            ; P = 2*beta*err(i)*x(i-254)
        SACH  *+,0,AR1       ; STORE b255(i+1).
*
        ZALR  *,AR3          ; LOAD ACCH WITH b254(i) & ROUND.
        MPYA  *-,AR2         ; b254(i+1) = b254(i) + P
*                            ; P = 2*beta*err(i)*x(i-253)
        SACH  *+,0,AR1       ; STORE b254(i+1).
*
        ZALR  *,AR3          ; LOAD ACCH WITH b253(i) & ROUND.
        MPYA  *-,AR2         ; b253(i+1) = b253(i) + P
*                            ; P = 2*beta*err(i)*x(i-252)
        SACH  *+,0,AR1       ; STORE b253(i+1).
          .
          .
          .
*
        ZALR  *,AR3          ; LOAD ACCH WITH b1(i) & ROUND.
        MPYA  *-,AR2         ; b1(i+1) = b1(i) + P
*                            ; P = 2*beta*err(i)*x(i-0)
        SACH  *+,0,AR1       ; STORE b1(i+1).
*
        ZALR  *,AR3          ; LOAD ACCH WITH b0(i) & ROUND.
        APAC  *-,AR2         ; b0(i+1) = b0(i) + P
        SACH  *+,0,AR1       ; STORE b0(i+1).
*
        RET                  ; RETURN TO CALLING ROUTINE.
```

5.7.3 Fast Fourier Transforms (FFT)

Fourier transforms are an important tool often used in digital signal processing systems. The purpose of the transform is to convert information from the time domain to the frequency domain. The inverse Fourier transform converts information back to the time domain from the frequency domain. Implementations of Fourier transforms that are computationally efficient are known as Fast Fourier Transforms (FFTs). The theory and implementation of FFTs on the TMS32020 has been discussed in an application report in the book, *Digital Signal Processing Applications with the TMS320 Family*. The TMS320C25 reduces the execution time of all FFTs by virtue of its 100-ns instruction cycle time.

In addition to the shorter cycle time, an addressing feature has been added to the TMS320C25 which provides execution speed and program memory enhancements for radix-2 FFTs. As demonstrated in Figure 5-5 and Figure 5-6 the inputs or outputs of an FFT are not in sequential order, i.e., they are scrambled. The scrambling of the data addressing is a direct result of the radix-2 FFT derivation. Observation of the figures and the relationship of the input and output addressing in each case reveal that the address indexing is a bit-reversed order, as shown in Table 5-1. As a result, either the data input sequence or the data output sequence must be scrambled in association with the execution of the FFT.

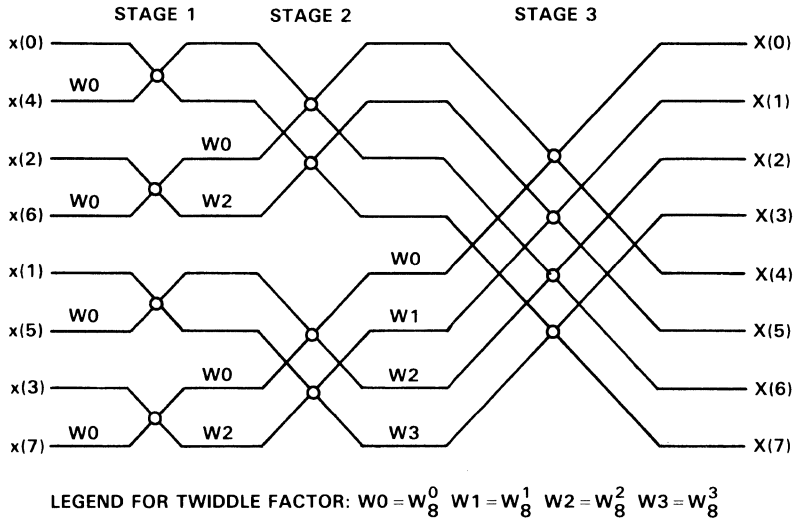


Figure 5-5. An In-Place DIT FFT with In-Order Outputs and Bit-Reversed Inputs

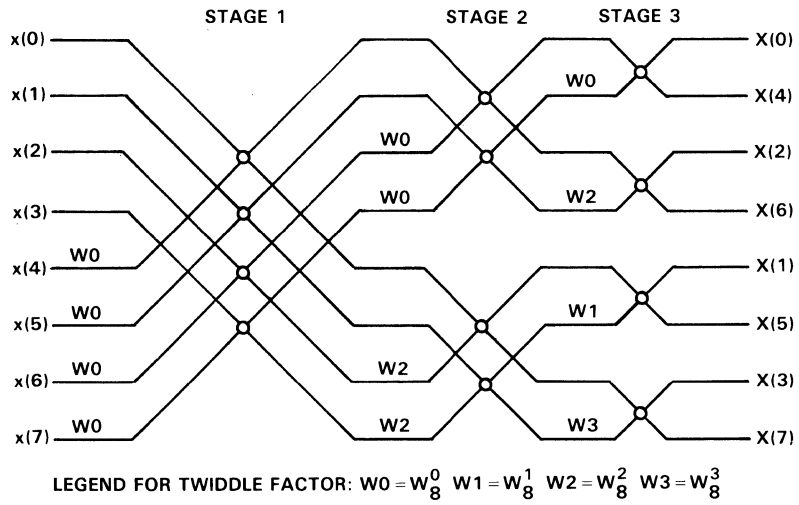


Figure 5-6. An In-Place DIT FFT with In-Order Inputs but Bit-Reversed Outputs

Table 5-1. 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

On the TMS32020 the bit reversal was handled by loading the accumulator with pairs of points that needed to be swapped and then storing them back in the swapped locations. A new addressing feature that uses reverse carry-bit propagation allows the TMS320C25 to scramble the inputs or outputs while it is performing the I/O. The addressing mode is part of the indirect addressing implemented with the auxiliary registers and the associated arithmetic unit. In this mode (a derivative of indexed addressing), a value (index) contained in AR0 is either added or subtracted from the auxiliary register being pointed to by the ARP. However, instead of propagating the carry bit in the forward direction, it is propagated in the reverse direction. The result is a scrambling in the address access.

The procedure for generating the bit-reversal address sequence is to load AR0 with a value corresponding to the length of the FFT and to load another auxiliary register, e.g., AR1, with the base address of the data array. Implementations of FFTs involve complex arithmetic; as a result, there are two data memory locations (one real and one imaginary) associated with every data sample. Generally, the samples are stored in memory in pairs with the real part in the even address locations and the imaginary part in the odd address location. This means that the offset from the base address for any given sample is twice the sample index. Real input data is easily transferred into the data memory and stored in the scrambled order, with every other location in the data memory representing the imaginary part of the data.

The following list shows the contents of auxiliary register AR1 when AR0 is initialized with a value of 8 (8-point FFT) and when data is being transferred by the code that follows.

```
        MSB                                         LSB
AR0:    0 0 0 0   0 0 0 0   0 0 0 0   1 0 0 0   8-Point FFT

AR1:    0 0 0 0   0 0 1 0   0 0 0 0   0 0 0 0   Base Address
```

```
        RPTK   7
        IN     *BR0+,PA0
```

```
AR1:    0 0 0 0   0 0 1 0   0 0 0 0   0 0 0 0   XR(0)

AR1:    0 0 0 0   0 0 1 0   0 0 0 0   1 0 0 0   XR(4)

AR1:    0 0 0 0   0 0 1 0   0 0 0 0   0 1 0 0   XR(2)

AR1:    0 0 0 0   0 0 1 0   0 0 0 0   1 1 0 0   XR(6)

AR1:    0 0 0 0   0 0 1 0   0 0 0 0   0 0 1 0   XR(1)

AR1:    0 0 0 0   0 0 1 0   0 0 0 0   1 0 1 0   XR(5)

AR1:    0 0 0 0   0 0 1 0   0 0 0 0   0 1 1 0   XR(3)

AR1:    0 0 0 0   0 0 1 0   0 0 0 0   1 1 1 0   XR(7)
```

Example 5-34 consists of lists of macros used in the implementation of FFTs. The first macro implements the bit reversal in the way necessary for the TMS32020 and is not necessary for implementations on the TMS320C25.

Example 5-34. FFT Macros

```
BITREV $MACRO  PR,PI,QR,QI
*
* BIT REVERSAL CODE - SWAP PR AND QR, SWAP PI AND QI.
*
       ZALH    :PR:
       ADDS    :QR:
       SACL    :PR:
       SACH    :QR:
       ZALH    :PI:
       ADDS    :QI:
       SACL    :PI:
       SACH    :QI:
       $END
*
COMBO  $MACRO  R1,I1,R2,I2,R3,I3,R4,I4
*
* CALCULATE PARTIAL TERMS FOR R3, R4, I3, AND I4.
*
       LAC     :R3:,14   ACC  := (1/4)(R3)
       ADD     :R4:,14   ACC  := (1/4)(R3+R4)
       SACH    :R3:,1    R3   := (1/2)(R3+R4)
       SUB     :R4:,15   ACC  := (1/4)(R3+R4)-(1/2)(R4)
       SACH    :R4:,1    R4   := (1/2)(R3-R4)
       LAC     :I3:,14   ACC  := (1/4)(I3)
       ADD     :I4:,14   ACC  := (1/4)(I3+I4)
       SACH    :I3:,1    I3   := (1/2)(I3+I4)
       SUB     :I4:,15   ACC  := (1/4)(I3+I4)-(1/2)(I4)
       SACH    :I4:,1    I4   := (1/2)(I3-I4)
*
* CALCULATE PARTIAL TERMS FOR R2, R4, I2, AND I4.
*
       LAC     :R1:,14   ACC  := (1/4)(R1)
       ADD     :R2:,14   ACC  := (1/4)(R1+R2)
       SACH    :R1:,1    R1   := (1/2)(R1+R2)
       SUB     :R2:,15   ACC  := (1/4)(R1+R2)-(1/2)(R2)
       ADD     :I4:,15   ACC  := (1/4)[(R1-R2)+(I3-I4)]
       SACH    :R2:      R2   := (1/4)[(R1-R2)+(I3-I4)]
       SUBH    :I4:      ACC  := (1/4)[(R1-R2)-(I3-I4)]
       DMOV    :R4:      I4   := R4 = (1/2)(R3-R4)
       SACH    :R4:      R4   := (1/4)[(R1-R2)-(I3-I4)]
       LAC     :I1:,14   ACC  := (1/4)(I1)
       ADD     :I2:,14   ACC  := (1/4)(I1+I2)
       SACH    :I1:,1    I1   := (1/2)(I1+I2)
       SUB     :I2:,15   ACC  := (1/4)(I1+I2)-(1/2)(I2)
       SUB     :I4:,15   ACC  := (1/4)[(I1-I2)-(I3-I4)]
       SACH    :I2:      I2   := (1/4)[(I1-I2)-(I3-I4)]
       ADDH    :I4:      ACC  := (1/4)[(I1-I2)+(I3-I4)]
       SACH    :I4:      I4   := (1/4)[(I1-I2)+(I3-I4)]
*
* CALCULATE PARTIAL TERMS FOR R1, R3, I1, AND I3.
*
       LAC     :R1:,15   ACC  := (1/4)(R1+R2)
       ADD     :R3:,15   ACC  := (1/4)[(R1+R2)+(R3+R4)]
       SACH    :R1:      R1   := (1/4)[(R1+R2)+(R3+R4)]
       SUBH    :R3:      ACC  := (1/4)[(R1+R2)-(R3+R4)]
       SACH    :R3:      R3   := (1/4)[(R1+R2)-(R3+R4)]
       LAC     :I1:,15   ACC  := (1/4)(I1+I2)
       ADD     :I3:,15   ACC  := (1/4)[(I1+I2)+(I3+I4)]
       SACH    :I1:      I1   := (1/4)[(I1+I2)+(I3+I4)]
       SUBH    :I3:      ACC  := (1/4)[(I1+I2)-(I3+I4)]
       SACH    :I3:      I3   := (1/4)[(I1+I2)-(I3+I4)]
       $END
```

```
*
ZERO    $MACRO  PR,PI,QR,QI
*
* CALCULATE Re[P+Q] AND Re[P-Q]
*
        LAC     :PR:,15   ACC  := (1/2)(PR)
        ADD     :QR:,15   ACC  := (1/2)(PR+QR)
        SACH    :PR:      PR   := (1/2)(PR+QR)
        SUBH    :QR:      ACC  := (1/2)(PR+QR)-(QR)
        SACH    :QR:      QR   := (1/2)(PR-QR)
*
* CALCULATE Im[P+Q] AND Im[P-Q]
*
        LAC     :PI:,15   ACC  := (1/2)(PI)
        ADD     :QI:,15   ACC  := (1/2)(PI+QI)
        SACH    :PI:      PI   := (1/2)(PI+QI)
        SUBH    :QI:      ACC  := (1/2)(PI+QI)-(QI)
        SACH    :QI:      QI   := (1/2)(PI-QI)
        $END
*
PIBY4   $MACRO  PR,PI,QR,QI,W
*
        LT      :W:       T REGISTER := W=COS(PI/4)=SIN(PI/4)
        LAC     :QI:,14   ACC  := (1/4)(QI)
        SUB     :QR:,14   ACC  := (1/4)(QI-QR)
        SACH    :QI:,1    QI   := (1/2)(QI-QR)
        ADD     :QR:,15   ACC  := (1/4)(QI+QR)
        SACH    :QR:,1    QR   := (1/2)(QI+QR)
        LAC     :PR:,14   ACC  := (1/4)(PR)
        MPY     :QR:      P REGISTER := (1/4)(QI+QR)*W
        APAC              ACC  := (1/4)[PR+(QI+QR)*W]
        SACH    :PR:,1    PR   := (1/2)[PR+(QI+QR)*W]
        SPAC              ACC  := (1/4)(PR)
        SPAC              ACC  := (1/4)[PR-(QI+QR)*W]
        SACH    :QR:,1    QR   := (1/2)[PR-(QI+QR)*W]
        LAC     :PI:,14   ACC  := (1/4)(PI)
        MPY     :QI:      P REGISTER := (1/4)(QI-QR)*W
        APAC              ACC  := (1/4)[PI+(QI-QR)*W]
        SACH    :PI:,1    PI   := (1/2)[PI+(QI-QR)*W]
        SPAC              ACC  := (1/4)(PI)
        SPAC              ACC  := (1/4)[PI-(QI-QR)*W]
        SACH    :QI:,1    QI   := (1/2)[PI-(QI-QR)*W]
        $END
*
PIBY2   $MACRO  PR,PI,QR,QI
*
* CALCULATE Re[P+jQ] AND Re[P-jQ]
*
        LAC     :PI:,15   ACC  := (1/2)(PI)
        SUB     :QR:,15   ACC  := (1/2)(PI-QR)
        SACH    :PI:      PI   := (1/2)(PI-QR)
        ADDH    :QR:      ACC  := (1/2)(PI-QR)+(QR)
        SACH    :QR:      QR   := (1/2)(PI+QR)
*
* CALCULATE Im[P+jQ] AND Im[P-jQ]
*
        LAC     :PR:,15   ACC  := (1/2)(PR)
        ADD     :QI:,15   ACC  := (1/2)(PR+QI)
        SACH    :PR:      PR   := (1/2)(PR+QI)
        SUBH    :QI:      ACC  := (1/2)(PR+QI)-(QI)
        DMOV    :QR:      QR   -> QI
        SACH    :QR:      QR   := (1/2)(PR-QI)
        $END
```

```
*
PI3BY4 $MACRO  PR,PI,QR,QI,W
*
       LT      :W:       T REGISTER := W=COS(PI/4)=SIN(PI/4)
       LAC     :QI:,14   ACC  := (1/4)(QI)
       SUB     :QR:,14   ACC  := (1/4)(QI-QR)
       SACH    :QI:,1    QI   := (1/2)(QI-QR)
       ADD     :QR:,15   ACC  := (1/4)(QI+QR)
       SACH    :QR:,1    QR   := (1/2)(QI+QR)
       LAC     :PR:,14   ACC  := (1/4)(PR)
       MPY     :QI:      P REGISTER := (1/4)(QI-QR)*W
       APAC              ACC  := (1/4)[PR+(QI-QR)*W]
       SACH    :PR:,1    PR   := (1/2)[PR+(QI-QR)*W]
       SPAC              ACC  := (1/4)(PR)
       SPAC              ACC  := (1/4)[PR-(QI-QR)*W]
       MPY     :QR:      P REGISTER := (1/4)(QI+QR)*W
       SACH    :QR:,1    QR   := (1/2)[PR-(QI-QR)*W]
       LAC     :PI:,14   ACC  := (1/4)(PI)
       SPAC              ACC  := (1/4)[PI-(QI+QR)*W]
       SACH    :PI:,1    PI   := (1/2)[PI-(QI+QR)*W]
       APAC              ACC  := (1/4)(PI)
       APAC              ACC  := (1/4)[PI+(QI+QR)*W]
       SACH    :QI:,1    QI   := (1/2)[PI+(QI+QR)*W]
       $END
```

Example 5-35 shows the bit-reversal addressing capability of the TMS320C25 for implementing an 8-point DIT FFT. On the TMS320C25 the following instructions input the data and store it in memory in the bit-reversed sequence:

```
       RPTK    7
       IN      *BR0+,PA0
```

This code combines the functions of input and bit-reversal addressing which were previously implemented separately on the TMS32020. The following implementation uses a separate bit-reverse macro:

```
       RPTK    7
       IN      *0+,PA0

       BITREV  X1R,S1I,X4R,X4I
       BITREV  X3R,S3I,X6R,X6I
```

Example 5-35. An 8-Point DIT FFT

```
X0R     EQU     00
X0I     EQU     01
X1R     EQU     02
X1I     EQU     03
X2R     EQU     04
X2I     EQU     05
X3R     EQU     06
X3I     EQU     07
X4R     EQU     08
X4I     EQU     09
X5R     EQU     10
X5I     EQU     11
X6R     EQU     12
X6I     EQU     13
X7R     EQU     14
X7I     EQU     15
W       EQU     16
WVALUE  EQU     >5A82       ; VALUE FOR SIN(45) OR COS(45)
*
* INITIALIZE FFT PROCESSING.
*
FFT     SPM     0           ; NO SHIFT OF PR OUTPUT
        SSXM                ; SET SIGN-EXTENSION MODE.
        ROVM                ; RESET OVERFLOW MODE.
        LDPK    4           ; SET DATA PAGE POINTER TO 4.
        LALK    WVALUE      ; GET TWIDDLE FACTOR VALUE.
        SACL    W           ; STORE SIN(45) OR COS(45).
*
* INPUT SAMPLES, STORING IN BIT-REVERSED ORDER.
*
        LARK    AR0,8       ; LOAD LENGTH OF FFT IN AR0.
        LRLK    AR1,>200    ; LOAD AR1 WITH DATA PAGE 4 ADDRESS.
        LARP    AR1
        RPTK    7
        IN      *BR0+,PA0   ; ONLY REAL-VALUED INPUT
*
* FIRST & SECOND STAGES COMBINED WITH DIVIDE-BY-4 INTERSTAGE
SCALING
*
        COMBO   X0R,X0I,X1R,X1I,X2R,X2I,X3R,X3I,
        COMBO   X4R,X4I,X5R,X5I,X6R,X6I,X7R,X7I.
*
* THIRD STAGE WITH DIVIDE-BY-2 INTERSTAGE SCALING
*
        ZERO    X0R,X0I,X4R,X4I
        PIBY4   X1R,X1I,X5R,X5I,W
        PIBY2   X2R,X2I,X6R,X6I
        PI3BY4  X3R,X3I,X7R,X7I,W
*
* OUTPUT SAMPLES, SUPPLYING IN SEQUENTIAL ORDER.
*
        LRLK    AR1,>200    ; LOAD AR1 WITH DATA PAGE 4 ADDRESS.
        RPTK    15
        OUT     *+,PA0      ; COMPLEX-VALUED OUTPUT
        RET
```

Table 5-2 provides a comparison of execution speed, program memory, and data memory for an 8-point DIT FFT implementation using the TMS32020 and TMS320C25.

Table 5-2. FFT Memory Space and Time Requirements

DEVICE	WORDS IN MEMORY		CPU CYCLES	TIME (μs)
	Data	Program		
TMS32020	17	169	216	43.2
TMS320C25	17	153	178	17.8

6. Hardware Applications

Information and examples on how to interface the TMS320C25 to external devices are presented in this section. The examples given are general enough in nature that they may be easily adapted to fit a particular system requirement.

The following buses, ports, and control signals provide system interface to the TMS320C25 processor:

- 16-bit address bus (A15-A0)
- 16-bit data bus (D15-D0)
- Serial port
- $\overline{PS}$, $\overline{DS}$, $\overline{IS}$ (program, data, I/O space select)
- R/$\overline{W}$ (read/write) and $\overline{STRB}$ (strobe)
- $\overline{READY}$ and $\overline{MSC}$ (microstate complete)
- $\overline{HOLD}$ and $\overline{HOLDA}$ (hold acknowledge)
- $\overline{INT}$(2-0) and $\overline{IACK}$ (interrupt acknowledge)
- XF (external flag) and $\overline{BIO}$ (branch control)
- $\overline{SYNC}$ (synchronization) and $\overline{BR}$ (bus request)

Major hardware applications discussed in this section are listed below.

6.1 External Local Memory Interface

The external local memory interface provides the versatility to interface the TMS320C25 to a wide variety of memory devices. For example, if speed and maximum throughput are desired, the TMS320C25 can run with zero wait states and perform memory accesses in a single machine cycle. The TMS320C25 can access slower memories by inserting one or more wait states into the memory access operation by using the READY input signal.

If the internal data RAM on the TMS320C25 is sufficient for system needs, a minimal memory configuration, such as the one shown in Figure 6-1, can be implemented. In the example, two (2K x 8) PROMs are used as program memory. No address decoding is performed, and the $\overline{PS}$ control signal is used as the chip enable.

Depending on the access time of the PROMs, the READY input can be either connected to a logic high for zero wait states or connected to the MicroState Complete ($\overline{MSC}$) pin for automatic one wait-state generation.

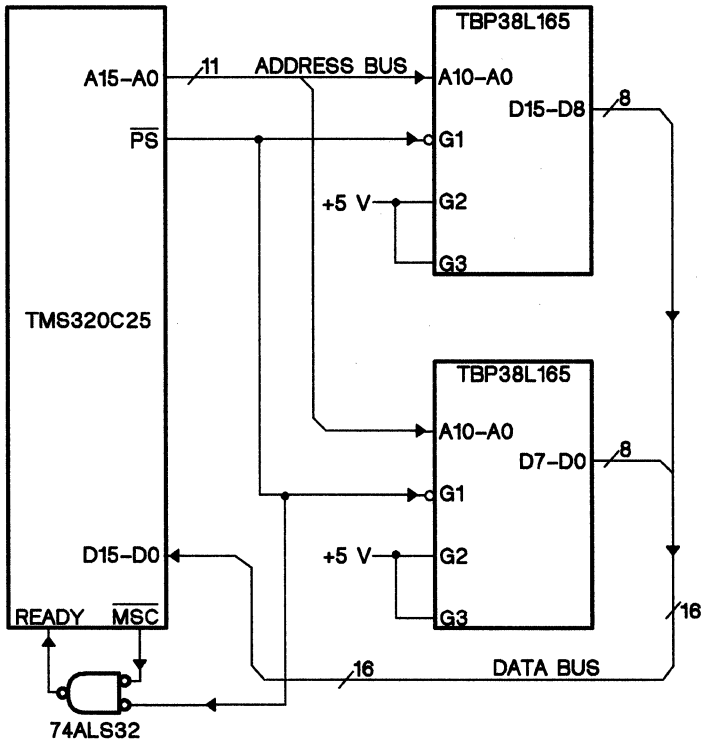


Figure 6-1. Minimal External Program Memory Configuration

6.2 Wait States

The number of cycles in a memory or I/O access is determined by the state of the READY input. At the start of quarter-phase 3, the TMS320C25 samples the READY input. If READY is high, the memory access ends at the next falling edge of CLKOUT1. If READY is low, the memory cycle is extended by one machine cycle, and all other signals remain valid. At the beginning of the next quarter-phase 3, this sequence is repeated. Figure 6-2 shows a one wait-state memory access. Note that for on-chip program and data memory accesses, the READY input is ignored.

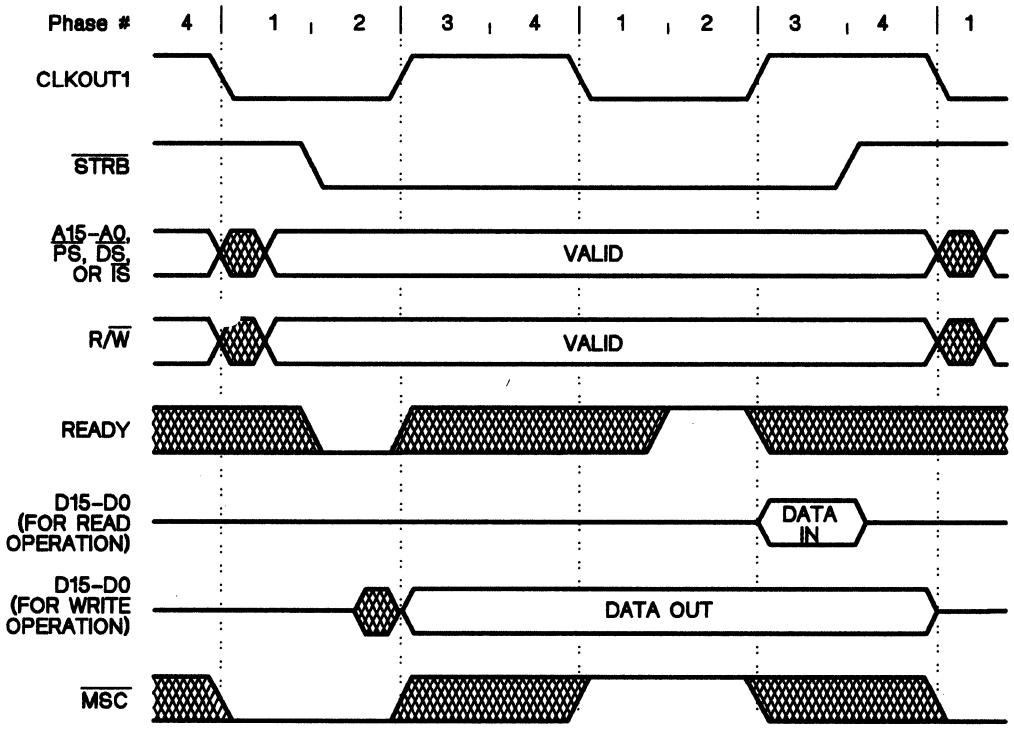


Figure 6-2. One Wait-State Memory Access Timing

The automatic generation of one wait state can be accomplished by the use of the MicroState Complete ($\overline{MSC}$) signal. The $\overline{MSC}$ output is asserted low during CLKOUT1 low to indicate the beginning of an internal or external memory or I/O operation (see Figure 6-2). By gating $\overline{MSC}$ with the address and $\overline{PS}$, $\overline{DS}$, and/or $\overline{IS}$, a one-wait READY signal can be generated (see Figure 6-3).

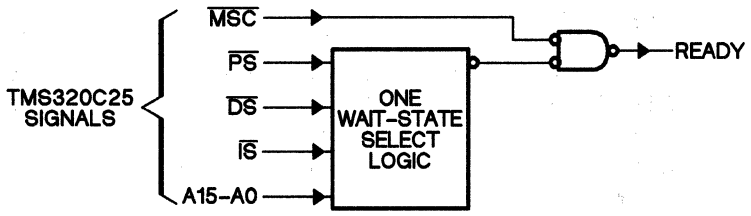


Figure 6-3. One Wait-State Generator Using MSC

6.3 Direct Memory Access

The TMS320C25 supports Direct Memory Access (DMA) to its external program/data memory using the $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$ signals. Direct memory access 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 halted processor's local off-chip memory. Here 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 provide the slave with the necessary data to complete a task.

In a typical TMS320C25 direct memory access scheme, the master may be a general-purpose CPU, another TMS320C25, or perhaps even an analog-to-digital converter. A simple TMS320C25 master-slave configuration is shown in Figure 6-4. The master TMS320C25 takes complete control of the slave's external memory by asserting $\overline{\text{HOLD}}$ low via its external flag (XF). This causes the slave to place its address, data, and control lines in a high-impedance state. By asserting $\overline{\text{RS}}$ in conjunction with $\overline{\text{HOLD}}$, the master processor can load the slave's local program memory with the necessary initialization code on reset or powerup. The two processors can be synchronized using the $\overline{\text{SYNC}}$ pin to make the transfer over the memory bus faster and more efficient.

After control of the slave's buses is given up to the master processor, the slave alerts the master of this fact by asserting $\overline{\text{HOLDA}}$. This signal may be tied to the master TMS320C25's $\overline{\text{BIO}}$ pin. The slave's XF pin may be used to indicate to the master when it has finished performing its task and needs to be reprogrammed or requires additional data to continue processing. In a multiple slave configuration, the priority of each slave's task may be determined by tying the slave's XF signals to the appropriate $\overline{\text{INT}}$(2-0) pin on the master TMS320C25.

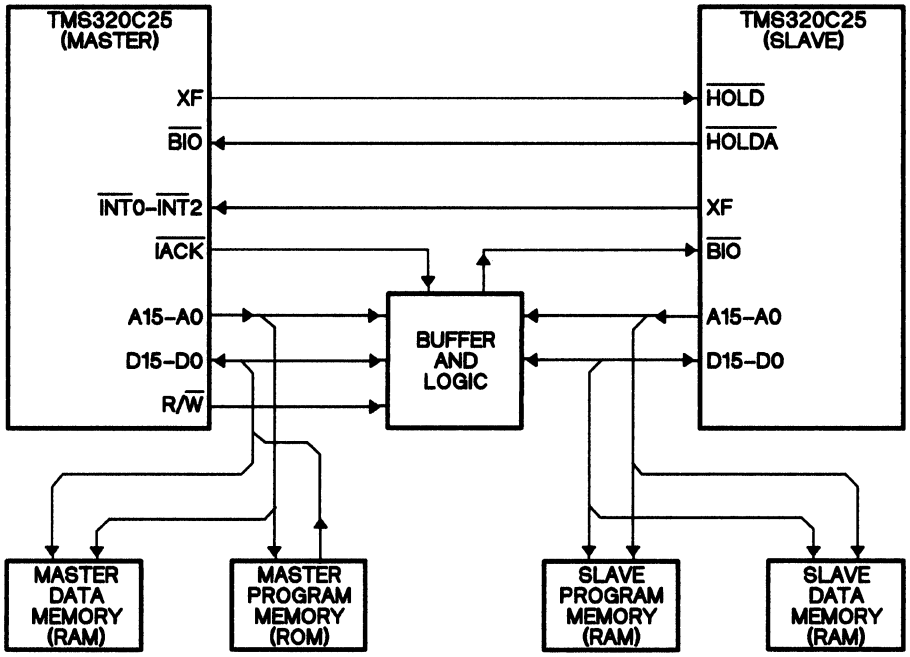


Figure 6-4. Direct Memory Access Using a Master-Slave Configuration

A PC environment presents another example of a potential direct memory access scheme where a system bus (the PC-bus) is used for data transfer. In this configuration, either the master CPU or a disk controller may place data onto the system bus, which can be downloaded into the local memory of the TMS320C25. Here the TMS320C25 acts more like a peripheral processor with multifunction capability. In a speech application, for example, the master can load the TMS320C25's program memory with algorithms to perform such tasks as speech analysis, synthesis, or recognition, and fill the TMS320C25's data memory with the required speech templates. In another application example, the TMS320C25 can serve as a dedicated graphics engine. Programs can be stored in TMS320C25 program ROM or downloaded via the system bus into program RAM. Data can come from PC disk storage or provided directly by the master CPU.

Figure 6-5 depicts a direct memory access using a PC environment. In this configuration, decode and arbitration logic is used to control the direct memory access. When the address on the system bus resides in the local memory of the peripheral TMS320C25, this logic asserts the $\overline{\text{HOLD}}$ signal of the TMS320C25 while sending the master a not-ready indication to allow wait states. After the TMS320C25 acknowledges the direct memory access by asserting $\overline{\text{HOLDA}}$, READY is asserted and the information transferred.

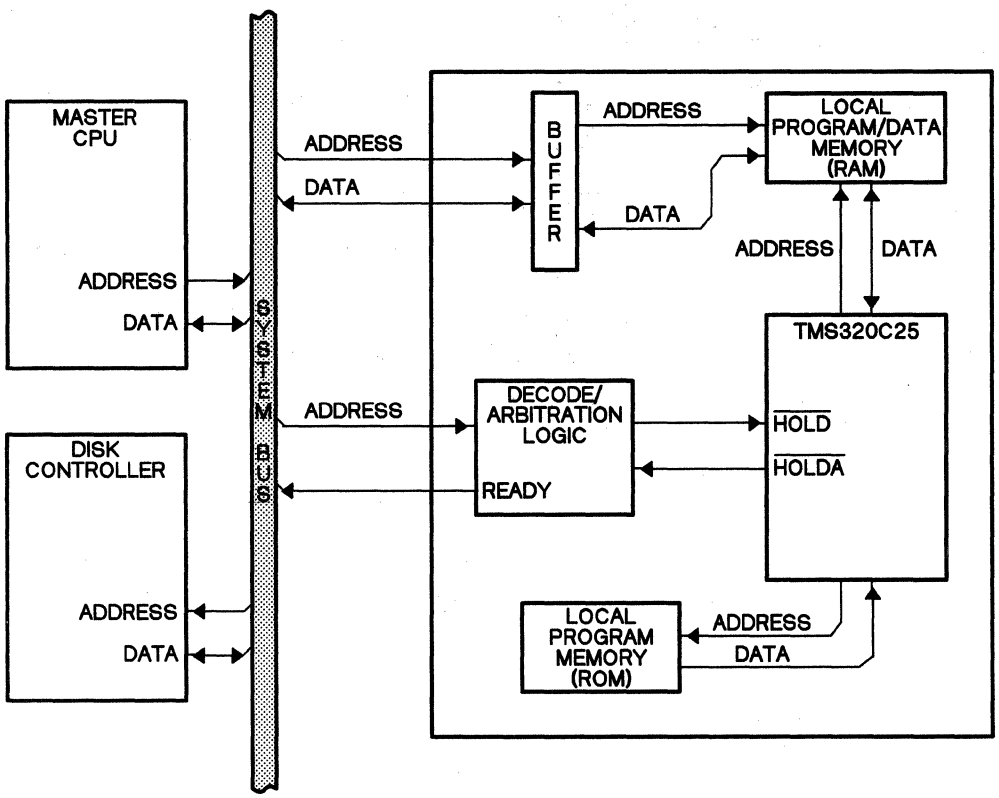


Figure 6-5. Direct Memory Access in a PC Environment

6.4 Global Memory

In various digital signal processing tasks, such as filters or modems, the algorithm being implemented may be divided into sections with a distinct processor dedicated to each section. In this multiple processor scheme, the first and second processors may share global data memory, as well as the second and third, the third and fourth, etc. Arbitration logic may be required to determine which section of the algorithm is executing and which processor has access to the global memory. With multiple processors dedicated to distinct sections of the algorithm, throughput may be increased via pipelined execution.

The external memory of the TMS320C25 can be divided into both global and local sections. Special registers and pins included on the TMS320C25 allow multiple processors to share up to 32K words of global data memory. This implementation facilitates efficient "shared data" multiprocessing where data is transferred between two or more processors. Unlike a direct memory access scheme, reading or writing global memory does not require one of the processors to be halted.

The size of the global memory is programmable between 256 and 32K locations in data memory by loading the global register (GREG). After global memory is defined in the GREG, the TMS320C25 asserts the $\overline{BR}$ (bus request) signal before each global memory access. The processor then inserts wait states until a bus grant is given by

asserting the READY line. Figure 6-6 illustrates such a global memory interface. Since the processors can be synchronized by using the $\overline{\text{SYNC}}$ pin, the arbitration logic may be simplified and the address and data bus transfers more efficient.

The $\overline{\text{SYNC}}$ pin on the TMS320C25 may also be used to synchronize several processors to allow for execution of redundant fail-safe systems. $\overline{\text{SYNC}}$ permits instruction broadcasting between several processors and lock-step execution after initial synchronization.

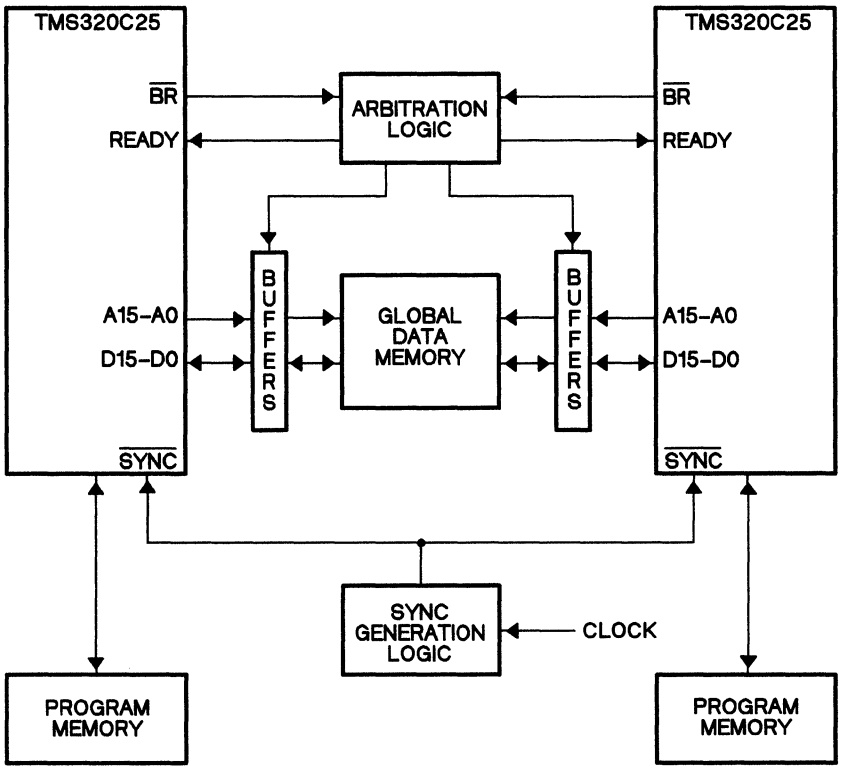


Figure 6-6. Global Memory Communication

6.5 Codec Interface

In some areas of telecommunications, speech processing, and other applications that require low-cost analog I/O devices, a codec may be useful. The combo-codec used here consists of nonlinear A/D and D/A converters with all the associated filters and data-holding registers.

The TMS320C25 serial port allows communication with serial devices such as codecs. The speed and versatility of the TMS320C25 allow it to compand (COMpress and exPAND) a PCM (Pulse Code Modulation) data stream, acquired by the codec, through the TMS320C25 execution of software conversion routines (see the application report, "Companding Routines for the TMS32010/TMS32020," in the book, *Digital Signal Processing Applications with the TMS320 Family*). Figure 6-7 shows an interface example of a Texas Instruments TCM2913 codec to the TMS320C25 serial port.

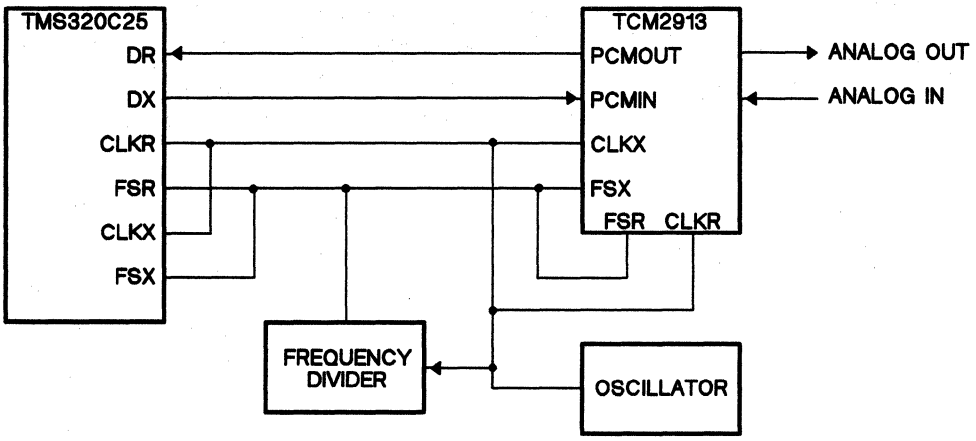


Figure 6-7. Codec Interface

In this configuration, all timing and synchronization signals are externally generated using independent oscillator and frequency-dividing hardware such as the 74AS867 or 74LS161 counters. Alternatively, the designer may decide to generate the timing signals from the TMS320C25 clock by subdividing its frequency.

In some circuits, it may be necessary to include an opamp at the analog output of the codec. In such cases or if variable output gain is required, a gain-setting resistor network must be provided as specified in the TCM2913 documentation.

Other linear A/D and D/A converters may be interfaced to the TMS320C25 through its parallel ports as well as the serial ports.

6.6 I/O Ports

I/O design on the TMS320C25 is treated the same way as memory. The I/O address space is distinguished from the local program/data memory space by the $\overline{IS}$ signal. $\overline{IS}$ goes low at the beginning of the memory cycle. All other control signals and timing parameters will be the same as those for the program/data external memory interface.

The TMS320C25 software instructions can access 16 input and 16 output ports. The four least significant bits of the address bus specify the particular port being accessed. A pair of 74AS138s can be used to fully decode these address bits (see Figure 6-8).

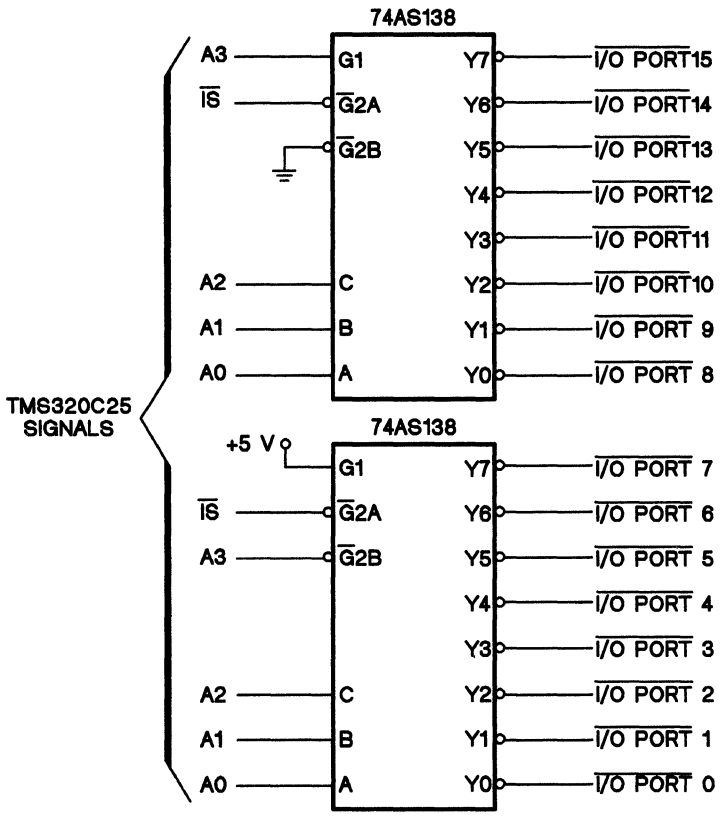


Figure 6-8. I/O Port Addressing

A simple interface between two processors can be implemented using up to 16 bidirectional I/O ports connected to the TMS320C25. An interprocessor communication path can be formed by memory-mapping peripherals to the I/O ports of the TMS320C25. In this manner, the TMS320C25 can connect to parallel A/Ds, registers, FIFOs, two-port memories, or other peripheral devices. In a multiprocessing scheme, intelligent peripherals can be memory-mapped into the I/O ports. Here the TMS320C25 can communicate with UARTs, general-purpose microprocessors, disk controllers, video controllers, or other peripheral processors.

Using an 8-bit general-purpose microprocessor, such as TI's TMS7042, for a keyboard interface is an example of a TMS320C25 I/O port multiprocessing scheme, as shown in Figure 6-9. The TMS7042 may be mapped into the TMS320C25 I/O space using latches to store the transferred data. In a single or multiple I/O port multiprocessing configuration, the four LSBs of the address bus are decoded to determine which of the 16 I/O ports on the TMS320C25 is being accessed. The TMS320C25 selects the I/O space ($\overline{IS}$) for its external bus and reads/writes data using the IN/OUT instructions.

Processor-controlled signals between the TMS320C25 and the peripheral device indicate when data is available to be read. This interprocessor communication is facilitated by using the input and output pins of the TMS7042 (or other peripheral processor). In an I/O multiprocessing configuration, the I/O port address space is

limited, and data transfers are relatively slow compared to a direct memory access or global memory configuration.

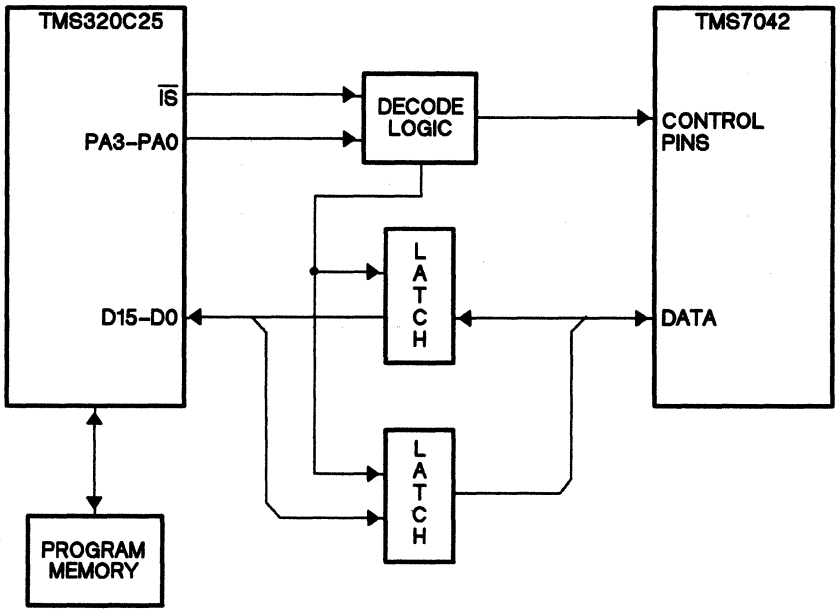


Figure 6-9. I/O Port Processor-to-Processor Communication

7. Assembler Directives

The TMS320C25 Macro Assembler translates mnemonic instructions and assembler directives specified by the TMS320C25 assembly language source code into executable object code. Some directives make sections of the program relocatable, others define constants for data or text, and still others provide linkage between separate program modules to form a complete executable program. After the Link Editor links a program as required, the TMS320C25 Simulator provides simulation for effective software development and program verification.

Given a file of TMS320C25 source code as input, the assembler outputs a listing file, an object file, an optional symbol table, and a cross-reference list. The assembler also provides a comprehensive set of error diagnostics.

Major topics discussed in this section are listed below.

7.1 Creation of TMS320C25 Source Code

The TMS320C25 assembly language consists of operation codes (called mnemonics) that correspond directly to binary machine instructions. An assembly language program is called a source program. Before it can be executed by the computer, the source program must be processed by the assembler to obtain a machine language program. This processing of a source program is called assembling. This consists of combining the binary values (which correspond to the operation code) with the binary address information to form the machine language instruction.

The TMS320C25 Assembler is a two-pass assembler that processes source code twice. On the first pass, the assembler maintains the location counter (which defines the program memory addresses assigned to the resulting words of object code), builds a symbol table, and produces a list file of the source code. On the second pass, the assembler produces the object code using the operation codes and the symbol table of the first pass.

An assembly language source program consists of source statements that may contain assembler directives, machine instructions, or comments. Source statements scanned by the assembler may contain four ordered fields (label, command, operand, and comment) separated by one or more blanks. Source statements containing an asterisk (*) in the first character position are comment statements, and as such, have no effect on the assembly. The source statement line may be as long as the source file format allows; however, the assembler truncates the source line to 60 characters without warning. Only comments may extend past column 60 without an error resulting.

The TMS320C25 Assembler uses the ASCII character set.

The syntax for source statements is as follows:

[<label>] <mnemonic> [<operand>] [<comment>]

A source statement may have a label that is user-defined. The label field begins in character position one of the source statement. At least one blank must separate the label from the command mnemonic, the mnemonic from the operand (when an operand is required), and the operand(s) from the comment field.

The last source statement of a source program, usually the END directive, is followed by the end-of-file statement for the source medium.

7.1.1 Label Field

The label field begins in character position one of the source statement and contains a label of up to six significant characters. The first character of the label must be alphabetic; additional characters may be alphanumeric. A label is optional for machine instructions and for many assembler directives. When the label is omitted, the first character position must contain a blank.

A source statement consisting of only a label field is a valid statement. It has the effect of assigning the current value of the location counter to the label as well as to the next source statement. This is equivalent to the following directive statement:

<label> EQU $ Where $ represents the current value of the location counter at that point in the assembly.

7.1.2 Command Field

The command field begins after the blank that terminates the label field, or in the first non-blank character past the first character position (which must be blank when the label is omitted). The command field is terminated by one or more blanks and may not extend past the right margin. The command field may contain one of the following:

- Assembler mnemonic of a machine instruction (e.g., MAC)
- Macro mnemonic (e.g., FACT)
- Assembler directive (e.g., DATA)

7.1.3 Operand Field

The operand field begins following the blank that terminates the command field and may not extend past the right margin of the source statement. The operand field is terminated by one or more blanks.

The operand field may contain one or more of the following:

- Constants
- Character strings
- Expressions

Symbols used in the operand field must be defined in the assembly, usually by appearing in the label field of a statement or in the operand field of a REF (external reference) or SREF (secondary external reference) directive. REF and SREF directives provide access to symbols defined in other programs.

7.1.4 Comment Field

The comment field begins after the blank terminating the operand field or the blank terminating the command field, as in the case of commands that have no operands. The comment field may extend to the end of the source statement (if required) and may contain any ASCII character including blank(s). The contents of the comment field up to the end of the source record are listed in the source portion of the assembly listing and have no other effect on the assembly.

7.2 Symbols

Symbols are used in the label field and the operand field. A symbol is a string of alphanumeric characters, ('A' through 'Z', '0' through '9', and '$'). The first character in a symbol must be 'A' through 'Z' or '$'. No character may be blank. When more than six characters are used in a symbol, the assembler prints all the characters, but accepts only the first six characters for processing (the assembler prints a warning indicating that the symbol has been truncated). Therefore, symbols must be unique in the first six characters. User-defined symbols are valid only during the assembly in which they are defined.

Symbols used in the label field become symbolic addresses. They are associated with locations in the program and must not be used in the label field of other statements. Mnemonic operation codes and assembler directive names may also be used as valid user-defined symbols when placed in the label field.

Any symbol appearing in the label field of a source statement, other than an EQU (define assembly-time constant) directive, is either absolute or relocatable depending on whether or not the statement is in an absolute (specified) block of the program.

7.2.1 Predefined Symbols

The predefined symbols are the dollar sign character ($) and the auxiliary register and port symbols. The dollar sign character is used to represent the current location within the program. The auxiliary register symbols are of the form 'ARn' where 'n' is 0 to 4. The port addresses are of the form 'PAn' where 'n' is 0 through 15.

Examples of valid predefined symbols:

$	Represents the current location
AR0	Represents Auxiliary Register 0
PA12	Represents Port Address 12

7.3 Constants

The assembler recognizes the following five types of constants, each internally maintained as a 16-bit quantity:

- Decimal integer constants
- Binary integer constants
- Hexadecimal integer constants
- Character constants
- Assembly-time constants

Decimal, binary, hexadecimal, and character constants are absolute. Assembly-time constants defined by absolute expressions are absolute, and assembly-time constants defined by relocatable expressions are relocatable.

7.3.1 Decimal Integer Constants

A decimal integer constant is written as a string of decimal digits. Decimal integers range in value from -32,768 to +32,767. Positive decimal integer constants in the range of 32,768 to 65,535 are considered negative when interpreted as two's-complement values.

Examples of valid decimal constants:

1000	Constant equal to 1000 or >3E8
-32768	Constant equal to -32768 or >8000
25	Constant equal to 25 or >19

7.3.2 Binary Integer Constants

A binary integer constant is written as a string of up to 16 binary digits (0 or 1) preceded by a question mark, '?'. If less than 16 digits are specified, the assembler right-justifies the given bits in the resulting constant.

Examples of valid binary constants:

?0000000000010011	Constant equal to 19 or >0013
?0111111111111111	Constant equal to 32767 or >7FFF
?11110	Constant equal to 30 or >001E

7.3.3 Hexadecimal Integer Constants

A hexadecimal integer constant is written as a string of up to four hexadecimal digits preceded by a 'greater than' sign, '>'. If less than four hexadecimal digits are specified, the assembler right-justifies the given bits in the resulting constant. Hexadecimal digits include the decimal values '0' through '9' and the letters 'A' through 'F'.

Examples of valid hexadecimal constants:

>78	Constant equal to 120 or >0078
>F	Constant equal to 15 or >000F
>37AC	Constant equal to 14252 or >37AC

7.3.4 Character Constants

A character constant is written as a string of one or two alphabetic characters enclosed in single quotes. Two consecutive single quotes represent each single quote contained within a character constant. If less than two characters are specified, the assembler right-justifies the given bits in the resulting constant. The characters are represented internally as eight-bit ASCII characters. A character constant consisting only of two single quotes (no character) is valid and is assigned the value 0000 (hexadecimal).

Examples of valid character constants:

'AB'	Represented internally as >4142
'C'	Represented internally as >0043
'N'	Represented internally as >004E
''''D'	Represented internally as >2744

7.3.5 Assembly-Time Constants

An assembly-time constant is a symbol given a value by an EQU directive. The value of the symbol, determined at assembly time, is considered to be absolute or relocatable according to the relocatability of the expression, not according to the relocatability of the location counter value. Absolute value symbols may be assigned values with expressions using any of the above constant types.

7.4 Character Strings

Some assembler directives require character strings in the operand field. A character string is written as a string of characters enclosed in single quotes. Two consecutive single quotes represent a single quote in a character string. The maximum length of the string is defined for each directive requiring a character string. The characters are represented internally as eight-bit ASCII.

Examples of valid character strings:

'SAMPLE PROGRAM'	Defines a 14-character string consisting of SAMPLE PROGRAM.
'PLAN "C"'	Defines an 8-character string consisting of PLAN 'C'.

7.5 Expressions

Expressions are used in operand fields of assembler directives and machine instructions. An expression is a constant or symbol, a series of constants or symbols, or a series of constants and symbols separated by arithmetic operators. Each constant or symbol may be preceded by a plus sign (unary plus), a minus sign (unary minus), or the # symbol (unary invert). Unary minus takes the two's complement of the expression, and unary invert takes the one's complement. The # symbol yields the value of the logical complement of the following constant or symbol. An expression does not contain embedded blanks. The valid range of values for an expression is -32,768 to +65,535.

An expression is relocatable when the number of relocatable symbols or constants added to the expression is one greater than the number of relocatable symbols or constants subtracted from the expression. (All other valid expressions are absolute.) When the first symbol or constant is unsigned, it is considered as added to the expression. For example, when all symbols in the following expressions are relocatable, the expressions are relocatable:

```
LABEL+1
LABEL+TABLE+-INC
-LABEL+TABLE+INC
```

7.5.1 Arithmetic Operators in Expressions

Arithmetic operators used in expressions are as follows:

+ for addition
- for subtraction
* for multiplication
/ for division

In evaluating an expression, the assembler first negates any constant or symbol preceded by a unary minus and then performs the arithmetic operations from left to right. The assembler does not assign precedence to any operation other than unary plus or unary minus. All operations are integer operations. The assembler truncates the fraction in division. When a unary minus follows a subtraction operator, the effective operation is addition. The unary minus cannot be applied to a relocatable expression or subexpression. For example, the expression 4+5*2 would be evaluated as 18, not 14; and the expression 7+1/2 would be evaluated as 4, not 7.

The assembler checks for a valid range of values. The warning message 'VALUE TRUNCATED' is given when a value is out of range.

An example of where a 'VALUE TRUNCATED' message is given:

```
B       65538
```

7.5.2 Parentheses in Expressions

The assembler supports the use of parentheses in expressions to alter the order of evaluation of the expression. Nesting of pairs of parentheses within expressions is also supported. Evaluation of portions of an expression within parentheses at the same nesting level may be considered to be simultaneous. Parentheses cannot be nested more than eight levels deep.

For example, the use of parentheses in the expression LAB1+((4+3)*7) results in the following operation:

1) Add four to three,
2) Multiply the resulting sum by seven, and
3) Add the resulting product to the value of LAB1.

7.5.3 Well-Defined Expressions

Some assembler directives require well-defined expressions in operand fields. For an expression to be well-defined, any symbols or assembly-time constants in the expression must have been previously defined. The evaluation of a well-defined expression must be absolute, and a well-defined expression must not contain a character constant. An example of a well-defined expression is as follows:

>1000+X Where X must have been previously defined.

7.5.4 Absolute and Relocatable Symbols in Expressions

The value of an expression containing absolute or relocatable symbols is either absolute or relocatable depending on a precise set of rules. An expression containing a relocatable symbol or relocatable constant immediately following a multiplication or division operator is illegal. When the result of evaluating an expression up to a multiplication or division operator is relocatable, the expression is also illegal. Table 7-1 defines the relocatability of the result for each type of operator.

If the current value of an expression is relocatable with respect to one relocatable section, a symbol of another section cannot be included until the value of the expression becomes absolute. The following are examples of legal expressions:

BLUE+1 The sum of the value of symbol BLUE plus one is legal and of the same type as BLUE.

GREEN-4 The result of subtracting four from the value of symbol GREEN is legal and of the same type as GREEN.

2*16+RED The sum of the value of symbol RED plus the product of two times 16 is legal, and of the same type as RED.

440/2-RED The result of dividing 440 by two and subtracting the value of symbol RED from the quotient (RED must be absolute for this to be a legal expression).

Table 7-1. Results of Operations on Absolute and Relocatable Items

IF A IS	AND B IS	RESULT OF A+B	RESULT OF A-B	RESULT OF A*B	RESULT OF A/B
ABS	ABS	ABS	ABS	ABS	ABS(B≠0)
ABS	RELOC	RELOC	illegal	Note 1	illegal
RELOC	ABS	RELOC	RELOC	Note 2	Note 3
RELOC	RELOC	illegal	Note 4	illegal	illegal

Notes:
1. Illegal unless A equals zero or one. If A is one, the result is relocatable; if A is zero, the result is an absolute zero.
2. Illegal unless B equals zero or one. If B is one, the result is relocatable; if B is zero, the result is an absolute zero.
3. Illegal unless B equals one. If B equals one, the result is relocatable.
4. Illegal unless A and B are in the same section. If A and B are in the same section, the result is absolute.

7.5.5 Externally Referenced Symbols in Expressions

As defined in the REF (external reference) and SREF (secondary external reference) directives, the assembler allows externally referenced symbols in expressions under the following conditions:

1) Only one externally referenced symbol is used in an expression.

2) The character preceding the referenced symbol must be a blank, a plus sign, or a comma. The portion of the expression preceding the symbol, if any, must be added to the symbol.

3) The portion of the expression following the referenced symbol must not include multiplication or division on the symbol (as for a relocatable symbol).

4) The remainder of the expression following the referenced symbol must be absolute.

The link editor resolves all externally referenced symbols automatically. However, the assembler limits the user to a total of 255 externally referenced symbols per module. Modules using more than 255 external symbols must be broken into smaller modules for assembly, and linked using the link editor.

7.6 Assembler Directives

Assembler directives are instructions that control the assembly process rather than produce object code for machine instructions. The TMS320C25 Assembler supports directives in the following categories:

- Directives that affect the location counter
- Directives that affect assembler output
- Directives that initialize constants
- Directives that provide linkage between programs
- Miscellaneous directives.

7.6.1 Directives That Affect the Location Counter

As the assembler reads the source statements of a program, the location counter is set to correspond to the memory locations assigned to the resulting object code. The thirteen assembler directives that affect the location counter are shown in Table 7-2. The first nine initialize the location counter and define its value as relocatable, absolute, or dummy. The next two directives set the location counter to provide a block of program memory for the object code. The last two directives define a block of an independently stored program segment.

Table 7-2. Assembler Directives That Affect the Location Counter

DIRECTIVES	MNEMONICS
Absolute origin	AORG
Relocatable origin	RORG
Dummy origin	DORG
Block starting with symbol	BSS
Block ending with symbol	BES
Data segment	DSEG
Data segment end	DEND
Common segment	CSEG
Common segment end	CEND
Program segment	PSEG
Program segment end	PEND
Independent program segment	EXEC
Independent segment end	XEND

7.6.2 Directives That Affect Assembler Output

Directives that affect assembler output (see Table 7-3) are primarily used to improve user interface. The first directive supplies a program identifier in the object code. The other five directives in this category format the source listing.

Table 7-3. Assembler Directives That Affect Assembler Output

DIRECTIVES	MNEMONICS
Program identifier	IDT
Output options	OPTION
Page title	TITL
Restart source listing	LIST
Stop source listing	UNL
Eject page	PAGE

7.6.3 Directives That Initialize Constants

Table 7-4 lists those directives that initialize constants. DATA and TEXT assign hexadecimal values in successive words of the object code. EQU initializes a constant for use during the assembly process.

Table 7-4. Assembler Directives That Initialize Constants

DIRECTIVES	MNEMONICS
Initialize word	DATA
Initialize text	TEXT
Define assembly-time constant	EQU

7.6.4 Directives That Provide Linkage Between Programs

Two pairs of directives, DEF/REF and SREF/LOAD, generate the information required to link program modules, thereby making it unnecessary to assemble an entire program in the same assembly. A long program may be divided into separately assembled modules to avoid a long assembly or to reduce the symbol table size. Modules common to several programs may also be combined as required. Program modules may be linked by the link editor to form a linked object module that may be stored on a library and/or loaded as required.

The DEF/REF directives enable program modules to be assembled separately and integrated into an executable program. The DEF directive places one or more symbols defined in the module into the object code of the assembled module, thus making them available for linking. The REF directive places symbols used in the module, but defined in another module, into the object code of the assembled module, allowing them to be linked.

The Link Editor's major function is to provide symbol resolution for external references and definitions created by the REF and DEF assembler directives (see Table 7-5). Each symbol defined in a program module and required by other program modules must be placed in the operand field of a DEF directive in the program module defining it and in the operand field of a REF directive in the program module referencing it. All program modules to be linked by the link editor must include an IDT directive with a character string enclosed in single quotes placed in its operand field as the

program module name. The link editor builds a list of symbols from DEF directives as it links the program modules, and matches symbols from REF directives to the symbols in the list. The link editor follows linking commands to determine the modules to be linked. If the module in which a routine is defined has the same name as the routine entry points, the link editor can automatically locate the required module in a designated library.

The Link Editor requires a link control file as input to specify the task name, to define the starting location for the data and program segments, and to indicate the object files to be linked. The following linker commands are the primary commands that should be included in a link control file:

```
FORMAT ASCII
TASK <taskname>
PROGRAM >0000
DATA >0000
INCLUDE <object code filenames separated by commas>
INCLUDE <  or listed in separate INCLUDE commands >
END
```

The Link Editor outputs two files when linking TMS320C25 object modules. The first file is a source listing file that shows the source statement number, a location counter value, the assembled object code, the source statement as entered, and a cross-reference listing of externally defined variables. The second file contains the actual load module of linked object code to be executed by the TMS320C25.

Table 7-5. Assembler Directives That Provide Linkage Between Programs

DIRECTIVES	MNEMONICS
External definition	DEF
External reference	REF
Secondary external reference	SREF
Force load	LOAD

7.6.5 Miscellaneous Directives

This category includes assembler directives not applicable to the other categories.

Table 7-6. Miscellaneous Assembler Directives

DIRECTIVES	MNEMONICS
Program end	END
Copy source file	COPY
Define MACRO library	MLIB

The Macro Library (MLIB) assembler directive provides the TMS320C25 Assembler with the name of a library containing macro definitions. The operand of this directive is a directory pathname (constructed according to the conventions of the host operating system) enclosed in single quotes. The macros listed in this directory are user-defined, one macro per file. The MLIB directive is defined only for hosts that support libraries.

7.7 Individual Directive Descriptions

Each TMS320C25 assembler directive is described in this section. Directives are listed in alphabetical order.

The majority of the instruction symbols used to describe the syntax of the assembler directives is identical to those symbols in Table 4-2. Those that are introduced for the first time or have definitions specific to this section are listed in Table 7-7.

Table 7-7. Assembler Directive Symbols

SYMBOL	MEANING
label	The contents of the label field
exp	An expression
wd-exp	A well-defined expression
comment	The contents of the comment field
string	A character string
/ /	Items within slashes can be used only if the operand field is not empty. When not empty, they are optional.
[ ]	Items within brackets are optional.
' '	Items within single quotes are character constants or character strings.

Syntax

[<label>] AORG [<wd-exp> /<comment>/]

When a label is used, it is assigned the value that the AORG directive places in the location counter.

Description

AORG places a value in the location counter and defines the succeeding locations as absolute. An absolute location is not affected by relocation. Upon encountering an AORG statement, the assembler places the value of the well-defined expression into the location counter. When no AORG is entered, no absolute addresses are included in the object program. When the operand field is not used, the length of all preceding absolute code replaces the value in the location counter.

Example 1

```
AORG  >1000+X
```

>1000+X must be a well-defined expression. If X has a value of 6, the location counter is set to >1006.

Example 2

```
HEX   AORG  >1000
```

The location counter is set to >1000. The label HEX is assigned the value >1000.

Syntax

[<label>] BES <wd-exp> [<comment>]

When used, a label is assigned the value of the location following the block.

Description

BES advances the location counter by the value in the operand field. The operand field contains a well-defined expression representing the number of words to be added to the location counter. BES assigns a label the value of the location following the block.

Example

```
BUFF2  BES  >10
```

BES reserves a 16-word buffer. If the location counter contains >100 when the assembler processes this directive, BUFF2 is assigned the value >110.

Syntax

[<label>] BSS <wd-exp> [<comment>]

When used, a label is assigned the value of the location of the first word in the block.

Description

BSS advances the location counter by the value of the well-defined expression (wd-exp) in the operand field. The well-defined expression represents the number of words to be added to the location counter. BSS assigns a label the value of the location of the first word in the block.

Example

```
BUFF1   BSS   >10
```

If the location counter contains >100 when the assembler processes this directive, BUFF1 is assigned >100. The location counter is set to >110.

Syntax

[<label>] CEND [<comment>]

When used, a label is assigned the value of the location counter prior to modification.

Description

CEND terminates the definition of a block of common-relocatable code by placing a value in the location counter and defining succeeding locations as program-relocatable. CEND results in setting the location counter to one of these values:

- The maximum value the location counter has ever attained as a result of the assembly of any preceding block of program-relocatable code, or
- Zero, if no program-relocatable code has been previously assembled.

If encountered in data- or program-relocatable code, CEND functions as a DEND or PEND, and a warning message is issued. Like DEND and PEND, CEND is invalid when used in absolute code.

Example

See CSEG for an example of the use of CEND.

Syntax

[<label>] COPY <file-name> [<comment>]

Description

COPY causes the assembler to read source statements from a different file. The file name may be one of the following:

- An access name recognized by the operating system, or
- A synonym form of an access name.

When end-of-file is reached, the assembler resumes processing source statements from the file or device previous to the COPY directive. A COPY directive may be placed in a file being copied. Nested copying of files can be performed by placing a COPY directive in a file being copied. Such nesting is limited by the assembler to eight levels; additional restrictions may be set by the host operating system.

Example

```
COPY  SFILE.ASM
```

COPY causes the assembler to take source statements from a file called SFILE.ASM.

Syntax

[<label>] CSEG ['<string>' /<comment>/]

When used, a label is assigned the value placed by the directive in the location counter.

Description

CSEG defines succeeding locations as common-relocatable. CSEG permits the construction and definition of independently relocatable data segments that several programs may access or reference at execution time. The segments are assembly language counterparts of FORTRAN COMMON. Information placed in the object code by the assembler permits the link editor to relocate all common segments independently and make appropriate adjustments to all addresses referencing locations within common segments. The difference between CSEG and DSEG is that locations within a particular common segment may be referenced by several different programs if each program contains a CSEG directive with the same operand or no operand.

If the operand field is not used, the CSEG directive defines the beginning (or continuation) of the blank common segment of the program. When used, the operand field contains a character string of up to six characters enclosed in quotes. (If the string length exceeds six characters, the assembler prints a truncation error message and retains the first six characters of the string.) If this string has not previously appeared as the operand of a CSEG directive, the assembler associates a new relocation section number with the operand, sets the location counter to zero, and defines succeeding locations as relocatable with respect to the new relocatable section. When the operand string has been previously used in a CSEG, the succeeding code represents a continuation of the particular common segment associated with the operand. The location counter is reset to the maximum value attained during the previous assembly of any portion of that particular common segment.

The following directives properly terminate the definition of a block of common-relocatable code: CEND, PSEG, DSEG, AORG, and END. The block is normally terminated with a CEND directive. The PSEG directive, like CEND, indicates that succeeding locations are program-relocatable. The DSEG and AORG directives effectively terminate the common segment by beginning a data segment or an absolute segment. The END directive terminates the common segment and the program.

Example

```
COM1A   CSEG      'ONE'

*   COMMON RELOCATABLE SECTION, NAMED 'ONE'
          .
          .
          .
        CEND
*
COM2A   CSEG      'TWO'
          .
          .
          .
*   COMMON-RELOCATABLE SECTION, NAMED 'TWO'
          .
          .
          .
COM2B   CEND
COM1C   CSEG      'ONE'
          .
          .
          .
*
COM1B   CEND
*
COM1L   DATA COM1B-COM1A    LENGTH OF SEGMENT 'ONE'
COM2L   DATA COM2B-COM2A    LENGTH OF SEGMENT 'TWO'
```

This example illustrates the use of CSEG and CEND. The three blocks of code between CSEG and CEND are common-relocatable. The first and third blocks are relocatable with respect to one common relocation counter; the second is relocatable with respect to another. The first and third blocks constitute the common segment 'ONE'; the value of the symbol COM1L is the number of words in this segment. The symbol COM2A is the symbolic address of the first word of common segment 'TWO'; COM2B is the common-relocatable (type 'TWO') word address of the location following the segment. (Note that the symbols COM2B and COM1C are of different relocation types and possibly different values.) The value of the symbol COM2L is the number of words in common segment 'TWO'.

Syntax

[<label>] DATA <exp>[,<exp>] [<comment>]

When used, a label is assigned the location where the assembler places the first word.

Description

DATA places one or more values in one or more successive words in program memory. The assembler evaluates each expression and places the value in a word as a 16-bit two's-complement number.

DATA should be used to place coefficients or other data words in program memory. During TMS320C25 execution, TBLR can be used to transfer the data words from program memory to data RAM. As many operands as desired may be used up to a total line length of 60 characters.

Example

```
KONS1  DATA  3200,1+'AB',-'AF',>F4A0,'A'
```

DATA initializes five words, starting with a word at location KONS1. The contents of the resulting words are >0C80, >4143, >BEBA, >F4A0, and >0041.

Syntax

[<label>] DEF <symbol>[,<symbol>] [<comment>]

When used, a label is assigned the current value of the location counter.

Description

DEF makes one or more symbols available to other programs. All symbols used in the DEF statement must be defined in the same module. Each symbol defined in a program module and required by other program modules must be placed in the operand field of a DEF directive. A program named 'ROUTINES' that DEFs a routine named 'SUBR1' is shown below. The label 'SUBR1' must be defined in the program.

```
          IDT       'ROUTINES'
          DEF       SUBR1,SUBR2
            .
            .
SUBR1     EQU       $
            .
            .
          RET
SUBR2     EQU       $
            .
            .
          RET
          END
```

Example 1

```
DEF  ENTER,ANS
```

This example causes the assembler to include symbols ENTER and ANS in the object code; these symbols are available to other programs.

Example 2

```
0001           0000   ABC    EQU    0
0002           0001   DEF    EQU    1
0003   0000                  AORG   0
0004                         DEF    ABC,DEF
0005                         END
NO ERRORS, NO WARNINGS
```

The object code for the above example is:

```
K0000NO$IDT  60000ABC  60001DEF  7F89AF 2.1 83.074 NO$IDT 1
```

The symbol name follows the four-digit hexadecimal numbers assigned to the symbol by EQU. The number 6 preceding the 4-digit hexadecimal number is an object code tag.

Syntax

[<label>] DEND [<comment>]

When used, a label is assigned the value of the location counter prior to modification.

Description

DEND terminates the definition of a block of data-relocatable code by placing a value in the location counter and defining succeeding locations as program-relocatable. DEND results in setting the location counter to one of these values:

- The maximum value attained by the location counter as a result of assembling any preceding block of program-relocatable code, or
- Zero, if no program-relocatable code has been previously assembled.

If encountered in common-relocatable or program-relocatable code, DEND functions as a CEND or PEND, and a warning message is issued. Like CEND and PEND, DEND is invalid when used in absolute code.

Syntax

[<label>] DORG <exp> [<comment>]

The label is assigned the value that the directive places in the location counter.

Description

DORG defines the succeeding locations as a dummy block or section. When assembling a dummy section, the assembler does not generate object code but operates normally in all other respects. The result is that symbols that describe the layout of the dummy section are available to the assembler during assembly of the remainder of the program. Any symbol in the expression must have been previously defined.

When the operand field is absolute, the location counter is assigned the absolute value. When the operand is relocatable, the location counter is assigned the relocatable value and the same relocation type as the operand. When this occurs, space is reserved in the section that has that relocation type.

Example 1

```
DORG    0
```

DORG causes the assembler to assign values relative to the start of the dummy section to the labels within the dummy section.

Example 2

```
RORG    0
  .
  .
  .     (code as desired)
  .
DORG    $
  .
  .
  .     (data segment)
  .
END
```

This example directive defines a data structure for the executable portion (procedure division) of a procedure that is common to more than one task. The executable portion of the module (following a RORG directive) should use the labels of the dummy section as relative addresses. The code corresponding to the dummy section must be assembled in another program module. In this manner, separate data portions (dummy sections) are available to the procedure portion, regardless of the memory area into which the data is loaded.

Example 3

```
CSEG    'COM1'
DORG    $           "$" HAS A COMMON-RELOCATABLE VALUE
  .
  .
  .
LAB1    DATA        $
MASK    DATA        >F000
  .
  .
  .
CEND
```

DORG may also be used with data-relocatable or common-relocatable operands to specify dummy data or common segments. In this example, no object code is generated to initialize the common segment COM1, but space is reserved. All common-relocatable labels describing the structure of the common block (including LAB1 and MASK) are available for use throughout the program.

Syntax

[<label>] DSEG [<comment>]

When a label is used, it is assigned the data-relocatable value that the directive places in the location counter.

Description

DSEG defines succeeding locations as data-relocatable. Either of these values is placed in the location counter:

- The maximum value the location counter can attain as the result of assembling any block of data-relocatable code, or
- Zero, if no data-relocatable code has been previously assembled.

DSEG defines the beginning of a block of data-relocatable code. The block is normally terminated with DEND. If several such blocks appear throughout the program, they constitute the data segment of the program. The entire data segment may be relocated independently of the program segment at link-edit time. This provides a convenient way to separate modifiable data from executable code.

In addition to the DEND directive, PSEG, CSEG, AORG, and END properly terminate the definition of a block of data-relocatable code. PSEG, like DEND, indicates that succeeding locations are program-relocatable. CSEG and AORG effectively terminate the data segment by beginning a common segment (CSEG) or an absolute segment (AORG). END terminates the data segment as well as the program.

Example

```
RAM     DSEG            START OF DATA AREA
   .
   .
   .
<Data-relocatable code>
   .
   .
   .
   .
ERAM    DEND

LRAM    EQU ERAM - RAM
```

The block of code between DSEG and DEND is data-relocatable. RAM is the symbolic address of the first word of this block; ERAM is the data-relocatable word address of the location following the code block. The value of the symbol LRAM is the length of words in the block.

Caution:

The TMS320C25 architecture provides separate data and program memory space, which results in two memory segments occupying the same address space. Data and program segment code must be distinguished. In particular, DATA and TEXT should not be used in DSEG to initialize an area within data memory.

Data memory is volatile RAM and cannot retain information from one powerup to the next, so the proper way to initialize memory is by the execution of instructions in program memory on powerup. BSS or BES can be used within the DSEG to establish the size and names of variables, scalers, arrays, etc., in data memory. No other directives or instructions should be placed in a DSEG or CSEG.

Syntax

[<label>] END [<symbol>] /<comment>/

When used, a label is assigned the current value of the location counter.

Description

END terminates the assembly. The last source statement of a program is the END directive. Any source statements or blank records following END are considered part of the next assembly.

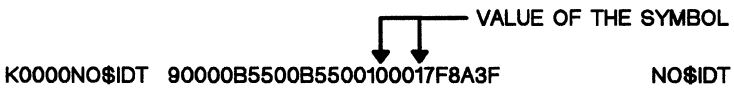
When used, the operand field contains a program-relocatable or absolute symbol that specifies to the link editor the entry point of the program. The entry point is the program address where execution of the assembled module begins. When the operand field is not used, no entry point is placed in the object code. If the entry point symbol is specified in the link control file, it must be REFed; otherwise the linker cannot find the entry symbol.

Example

```
          AORG    0
          NOP
ENTRY     NOP
          END     ENTRY
```

The symbol ENTRY is assigned the value 1 by the assembler. Since ENTRY appears as the operand of END, the value of the symbol appears as a four-digit hexadecimal character 1, as seen in the sample printout below.

Sample Printout:

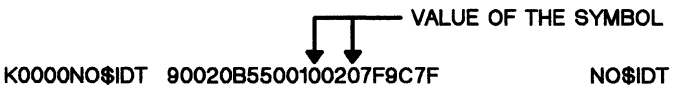


Example

```
          AORG    >20
ENTRY     NOP
          END     ENTRY
```

The symbol ENTRY is assigned the value >20. As in Example 1, the value appears in the object code following the tag character 1, as shown in the sample printout below.

Sample Printout:



Syntax

[<label>] EQU <exp> [<comment>]

The label field contains the symbol to be given a value.

Description

EQU assigns a value to a symbol. <exp> may not contain a symbol appearing in a REF directive nor contain forward references. Symbols in the operand field must be previously defined. Certain symbols, such as AR0 and PA0, have predefined values.

Example 1

```
SUM     EQU     AR1
```

The EQU directive assigns an absolute value to the symbol SUM, making SUM available as a register address.

Example 2

```
TIME    EQU     HOURS
```

This example assigns the value of the previously defined symbol HOURS to the symbol TIME. When HOURS appears in the label field of a machine instruction in a relocatable block of the program, the value is a relocatable value. After execution of EQU, the two symbols may be used interchangeably.

Syntax

[<label>] EXEC <pma> [<comment>]

When used, a label is assigned the value that the directive places in the location counter.

Description

EXEC places the value <pma> in the location counter and defines succeeding locations as independently stored program segments. EXEC defines the beginning of a block of independent code. The block is terminated by XEND. Directives that affect the value in the location counter, such as BSS, cannot be used in the program segment defined by EXEC. Use of this type of directive terminates the EXEC segment.

EXEC enables execution of a program segment at its actual loading address. The value placed in the location counter is the actual loading address of the independent segment.

Example

```
EXEC1     EXEC      >1F40
            .
            .
            .
          XEND
```

Syntax

[<label>] IDT '<string>' [<comment>]

When used, a label assumes the current value of the location counter.

Description

IDT assigns a name to the object module produced. The operand field contains the module name <string>, a character string of up to eight characters within single quotes. When a character string of more than eight characters is entered, the assembler prints a truncation warning message and retains the first eight characters as the program name.

Program modules to be linked by the link editor must include an IDT. The module names in the character strings of IDTs should be unique. The <string> on IDT is not automatically a DEFed symbol.

Example

```
0001              IDT     'EXAMPLE'
0002    0001      ONE     EQU 1
0003    0002      TWO     EQU 2
```

IDT assigns the name EXAMPLE to the module being assembled. The module name is then printed in the source listing as the operand of IDT and appears in the page heading of the source listing. The module name is also placed in the object code and is used by the link editor to determine the entry point for the module. The entry point must also appear as a symbol in a REF directive.

> **Note:**
>
> Uppercase letters and numerals are recommended within the quotes. The assembler accepts lowercase letters and special characters, but ROM loaders (for example) do not.

Syntax

[<label>] LIST [<comment>]

When used, the label assumes the current value of the location counter.

Description

LIST restores printing of the source listing. LIST is required only when UNL (stop source listing) is in effect and causes the assembler to resume listing. LIST is not printed in the source listing, but the line counter is incremented. The assembler does not print the comment.

Example

```
LIST
```

The printing of the source listing is restored.

Syntax

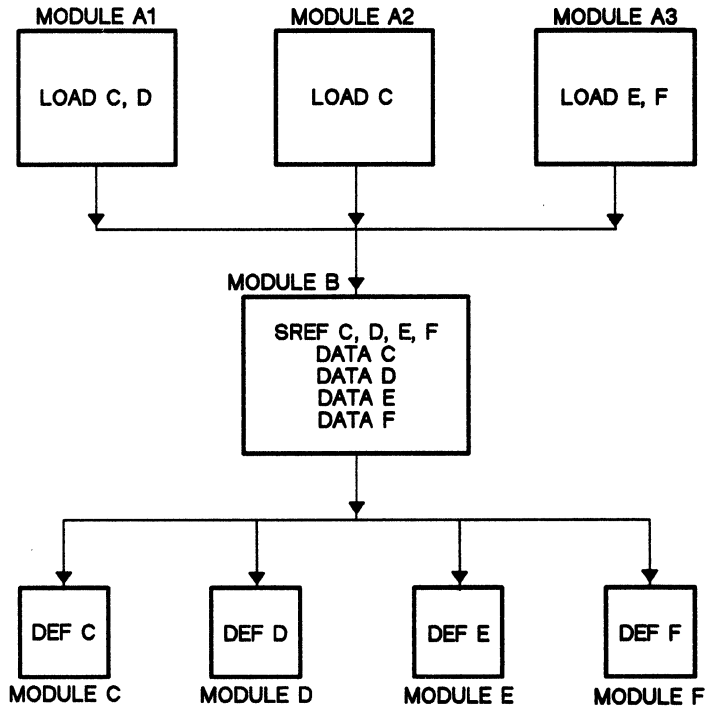
[<label>] LOAD <symbol>[,<symbol>] [<comment>]

When used, a label is assigned the current value of the location counter.

Description

LOAD makes one or more symbols available to other programs. The LOAD directive is like DEF, except that the symbols need not be used in the module containing LOAD. The symbols used in LOAD must be defined in some other module during link edit time. LOADs are used with SREFs. If a one-to-one matching of LOAD and SREF pairs does occur, there will be no unresolved references during link editing.

Example



Module A(n) uses a branch table in module B to obtain one module: either C, D, E, or F. Module B has an SREF for C, D, E, and F. SREF does not require that symbols C, D, E, and F have corresponding symbols defined in another module, so modules need not be included in one link editing time. Module C has a DEF for C; module D has a DEF for D; module E has a DEF for E; and module F has a DEF for F. Module A1 has a LOAD for modules C and D; module A2 has a LOAD for module C; and module A3 has a LOAD for modules E and F.

LOAD and SREF permit module B to be written to in order to handle a highly involved case and still be linked together without unnecessary modules. A(n) only has LOAD directives for its required modules. This is especially useful when developing large codes that may have more than a hundred modules. Not all modules are required to test a particular function.

If the link control file included A1 and A2, modules C and D would be pulled in from a specified library while modules E and F would not. If the link control file included A3, modules E and F would be pulled in while modules C and D would not. If the link control file included A2, module C would be pulled in while modules D, E, and F would not.

```
TASK TSTLOAD
FORMAT ASCII
PROGRAM 0
INCLUDE E:A1.MPO
INCLUDE E:B.MPO
FIND A:*.MPO
END
```

In this example using a PC/MS-DOS computer, the A:*.MPO is a selection of files that contain 990-tagged object modules for modules C, D, E, and F. In this case, only modules C and D are to be linked into the LOAD object module.

Syntax

[<label>] MLIB '<pathname>' [<comment>]

When used, a label assumes the current value of the location counter.

Description

MLIB provides the assembler with the name of a library containing macro definitions. The operand of MLIB is a directory pathname (constructed according to the conventions of the host operating system) enclosed in single quotes (see IDT and TITL directives). The operand field contains the pathname, a character string of up to 48 characters enclosed in single quotes; longer strings cause a truncation error message.

Example 1

```
MLIB 'DRC1:[PROJECT.STDMACS]'
MLIB 'DRC1:[PROJECT.DSPMACS]'
```

When the program finds macro call SUBMAC (not previously defined), the above example causes the macro to search first for a file named:

```
DRC1:[PROJECT.DSPMACS]SUBMAC.ASM
```

Then, if that file is not found, the macro searches for a file named:

```
DRC1:[PROJECT.STDMACS]SUBMAC.ASM
```

in that order.

Example 2

```
MLIB 'DRCO:[MOORE.MACLIB].ASM32'                    (VAX)
MLIB 'A:'                              (MS-DOS VER 1.25)
```

This example shows the typical use of MLIB on other systems.

> **Note:**
>
> On VAX/VMS systems, the filename of all files in the macro library must have an extension name of ".ASM". For example, if the statement: MLIB 'DRC1:[MACROS]' has been used, the VAX/VMS version of the macro library processor would expect to find files such as `MYMACRO.ASM`, `NEWMAC.ASM`, etc., within the macro library 'MACROS'.
>
> On PC/MS-DOS systems, the filename of all files in the directory that are to be found as macros must not have an extension. For example, if the statement: MLIB 'E:' has been used, then the PC/MS-DOS version of the macro library processor would expect to find files such as `MYMACRO`, `NEWMAC`, etc., within the current level directory.

Syntax

[<label>] OPTION <option list> [<comment>]

When used, the label assumes the current value of the location counter.

Description

OPTION selects several options for the assembler listing output. The <option-list> operand is a list of keywords, separated by commas, where each keyword selects a listing feature. The available <option-list> features are as follows:

DUNLST: Limit the listing of DATA directives to one line.

FUNLST: Turn off all list options.

NOLIST: Inhibit all listing output (overrides LIST directive).

SYMLST: Produce a symbol table list in the object file.

TUNLST: Limit the listing of TEXT directives to one line.

XREF: Produce a symbol cross-reference listing.

Example

```
OPTION  XREF
```

This example results in the production of a symbol cross-reference listing.

Syntax

[<label>] PAGE [<comment>]

When used, a label assumes the current value of the location counter.

Description

PAGE causes the assembler to continue the source program listing on a new page. PAGE is not printed in the source listing, but the line counter is incremented. The assembler does not print the comment. Using PAGE to divide the source listing into logical divisions improves program documentation.

Example

```
PAGE
```

PAGE causes the assembler to list a next source statement as the first statement on a new page in the source listing.

Syntax

[<label>] PEND [<comment>]

When used, a label is assigned the value of the location counter prior to modification.

Description

PEND ends a segment that is program-relocatable. This directive is provided as the program-segment counterpart to DEND and CEND. PEND, like DEND and CEND, places a value in the location counter and ends a segment that has defined succeeding locations as program-relocatable. Since PEND properly appears only in program-relocatable code, the relocation type of succeeding locations remains unchanged.

The value placed in the location counter by PEND is the maximum value attained by the location counter as a result of the assembly of all preceding program-relocatable code. PEND is invalid when used in absolute code.

Example

See PSEG.

Syntax

[<label>] PSEG [<comment>]

When used, a label is assigned the value that the directive places in the location counter.

Description

PSEG places a value in the location counter and defines succeeding locations as program-relocatable. The location counter is set to one of the following values:

- The maximum value the location counter attained as a result of the assembly of any preceding block of program-relocatable code, or
- Zero, if no program-relocatable code was previously assembled.

PSEG is provided as the program-segment counterpart to DSEG and CSEG. Together, the three directives provide a consistent method of defining the various types of relocatable segments.

Example

The following two sequences of directives are functionally identical:

SEQUENCE 1

```
DSEG
 .
 .
 .
<Data-relocatable code>
 .
 .
 .
DEND
CSEG
 .
 .
 .
<Common-relocatable code>
 .
 .
 .
CEND
PSEG
 .
 .
 .
PEND
 .
END
```

SEQUENCE 2

```
DSEG
 .
 .
 .
<Data-relocatable code>
 .
 .
 .
 .
CSEG
 .
 .
 .
<Common-relocatable code>
 .
 .
 .
 .
PSEG
 .
 .
 .
 .
 .
END
```

Syntax

[<label>] REF <symbol>[,<symbol>] [<comment>]

When used, a label is assigned the current value of the location counter.

Description

REF provides access to one or more symbols defined in other programs. Each symbol from another program module must be placed in the operand field of REF or SREF in the program module that requires the symbol. Below is a program named 'MAIN' that REFs a routine named 'SUBR1'. SUBR1 is not defined in this file.

```
IDT       'MAIN'
REF       SUBR1
  .
  .
  .
CALL      SUBR1
  .
  .
  .
END
```

If a symbol is listed in the REF statement, a corresponding symbol must also be present in a DEF statement in another source module. If a one-to-one matching of symbols does not occur, then an error occurs at link edit time. The link editor generates a summary list of all unresolved references.

Example

```
REF  ARG1,ARG2
```

This example causes the assembler to include symbols ARG1 and ARG2 in the object code so that the corresponding addresses may be obtained from other programs.

Syntax

[<label>] RORG [[<exp>] /<comment>/]

When a label is used, it is assigned the value that the directive places into the location counter.

Description

RORG defines succeeding locations as program-relocatable and initializes the location counter to either the value following the previous relocatable code of the program or to zero if no relocatable code has been previously assembled.

Since the location counter begins at zero, the length of a segment and the next available address within that segment are identical. For example, if a segment begins at >0 and ends at >E, the length is >F. The next available address is >F.

When the operand field is used, the operand must be an absolute or a relocatable expression that contains only previously defined symbols. (Symbols are defined by the EQU directive.) When the operand field is not used, previous data segments and specific common segments of a program replace the value of the location counter. If RORG appears in absolute code, a relocatable operand must be program-relocatable. RORG changes the location counter to program-relocatable and replaces its value with the operand value. In relocatable code, the relocation type of the operand must match that of the current location counter. The operand value replaces the current location counter value, and the relocation type of the location counter remains unchanged.

Example 1

```
RORG $-10 OVERLAY TEN WORDS
```

The $ symbol refers to the present location counter value. This has the effect of backing up the location counter by ten words. The instructions and directives following RORG replace the ten previously assembled words of relocatable code, permitting correction of the program without removing source records. If a label had been included, the label would have been assigned the value placed in the location counter. RORG would have no effect except at the end of an absolute block or a dummy block.

Example 2

```
SEG2 RORG
```

The location counter contents depend upon preceding source statements. Assume that after defining data for a program that occupied >44 words, AORG initiated an absolute block of code. The absolute block is followed by the RORG directive from Example 1. This places >0044 in the location counter and defines the location counter as relocatable. Symbol SEG2 is a relocatable value, >0044.

Syntax

[<label>] SREF <symbol>[,<symbol>] [<comment>]

When a label is used, the current value of the location counter is assigned to the label.

Description

SREF provides access to one or more symbols defined in other programs. Unlike REF, SREF does not require that a symbol have a corresponding symbol listed in a DEF statement of another source module. The SREFed symbol will be an unresolved reference, but is not included in the summary list of unresolved references.

Example

```
SREF   ARG1,ARG2
```

This example causes the assembler to include symbols ARG1 and ARG2 in the object code so that the corresponding addresses may be obtained from other programs.

Syntax

[<label>] TEXT [-]'<string>' [<comment>]

When used, a label is assigned the location at which the assembler places the first character.

Description

TEXT places one or more characters of a string of characters in successive words of program memory. The assembler negates the last character of the string when the string is preceded by a unary minus (-) sign. The operand field contains a character string of up to 52 characters enclosed in single quotes, which may be preceded by a unary minus sign.

Example

```
MSG1  TEXT  'EXAMPLE'  MESSAGE HEADING
```

In this example, TEXT places the eight-bit ASCII character representations in memory and fills the unused byte of the last word with an ASCII space (>20). This space is considered as the last character if the negate option is specified. The result is >4558, >414D, >504C, and >4520. The label MSG1 is assigned the first word's address, which contains the value >4558.

Syntax

[<label>] TITL '<string>' [<comment>]

When used, a label field assumes the current value of the location counter.

Description

TITL supplies a title to be printed in the heading of each page of the source listing. Unlike IDT, TITL is not printed in the source listing. When a title is desired in the heading of the listing's page, TITL must be the first source statement submitted to the assembler. The assembler does not print the comment because TITL is not printed. The line counter is incremented.

The operand field contains the title (string) and a character string of up to 50 characters enclosed in single quotes. When more than 50 characters are entered, the assembler retains the first 50 characters as the title and prints a syntax error message.

When TITL is the first source statement in a program, the title is printed on all pages until another TITL is processed. Otherwise, the title is printed on the next page after TITL is processed, and on subsequent pages until another TITL is processed.

Example

```
TITL  '**REPORT GENERATOR**'
```

This example causes the title **REPORT GENERATOR** to be printed in the page headings of the source listing.

Syntax

[<label>] UNL [<comment>]

When used, the label assumes the value of the location counter.

Description

UNL halts the source listing output until the occurrence of a LIST directive. UNL is not printed in the source listing, but the source line counter is incremented. UNL is frequently used in macro definitions to inhibit the listing of the macro expansion. The assembler does not print the comment.

UNL can be used to reduce assembly time and the size of the source listing.

Example

```
NOPRINT   UNL   STOP LISTING
```

The source listing is halted until a LIST directive occurs.

Syntax

[<label>] XEND [<comment>]

When used, a label is assigned the value placed in the location counter by the XEND directive.

Description

XEND terminates the block definition of an independently stored program segment, previously defined by EXEC. The command field contains XEND. XEND results in setting the location counter to the value attained by the location counter before EXEC was issued, plus the difference between the most recent value in the location counter and the loading address of EXEC.

Without using EXEC, a warning message is issued.

Example

See EXEC.

7.8 Source Listing Format

The source listings show the source statements and the resulting object code. Each page of the source listing has a title line at the top, on which is printed a title supplied by a TITL (title) directive. If TITL is not used, the title line is left blank. A page number is printed to the right of the title. The printer inserts a blank line below the title line and prints a line for each source statement listed.

Each source statement line contains a source statement number, a location counter value, the assembled object code, and the source statement as entered. A source statement may result in more than one word of object code. The assembler prints the location counter value and object code on a separate line for each additional word. Each added line is printed immediately following the source statement line. Figure 7-1 is an example of a source statement line.

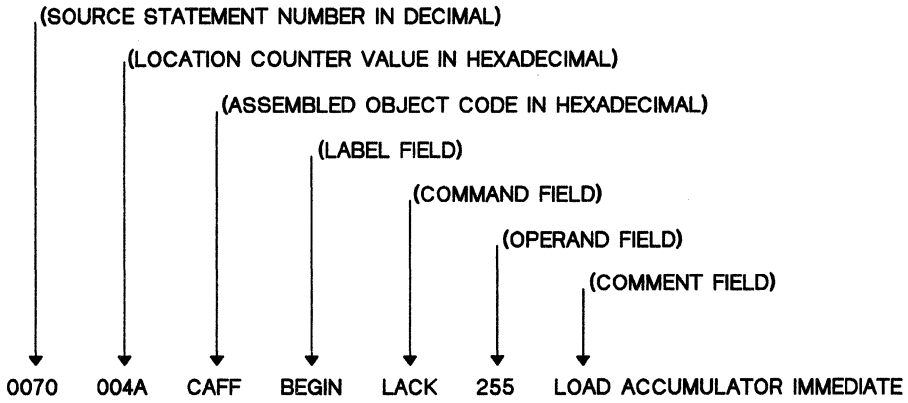


Figure 7-1. Source Statement Line Example

The source statement number, 0070 in the example, is a four-digit decimal number. Source records are numbered in the order in which they are entered, including those source records that are not listed (e.g., TITL, LIST, UNL, and PAGE directives are not listed; source records between UNL and LIST are not listed). The difference between two source record numbers printed immediately in line indicates the number of source records entered and not listed. Source records generated by a macro call, however, are renumbered starting at line number 0001. The original line-numbering sequence continues after the macro expansion is complete.

The next field in the source listing contains a hexadecimal location counter value. In the example, 004A is the location counter value. Since not all directives affect the location counter, the location counter field is blank for those directives that do not affect it, such as the IDT (program identifier), DEF (external definition), END (program end), EQU, REF, and SREF directives.

The third field is the object code field which contains the hexadecimal representation of the object code, (>CAFF in the above example). All machine instructions and the DATA and TEXT directives use this field to list object code. The EQU directive places the value corresponding to the label in the object code field.

The fourth field contains the characters of the source statement as they were scanned by the assembler. The maximum line length that the assembler will accept is 60 characters. Spacing in this field is determined by the spacing in the source statement. The four fields contained in source statements are aligned in the listing only when they are aligned in the source statements or when tab characters are used. Each of the four fields must be separated by at least one blank space.

7.9 Object Code

A major advantage of the TMS320C25 Macro Assembler is its ability to generate relocatable object code modules that can then be linked by the link editor to form an executable program. The ability to relocate modules simplifies the programming task. Programs designed as a set of modules are easier to code, test, and debug, and are also easier to understand and maintain. Relocatability also permits multiple programmers to work on a program's components. Relocatable code includes information that allows a link editor to place the code in any available area of memory, thus providing the most efficient use of available memory. Absolute code must be loaded into a specified area of memory and cannot be moved.

Object code generated by an assembler constitutes the assembled program, and consists of machine language instructions, addresses, and data. The code includes absolute, program-relocatable, data-relocatable, and common-relocatable segments.

In assembly language source programs, symbolic references to locations within a relocatable segment are called relocatable addresses. These addresses are represented in the object code as displacements from the beginning of a specified segment. A program-relocatable address, for example, is a displacement into the program segment. At load time, all program-relocatable addresses are adjusted by a value equal to the load address (the load address defines the beginning of the module). Data-relocatable addresses are represented by a displacement into the data segment. There may be several types of common-relocatable addresses in the same program, since distinct common segments may be relocated independently of each other.

The assembler produces object code that may be linked to other object code modules or programs, and is loaded directly into the processor. Object code consists of records containing up to 71 ASCII characters. Corrections on record data can be made via a keyboard, making reassembly unnecessary. Figure 7-2 is an example of object code.

```
K0000FACT    91006BCA01B6000BCA01B6001BCA02B6002BCA03B6003B3C037F240F   FACT 1
BA002BCE14B6003BCA04B6004B3C04BA003BCE14B6004B3C04BA003BCE14B60047F16BF FACT 2
BCA05B6005B3C05BA004BCE14B6005B3C05BA003BCE14B6005B3C05BA002BCE147F151F FACT 3
B6005BCA06B6006B3C06BA005BCE14B6006B3C06BA004BCE14B6006B3C06BA0037F169F FACT 4
BCE14B6006B3C06BA002BCE14B6006BCA07B6007B3C07BA006BCE14B6007B3C077F14BF FACT 5
BA005BCE14B6007B3C07BA004BCE14B6007B3C07BA003BCE14B6007B3C07BA0027F158F FACT 6
BCE14B60077FD8BF                                                        FACT 7
:       FACT    8/7/84   16:42:51     ASM32020    0.6 84.140             FACT 8
```

Figure 7-2. Sample Object Code

7.9.1 Object Code Format

Object code is formatted to contain records made up of fields sandwiched between tag characters. Table 7-8 lists field and tag character information.

A tag character occupies the first position on each record of object code and identifies the fields it precedes. The specific tag character used depends on the function of the field with which it is associated. The following paragraphs detail the various tag characters and their associated fields.

Tag character K is placed at the beginning of each program and is followed by two fields. Field one contains the number of words of program relocatable code; field two contains the program identifier assigned to the program by an IDT directive. When no IDT is entered, NO$IDT is put into field two. The link editor uses the program

identifier to identify the program, and the number of words of program-relocatable code to determine the load bias for the next module or program.

The tag character M is used when data or common segments are defined in the program and is followed by three fields. Field one contains the length, in words, of data- or common-relocatable code; field two contains the data or common segment identifier; and field three contains a common number. The identifier is a six-character field containing the name $DATA (padded on the right by one blank) for data segments and $BLANK for blank common segments. If a named common segment appears in the program, an M tag appears in the object code, with an identifier field corresponding to the operand in the defining CSEG directive(s). Field three of the M tag consists of a four-character hexadecimal number defining a unique common number to be used by other tags referencing or initializing data of that particular segment. For data segments, this common number is always zero. For common segments (including blank common), common numbers are assigned in increasing order. The maximum number of common segments that a program may contain is 127.

Tag characters 1 and 2 are used with entry addresses. The associated field is used by the linker to determine the entry point where execution starts when linking is complete. Tag character 1 is used when the entry address is absolute; tag character 2 is used when the address is relocatable. The field lists the address in hexadecimal.

Tag characters 9, A, S, and P are used with load addresses required for data words to be placed at other than the next immediate memory addresses. Tag character 9 is used when the load address is absolute; A when the load address is program-relocatable; S when the load address is data-relocatable; and P when the load address is common-relocatable. Field one contains the load address. Field two is only used with P and contains the common number.

Tag characters B, C, T, and N are used with data words. Tag character B is used when the data is absolute (i.e., an instruction word or a word containing text characters or absolute constants). B is used for absolute word data (16 bits). Tag character C is used for a word containing a program-relocatable address; tag character T for a word containing a data-relocatable address; tag character N for a word containing a common-relocatable address. Field one contains the data word. The linker places the data word in the memory location specified in the preceding load address field or in the memory location that follows the preceding data word. Field two is only used with N and contains the common number.

Tag characters #, %, and & are also used when an instruction's multibit field refers to a data element in a DSEG, PSEG, or CSEG. Tag character # identifies an instruction containing a reference to a multibit data-relative item. The second field following the tag contains a mask indicating to the link editor the width of the field (mask = >007F indicates the least significant 7 bits). The link editor generates the final version of this instruction by adding the beginning location of the data segment to the masked data word, and re-inserting the sum in the multibit field within the data word. Note that field overflow may occur in the link edit operation, and error messages may be generated that were not evident at assembly time, which may give unpredictable results. The description of the % tag is the same as above, except that it represents the use of a program-relative item as the operand. The fields used with the & tag are identical to the # and % tags, except that the second field is the common number, and the mask becomes the third field.

Tag characters 5, 6, and W are used for external definitions. Tag character 5 is used when the location is program-relocatable; 6 when the location is absolute; and W when the location is data- or common-relocatable. The link editor uses the fields to provide the desired linking to the external definition. Field one contains the

location of the last appearance of the symbol; field two contains the symbol of the external definition; and field three of tag character W contains the common number.

Tag character 4 is used for external references when the last appearance of the externally referenced symbol is in absolute code. Tag character 4 is associated with two fields: field one contains the location of the last appearance of the symbol, and field two contains the symbol itself.

Tag character E is used for external references. An E tag is used when a non-zero quantity is added to a reference. Field 1 identifies the reference by occurrence in the object code (0, 1, 2, ...). In other words, the value in field one is an index to references identified by the 4 and Y tags in the object code. The list is maintained by order of occurrence (i.e., the first entry in the list is the symbol located in field two of the first 4 or Y tag). Field 2 contains the value to be added to the reference after the reference is resolved.

Tag character ! is used when a multibit field of an instruction refers to an external reference. The format of the ! sequence is:

! (external symbol number) (opcode/offset) (mask)

This tag and its associated fields are processed the same as that of the # tag.

Tag characters G, H, and J are used when the symbol table option (see the OPTION directive) is specified. Tag character G is used when the location or value of the symbol is program-relocatable; H when the location or value of the symbol is absolute; and J when the location or value of the symbol is data- or common-relocatable. Field one contains the location or value of the symbol; field two contains the symbol to which the location is assigned; field three is used only with tag character J and contains the common number.

Tag character U is generated by the LOAD directive. The symbol specified is treated as if it were the value specified in an INCLUDE command to the loader. Field one contains zeroes, and field two contains the symbol for which the loader will search for a definition.

Tag character Y is used for secondary external references when the last appearance of the externally referenced symbol is in absolute code. Tag character Y is associated with two fields: field one contains the location of the last appearance of the symbol, and field two contains the symbol itself.

Tag character 7 precedes the checksum, and is placed at the end of the set of fields in the record. The checksum is an error detection word formed as the record is being written. The checksum is the two's complement of the sum of the characters' 8-bit ASCII values from the first tag of the record through the checksum tag (tag character 7).

Tag character 8 is also associated with the checksum field but is used when the checksum field is to be ignored (as when changing the object code).

Tag character D, used to specify a load bias, has an associated field containing the absolute address used by the loader to relocate symbols. The link editor does not accept the D tag.

Tag character F is placed at the end of the record, and it may be followed by blanks.

The end of each record is identified by the tag character 7, followed by the checksum field and tag character F. The assembler fills the rest of the record with blanks and a sequence number, and begins a new record with the appropriate tag character.

The last record of an object module has a colon (:) in the first character position of the record, followed by the module name, assembly date, and assembly time.

Table 7-8 defines the object record format and tags.

Table 7-8. Object Record Format and Tags

TAG	1ST FIELD	2ND FIELD	3RD FIELD
	(MODULE DEFINITION)		
K	PSEG LENGTH	PROGRAM ID(8)	
M	DSEG LENGTH	$DATA	0000
M	BLANK COMMON LENGTH	$BLANK	COMMON #
M	CSEG LENGTH	COMMON NAME(6)	COMMON #
	(ENTRY POINT DEFINITION)		
1	ABSOLUTE ADDRESS		
2	P-R ADDRESS		
	(LOAD ADDRESS)		
9	ABSOLUTE ADDRESS		
A	P-R ADDRESS		
S	D-R ADDRESS		
P	C-R ADDRESS	COMMON #	
	(DATA WORD)		
B	ABSOLUTE 16-BIT VALUE		
C	P-R ADDRESS		
T	D-R ADDRESS		
N	C-R ADDRESS	COMMON #	
#	OPCODE/DR ADDRESS	MASK	
%	OPCODE/PR ADDRESS	MASK	
&	OPCODE/CR ADDRESS	COMMON #	MASK
	(EXTERNAL DEFINITIONS)		
6	ABSOLUTE VALUE	SYMBOL(6)	
5	P-R ADDRESS	SYMBOL(6)	
W	D-R/C-R ADDRESS	SYMBOL(6)	COMMON #
	(EXTERNAL REFERENCES)		
4	ABSOLUTE ADDRESS OF CHAIN	SYMBOL (6)	
E	SYMBOL INDEX NUMBER	ABSOLUTE OFFSET	
!	SYMBOL INDEX NUMBER	OPCODE/OFFSET	MASK
	(SYMBOL DEFINITIONS)		
G	P-R ADDRESS	SYMBOL(6)	
H	ABSOLUTE VALUE	SYMBOL(6)	
J	D-R/C-R ADDRESS	SYMBOL(6)	COMMON #

Table 7-8. Object Record Format and Tags (Concluded)

TAG	1ST FIELD	2ND FIELD	3RD FIELD
(FORCE LOAD)			
U	0000	SYMBOL(6)	
(SECONDARY EXTERNAL REFERENCE)			
Y	ABSOLUTE ADDRESS OF CHAIN	SYMBOL(6)	
(CHECKSUM)			
7	VALUE		
(IGNORE CHECKSUM)			
8	ANY VALUE		
(LOAD BIAS)			
D	ABSOLUTE ADDRESS		
(END OF RECORD)			
F			
(END OF MODULE)			
:	(IMPLEMENTATION DEPENDENT)		
(PROGRAM ID [SYMT OPTION])			
I	P-R ADDRESS	PROGRAM ID(8)	

NOTES:
1. All field widths are four characters unless otherwise specified.
2. If the first tag is >01, the file is in compressed object format.
3. P-R denotes program segment, relative address.
   D-R denotes data segment, relative address.
   C-R denotes common segment, relative address.

7.9.2 Changing Object Code

Object code may be corrected without reassembling a program by changing or adding one or more records. One additional tag character is recognized by the loader to permit specifying a load point. The additional tag character, D, may be used in object records changed or added manually.

Tag character D is followed by a load bias (offset) value. The loader uses this value instead of the load bias computed by the loader itself. The loader adds the load bias to all relocatable entry addresses, external references, external definitions, load addresses, and data. The effect of tag character D is to specify that area of memory into which the loader loads the program. The tag character D and the associated field must be placed ahead of the object code generated by the assembler.

Correction of the object code may only require changing a character or a word in an object code record. Records may be duplicated up to the character or word in error. Then the correct data replaces the incorrect data, and the remainder of the record up to tag character 7 is duplicated. When the checksum is verified as the record is loaded, the changes made cause a checksum error. The tag character 7 should be changed to 8. This causes the checksum error resulting from the record change to be ignored.

When more extensive changes are required, an additional object code record(s) may be written. Each record is begun with a tag character 9, A, S, or P (load address tag characters), followed by an absolute load address or a relocatable load address. This may be an address into which an existing object code record places a different value. The new value on the new record overrides the old value when the new record

follows the old record in the loading sequence. The load address is followed by a tag character B, C, T, or N (data word tag characters) and an absolute data word or a relocatable data word. Additional data words preceded by appropriate tag characters may follow. When additional data is placed at a nonsequential address, another load address tag character is written followed by the load address and data words preceded by tag characters. When the record is full, or all changes have been written, tag character F is written to end the record.

When additional memory locations are loaded as a result of changes, field one of tag character K containing the number of words of relocatable code must be changed. For example, if the object field written by the assembler contained 1000 hexadecimal words of relocatable code and the change has added eight words in a new object record, additional memory locations will be loaded. In the object code file, the value following the tag character K is changed from 1000 to 1008. The tag character 7 is also changed to 8 in that record.

When added records place corrected data in locations previously loaded, the added records must follow the incorrect records. The loader processes the records as they are read from the object medium. The last record that affects a given memory location determines the contents of that location at execution time.

The object code records that contain the external definition fields, the external reference fields, the entry address field, and the final program start field, must follow all other object records. An additional field or record may be added to include reference to a program identifier. The tag character is 4, and the hexadecimal field contains zeros. The second field contains the first six characters of the IDT character string. External definitions may be added using tag character 5 or 6, followed by the relocatable or absolute address, respectively. The second field contains the defined symbol, filled to the right with blanks when the symbol contains less than six characters.

7.10 Cross-Reference Listing

The assembler prints an optional cross-reference listing following the source listing. (The cross-reference listing is created by using the OPTION directive.) The format of the listing is shown in Figure 7-3.

```
LABEL    VALUE  DEFN   REFERENCES


BASE2    029B   0095
BC       0236   0009   0003   0025   0030   0035   0060   0061   0064   0067   0069
BCDONE   REF    0004   0082   0084   0086   0088   0090   0092   0094
CTXT0    023B   0014   0020   0079
CTXT1    023C   0015   0021   0077
CTXT2    023D   0016   0022   0078
IORT     SREF   0005
IORT1B   UNDF          0039   0043
IORT1F   0256   0040   0028
IORT2F   025B   0044   0036
IORT3F   0281   0076   0072
IORTB1   0298   0093   0058
IORTB2   0295   0091   0055
IORTB3   0292   0089   0052
IORTB4   028F   0087   0049
```

Figure 7-3. Cross-Reference Listing Format

As shown in Figure 7-3, the assembler prints in the LABEL column each symbol defined or referenced in the assembly. The VALUE column contains a four-digit hexadecimal number and is possibly followed by either a character or a name that represents the attributes of the symbol. A four-digit hexadecimal number represents the value assigned to the symbol. The characters following the four-digit number or the names that may be in the VALUE column have their meanings listed in Table 7-9. The number of the statement in which the symbol is defined appears in the DEFN (definition) column. For undefined symbols, this column is left blank. The REFERENCES column lists the number of statements that reference the symbol. A blank in this column indicates the symbol was defined but never used.

Table 7-9. Assembly Symbol Attributes

CHARACTER OR NAME	MEANING
REF	External reference (REF)
UNDF	Undefined
SREF	Secondary reference (SREF)
'	Symbol defined in a program segment
"	Symbol defined in a data segment
+	Symbol defined in a common segment

7.11 Assembler Error Messages

The assembler issues the following three types of error messages:

- Nonfatal
- Fatal
- Informative

When the assembler completes an assembly, it indicates any errors it encounters in the assembly listing. The assembler indicates errors following the source line in which they occur. The errors are referenced by number. At the end of a module (as delineated by the IDT/END directive pair), the corresponding messages are printed. Table 7-10 lists non-fatal error messages, and Table 7-11 lists fatal messages. In Table 7-12, assembly information messages are given.

Table 7-10. Non-Fatal Error Listing

MESSAGE	EXPLANATION/RESPONSE
WARNING - 'CEND' ASSUMED	Occurs when CSEG is not terminated by CEND.
WARNING - 'DEND' ASSUMED	Occurs when DSEG is not terminated by DEND.
WARNING - 'PEND' ASSUMED	Occurs when PSEG is not terminated by PEND.

WARNING - 'DSEG' ASSUMED	This is a warning that the following two statements have the same result:
	`CSEG 'DATA'`
	`DSEG`

WARNING - SYMBOL TRUNCATED	The maximum length for a symbol is 6 characters. The assembler ignores the extra characters.
WARNING - STRING TRUNCATED	Check the syntax for the directive question to determine the maximum length for the string.
WARNING - TRAILING OPERAND(S)	The assembler found fewer or more operands than expected in the flagged instruction.
WARNING - BYTE VALUE TRUNCATED	A value to be used as a byte is larger than can be loaded into the space for a byte.
WARNING - NULL STRING DEFINED	An empty string (i.e., length = 0) is defined for string input, for directives that require a null string operand.

Table 7-11. Fatal Error Listing

MESSAGE	EXPLANATION/RESPONSE
ABSOLUTE VALUE REQUIRED	A relocatable symbol was used where an absolute symbol was expected.
DISPLACEMENT TOO BIG	The maximum value of the operand was exceeded.
INVALID EXPRESSION	This may indicate invalid use of a relocatable symbol in arithmetic.
EXPRESSION OUT OF BOUNDS	Range limit for the value of the operand was exceeded.
DUPLICATE DEFINITION	The symbol appears as an operand of a REF statement, as well as in the the label field of the source, or the symbol appears more than once in the label field of the source.
INVALID RELOCATION TYPE	An absolute variable cannot be made relocatable.
INVALID OPCODE	The command field of the source record has an entry that is not a defined instruction, directive, or macro.
INVALID OPTION	The option given in the OPTION directive is invalid. An option is often misspelled.
INVALID REGISTER VALUE	The register specified is too large or too small. Only values of 0 to 4 are allowed for AR0 to AR4, respectively.
INVALID SYMBOL	The symbol has invalid characters in it.
VALUE TRUNCATED	The value is too big for the field and has been truncated. This message also appears when a label string exceeds its maximum length.
SYMBOL USED IN BOTH REF AND DEF	
COPY FILE OPEN ERROR	File does not exist or is already being used.
EXPRESSION SYNTAX ERROR	Unbalanced parentheses or invalid operations on relocatable symbols.
INVALID ABSOLUTE CODE DIRECTIVE	The directives PEND, DEND and CEND have no meaning in absolute code.
LABEL REQUIRED	The flagged directive must have a label.
BLANK MISSING	A blank or blanks must separate each field of the source statement.
COMMA MISSING	Expected a comma but did not find one. Usually means that more operands were expected.
COPY FILENAME MISSING	Filename specified cannot be found.
SYMBOL REQUIRED	OPTION, DEF, REF, SREF, and LOAD directives require symbols as operands.
OPERAND MISSING	An operand must be supplied.
CLOSE (') MISSING	All strings must be enclosed in quotes.
CLOSE (')') MISSING	Mismatched parenthesis.
STRING REQUIRED	TEXT directive used with no text following.
PASS1/PASS2 OPERAND CONFLICT	A symbol in the symbol table did not have the same value in PASS1 and PASS2.
SYNTAX ERROR	Error in syntax.
UNDEFINED SYMBOL	The symbol has not been REFed, or it has been DEFed but not used.
DIVIDE BY ZERO	An expression or well-defined expression contains invalid division.
ILLEGAL SHIFT COUNT	The shift count requested is not valid.

Table 7-12. Assembly Information Message Listing

MESSAGE	EXPLANATION/RESPONSE
OPCODES REDEFINED	As a result of an MLIB directive, one or more assembler opcodes have been redefined by a MACRO within a MACRO directory. The user should take action if this is not intended.
MACROS REDEFINED	As a result of an MLIB directive, one or more currently defined MACROS have been redefined by a MACRO (of the same name) within a MACRO directory. The user should take action if this is not intended.

8. Assembler Macros

The TMS320C25 Macro Assembler supports macro calls and definitions along with macro-conditional assembly for simplifying programming and consolidating frequently repeated source code. Macros may be defined with the assembler input or in a library (directory) of external files to be included at link time.

Major topics discussed in this section are listed below.

8.1 Macro Definitions

The TMS320C25 Macro Assembler recognizes a macro definition language that is used to simplify programming. A macro definition is a set of source statements (machine instructions, macro statements, and assembler directives), which constitute a template for generating other statements within a source program. Macro definitions consist of model statements and statements containing macro verbs. They are used to define macros and macro variables and to determine which model statements are assembled.

When the assembler processes a macro call, it substitutes the predefined statements of the macro definition for the macro call statement in the source program, and assembles the substituted statements as if they had been included in the source program.

Macro definitions are usually created by including lines of code in a predefined format within the assembler source file. In general, this format requires a symbolic line marking the start of a macro definition. The macro name is placed in the line's label field, and the string '$MACRO' is placed in the operand field. A list of formal parameters separated by commas may be placed in the operand field.

The elements of the macro assembly language are labels, strings, constants, variables, operators, keywords, and verbs. A macro definition consists of model statements and statements containing macro verbs used to define the macro and macro variables and determine which model statements are assembled. All macro statements that do not contain verbs are processed as model statements. A model statement results in an assembly language source statement.

Macros may be defined in-line with the normal assembler source code, provided that the macro definition appears before that macro is called. Macro definitions are usually placed at the top of the assembler source file. This allows easy reference to the macro definitions since they are in one location.

Macros may also be defined in external files. These files are simply text files (like the assembler source file) that contain macros defined in the same manner as those defined in the main assembler source file. Only one macro may be defined in a file. The assembler is informed of the existence of a macro library (i.e., a collection of macro files) by means of the MLIB directive (see Section 7.6.5). An example of the use of the MLIB directive is:

```
MLIB 'E:'
```

The string enclosed in the quotes represents a directory name in the format required by the host operating system.

To illustrate the use of a macro library, a library of macro definitions is assumed to be contained in a directory named 'E:' and a file named 'CPXADD' that is a member of that directory. If the macro call

```
LABEL   CPXADD   CX1,CX2
```

is found in the assembler source, the in-memory macro table is first searched for the definition of CPXADD. CPXADD will be in the macro table if CPXADD was previously defined in the assembler source file or was previously encountered and read from a macro file. If the definition is not found in the macro table, a search of the normal assembler opcode/directive table is made. If found in the assembler opcode/directive table, the opcode is assembled as a normal machine instruction. If not, an attempt is made to find the file whose name is formed by appending the macro name to the MLIB name. If more than one MLIB directive has been encountered, the most recently defined library is searched first; then all remaining libraries are searched. If the file

is found, the macro definition is copied into the assembler's macro file (in a compressed format), and an entry is made in the macro table for later use.

Because of the sequential search for matching definitions (library search following the opcode/directive table search), a macro defined in a library will not automatically redefine a machine instruction, although this is easily done using an in-line macro definition. To extend this capability to the macro library, that library should include a text file named 'MLIST', which contains the names of the opcodes and currently defined macros (one name per line, starting with column one) which are to be redefined.

A typical MLIST file may be constructed as follows, using the appropriate system text editor:

```
file name      <MLIB directory name>.  MLIST

 record 1          ADD                 (opcode)
 record 2          LACK                (opcode)
 record 3          DMOV                (opcode)
 record 4          FSUB                (macro)
                                       eof(MLIST)
```

This MLIST file is read when the MLIB directive is processed. If a name found there matches a currently defined opcode or a name in the macro table, the matching entry is removed from its table. This forces a search of the libraries, since the name will not be found elsewhere. When a name is found matching an opcode, the message:

```
' ****  OPCODES REDEFINED'
```

is printed in the assembler listing, following the printing of the MLIB statement. A similar message:

```
' **** MACROS REDEFINED'
```

appears when currently defined macros are redefined. If this is intended, then no action is required; if not, then some action is necessary, such as the deletion of some or all of the records in the MLIST file.

The name of a macro in file should be the same as the file name; otherwise, some inefficiency in macro usage will result. If the file named CPXADD contains a definition line such as:

```
CPXMUL    $MACRO    MR, MD
```

an entry for a macro named CPXMUL will be made in the internal macro table. The next call to CPXADD will be recognized as undefined and reentered into the internal macro table as CPXMUL.

Note that the use of an MLIST file to override the assembler opcode table can result in unpredictable behavior of the assembler. Care should be taken in using this option.

8.1.1 Sample Macros

The following example defines a macro named INCX. $MACRO identifies the beginning of the macro definition, and $END identifies the end of the macro definition. LACK 1, ADD X, and SACL X are model statements which will be placed into the source program upon a macro call. The macro INCX may be used in the source program as often as necessary.

```
INCX    $MACRO
        LACK    1
        ADD     X
        SACL    X
        $END
```

The macro INCX may be called by simply placing the line INCX within the source file. The macro assembler will replace this line with the remainder of the definition, i.e.:

```
LACK    1
ADD     X
SACL    X
```

X must be a symbol representing a memory address in the source program assigned by the EQU directive. INCX is limited because the macro can only be used with the single memory location X. The INC macro can be used with any memory location:

```
INC     $MACRO  M
        LACK    1
        ADD     :M.S:
        SACL    :M.S:
        $END
```

M is a macro parameter that is replaced by the actual parameter when the macro is called. M.S is the string component of this variable, i.e., the symbol representation of the variable. For example, the line INC Y will be replaced by:

```
LACK    1
ADD     Y
SACL    Y
```

Likewise, INC Z will be replaced by:

```
LACK    1
ADD     Z
SACL    Z
```

Another component of a macro variable is the value component, as shown in the following example:

```
ADDK    $MACRO  X,NUM       X and NUM are parameters.
        LACK    :NUM.V:     NUM V is the value component
                            of parameter NUM.
        ADD     :X.S:
        SACL    :X.S:
        $END
```

The macro call ADDK Y,3 will result in:

```
LACK    3
ADD     Y
SACL    Y
```

8.2 Labels

A macro label consists of one to six characters. The first character must be alphabetic, optionally followed by alphanumeric characters. Macro labels are used to determine the sequence of processing of statements in a macro definition when the statements are not to be processed in order.

Examples of valid macro labels:

```
L1

NXTPNT

C
```

8.3 Strings

Macro strings consist of one or more characters with enclosing quotes. Macro strings are defined in the same manner as the character string used in the assembly language source statement (see Section 7.4).

Example of strings:

```
'ONE'

'   '  (three blank spaces)
```

8.4 Constants

Constants for macros are defined in the same manner as constants in the assembly language source statements (see Section 7.3).

Examples of constants:

```
>9F3C

$       (current location counter value)
```

8.5 Variables

Variables are symbols, used within a macro, which take on values through various mechanisms in the macro definition language. The maximum length of a variable is six characters. Macro variables are strictly local, i.e., they are available only to the macro that defines them. Macro $VAR (variable declaration verb) statements declare variables for a macro definition.

The macro assembly language permits concatenation of macro variable components with strings, characters of model statements, and other macro variables. Variables are represented in the same manner as symbols in the Assembler Symbol Table (AST). This table maintains all the references to the variables, symbols, and labels used.

8.5.1 Parameters

Parameters are macro variables that are declared in the $MACRO (macro definition verb) statement at the beginning of the macro definition. The sequence of parameters in the operand field of the $MACRO statement corresponds to the sequence of operands in the operand field of the macro call. In the expansion of a macro call, the parameters have values that are associated with the corresponding operands in the macro call.

Examples of $MACRO statements with parameters:

```
LABEL   $MACRO    A,B3

NAME    $MACRO    O,RC,AMT
```

8.5.2 Macro Symbol Table (MST)

The macro translator maintains a Macro Symbol Table (MST) similar to the Assembler Symbol Table (AST). Each entry consists of four components: string, value, length, and attributes of a variable or parameter. The macro assembler places parameters in the MST while processing a macro call. Variables are placed in the MST as the assembler processes the macro $VAR statements that declare variables.

An entry's string component in the MST contains a character string assigned to the macro variable or parameter by the macro expander. The value component contains the numerical equivalent of the string component if the string component is an integer. The value component can also contain the numerical value of the symbol if the string component is a symbol in the AST.

If a parameter is an operand list, the value is the length of the list. The length component contains the number of characters in the string component. The attribute component of the MST is a bit vector, the bits of which correspond to the attributes of the variable or parameter.

Macro definition example:

```
ADDK $MACRO X,NUM
```

The $MACRO directive defines the beginning of the definition of the macro ADDK with parameters X and NUM.

Macro call example:

```
ADDK    VAR1,3
```

With the MST now containing parameters X and NUM, the string component of parameter X is the character string VAR1. The attribute component indicates that the parameter is supplied in a macro call. The length component is four. The string component of parameter NUM is the ASCII character 3. The value component is three (expressed as a binary number) and the length component is one. The attribute component indicates that the parameter is supplied in the macro call.

Each macro variable component may be accessed individually. Reference to a variable component is made in either binary mode or string mode. In the binary mode, the referenced macro-variable component is treated as a signed 16-bit integer. Binary mode access is made by writing the variable name and component. When a reference is made to the string component of a macro variable in binary mode, the 16-bit integer value of the ASCII representation of the first two characters of the string is obtained. In the macro definition and call examples above, the binary-mode value of the string component of X is >5641, which is the ASCII representation for VA.

String-mode access of macro-variable components is signified by enclosing the variable in a pair of colons, e.g., :X:. Colons are always used in pairs to enclose a variable name. If a component qualifier is used, the pair of colons enclose the entire qualified name.

8.5.3 Variable Qualifiers

Parameter or variable components may be specified using the names shown in Table 8-1. The variable name is followed by a period '.' and a single-letter qualifier. The following examples refer to previous macro examples using ADDK.

Examples of qualified variables:

`X.S` String component of variable VAR1. X.S equals the binary equivalent for VA or >5641. A string mode indicated as :X.S: is equal to the string 'VAR1'.

`X.A` Attribute component of variable X. This component may be accessed by use of logical operators and attribute keywords (described in Table 8-3).

`X.V` Value component of variable X.

`X.L` Length component of variable X. In the first example of the macro call for the macro ADDK,:X.L: = 4.

Table 8-1. Variable Qualifiers

QUALIFIER	MEANING
S	The string component of the variable
A	The attribute component of the variable
V	The value component of the variable
L	The length component of the variable

If a variable is not followed by a period '.' and a single-letter qualifier, it is referred to as an unqualified variable. Except in an $ASG statement, an unqualified variable defaults to the string component of the variable. In the two following examples, the concatenated strings are equivalent:

Example 1:	:CT.S:'WAY'	Variable CT qualified
Example 2:	:CT:'WAY'	Variable CT unqualified

In model statements, binary references to macro variables MUST be qualified.

All symbols in the Assembler Symbol Table (AST) have symbol components. (All components of macro parameters and the values of all AST symbols are directly accessible.) In order for other components to be accessed in a macro, the symbol must be assigned to the string component of a macro variable, using $ASG (value assignment verb). The additional qualifiers shown in Table 8-2 may be used with the macro variable to access the symbol components of the AST symbols.

Table 8-2. Variable Qualifiers for Symbol Components

QUALIFIER	MEANING
SS	String component of a symbol that is the string component of a variable
SV	Value component of a symbol that is the string component of a variable
SA	Attribute component of a symbol that is the string component of a variable
SL	Length component of a symbol that is the string component of a variable

Assuming that V1.S is defined as MASK and the statement MASK EQU >FF has been previously encountered in the assembly language source program, the following examples of qualified variables specify symbol components of string components of variables:

`V1.SS` String component of the symbol MASK. Null unless a macro instruction has caused a string to be associated with it by using a $ASG statement.

`V1.SV` Value component of the symbol MASK, i.e., >FF. In string mode, :V1.SV: equals the characters '255'.

`V1.SA` Attribute component of the symbol MASK. May be accessed by logical operators and keywords.

`V1.SL` Length component of the symbol MASK. If a string has been assigned to MASK, then V1.SL is the length of that string.

Concatenation is especially useful when a previously defined string is augmented with additional characters. The string ONE may be represented by a qualified variable such as CT.S. In that case, concatenation is expressed as:

```
:CT.S:' WAY'
```

and provides the same result as writing:

```
ONE WAY
```

If the qualified variable CT.S represented the string 'TWO', then the result of the concatenation in the example would be TWO WAY. Strings and qualified variables may be concatenated as required and the variable need not be first. Components of variables that are represented by a binary value (e.g., CT.V and CT.L) are converted to their ASCII decimal equivalents before concatenation. For example,

```
:CT.S:' WAY ':CT.L:
```

is expanded as

```
ONE WAY 3
```

since the length component of the variable CT is three.

8.6 Operators

Three types of operators are available for use in the macro assembler: arithmetic, relational, and logical operators.

8.6.1 Arithmetic Operators

Arithmetic operators, using the functions of +, -, * (multiply), and / (divide), generate operand values.

Example of an arithmetic operator:

```
LABEL   EQU     $+4     (current location counter value + 4)
```

8.6.2 Relational Operators

Relational operators compare the values of two variables, or a variable and a constant, and return the answer of TRUE or FALSE. The relational operators are as follows:

=	Equal
>	Greater than
<	Less than
≠	Not equal

Examples of relational operators:

`$IF`	`A.V>3`	Process succeeding block if value component of variable A is >3.
`$IF`	`B.L≠A.L`	Process succeeding block if length component of variable B is not equal to length component of variable A.

8.6.3 Logical Operators

Logical (Boolean) operators perform the desired operation and return either TRUE or FALSE. The following are logical operators:

&	AND
++	OR
--	NOT

Example of a logical operator:

`$IF`	`(A.V>3)&(B.L≠A.L)`	Process succeeding block if both expressions in parentheses are true.

8.7 Keywords

The attribute component of assembler symbols and macro parameters contains information on various attributes of those symbols and parameters. The macro assembly language recognizes certain keywords that are used to access that information. A keyword is used with a logical operator and the attribute component to test or to set a specific attribute of a symbol or parameter.

8.7.1 Symbol Attribute Component Keywords

The keywords listed in Table 8-3 may be used with a logical operator and the symbol attribute component (.SA) to test or set the corresponding attribute component in the Assembler Symbol Table (AST).

Table 8-3. Symbol Attribute Component Keywords

KEYWORD	SYMBOL MEANING
$REL	Relocatable
$REF	An operand of an REF directive
$DEF	An operand of a DEF directive
$STR	Assigned a component string
$MAC	Defined as a macro name
$UNDF	Not defined

Note that the use of these attributes in conditional assembly (see $IF) can lead to pass conflict errors if the symbol has not been defined before the macro call.

Examples using symbol attribute component keywords:

`V1.SA&$STR` The result of an AND operation between the attribute component of the symbol MASK (assuming V1.S has been defined as MASK) and a flag corresponding to keyword $STR. The expression is TRUE when the contents of the string component of MASK are not null; otherwise, the expression is FALSE.

`V1.SA++$REL` The result of an OR operation between the attribute component of the symbol MASK and the flag corresponding to keyword $REL.

8.7.2 Parameter Attribute Component Keywords

The keywords listed in Table 8-4 may be used with a logical operator and the macro symbol attribute component to test or set the corresponding attribute in the MST attribute component or attributes of all variables in the MST.

Table 8-4. Parameter Attribute Component Keywords

KEYWORD	SYMBOL MEANING
$PCALL	Appears as a macro instruction operand.
$POPL	An operand list (the value component contains the number of operands in the list).
$PSYM	A symbolic memory address (recognized when the variable is preceded by an @ character).

Examples using parameter attribute component keywords:

`P6.A&$PCALL` The result of an AND operation between the attribute component of variable P6 and the flag corresponding to keyword $PCALL. The expression is TRUE when variable P6 is a parameter supplied in a macro call; otherwise, the expression is FALSE.

`RA.A++$PSYM` The result of an OR operation between the attribute component of variable RA and the flag corresponding to keyword $PSYM.

8.8 Verb Statements

The following verbs may be used in macro statements:

$ASG
$ELSE
$END
$ENDIF
$IF
$MACRO
$VAR

Any statement in a macro definition not containing a macro verb in the operation field is processed as a model statement.

The macro verb statements are listed in alphabetical order and described in the following pages. The syntax and an example are also given.

8.8.1 $ASG (Value Assignment Verb)

The $ASG statement assigns values to the components of a variable. Variables that are not parameters do not have values for any components until values are assigned using $ASG statements. Components of variables with previously assigned values may be assigned new values with $ASG statements.

Syntax: `$ASG <expression/string> TO <var> [<comment>]`

The expression operand may be any valid assembler expression and may contain binary-mode variable references and keywords.

A string may be one or more characters enclosed in single quotes or the concatenation of such a literal string with the string mode value of a qualified variable. The <var> may be either an unqualified or a qualified variable.

When the operands are both unqualified variables, all components are transferred to target variables. When the destination variable is qualified, only the specified component receives the corresponding component of the expression or string. An exception to this is when a string is assigned to the string component of a variable or symbol, the length component of that variable or symbol is set to the number of characters in the assigned string. If the attribute component of the destination variable is to be changed, only those attributes that can be tested using keywords are changed. Other attributes maintained by the macro assembler may or may not be changed as appropriate. A qualified variable that specifies the length component is illegal as a destination in a $ASG statement, and will NOT set the length component.

The following examples illustrate the use of $ASG. Variables P3, V3, and CT are assumed to have been previously declared either as parameters in a $MACRO statement or as variables in a $VAR statement.

```
$ASG P3 TO V3
```
Assigns all the components of variable P3 to variable V3.

```
$ASG :P3.S:'ES' TO P3.S
```
Concatenates string 'ES' to the string component of variable P3, and set the string component to the result. This adds 2 to the length component of P3.

```
$ASG  CT.A++$PSYM TO CT.A
```
Sets the flag in the attribute component of variable CT to indicate the symbolic address attribute.

The $ASG statement may be used to modify symbol components as shown in the following examples. Assume that P3.V = 6 and P3.S = SUB.

```
$ASG 'TEN' TO G.S
```
Assigns 'TEN' as the string component of variable G. When 'TEN' is a symbol in the AST, this statement allows the use of symbol component qualifiers to modify the components of symbol TEN.

```
$ASG P3.V TO G.SV
```
Sets the value component of the symbol in the string component of variable G to the value component of variable P3. In this case, the value component of TEN is set to six.

```
$ASG 'A':P3.S:'S' TO G.SS
```
Concatenates string 'A', the string component of variable P3, and string 'S', and places the result in the string component of the symbol in the string component of variable G. Also sets the length component of the same symbol. Thus, the string component of TEN is ASUBS, and the length component is five.

Keywords in a $ASG statement must be used with a Boolean (logical) operator and an attribute component of a variable in the source field. The attribute component must come first. When quoted strings are assigned to the string component of some variable, that string may later appear in the list of undefined symbols.

8.8.2 $ELSE (Alternate Else Verb)

The $ELSE statement begins an alternate block to be processed if the preceding $IF expression was false.

Syntax: `$ELSE [<comment>]`

Example: See $IF.

8.8.3 $END (Macro Definition Termination Verb)

The $END statement marks the end of the group of statements of the macro definition named in the operand. When executed, the $END statement terminates the processing of the macro definition.

Syntax: `$END [<MACRO NAME>] [<COMMENT>]`

Example:
```
$END FIX   Terminates the definition of the FIX macro.
```

8.8.4 $ENDIF (IF Termination Verb)

The $ENDIF statement terminates the conditional processing initiated by an $IF statement in a macro definition.

Syntax: `$ENDIF [<comment>]`

Example: See $IF.

8.8.5 $IF (Conditional If Verb)

The $IF statement provides conditional processing in a macro definition. The condition of the $IF statement determines whether or not a block of statements is processed, or which of two blocks of statements is processed. A block may consist of zero or more statements.

An $IF statement is followed by a block of macro language statements terminated by an $ELSE statement or an $ENDIF statement. When the $ELSE statement is used, it is followed by another block of macro statements terminated by an $ENDIF statement. When the expression in the $IF statement has a nonzero value (or is evaluated as TRUE), the block of statements following the $IF statement is processed. When the expression in the $IF statement has a zero value (or is evaluated as FALSE), the block of statements following the $IF statement is skipped. When the $ELSE statement is used and the expression in the $IF statement has a nonzero value, the block of statements following the $ELSE statement and terminated by the $ENDIF statement is skipped.

Syntax: `$IF <expression> [<comment>]`

The <expression> may be any expression as defined for the $ASG statement and may include qualified variables and keywords. The expression defines the condition for the $IF statement.

Note that the expression is always evaluated in binary mode. Specifically, the relational operations (<,>,=,≠) operate only on the binary mode values of macro variables. Logical operators may be nested. In addition, $IF blocks may be nested up to 44 levels.

Example:

```
$IF       KY.SV
  .
  .       BLOCK A
ELSE
  .
  .       BLOCK B
$ENDIF
  .
  .
IF --     (T.A &$PCALL)
  .
  .       BLOCK A
  .
  .
$ENDIF
  .
  .
IF        T.L=5
  .
  .       BLOCK A
  .
  .
```

Process the statements of BLOCK A when the value component of the symbol in the string component of variable KY contains a non-zero value. Process the statements of BLOCK B when the component contains zero. After processing either block of statements, continue processing at the statement following the $ENDIF statement.

Process the statements of BLOCK A when the attribute component of parameter T indicates that parameter T is not supplied in the macro instruction. If parameter T is supplied, do not process the statements of BLOCK A. Continue processing at the statement following the $ENDIF statements in either case.

Process the statements of BLOCK A when the length component of variable T is equal to 5; otherwise, do not process the statements of BLOCK A. Continue processing at the statement following the $ENDIF statement.

8.8.6 $MACRO (Macro Definition Verb)

The $MACRO statement must be the first statement of a macro definition. It assigns a name to the macro and declares the parameters for the macro. The macro name consists of one to six alphanumeric characters, the first of which must be alphabetic. Each <parm> is a parameter for the macro. The operand field may contain as many parameters as the size of the field allows and must contain all parameters used in the macro definition. The comment field may not be used if there are no parameters.

Syntax: `<macro name>  $MACRO  [<parm-list>] [<comment>]`

where <parm-list> is a sequence of parameters separated by commas. The macro definition is used in the expansion of macro calls where that macro name appears in the instruction field.

Syntax for a call:

```
<macro name>  [<operand-list>] [<comment>]
```

where <operand-list> is a sequence of operands, separated by commas. The macro name specifies the macro definition to be used. Each operand may be any expression or address type recognized by the assembler, or a character string enclosed in quotes. Alternatively, a list that is a group of operands enclosed in parentheses and separated by commas (when two or more operands are in the list) may be used. A list is processed as a set after removal of the outer parentheses during macro expansion.

Operands (or lists) may be nested in parentheses in the macro call for use within macro definitions. For example, if the macro ONE is defined as

```
ONE     $MACRO    P1,P2
```

then the statement

```
ONE     PAR1,PAR2
```

results in PAR1 being associated with P1, and PAR2 being associated with P2. Similarly, the statement

```
ONE     PAR1,(PAR21,PAR22)
```

results in PAR1 being associated with P1, and PAR21,PAR22 being associated with P2.

The macro expander is responsible for replacing the macro call with the appropriate source code. Processing of each macro call in a source program causes the macro expander to associate the first parameter in the $MACRO statement with the first operand or operand list on the macro call line and the second parameter with the second operand or operand list, etc. Each parameter receiving a value has the $PCALL attribute (see Table 8-4) set in the MST. When the macro definition has more parameters specified than the number of operands in the macro call, the $PCALL attribute is not set for the excess parameters. The $PCALL attribute also is not set if an operand is null (i.e., the call line has two commas adjacent or an operand list has zero operands). Expansion of the macro can be controlled by the number of operands by using the $PCALL attribute and $IF statements.

For example, a macro definition containing

```
AMAC    $MACRO    P1,P2,P3
```

when called by

```
AMAC    AB1,AB2
```

sets $PCALL parameters P1 and P2, but not P3. Similarly,

```
AMAC    XY,,XY3
```

causes $PCALL to be set for P1 and P3, but not P2.

When the macro call has more operands than the number of parameters in the $MACRO statement, the excess operands are combined with the operand or list corresponding to the last parameter to form a list (or a longer list). In the macro statements shown below, the operands of the two macro calls would be assigned to the parameters in the same way.

Macro Call 1:

```
ONE         EQU          9
TWO         EQU          43
THREE       EQU          86
FIX         $MACRO       P1,P2              MACRO FIX
              .
              .
              .
            $END
              .
              .
              .
            FIX          ONE,TWO,THREE      MACRO CALL
            FIX          ONE,(TWO,THREE)    MACRO CALL
```

Macro Call 2:

```
A       EQU         7
B       EQU         15
C       DATA        17
D       DATA        63
E       EQU         95
F       EQU         47
G       EQU         58
H       EQU         101
I       EQU         119
PARM    $MACRO      P1,P2,P3,P4,P5,P6,P7,P8,P9
          .
          .
          .
        $END
          .
          .
          .
        PARM        A,,B,(),C,(D),E,(G,(H,I))
```

For the above macro call, the parameter assignments for PARM are as follows:

P1.S = A
P1.A = $PCALL
P1.L = 1
P1.V = 7

P2.S = (no string)
P2.A = (all false)
P2.L = 0
P2.V = 0

P3.S = B
P3.A = $PCALL
P3.L = 1
P3.V = 15

P4.S = (no string)
P4.A = $POPL
P4.L = 0
P4.V = 0

P5.S = C
P5.A = $PCALL
P5.L = 1
P5.V = 17

P6.S = D
P6.A = $PCALL,$POPL
P6.L = 1
P6.V = 1

P7.S = E
P7.A = $PCALL
P7.L = 1
P7.V = 95

P8.S = G,(H,I)
P8.A = $PCALL,$POPL
P8.L = 7
P8.V = 2

P9.S = (no string)
P9.A = (all false)
P9.L = 0
P9.V = 0

A macro definition supercedes previous macro definitions and native instructions with the same name. Symbolic operands that appear in a macro call are treated as symbolic operands in native instructions; i.e., if they are not defined with the program in which they appear, they are listed as undefined symbols.

8.8.7 $VAR (Variable Declaration Verb)

The $VAR statement declares the variables for a macro definition. The $VAR statement is required only if the macro definition contains one or more variables other than parameters. More than one $VAR statement may be included, and each $VAR statement may declare more than one variable. Each <var> in the operand is a variable.

Syntax: `$VAR  <var>[,<var>] [<comment>]`

Example:
```
$VAR   A,CT,V3        THREE VARIABLES FOR A MACRO
```

The example declares variables A, CT, and V3, which must not have been declared as parameters.

The $VAR statement does not assign values to any components of the variables; that is the function of the $ASG statement. $VAR statements may appear anywhere in the macro definition to which they apply, provided each variable is declared before the first statement that uses the variable. Placing $VAR statements immediately following the $MACRO statement is recommended for clarity in reading the source code.

8.9 Model Statements

A macro definition consists of model statements and statements that contain macro verbs. Processing a model statement results in an assembly language statement. This statement may be composed of the usual elements of an assembly language statement combined with string mode qualified variable components.

Examples of model statements:

`IN  *+,PA7,1` — An assembly language source statement that contains a machine instruction.

`:P7.S: LAR :P2.S:,R8 :V4.S:` — The string component of variable P7, followed by one blank, LAR, and one more blank, is concatenated to the string. The string component of variable P2 is concatenated to the result, to which R8 and three blanks are concatenated. A final concatenation places the string component of variable V4 in the model statement. The result is an assembly language machine instruction having the label and comment fields and part of the operand field supplied as string components.

`:MS.S:` — The string component of variable MS. Preceding statements in the macro definition must place a valid assembly language source statement in the string component to prevent assembly errors.

Note that conditional assembly directives may not appear as operations in a model statement. Comments supplied in model statements may not contain periods since the macro assembler scans them. Improper use of punctuation may cause syntax errors.

8.10 Macro Examples

Macros may simply substitute a machine instruction for a macro instruction, or they may include conditional processing, access the Assembler Symbol Table (AST), and employ recursion. Several examples of macro definitions are described in the following paragraphs.

8.10.1 ID (Identification Macro)

The ID macro, an example of a macro with a default value, supplies two DATA directives to the source program. The ID macro consists of nine other macro statements, four of which are model statements. The definition is as follows:

```
ID      $MACRO  WS,PC          Defines ID with parameters WS and PC.
        DATA    :WS.S:         Model statement: places a DATA directive with
                               the string of the first parameter as the operand
                               in the source program.
        $IF     PC.A&$PCALL    Tests for presence of parameter PC.
        DATA    :PC.S:,15      Model statement: places a DATA directive in
                               the source program. The first operand is the
                               string of the second parameter, and the second
                               operand is 15. This statement is processed if
                               the second parameter is present.
        $ELSE                  Starts the alternate portion of the definition.
        DATA    START,15       Model statement: places a DATA directive in
                               the source program. The first operand is label
                               START, and the second operand is 15. This
                               statement is processed if the second parameter
                               is omitted.
START   EQU     $              Model statement: places label START in the
                               source program. This statement is processed
                               if the second parameter is omitted.
        $ENDIF                 Ends the conditional processing.
        $END                   Ends the macro.
```

Syntax: `[<label>] ID <exp>[,<exp>] [<comment>]`

The addresses may be expressions or symbols.

Example of a macro call for macro ID:

```
        ID      WORK1,BEGIN
```

The resulting source code would be

```
        DATA    WORK1
        DATA    BEGIN,15
```

If only one operand is supplied, the macro instruction could be coded as follows:

```
ID      WORK2
```

This would result in the following source code:

```
          DATA      WORK2
          DATA      START,15
START     EQU       $
```

This form of the macro instruction imposes two restrictions on the source program. The source program may only call the ID macro with a single parameter once. This is necessary to prevent the use of the label 'START' more than once. Problems with labels supplied in macros may be prevented by reserving certain characters for use in macro-generated labels. A macro definition may maintain a count of the number of times it is called, and use this count in each label generated by the macro.

8.10.2 GENCMT (Generate Comment Macro)

The GENCMT macro implements only those comments that appear in the macro definition and the expansion of the macro. In the following example, the first five lines define the macro, followed by :V.S: that expands the macro definition.

Example of assembler list file:

```
0001                        IDT         'GENCMT'
0002             GENCMT     $MACRO
0003                        $VAR V
0004             * THIS IS A MACRO DEFINITION COMMENT.
0005                        $ASG '*' TO V.S
0006             :V.S: THIS IS A MACRO EXPANSION COMMENT.
0007                        $END
0008             *
0009             *
0010                        GENCMT
0001             * THIS IS A MACRO EXPANSION COMMENT.
0011  0000 0000            DATA 0,1
      0001 0001
0012                        GENCMT
0001             * THIS IS A MACRO EXPANSION COMMENT.
0013                        GENCMT
0001             * THIS IS A MACRO EXPANSION COMMENT.
0014  0002 0004            DATA 4
0015                        END
NO ERRORS, NO WARNINGS
```

8.10.3 FACT (Factorial Macro)

The FACT macro, an example of the recursive use of macros, produces the assembly code necessary to calculate the factorial of N where N is an immediate value, and store that value at data memory address LOC. FACT accomplishes this by calling FACT1, which calls itself recursively.

Example:

```
FACT    $MACRO N,LOC          * N IS AN INTEGER CONSTANT AND
*                             * LOC IS THE DATA MEMORY ADDRESS
*                             * WHERE N! IS TO BE STORED.
*
        $IF N.V<2
        LACK 1                * 0!= 1! = 1
        SACL :LOC:
        $ELSE
        LACK :N.V:            * N >= 2 SO, STORE N AT LOC,
        SACL :LOC:            * DECREMENT N, AND DO THE
        $ASG N.V-1 TO N.V     * FACTORIAL OF N-1.
        FACT1 :N.V:,:LOC:
        $ENDIF
        $END
*
FACT1   $MACRO M,AREA
        $IF M.V>1
        LT   :AREA:           * MULTIPLY PRESENT FACTORIAL
        MPYK :M.V:            * BY PRESENT POSITION.
        PAC
        SACL :AREA:           * SAVE RESULT.
        $ASG M.V-1 TO M.V     * DECREMENT POSITION.
        FACT1 :M.V:,:AREA:    * RECURSIVE CALL.
        $ENDIF
        $END
```

8.11 Macro Error Messages

Table 8-5 lists and defines the macro error messages, and gives correction information.

Table 8-5. Macro Error Messages

MESSAGE	DESCRIPTION
MACRO LINE TOO LONG	In a macro definition, macro directive lines may be only 58 characters long. Model statements, when fully expanded, may be only 60 characters long.
LONG MACRO VARIABLE QUALIFIER	Macro variable qualifiers may be only one or two characters long.
TOO MANY MACRO VARIABLES	The total number of macro parameter variables and labels in a single macro definition may not exceed 128.
INVALID MACRO QUALIFIER	The only valid macro qualifiers are: S, V, L, A, SS, SV, SL, and SA.
VARIABLE ALREADY DEFINED	A macro variable cannot be redefined within a macro.
IF LEVEL EXCEEDED	The maximum nesting level of $IF directives is 44.
MACRO ASSEMBLER PROGRAM ERROR	The macro assembler has detected an internal error. These can be caused by incorrect syntax.

9. Link Editor

The Link Editor combines separately generated object modules with associated procedures and overlays to form a single, linked, relocatable object module that can be installed and executed on various computer systems. The object code is generated by an assembler supplied with the TMS320C25 software development system. The link editor is currently available for the VAX (VMS) and TI/IBM PC (MS/PC-DOS) operating systems.

This section describes the Link Editor, its files and control commands, and gives examples of various linking procedures. Included in this section are the following major topics:

9.1 Description

The Link Editor provides symbol resolution for external references and definitions created by the REF and DEF assembler directives (see Section 6). Without this function, all modules would have to be compiled or assembled at once, and modules written in different languages could not be mixed.

The Link Editor builds a list of symbols from the REF tags in the object modules that are included in the linking process. The Link Editor then resolves references by matching DEF tag symbols with the REF tags and inserting the correct values for these symbols in the linked object code.

The Link Editor can position the three defined segments (program, data, and common) to prescribed boundaries for eventual ROM/RAM partitioning. Program, data and common segments are defined by the PSEG, DSEG, and CSEG assembler directives, respectively. If these directives are not used, the entire object module is tagged as a program segment.

When PSEG, DSEG, and CSEG tags are encountered in the included modules, the Link Editor reorganizes segments from each module into three segments in the linked output. The first segment contains the PSEGs of all included modules, the second segment contains the DSEGs, and the third segment the CSEGs of all included modules. The beginning location for each segment can be user-defined.

The Link Editor also allows overlays and procedure/task segmentation. However, if the system being used loads only one module at a time, procedure/task segmentation and overlays cannot be used because they produce multiple output modules.

9.2 Program Definition

To use the Link Editor, each program must be defined as a phase or a task. Below are the definitions of each.

Phase The smallest functional unit that can be loaded as a logical entity during execution in an overlay structure.

Each phase is identified by a name and a level number. The root phase is at level 0 and is that portion of the program that must remain memory resident. Other phases (level 1 and above) that do not have to be simultaneously memory-resident can overlay each other.

Task A complete program containing both variable data and executable code or the variable data portion of a program (for procedure/task segmentation).

9.3 Link Editor Files

Executing the Link Editor utility begins by accessing the Linker and then responding to prompts for the link control file, linked output file, and listing file. The Link Editor utility uses the following five files in the linking process:

- Link control file
- Object modules
- Libraries
- Linked output file
- Listing file.

Each file is given a pathname so that when that pathname is entered, the Link Editor can search for that file. The pathnames for the link control file, object modules, declared libraries, the linked output file, and the listing file are in the listing file. An example of pathname structure (default value) for the link control file is given for the two operating systems currently available for the TMS320C25 Link Editor.

Pathname	System
[PROJECT.MACK]SEGMENT.CON	VAX (VMS)
A:PARTIAL.CTL	TI/IBM PC (MS/PC-DOS)

Each of the link editor files is described in the succeeding subsections.

9.3.1 Link Control File

The link control file is an input file that controls the operation of the Link Editor. This file contains a set of link control commands called a control stream which defines the modules to be linked and how they are to be linked. The Link Editor links the object modules in the order specified by the linker commands. See Table 9-2 for a summary of all the linker commands.

The link control file must be created ahead of time. Entering a pathname instructs the editor to look for a file containing the necessary control commands.

9.3.2 Object Modules

Object modules are the input programs that are to be linked together. They are contained in files and must consist of either ASCII or compressed 990-tagged object code. The ASCII 990-tagged object code is the type of code generated by the assembler supplied with the TMS320C25 Software Development System. The object code consists of ASCII tags followed by data fields (see Section 7.9 for a description of object code format).

As the Link Editor finishes writing out an object module, it names the module and gives the number of object records it contains. When the link terminates normally, the last line written reads `'*** LINKING COMPLETED'`. The date and time at the end of the link are printed on the last line. The date and time captured at the beginning of the link are printed at the top of every page and on the last card of every module in the linked object.

Object modules can be explicitly user-defined with the INCLUDE command in the control file, or automatically included by the Link Editor as a result of a search for unresolved references.

9.3.3 Libraries

Libraries are directories or files containing collections of object modules. An object library may be either random or sequential. A random library is a directory of object modules in separate files, whereas a sequential library is a file containing one or more object modules concatenated together. See Section 8.5.4 for examples of library creation.

Libraries are used to automatically resolve the REF and DEF tag symbols between object modules specified in INCLUDE commands.

Two types of symbol resolution are implemented:

- Automatic symbol resolution by default (the AUTO command) when the END command is detected in the control file unless the NOAUTO command has been used.
- Symbol resolution at a user-defined point in the linking process when a SEARCH or FIND command is used. The SEARCH command is used with random libraries and the FIND command with sequential libraries.

Libraries defined by the LIBRARY command are searched in the same order they are defined. Any additional unresolved references created by modules to satisfy references are also resolved automatically. Automatic symbol resolution still occurs at the end of the linking process for any remaining unresolved references unless a NOAUTO command is in the control file.

9.3.4 Linked Output File

The linked output file is an 80-character output file containing the 990-tagged object format load module in the "LINKED OUTPUT" file. This load module appears in ASCII or compressed format, depending on the use of the FORMAT command in the object link control file. The response to the linked output file name specifies the destination of the load module.

9.3.5 Listing File

The listing file consists of a listing that includes the control stream and a link map that lists the modules with their origins and lengths. The link map consists of the following four sections:

1) Individual constituent object modules
2) Common segments
3) Symbols (external)
4) Unresolved references (identified even if the NOMAP option has been selected).

The response to the listing file access name specifies the destination of the listing generated during the link edit. The pathnames for the control file, the listing file, the linked object file, and declared libraries are in the listing file. Messages are listed for detected errors in the listing file.

The Link Editor creates two temporary files on the work file disk. Therefore, sufficient space for two disk or diskette files must be available.

9.4 Linker Commands

Link control commands define the modules to be linked and how they are linked. This section gives some rules for entering a command in the link control file.

A command set summary of all the linker commands, arranged according to function, is provided for easy reference. Each command in the summary table is next described individually. Linker syntax and example(s) are also given for each command. The commands are listed in alphabetical order.

9.4.1 Entering a Command

When entering a command in the control file, these rules should be followed:

- Either the entire command or only the first four characters may be specified.
- At least one space must separate the command from its parameters.
- Comments may be entered either on a separate line or following the command parameters.
- All comments must be preceded by a semicolon (;).
- The command must be contained within the first 72 characters of the line.

9.4.2 Linker Command Set

Table 9-1 lists the symbols used in the syntax definitions of the linker commands.

Table 9-1. Linker Syntax Symbols

SYMBOL	MEANING
< >	User-defined parameters.
[ ]	Optional parameters. They may be omitted.
{ }	Alternative parameters, one of which must be entered.
...	The preceding parameter may be repeated.
( )	Indicates "contents of".
<acnm>	An access name for a file or library must be entered for the parameter.
<base>	The starting location of a segment, expressed as either a decimal or hexadecimal number up to five digits in length.
<level>	The level of a phase.
<name>	The name of a specified area. Consists of one to eight alphanumeric characters, the first of which must be alphabetic.
(<name>)	The name of a member in a library.
<value>	The number of lines, between 16 and 60, to be printed on a page. Replaces the default value of 60.
>	Represents hexadecimal, as does also a leading zero.
	Words shown in capital letters and special characters not listed here must be entered as shown.

The link command set summary of Table 9-2 is arranged according to function and alphabetized within each functional grouping. Of the four groups, the first group consists of basic commands that are required to perform basic Link Editor functions. The second group consists of ROM/RAM partitioning commands. The third group includes those miscellaneous commands that perform auxiliary link editor functions, such as specifying default conditions and procedure/task segmentation. The fourth group consists of the partial link commands.

Table 9-2. Linker Command Set Summary

BASIC COMMANDS	
Command	**Function**
END	Indicates the end of the control stream. This is a required command.
FIND	Specifies a search of only sequential libraries for unresolved references at this point in the control stream.
FORMAT	Defines the format of the linked output module as ASCII or COMPRESSED code. The default is ASCII object code.
INCLUDE	Defines one or more modules to be included in the linking process. At least one INCLUDE command is required in each control stream.
LIBRARY	Defines a random library directory.
PHASE	Defines the level and name of a phase in a program. Either the PHASE or the TASK command must appear in each control stream. Multiple phases are allowed when overlays are used.
SEARCH	Specifies a search of defined random libraries for unresolved references at this point in the control stream.
TASK	Defines a phase to be installed and executed as a task or standalone program. A name is assigned the task.
ROM/RAM PARTITIONING COMMANDS	
Command	**Function**
ALLOCATE	Controls the relative positioning of the program, data, and common segments (PSEG, DSEG, and CSEG assembler directives, respectively).
COMMON	Specifies the starting location of the common segment in the linked output.
DATA	Specifies the starting location of the data segment in the linked output.
PROGRAM	Specifies the starting location of the program segment in the linked output.
AUXILIARY FUNCTION COMMANDS	
Command	**Function**
ADJUST	Aligns a phase or a module within a phase on a specified boundary.
AUTO	Specifies automatic symbol resolution at the end of the control stream (default condition).
DUMMY	Suppresses generation of the linked output file. Useful for error identification or when only a listing file is required.
ENTRY	Specifies a symbol for an entry tag to be produced.
NOAUTO	Inhibits automatic symbol resolution, allowing the user to explicitly control library searching for unresolved references.
NOMAP	Suppresses the output of the link map listing by omitting the module, common, and symbol maps from the listing.
NOPAGE	Inhibits page ejects between the link maps of each phase.
NOSYMT	Omits symbol tables from included modules in the linked output file (default condition).
PAGE	Causes page ejects between link maps for each phase (default condition).
PROCEDURE	Defines a phase of the link edit structure which can be installed as a procedure. Used for procedure/task segmentation only. An alternate version of this command can be used to support levels 1 and 2.
REPLACE	Replaces one external symbol name for another in the next object file read in.
SYMT	Includes symbol tables in linked output files.
PARTIAL LINK COMMANDS	
Command	**Function**
ALLGLOBAL	Declares all external definitions in included modules as global symbols for subsequent relinking (default condition).
GLOBAL	Identifies the symbols defined in included modules to be processed as global symbols for subsequent relinking.
NOTGLOBAL	Declares either specified externally defined symbols or all externally defined symbols in included modules as local symbols.
PARTIAL	Performs a partial link and outputs either ASCII or compressed object code. The output of a partial link must be linked again without the PARTIAL command before the program can be loaded and executed.

9.4.3 Individual Command Descriptions

Each command in the linker command set summary is described in the following pages. Information, such as linker syntax, a description, and example(s), is given for each command. The commands are listed in alphabetical order.

Syntax

ADJUST [<n>]

where <n> = a decimal number less than 16 specifying a power-of-two bytes. A value greater than 15 causes an error. When the parameter is omitted or equal to zero, alignment is on the next word boundary.

Description

The ADJUST command specifies the alignment of a phase or of a module within a phase on a specified boundary.

When the ADJUST command appears immediately before a PHASE command, the next phase and all subsequent phases of the same level and with the same parent node are aligned on the specified boundary, relative to the beginning of the program.

If the ADJUST command follows a PHASE command but precedes all INCLUDE commands in the phase, the effect is the same as above. When the ADJUST command follows a PHASE command but precedes an INCLUDE command, the next module in that phase is aligned on the specified boundary, relative to the beginning of the phase.

Syntax

ALLGLOBAL

Description

The ALLGLOBAL (partial linking) command declares all external definitions in included modules as global symbols. ALLGLOBAL is a default condition.

Global symbols are externally defined in the linked output module and therefore may be re-linked in a subsequent linking process.

Syntax

ALLOCATE

Description

The ALLOCATE command controls the relative positioning of program, data, and common segments (PSEG, DSEG, and CSEG directives, respectively). ALLOCATE has no parameters.

ALLOCATE directs the Link Editor to reserve space for all outstanding data and common segments as if no more object modules were to be included in the link. The primary purpose of the ALLOCATE command is to aid the user in sharing non-reentrant procedures between different tasks.

The ALLOCATE command only works if all read/write data is contained in data segments (DSEGs) or common segments (CSEGs).

Syntax

AUTO

Description

The AUTO command specifies automatic symbol resolution using defined libraries at the end of the linking process. The AUTO command has no parameters and is optional. It is the default condition.

Syntax

COMMON {<base>[,<name>] [,<name>]...}

where <base> = the starting location of the common segment. It can be expressed as either a decimal or hexadecimal number up to five digits in length.

<name> = the name of the common segment. Any unnamed common segment begins after the last data area encountered. The commons are allocated in the order in which the definitions appear in the object module.

Description

The COMMON command defines the starting address for the specified common segment (CSEG). Commons that are loaded at the specified address must be specifically identified within this command. The COMMON command is only valid when used with the PROGRAM command and is ignored if used alone.

More than one COMMON command may be used, and a continuation can be performed by repeating the command using a previously named common instead of a starting address. The COMMON command cannot be used in partial links.

Example 1

```
COMMON   01000,COMA      Begin common COMA at location
                         >1000.
```

Example 2

```
COMM     >1000,COMA      Results are the same as the
                         preceding example.
```

Example 3

```
COMMON   COMA,COMB       Begin common COMB immediately
                         following COMA.
```

Example 4

```
COMM     4096,COMA,COMB  Results are the same as the
                         two preceding examples.
```

Syntax

DATA <base>

where <base> = the starting location of the data segment. It can be expressed as either a decimal or hexadecimal number up to five digits in length.

Description

The DATA command defines the absolute starting address for the data segment (DSEG) in the linked output. The DATA command is only valid when used with the PROGRAM command and is ignored if used alone.

The DATA command may appear more than once in the control stream, but the first DATA command must appear before the first INCLUDE command. If the DATA command is omitted, the starting location for each data area defaults to the end of the corresponding program area.

The DATA command cannot be used in partial links.

Example 1

```
DATA    01000       Begin data segment at location >1000.
```

Example 2

```
DATA    4096        Same as the preceding example.
```

Syntax DUMMY

Description The DUMMY command supresses generation of the linked output file. This command is useful for error identification or when only a listing file is needed. DUMMY has no parameters.

Syntax

END

Description

The END command specifies the end of the control stream. The command is required in every control stream.

Syntax

ENTRY <symbol>

Description

The ENTRY command specifies a symbol for the entry point in order to produce an entry tag. This overrides all entry tags received in input object modules.

Syntax

FIND <acnm>[,<acnm>][,<acnm>]...

where <acnm> = the access name of the sequential library that is to be searched for unresolved references.

Description

The FIND command specifies a search of sequential libraries for members representing unresolved references. Only one pass is made through a library in response to a single FIND command. The search occurs at the point in the linking process where the FIND command occurs.

The FIND command functions as a SEARCH command but applies to sequential libraries only. The FIND command is listed as a SEARCH command in the link map.

Example

```
FIND   A:*.EXT    For PC/MS-DOS system.
```

Syntax

FORMAT {ASCII,COMPRESSED}

Description

The FORMAT command defines the format of the linked output module.

The format specified may be either ASCII or COMPRESSED object code. In ASCII format, each integer value in the object is represented as a four-byte character string. ASCII format is also called 990-tagged object format and is the default condition. Compressed format is more efficient to use since each integer value is represented as a two-byte word.

Syntax

GLOBAL [symbolname][,symbolname]...

where symbolname = a symbol that is to be processed as a global symbol. It is defined at assembly time and consists of six characters or less, the first of which must be alphabetic.

Description

The GLOBAL command is a partial linking command, identifying the symbols defined in included modules to be processed as global symbols. Global symbols are externally defined in the output module that may be relinked.

Each parameter specifies a symbol that is to be processed as a global symbol. The command may include several parameters and may appear more than once in the command stream. If no parameters are specified, the command functions as an ALLGLOBAL command.

Symbols defined by the GLOBAL command are not affected by the NOTGLOBAL command (no parameters) that declares all symbols to be local.

Syntax

INCLUDE {<acnm>[,<acnm>]...,(<name>)[,(<name>)]...}

where <acnm> = the access name of a file containing the object module(s) to be included in the linking process.

(<name>) = a member in a library.

Description

The INCLUDE command specifies modules to be included in the linking process. This command is required in the control stream. More than one INCLUDE command may be used as needed.

A PROCEDURE, TASK, or PHASE command must precede the first INCLUDE command.

If the <name> parameter is used, enclose only the file name or module name of the object modules (rather than the entire access name) in parentheses. The specified <name> must be of a file contained in a defined random library. The Link Editor searches the defined libraries for the specified module.

If no parameters are given, in-line text format (not compressed) object code is assumed. The in-line object (see Example 3) is delimited by either end-of-file or by a record with '/*' in columns one and two. This method is suitable, for example, when the control file is read in from a card reader (in which case, end-of-file is denoted by a '/*' card).

Example 1

```
INCLUDE     (X)          Search defined random libraries for
                         a file named X and include the
                         module(s) in that file.
```

Example 2

```
INCLUDE     TEST.MPO     Include the module TEST.MPO from the
                         default directory on a PC/MS-DOS
                         system.
```

Example 3

```
INCLUDE
K006CCARTMOND50020LBL2B150021LBL2B240000LBL2C240000LBL2D2A00207F1E7F
BCE26BCE26BFE80E00000000BFE80E00010000BCE1BB0201B0388B4802B48A97F1E1F
BFFEEBCE27BCE50BCE04BCE05BCE01B567BB568CBCE00BCE0FBCE1FB807AB81A87F048F
:       CARTMOND   4/10/85  10:53:14    ASM32020 PC 1.0 85.092
/*
```

Syntax

LIBRARY <acnm>[,<acnm>]...

where <acnm> = the access name of the directory that is to be defined as a library.

Description

The LIBRARY command defines random library directories. Random libraries must consist of a directory, and the files in the directory must contain 990-tagged object modules. Sequential libraries, consisting of a sequential file of object modules, are indicated using the FIND command.

Example

```
LIBR   A:*.EXT      Define drive A: as a random
                    library of files with extension
                    .EXT on a PC/MS-DOS system.
```

Syntax

NOAUTO

Description

The NOAUTO command inhibits automatic symbol resolution at the end of the linking process. This command allows the user to explicitly control library searching for unresolved references through use of the SEARCH and FIND commands. NOAUTO has no parameters.

Syntax

NOMAP

Description

The NOMAP command specifies that the module, common, and symbol maps are to be omitted from the listing. This gives some improvement in terms of speed and number of symbols that can be processed. The following information is still printed on the listing file:

- Length of task and procedure(s)
- Unresolved references
- Release number of the Link Editor.

NOMAP must appear before any PHASE or TASK commands are used.

Syntax NOPAGE

Description The NOPAGE command sets no page ejects between the link maps for each phase. New pages are started for the listing of the first phase and when the number of lines per page has been exceeded.

Syntax

NOSYMT

Description

The NOSYMT command omits symbol tables from included modules in the linked output file. This provides for more compact object code but does not allow symbolic debugging.

The NOSYMT command may appear anywhere in the control file. However, if an overlay structure is used, the NOSYMT command must appear in the root phase (phase 0).

NOSYMT is the default option and is the inverse of SYMT.

Syntax

NOTGLOBAL [symbolname][,symbolname]...

where symbolname = a symbol which is to be processed as a local symbol. It is defined at assembly time and consists of six characters or less, the first of which must be alphabetic.

Description

The NOTGLOBAL command is a partial linking command, declaring that either specified externally defined symbols or all externally defined symbols in the included modules are to be processed as local (not global) symbols.

Local symbols are not externally defined in the partially linked output module and thus can only be referenced by modules included in the current partial link.

The command may include several parameters and may appear more than once in the command stream. If no parameters are specified, all symbols are processed as local, except those specified in the GLOBAL command.

Syntax

PAGE [value]

where value = the number of lines to be printed on a page, replacing the default value of 60. The value parameter is optional, but when present, the value must be between 16 and 60.

Description

The PAGE command causes page ejects to separate the beginning of each link map for each phase. This is the default condition.

Syntax

PARTIAL

Description

The PARTIAL command performs a partial link and outputs either ASCII or compressed object code. The output of a partial link is not executable and must be linked again without the PARTIAL directive before the program can be loaded and executed.

The PARTIAL command causes the Link Editor to do the following:

1) Resolve all external references defined by any module included in the partial link.

2) Retain all entry points in the partial link as an entry in the output (subject to GLOBAL, NOTGLOBAL, ALLGLOBAL commands).

3) Retain the common tags and update common numbers.

4) Output one data section that is the total of all input data sections.

Partial linking is allowed for single phases only, and the control stream must contain either a TASK or PHASE 0 command. If partial linking of overlays is required, each phase must be partially linked separately as a phase 0. The phase level and name may be redefined in subsequent links. The following commands are invalid with partial links: ALLOCATE, PROGRAM, DATA, COMMON, and DUMMY.

Syntax

PHASE <level>,<name>

where <level> = the level of the phase. Levels specified greater than zero can be used for overlay structures only. Level 0 defines the root (memory-resident) phase. Each subsequent PHASE command defines the level and name of an overlay.

<name> = the name of the phase. It consists of one to eight alphanumeric characters, the first of which must be alphabetic. The name supplied becomes the IDT name, placed on the last card of the object module produced and on the identification fields of ASCII-formatted object records.

Description

The PHASE command defines the level and name of a phase in a program.

PHASE 0 and TASK commands are logically identical; one and only one of these two commands must appear in each control stream.

The Link Editor produces an output module for each phase of the program. PHASE commands are followed by INCLUDE commands that define the modules included in the phase. Multiple phases are allowed when overlays are used.

Example 1

```
PHASE   0,MAIN    Define phase MAIN at level 0.
```

Example 2

```
PHAS    2,DISK    Define phase DISK at level 2.
```

Syntax

PROCEDURE {<name>,<level,name>}

where <name> = the identifier of the procedure to be used. The parameter consists of one to eight alphanumeric characters, the first of which must be alphabetic.

<level> = the level of the phase.

Description

The PROCEDURE command provides procedure/task segmentation by defining a phase of the link edit structure, which can be installed as a procedure (a re-entrant procedure may be shared among several tasks). The name supplied becomes the IDT name, placed on the last record of the object module produced and on the identification field of ASCII-formatted object records. This command is useful in ROM/RAM partitioning for generating load modules with a level of root phase 0.

When used, the PROCEDURE command must precede the TASK command, all PHASE commands, and the INCLUDE command that defines the procedure module.

The PROCEDURE command is used with the INCLUDE commmand to define the procedure. The PROCEDURE command defines the name of the procedure, and the INCLUDE command defines the modules that are to be in that procedure. Procedures contain the program segment (PSEG), which may be the entire program but is usually only the executable code and read-only data.

A generalization of the standard PROCEDURE command is supported for levels 1 and 2. In place of a single first procedure, any number of other level-one procedures can be defined, any of which can be resident in memory at a given time under the user's control. The length of the first procedure area is the maximum of the lengths of the individual level-one modules. Analogous properties apply to second-level PROCEDURES. Modules brought in by automatic call to satisfy references in any procedure module are placed in the root.

Example 1

```
PROCEDURE    FORLIB     Define procedure FORLIB.
```

Example 2

```
PROC         RUNLIB     Define procedure RUNLIB.
```

Example 3

```
PROCEDURE    2,FILEMG   Define a procedure FILMG at level 2.
```

Syntax

PROGRAM <base>

where <base> = the starting location of the program segment. It can be expressed as a decimal or hexadecimal number up to five digits in length.

Description

The PROGRAM command defines the absolute starting address for the program segment (PSEG) in the linked output.

The PROGRAM command may be used more than once. The first PROGRAM command must appear before the first INCLUDE command. Use of the PROGRAM command by itself or with the DATA and COMMON commands causes the linked output to be loaded at the specified address (base).

Example 1

```
PROGRAM   01F00   Begin program segment at location
                  >1F00.
```

Example 2

```
PROG      >1F00   Same as the preceding example.
```

Example 3

```
PROG      7936    Begin program segment at location
                  7936 (>1F00).
```

Syntax

REPLACE <oldsym(newsym)>[,<oldsym(newsym)>]...

where oldsym = the currently existing external symbol representing a reference, definition, or common name.

(newsym) = the new external symbol to replace the oldsym.

Description

The REPLACE command specifies that in the next file read in, each occurrence of 'oldsym' as an external symbol is replaced by 'newsym'. The command applies to every module in a file containing multiple modules. It applies only to the first file in an INCLUDE command list. If the command immediately precedes a FIND, SEARCH, or END command, it still applies to the next single file read in.

If 'oldsym' is $DATA and an affected module contains a DSEG, the link editor converts the DSEG to a common with the name 'newsym'. This means that no data segment is identified in the listing, and if other instances of the common name occur, the common may be extended in length or promoted (moved up to a lower-numbered phase).

Note that data segments can be shared by using the REPLACE command to convert them to a common. Appropriately used, this permits a module to share a data segment in an ancestor phase and places no restrictions on the order of definition of segments with different lengths.

Syntax

SEARCH [<acnm>][,<acnm>]...

where <acnm> = the access name of random libraries to be searched. The order of these access names determines the order of the search. If no <acnm>s are specified, the libraries defined by the LIBRARY commands define the search ordering.

Description

The SEARCH command directs the Link Editor to search for unresolved references at any point in the control stream.

If a SEARCH command is given in a phase other than the TASK or PHASE 0 phase, searching is performed only for symbols that are unresolved in that phase. Unresolved references that were established in or promoted to other phases are ignored.

A SEARCH command in a TASK phase causes searching to be done for every phase (for the given phase and all its descendant and previous phases). The only way the SEARCH command can be applied to more than one phase is by re-entering a phase defined earlier. This is permitted only for the task phase and for the purpose of doing SEARCHes and FINDs.

Example 1

```
SEARCH              Search defined libraries for
                    unresolved references.
```

Example 2

```
SEARCH   A:*.EXE    Search drive A: as a library of
                    files with extension .EXE on a
                    PC/MS-DOS system.
```

Syntax

SYMT

Description

The SYMT command causes the Link Editor to include symbol tables in the linked output file when the linker input files contain such symbols. These symbols were provided in the assembler as a result of selecting the SYMLST option (see the OPTION directive in Section 7.7). Although symbol tables make the linked module larger, they are useful for symbolic debugging.

SYMT is the inverse of the NOSYMT option.

Syntax

TASK [<name>]

where <name> = the identifier of the task module. The <name> can have up to eight characters. The name supplied becomes the IDT name, placed on the last record of the object module produced and on the ID fields of ASCII-formatted records. If the parameter is omitted, the IDT name of the first included module is used as the task name.

Description

The TASK command defines a phase that can be installed and executed as a task or standalone program, and assigns a name to the task.

A task is either a complete program, containing both variable data and executable code, or it is the variable data portion of a program (procedure/task segmentation). The TASK and PHASE 0 commands are logically identical; one and only one of these two commands must appear in each control stream.

When task/procedure segmentation is used, the TASK command must follow all PROCEDURE commands and precede all PHASE and INCLUDE commands that define the task module. The TASK command can be given after overlays have been defined (to re-enter the root phase).

Example 1

```
TASK    FORPRG    Define task named FORPRG.
```

Example 2

```
TASK              Define task and assign it the IDT name
                  of the first included module.
```

9.5 Linking Examples

Examples showing how and when to use the link control commands are provided in this section. Among the examples are a simple link (Section 8.5.1), ROM/RAM partitioning (Section 8.5.2), and a partial link (Section 8.5.3). In addition, examples are given for creating random and sequential libraries (Section 8.5.4).

Three separately assembled modules, MAIN, RESET, and INTRPT, are to be linked together. Figure 9-1, Figure 9-2, and Figure 9-3 contain the assembly language source for each module. The TMS320C25 Assembler produces 990-tagged object code that the Link Editor requires as input.

The first and third linking examples assume that each module is contained in a separate file named MAIN.MPO, RESET.MPO, and INTRPT.MPO, respectively, and that the three files are listed on a diskette in a TI/IBM PC (MS/PC-DOS) operating system. The second example is similar, but bases its file access on the VAX/VMS operating system.

```
              IDT       'MAIN'
*
              DEF       MAIN
              REF       RESET,INTRPT
* DATA PAGE 6 RAM DEFINITION
NEXTO         EQU       0
SAMPLE        EQU       32
*
              PSEG
BEGIN         B         RESET
INTO          B         INTRPT
*
              BSS       28
MAIN          LARP      AR1
              LDPK      6
LOOP          IDLF
              LALK      INPUT
              TBLR      SAMPLE
              LRLK      AR1,>0300+SAMPLE
              CNFP
              MPYK      0
              ZAC
              RPTK      31
              MACD      >FF00,*-
              APAC
              SACL      *
              CNFD
              LALK      OUTPUT
              TBLW      NEXTO
              B         LOOP
              PEND
*
              CSEG      'IO'
INPUT         BSS       1
OUTPUT        BSS       1
              CEND
*
              END
```

Figure 9-1. Source for Module MAIN

```
            IDT     'RESET'
*
            DEF     RESET
            REF     MAIN
            PSEG
RESET       LARP    AR1
            LRLK    AR1,>0300
            ZAC
            RPTK    35
            SACL    *+
            LRLK    AR1,>0200
            RPTK    31
            BLKP    CFIR,*+
            B       MAIN
*
CFIR        DATA    176,-203,297,-398,493,-566,598,-567
            DATA    448,-212,-176,772,-1684,3193,8,7
            DATA    6,5,3193,-1684,772,-176,-212,448
            DATA    -567,598,-566,493,-398,297,-203,176
            PEND
*
            END
```

Figure 9-2. Source for Module RESET

```
            IDT       'INTRPT'
*
            DEF       INTRPT
* DATA PAGE 0 RAM DEFINITION
STATUS      EQU       96
TEMP1       EQU       97
TEMP2       EQU       98
            PSEG
*
INTRPT      SST       STATUS
            LDPK      0
            IN        TEMP1,PA0
            LALK      LASTIN
            TBLR      TEMP2
            TBLW      TEMP1
            LAC       TEMP2
            SUB       TEMP1
            SACL      TEMP1
            LALK      INPUT
            TBLW      TEMP1
            LALK      OUTPUT
            TBLR      TEMP1
            LALK      LASTO
            TBLR      TEMP2
            LAC       TEMP2
            ADD       TEMP1
            SACL      TEMP1
            OUT       TEMP1,PA1
            LALK      LASTO
            TBLW      TEMP1
            LST       STATUS
            RET
            PEND
*
            CSEG      'IO'
INPUT       BSS       1
OUTPUT      BSS       1
            CEND
*
            DSEG
LASTIN      BSS       1
LASTO       BSS       1
            DEND
*
            END
```

Figure 9-3. Source for Module INTRPT

9.5.1 Simple Linking

Every control stream must contain either a TASK or PHASE 0 command to define the name of the program being linked. In addition, the control stream must contain one or more INCLUDE commands to define modules that are being linked. The control stream is terminated with an END command. The following is an example control stream on the TI/IBM PC MS/PC-DOS operating system for linking the three example object modules generated for MAIN, RESET, and INTRPT.

```
PHASE     0,SIMPLE
INCLUDE   MAIN.MPO
INCLUDE   RESET.MPO
INCLUDE   INTRPT.MPO
END
```

The three modules may be specified in one INCLUDE command rather than with three separate commands. The diskette containing the modules (A:) may also be defined as a library. When this is done, only the file name (enclosed in parentheses) need be specified. The Link Editor searches the defined library for the required files. An example of a control stream using the INCLUDE command is as follows:

```
PHASE     0,SIMPLE
LIBRARY   A:*.MPO
INCLUDE   (MAIN),(RESET),(INTRPT)
END
```

Since MAIN references RESET and INTRPT, and the directory containing these modules has been defined as a library, MAIN is the only module that must be specified in the INCLUDE command, as shown in the following example:

```
PHASE     0,SIMPLE
LIBR      A:*.MPO
INCL      (MAIN)
END
```

At the end of the control stream, the Link Editor automatically searches the defined library for unresolved references and includes the modules that satisfy the references in the linking process.

The Link Editor produces a listing of the linking process and writes it to a specified file. Figure 9-4 is an example of the listing file produced. The listing file for this example consists of three pages. The first page contains a copy of the link control stream. The second page lists the parameters used when the Link Editor was initialized (access names of the control file, linked output file, and listing file) and the format of the linked output. Since the FORMAT command was not included in the control stream, the default, ASCII, is used.

The third page contains the link map, which is generated to facilitate debugging. The link map lists the origins and lengths of the phase being linked, the modules included in the link, and any common segments. The origins are relative to the beginning of the phase. The order in which the included modules are linked is indicated by the number listed next to the module name. The link map also lists the symbols defined in the included modules, indicating the module in which the symbol is defined (number) and the resolved location of the symbol (value). An asterisk (*) preceding the symbol name indicates that the symbol is not referenced in the included modules. An asterisk to the right means the symbolic value is absolute.

```
PC/CrossWare Family Linker v.2.3 85.084 6/21/85 08:49:35 PAGE 1
COMMAND LIST

        PHASE     0,SIMPLE
        INCLUDE   A:MAIN.MPO
        INCLUDE   A:RESET.MPO
        INCLUDE   A:INTRPT.MPO
        END

PC/CrossWare Family Linker v.2.3 85.084 6/21/85 08:49:35 PAGE 2
LINK MAP

CONTROL FILE = A:SIMPLE.CTL

LINKED OUTPUT FILE = A:SIMPLE.LOD

LIST FILE = A:SIMPLE.MAP

OUTPUT FORMAT = ASCII

PC/CrossWare Family Linker v.2.3 85.084 6/21/85 08:49:35 PAGE 3

PHASE 0   SIMPLE MODULE   ORIGIN = 0000   LENGTH = 0083

MODULE  NO  ORIGIN  LENGTH   TYPE      DATE       TIME     CREATOR

MAIN     1   0000    0036   INCLUDE  06/21/85  08:48:34   ASM320
RESET    2   0036    002D   INCLUDE  06/21/85  08:49:03   ASM320
INTRPT   3   0063    001C   INCLUDE  06/21/85  08:49:18   ASM320
$DATA    3   007F    0002

COMMON          NO          ORIGIN    LENGTH

IO               1           0081      0002

                      D E F I N I T I O N S

NAME     VALUE NO    NAME     VALUE NO    NAME     VALUE NO

INTRPT   0063   3    MAIN     0020   1    RESET    0036   2

LENGTH OF REGION FOR TASK                = 0083

NUMBER OF RECORDS FOR MODULE SIMPLE      =   10

TOTAL RECORDS WRITTEN                    =   10

****  LINKING COMPLETED    06/21/85   08:49:42
```

Figure 9-4. Listing File for a Simple Link

9.5.2 ROM/RAM Partitioning

Each example module has a program segment defined by the PSEG assembler directive, a data segment defined by the DSEG directive, and a common defined by the CSEG directive. Program segments generally contain instructions and nonvariable data (read only). Data segments generally contain variable data (read/write) and are labeled by the Link Editor as $DATA. Common segments contain variable data that may be shared by more than one module.

The Link Editor automatically reorganizes the output so that all the program segments of the included modules are together, followed by the data segments and then the common segments. The link control commands PROGRAM, DATA, and COMMON can be used to specify the beginning location of each output segment. These commands cannot be used with a PROCEDURE command or a PHASE command with a level greater than zero.

The following is an example of the control stream for a VAX/VMS operating system, which is used to partition the program and data segments into potential ROM and RAM locations.

```
PHASE       0,SEGMENT
PROGRAM     >0000
DATA        >2000
COMMON      >3000,IO
INCLUDE     [PROJECT.MACK]MAIN.MPO
INCLUDE     [PROJECT.MACK]RESET.MPO
INCLUDE     [PROJECT.MACK]INTRPT.MPO
END
```

The example assumes that location >0000 is in ROM and locations >2000 and >3000 are in RAM. This control stream causes the program segment of MAIN to begin at location >0000, followed by the program segment of RESET, and then INTRPT. The data segment begins at location >2000. The common segment that is to be shared by the modules begins at location >3000. Note that if the common segment is not specifically named in the COMMON command, the segment begins immediately following the last data segment.

Figure 9-5 contains the listing produced by this link. Use of the PROGRAM, DATA, and COMMON commands causes the phase length to be listed as zero and the origins to be listed as absolute locations. An asterisk(*) preceding the symbol name indicates that the symbol is not referenced in the included modules. An asterisk following the value of a symbol name indicates an absolute location. Linking absolute code generated by the assembler (AORG assembler directive) also causes the phase length to be listed as zero and the origins to be absolute locations.

```
VAX/32020 LINKER VERSION v.2.3 85.084 6/21/85 08:49:53 PAGE 1
COMMAND LIST

        PHASE     0,SEGMENT
        PROGRAM   >0000
        DATA      >2000
        COMMON    >3000
        INCLUDE   [PROJECT.MACK]MAIN.MPO
        INCLUDE   [PROJECT.MACK]RESET.MPO
        INCLUDE   [PROJECT.MACK]INTRPT.MPO
        END

VAX/32020 LINKER VERSION v.2.3 85.084 6/21/85 08:49:53 PAGE 2
LINK MAP

CONTROL FILE = [PROJECT.MACK]SEGMENT.CTL

LINKED OUTPUT FILE = [PROJECT.MACK]SEGMENT.LOD

LIST FILE = [PROJECT.MACK]SEGMENT.MAP

OUTPUT FORMAT = ASCII

VAX/32020 LINKER VERSION v.2.3 85.084 6/21/85 08:49:53 PAGE 3

PHASE 0   SEGMENT MODULE   ORIGIN = 0000   LENGTH = 0000

MODULE  NO   ORIGIN  LENGTH   TYPE     DATE      TIME     CREATOR

MAIN     1    0000    0036   INCLUDE  06/21/85  08:48:34  ASM320
RESET    2    0036    002D   INCLUDE  06/21/85  08:49:03  ASM320
INTRPT   3    0063    001C   INCLUDE  06/21/85  08:49:18  ASM320
$DATA    3    2000    0002

COMMON          NO          ORIGIN    LENGTH

IO               1           3000*     0002

                      D E F I N I T I O N S

NAME     VALUE NO    NAME     VALUE NO    NAME     VALUE NO

INTRPT   0063*  3    MAIN     0020*  1    RESET    0036*  2

LENGTH OF REGION FOR TASK                = 0000

NUMBER OF RECORDS FOR MODULE SEGMENT     =   10

TOTAL RECORDS WRITTEN                    =   10

**** LINKING COMPLETED    06/21/85   08:50:05
```

Figure 9-5. Listing File for ROM/RAM Partitioning

9.5.3 Partial Linking

This section shows how to generate a partial link and then include the output of the partial link in a subsequent link. Only ASCII object code can be used in partial linking on the TMS320C25 device.

The PARTIAL command is used in the control stream to specify a partial link. In this example, modules RESET and INTRPT are to be linked together in a partial link. The output of the partial link is not executable and must be linked again without the PARTIAL command so that the output of this partial link will then be linked with module MAIN to produce an executable module. The following is the control stream for the partial link, using the TI/IBM PC (MS/PC-DOS) operating system:

```
PARTIAL
PHASE     0,PARTIAL
INCLUDE   A:RESET.MPO
INCLUDE   A:INTRPT.MPO
END
```

All commands pertaining to partial links must be issued before any INCLUDE, SEARCH, and FIND commands. The PARTIAL command must be given before the first INCLUDE command in the control stream. In a partial link, only one phase is allowed and must be defined by the PHASE 0 or TASK command.

The ALLGLOBAL, GLOBAL, and NOTGLOBAL commands are used with the PARTIAL command to define the scope of DEF tags in modules included in the partial link. These symbols are specified as either global or local. All externally-defined symbols are processed as global symbols. Global symbols are externally defined in the partially linked output modules and may be referenced in a subsequent link. Local symbols are not externally defined in the partially linked output module; therefore, they may be referenced in the current partial link. Since none of these commands are included in the control stream, the default, ALLGLOBAL, is used.

The output of the partial link can now be linked with module MAIN to produce an executable module, using the following control stream:

```
PHASE     0,PROJ
INCLUDE   B:MAIN.MPO
INCLUDE   B:PARTIAL.MPO
END
```

The listing and object modules from a partial link using the PARTIAL command are given in Figure 9-6.

The second part of the link, in which the output of the partial link is relinked without using the PARTIAL command, is performed next. The listing and object files for relinking the output of the partial link are shown in Figure 9-7.

```
PC/CrossWare Family Linker v.2.3 85.084 6/21/85 08:51:09 PAGE 1
COMMAND LIST

       PARTIAL
       PHASE     0,PARTIAL
       INCLUDE   A:RESET.MPO
       INCLUDE   A:INTRPT.MPO
       END

PC/CrossWare Family Linker v.2.3 85.084 6/21/85 08:51:09 PAGE 2
LINK MAP

CONTROL FILE = A:PARTIAL.CTL

LINKED OUTPUT FILE = A:PARTIAL.MPO

LIST FILE = A:PARTIAL.MAP

OUTPUT FORMAT = ASCII

PC/CrossWare Family Linker v.2.3 85.084 6/21/85 08:51:09 PAGE 3

PHASE 0   PARTIAL MODULE   ORIGIN = 0000   LENGTH = 004D

MODULE  NO   ORIGIN  LENGTH   TYPE      DATE      TIME     CREATOR

RESET    1    0000    002D   INCLUDE  06/21/85  08:49:03   ASM320
INTRPT   2    002D    001C   INCLUDE  06/21/85  08:49:18   ASM320
$DATA    2    0000    0002

COMMON          NO          ORIGIN    LENGTH

IO               2           0000      0002

                      D E F I N I T I O N S

NAME     VALUE NO    NAME     VALUE NO

*INTRPT  002D   2    *RESET   0000   1

                       UNRESOLVED REFERENCES

MAIN           1

LENGTH OF REGION FOR TASK                = 004D

NUMBER OF UNRESOLVED REFERENCES          =    1

NUMBER OF RECORDS FOR MODULE PARTIAL     =    9

TOTAL RECORDS WRITTEN                    =    9

**** LINKING COMPLETED   06/21/85   08:51:17
```

a. Listing File for a Partial Link

Figure 9-6. Listing and Object Files for a Partial Link

```
K0049PARTIAL M0002$DATA 0000M0002IO    000250000RESET 5002DINTRPT7F0EAF
40000MAIN  A0000B5589BD100B0300BCA00BCB23B60A0BD100B0200BCB1FBFCA07F136F
C000DBFF80E00000000B00B0BFF35B0129BFE72B01EDBFDCAB0256BFDC9B01C07F140F
BFF2CBFF50B0304BF96CB0C79B0008B0007B0006B0005B0C79BF96CB0304BFF507F12EF
BFF2CB01C0BFDC9B0256BFDCAB01EDBFE72B0129BFF35B00B0A002DB7860BC8007F0C4F
B8061BD001T0000B5862B5961B2062B1061B6061BD001N00000002B5961BD0017F225F
N00010002B5861BD001T0001B5862B2062B0061B6061BE161BD001T0001B59617F219F
B5060BCE267FD8AF
:     PARTIAL    06/21/85  08:51:09    XLNKPC  v2.3    85.084
```

b. Object File for a Partial Link

Figure 9-6. Listing and Object Files for a Partial Link (Concluded)

The second part of the link, in which the output of the partial link is relinked without using the PARTIAL command, is performed next. The listing and object files for relinking the output of the partial link are shown in Figure 9-7.

```
PC/CrossWare Family Linker v.2.3 85.084 6/21/85 08:51:33 PAGE 1
COMMAND LIST

       PHASE     0,PROJ
       INCLUDE   A:MAIN.MPO
       INCLUDE   A:PARTIAL.MPO
       END

PC/CrossWare Family Linker v.2.3 85.084 6/21/85 08:51:33 PAGE 2
LINK MAP

CONTROL FILE = A:PROJ.CTL

LINKED OUTPUT FILE = A:PROJ.LOD

LIST FILE = A:PROJ.MAP

OUTPUT FORMAT = ASCII

PC/CrossWare Family Linker v.2.3 85.084 6/21/85 08:51:33 PAGE 3

PHASE 0   PROJ MODULE   ORIGIN = 0000   LENGTH = 0083

MODULE  NO   ORIGIN  LENGTH   TYPE      DATE       TIME     CREATOR

MAIN     1    0000    0036   INCLUDE  06/21/85  08:48:34   ASM320
PARTIAL  2    0036    0049   INCLUDE  06/21/85  08:51:09   XLNKPC
$DATA    2    007F    0002

COMMON          NO          ORIGIN    LENGTH

IO               1           0081      0002

                        D E F I N I T I O N S

NAME     VALUE NO    NAME     VALUE NO    NAME     VALUE NO

INTRPT   0063   2    MAIN     0020   1    RESET    0036   2

LENGTH OF REGION FOR TASK                 = 0083

NUMBER OF RECORDS FOR MODULE PROJ         =   10

TOTAL RECORDS WRITTEN                     =   10

**** LINKING COMPLETED    06/21/85   08:51:40
```

a. Listing File for Relinking the Partial Link Output

```
K0083PROJ    A0000BFF80C0036BFF80C0063A0020B5589BC806BCE1FBD0017F1D8F
C0081B5820BD100B0320BCE05BA000BCA00BCB1FB5C90BFF00BCE15B6080BCE047F129F
BD001C0082B5900BFF80C0022A0036B5589BD100B0300BCA00BCB23B60A0BD1007F193F
B0200BCB1FBFCA0C0043BFF80C0020B00B0BFF35B0129BFE72B01EDBFDCAB02567F0EDF
BFDC9B01C0BFF2CBFF50B0304BF96CB0C79B0008B0007B0006B0005B0C79BF96C7F10CF
B0304BFF50BFF2CB01C0BFDC9B0256BFDCAB01EDBFE72B0129BFF35B00B0A00637F0C9F
B7860BC800B8061BD001C007FB5862B5961B2062B1061B6061BD001C0081B59617F1C0F
BD001C0082B5861BD001C0080B5862B2062B0061B6061BE161BD001C0080B59617F1DAF
B5060BCE267FD8AF
:     PROJ       06/21/85  08:51:33    XLNKPC  v2.3    85.084
```

b. Object File for Relinking the Partial Link Output

Figure 9-7. Listing and Object Files for Relinking the Partial Link Output

9.5.4 Library Creation

The linker can accommodate two object library types: random and sequential.

A random library can be created almost automatically. Whenever one or more object files are placed in the same directory or sub-directory, that directory becomes a random library. Some examples of random libraries are as follows:

DUA0:[USER07.PROJ2710.PARTS]
(VAX), where the example indicates a directory containing object file members named (e.g., PART1.OBJ, INITIAL.OBJ, CLEANUP.OBJ, etc.).

A:*.MPO
(MS-DOS), where the name indicates a drive and all files with the extension .MPO (e.g., QUICK.MPO, BTREE.MPO, SHELL.MPO, etc.).

The creation of sequential libraries is more involved. Since sequential libraries offer no advantages over random libraries, their use is probably restricted to those users of systems not supporting random libraries, i.e., not supporting multilevel directories or "wild-card" file specifications.

A sequential library is a single file and consists of a "dictionary," followed by one or more concatenated object modules. The user must order the elements in a sequential library so that no object segment contains an external reference to a preceding segment. The concatenated object files may be created by assembling a source file created by concatenating the source files of several proposed members of the sequential library. Such a source file may appear as shown in Figure 9-8.

```
          IDT    'TRESRT'
          TITL   'THIS IS THE FIRST LIB MEMBER'
          DEF    TRESRT,QUICK
*
QUICK     EQU    $
           .
           .
           .
TRESRT    EQU    $
           .
          END
*
          IDT    'ELEM'
          TITL   'STILL IN SEQ LIB'
          DEF    ELEM
           .
           .
           .
          END
*
          IDT    'LASTPROG'
          TITL   'ET CETERA'
           .
           .
           .
          END
```

Figure 9-8. Source File for Sequential Library Creation

The assembler output contains all modules within the same file, yet distinct. The use of a text editor allows the creation of such a file by appending the object files that result from independent assembly of the proposed library members. The dictionary structure must be created by use of the text editor. It precedes the first object module and must match the following pattern:

First line, the library 'IDT' record: 0llllaaaaaaaaF

where 0 = tag
llll = length of the dictionary
aaaaaaaa = library name
F = end-of-record tag
user-defined information out to 80th character

Example: 00000SEQLIB01F 04/10/85 08:15:00

One entry for each object module included in the library:
FIllllaaaaaaaa;

where F = tag
I = IDT marker
llll = length of PSEG of module
aaaaaaaa = IDT of the module
';' = record end marker

Example: FI00CEFASTSORT;

For each DEF within a module: FEtllllaaaaaa{,tllllaaaaaa};

where F = tag
E = tag
t = type of DEF ("A" = absolute, "R" = relative)
llll = value of the DEF'd symbol
aaaaaa = 6-character name of symbol DEF'd in this module
, = if more (up to five DEFs allowed per record
; = if no more

Example: FEA0050PACK ,R00ACUNPAK ,A0E08ENCODE;

For each REF within the module: FR aaaaaa{, aaaaaa}

where F = tag
R = tag
' ' = five blanks
aaaaaa = 6-character name of symbol REF'd
, = if more (up to five REFs allowed per record)
; = if no more

Example: FR EXREF , ESYM , X

For each COMMON segment contained within the module:
FCllllaaaaaa{,llllaaaaaa};

where F = tag
C = tag
llll = length of COMMON segment
aaaaaa = 6-character COMMON segment name

Example: FC0140COMNAM,0266$BLANK;

For each DATA segment defined within the module:
FDllll {,llll }

where F = tag
D = tag
llll = length of DATA segment
' ' = six blanks
up to five DSEGs allowed, separated by commas
terminated by ;

Example: FD0010 ,0032

This set of records is repeated for each object module in the library. The final record, just prior to the first record of the first library member, must contain a colon in the first character position. The remainder of this colon record is not specified. It can be used for date, time, and other user-defined information.

9.6 Link Editor Error Messages

Messages are listed for detected errors in the listing file. The Link Editor error messages are named and described below.

When the error-message description indicates that a malfunction of the link editor has occurred, please contact the Texas Instruments Customer Response Center (CRC) hotline number, 1-800-232-3200, extension 2171, for assistance. 713-879 2320

'(' EXPECTED: The REPLACE command expects a parenthesis.

ADDRESS SPACE HAS OVERFLOWED IN THIS MODULE: The maximum address required to represent this module is >10000 or greater. No valid object module can be produced for this phase. The linker continues to produce the map, but with increased likelihood that it will abort from internal errors.

ADDRESS SPACE TRUNCATED FOR TAG = X IN THE SEGMENT STARTING AT YYYY: The 320-specific tags have a seven-bit address field that has overflowed.

ALIGNMENT VALUE MUST BE IN THE RANGE 0..15: The value in the ADJUST command is out of range.

AN ACTIVE BUFFER SHOULD HAVE BEEN CLOSED: A buffer that needs to be closed is still marked as active.

ATTEMPT MADE TO WRITE TO A NIL SEGMENT: The linker attempted to write to a nonexistent segment. Indicates a malfunction of the link editor; call hotline immediately.

ATTEMPT MADE TO WRITE TO INACTIVE SEGMENT: The linker has attempted to write to an unopen segment. Indicates a malfunction of the link editor; call hotline immediately.

ATTEMPT TO ACTIVATE AN ALREADY ACTIVE SEGMENT: The linker has attempted to open a segment that is already active. Indicates a malfunction of the link editor; call hotline immediately.

ATTEMPT TO ACTIVATE NIL SEGMENT: The linker has attempted to open a nonexistent segment. Indicates a malfunction of the link editor; call hotline immediately.

ATTEMPT TO ALLOCATE AFTER DSEG OF CSEG DEFINED: The ALLOCATE command has been given after data and/or common segments have been encountered.

ATTEMPT TO MOVE NON-COMMON SEGMENT: An attempt has been made to move a segment that is not a common. Indicates a malfunction of the link editor; call hotline immediately.

ATTEMPT TO ORDER A NIL SEGMENT: A common that was never defined cannot be placed in the stream of commons. Indicates a malfunction of the link editor; call hotline immediately.

ATTEMPT TO REDEFINE COMMON ORIGIN: Directives to place a common origin provide information that conflicts with information that has been already determined.

BAD CHAIN FOR XXXX TO YYYY: In a partial link, the reference chain for XXXX points to the address YYYY that is outside the scope of the segment.

BAD INDEX - COMMAND FORMAT NOT RECOGNIZED: In a partial link, an error was detected in the control record such that the current index is negative.

BAD TAG IN CHAIN FOR XXXX AT YYYY: In a partial link, an invalid tag was encountered in the processing of the reference chain. If this error occurs, the object module has been damaged.

CANNOT ORDER COMMONS FROM DIFFERENT MODULES: Commons that are overlayed in procedures of the same phase level cannot be ordered together. This usually occurs in a partial link.

CAN'T ASSIGN LUNO TO LIBRARY: The limit has been exceeded in the number of files that can be opened at one time.

CAN'T OPEN FILE: A file that should be opened cannot be opened. The system return code is identified in the immediate preceding warning.

CAN'T OPEN LIBRARY: An error code was returned or an attempt made to open the directory.

CHAIN TO UNINITIALIZED LOCATION FOR XXXX AT YYYY: The reference chain for value XXXX points from YYYY to an insignificant address. Processing of the chain is discontinued.

COMMAND NOT VALID WITH PARTIAL LINK: The specified command is not valid in producing a partial link.

COMMAND ONLY VALID WITH PARTIAL LINK: The GLOBAL, NOTGLOBAL, and ALLGLOBAL commands must follow a PARTIAL command in a control stream.

COMMON HAS NOT BEEN PLACED VALIDLY: An attempt has been made to place a common at two or more locations.

COMMON NAME TRUNCATED: The common name has been truncated to six characters. This is a trivial warning and processing proceeds with the truncated name.

COMMON NUMBER INVALID: The common number given an M tag is not in the valid range.

COMMON ORIGIN HAS ALREADY BEEN SET: An attempt has been made to define a common origin already set with a previous COMMON command.

COMMON ORIGIN INVALID: The origin for a common is not valid.

COMMON ORIGIN WAS NOT DEFINED: The first common name specified in a COMMON command was never defined in the link.

COMMON SEGMENT HAS NO SYMBOL DEFINED: A symbol has been given for a common segment, but the segment itself does not exist.

COMMON SYMBOL IS NOT VALID: The given common symbol is not legal.

COMMON SYMBOL WAS NEVER DEFINED: A common segment was not found corresponding to the common symbol.

COMPRESSED FORMAT NOT SUPPORTED FOR 320 INPUT MODULES: The linker only recognizes ASCII-formatted input modules for these object codes.

CONFLICTING COMMON SYMBOL FOUND: A common symbol that is not consistent with previous commons has been detected.

CURRENT SEGMENT HAS NOT BEEN DEACTIVATED: The segment that should be closed has been left active.

DUPLICATE SYMBOL DEFINITION ENCOUNTERED: Two definitions have been encountered for the same external symbol.

ENTRY NAME TRUNCATED: The entry name has been truncated to six characters. This is a trivial warning and processing proceeds with the truncated name.

EXTERNAL REFERENCE INDEX OUT OF RANGE: The index number specified by an E tag in an input object module is not valid.

EXTERNAL SYMBOL TRUNCATED: A symbol specified in a GLOBAL or NOTGLOBAL command exceeds six characters in length.

FATAL ERROR DETECTED -- *LINKER ABORTING*: An error that the linker cannot recover from has been detected. The user should repeat the process. If the message occurs again, then either check the procedures used or call the hotline for assistance.

FIRST PHASE HAS ALREADY BEEN DEFINED: A NOMAP command has appeared after a TASK or a PHASE command has been issued in the control stream.

HEAP ERROR ENCOUNTERED IN PASS2: A heap error was detected while trying to allocate data space for the second pass. Indicates a malfunction of the link editor; call hotline immediately.

ILLEGAL INTERMEDIATE TAG ENCOUNTERED AT XXXX: An encoded tag was encountered that was not valid. Indicates a malfunction of the link editor; call hotline immediately.

ILLEGAL TAG FOUND IN INTERMEDIATE FILE; TAG=X: An invalid tag was found in the intermediate file. Indicates a malfunction of the link editor; call hotline immediately.

INTERMEDIATE FILE OVERFLOW: The maximum number of records for intermediate object representation has been exceeded. Obtain more file space by making individual object modules smaller or call the hotline for assistance.

INTERMEDIATE RECORD NUMBER INVALID: The record index for the intermediate storage is not in the legal range.

INTERNAL LINKER ERROR IN AUTOCALL: An error has occurred in the automatic-call algorithm. This error should never occur. If it does, unresolved references may be the result of modules not having been read in; in other respects, the object module produced should be good. Relink using specific INCLUDES for the missing modules or call the hotline for assistance.

INVALID ATTEMPT TO MOVE FIRST COMMON: The linker has attempted to move a common that was specified as the first common by a COMMON command. Indicates a malfunction of the link editor; call hotline immediately.

INVALID ATTEMPT TO READ BUFFER: The linker has attempted to activate a buffer that is not of the correct type. Indicates a malfunction of the link editor; call hotline immediately.

INVALID LEVEL FOR PHASE: The level argument to a PHASE command is not appropriate. The first phase established must be a TASK or a phase of level 0. If the current level is N, a new phase must have level $\leq N+1$.

INVALID PROCEDURE LEVEL: The level for procedures must be 1 or 2.

INVALID PROCEDURE SPECIFIED: An illegal procedure has been declared.

INVALID SYMBOL NAME FOR REPLACE: The REPLACE command has encountered an illegal symbol name.

INVALID VALUE FOR LINES PER PAGE: The argument to a page command is not recognized as a positive integer or is out of the range of 16 to 60 lines per page.

LAST COMMON IN LIST IS NIL: The last common in a list of ordered commons does not exist. Indicates a malfunction of the link editor; call hotline immediately.

LAST MODULE FOR PHASE IS NIL: The linker cannot find the information about the current phase. Indicates a malfunction of the link editor; call hotline immediately.

MAP RECORD INDEX IS OUT OF RANGE: The map record is full or the index has been changed to an invalid value. Indicates a malfunction of the link editor; call hotline immediately.

MEMBER NAME TOO LONG: The member name exceeds eight characters. The command is not processed.

MEMBER NAME TRUNCATED: The member name has been truncated to eight characters. This is a trivial warning and processing proceeds with the truncated name.

MINIMUM NUMBER OF LINES PER PAGE IS 16: A PAGE N command may not specify a value of N less than 16.

MODULE LENGTH IS ZERO: The length for the module has been incorrectly specified as zero.

MODULE ORIGIN IS NOT ZERO: The origin for a module must be zero before the relocation is applied.

NIL COMMON SEGMENT WAS ACTIVATED: An attempt was made to activate a common segment that does not exist.

NIL SEGMENT FOR M TAG SYMBOL: An M tag definition applies to a segment that does not exist. Indicates a malfunction of the link editor; call hotline immediately.

NO PHASE IS DEFINED: No PROCEDURE, TASK, or PHASE 0 has been defined. A command has been given which requires that object modules be read in or that some phase be active.

NO TASK PHASE IS DEFINED: No TASK or PHASE 0 has been defined. A valid set of linked object modules cannot be produced.

NOTGLOBAL MUST PRECEDE A GLOBAL COMMAND: The GLOBAL command is only valid if it is preceded by a NOTGLOBAL command with no parameter.

OBJECT CARD INDEX ERROR DETECTED: After writing an object record, the index into the record was not equal to one. Indicates a malfunction of the link editor; call hotline immediately.

ORIGIN CANNOT BE WRITTEN TO INACTIVE SEGMENT: The linker has attempted to write origin information to a segment that is not open. Indicates a malfunction of the link editor; call hotline immediately.

OVERWRITTEN BLOCKS FOR XXXX TO YYYY: Absolutely placed object code overlaps at the given address.

OVERWRITTEN SEGMENTS STARTING AT XXXX IN MODULE NNNNNNNN: Overlapping segments have been detected starting at location XXXX. The link map specifies which segment starts at that point. This is flagged as a warning.

PARTIAL COMMAND INVALID IN CONTEXT: The PARTIAL command was specified in the control stream after a command that is inconsistent with partial links (e.g., DUMMY, PROGRAM, DATA, COMMON, ALLOCATE, PROCEDURE, PHASE 1, etc.)

PHASE LEVEL EXPECTED: The level argument to a PHASE command must be a zero or a positive integer.

PHASE LEVEL SPECIFIED IS NOT VALID: The level specified in a PHASE command is not in the valid range.

PHASE NAME TRUNCATED: The phase name has been truncated to eight characters. This is a trivial warning, and processing proceeds with the truncated name.

PHASE SEQUENCE IS NOT VALID: The order in which the phases have been declared is not legal.

PREMATURE END OF CONTROL FILE: The control file has ended before an END command was encountered. No further processing is done.

PROCEDURE CANNOT HAVE BROTHERS: A procedure cannot have phases at the same level defined with it.

PROCEDURE NAME TRUNCATED: The procedure name has been truncated to eight characters. This is a trivial warning, and processing proceeds with the truncated name.

PROC 1 MUST BE DUMMIED TO DUMMY PROC 2: In order to dummy the second procedure, the first procedure must also be dummied.

PROC 1 SYMBOL NUMBER IS NOT ZERO: The symbol number for the procedure must be zero.

PROC 2 SYMBOL NUMBER IS NOT ZERO: The symbol number for the procedure must be zero.

RELOCATABLE ADDRESS IS NOT VALID;SEGMENTS SHOULD BE PLACED AT ABSOLUTE LOCATIONS: Certain TMS320-specific tags require that segments to which they refer be placed at absolute addresses.

SEGMENT BUFFER HAS BEEN DAMAGED: The current segment does not contain the expected information. Indicates a malfunction of the link editor; call hotline immediately.

SEGMENT ORIGIN IS ZERO: The origin for a segment has erroneously changed to zero. Indicates a malfunction of the link editor; call hotline immediately.

TASK OR PHASE 0 IS ALREADY DEFINED: A PROCEDURE command cannot be given once a task phase has been defined.

TASK OR PHASE 0 MUST BE DEFINED BEFORE OVERLAY: An overlay has been defined before the task or root phase.

THE CURRENT SEGMENT IS NIL: The segment that is being examined does not exist. Indicates a malfunction of the link editor; call hotline immediately.

THE INPUT OBJECT MODULE HAS BEEN DAMAGED: Unexpected or invalid tags and values have been encountered in the input object module.

THE MAP RECORD IS NIL: An attempt has been made to place information into the map record when the record does not exist. Indicates a malfunction of the link editor; call hotline immediately.

THE OVERWRITTEN BLOCKS ARE NOT COMPATIBLE: The types of the overlapping blocks are not the same, and a valid object module cannot be produced.

THE PHASE TYPE IS NOT TASK, OVLY, OR PROC: This error should never occur, because the only valid types are TASK, OVLY, and PROC.

THE SEGMENT TYPE IS NOT PSEG, DSEG, OR CSEG: The only valid segment types are PSEG, DSEG, and CSEG.

TOO MANY SYMBOLS HAVE BEEN DEFINED: The statically allocated arrays that contain the values for symbols (mostly external symbols and phase lengths, origins, etc.) have overflowed. The number of symbols allowed in a symbol table is 1110.

UNABLE TO PROPERLY ORDER COMMONS: The linker cannot order the commons as specified.

UNEXPECTED TAG: The input object module is not of the expected format. It may not really be an object module. Processing stops.

UNRECOGNIZED FORMAT: The argument to the FORMAT command is not recognized.

UNRECOGNIZED COMMAND: The command on the most recent line is not recognized as a linker command. The line is ignored.

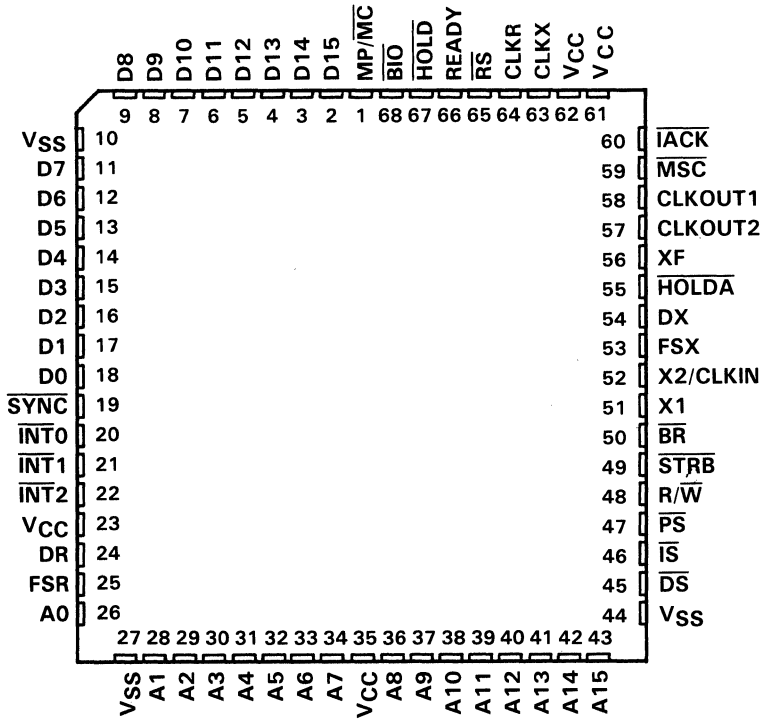
UNSUPPORTED INTER-SEGMENT LINK FOR XXXX FROM YYYY TO ZZZZ: This message is printed by the second pass, and applies to the module in the linked output that is next identified. An external reference chain has pointed from one PSEG, DSEG, or CSEG into another. XXXX is the value of the external symbol to be filled in. (The second pass cannot identify it by symbol name. The name can often be found by examining the symbol definitions. A symbol with a value of zero may be an unresolved reference.) YYYY, the address where the chain starts, identifies the offending module. ZZZZ is the address to which the chain points. The only deficiency in the linked object is that incorrect values remain where the value of external symbol XXXX should have been inserted.

TMS320C25
DIGITAL SIGNAL PROCESSOR

MAY 1986

- 100-ns Instruction Cycle Time
- 544 Words of Programmable On-Chip Data RAM
- 4K Words of On-Chip Program ROM
- 128K Words of Data/Program Space
- Sixteen Input and Sixteen Output Channels
- 16-Bit Parallel Interface
- Directly Accessible External Data Memory Space
- Global Data Memory Interface
- 16-Bit Instruction and Data Words
- 32-Bit ALU and Accumulator
- Single-Cycle Multiply/Accumulate Instructions
- 0 to 16-Bit Scaling Shifter
- Bit Manipulation and Logical Instructions
- Instruction Set Support for Floating-Point Operations, Adaptive Filtering, and Extended-Precision Arithmetic
- Block Moves for Data/Program Management
- Repeat Instructions for Efficient Use of Program Space
- Eight Auxiliary Registers and Dedicated Arithmetic Unit for Indirect Addressing
- Serial Port for Direct Codec Interface
- Synchronization Input for Synchronous Multiprocessor Configurations
- Wait States for Communication to Slow Off-Chip Memories/Peripherals
- On-Chip Timer for Control Operations
- Three External Maskable User Interrupts
- Input Pin Polled by Software Branch Instruction
- Programmable Output Pin for Signalling External Devices
- 1.8-μm CMOS Technology
- Single 5-V Supply
- On-Chip Clock Generator

68-PIN FN
PLASTIC LEADED CHIP CARRIER PACKAGE
(TOP VIEW)



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

PIN NOMENCLATURE

SIGNALS	I/O/Z†	DEFINITION
V_{CC}	I	5-V supply pins
V_{SS}	I	Ground pins
X1	O	Output from internal oscillator for crystal
X2/CLKIN	I	Input to internal oscillator from crystal or external clock
CLKOUT1	O	Master clock output (crystal or CLKIN frequency/4)
CLKOUT2	O	A second clock output signal
D15-D0	I/O/Z	16-bit data bus D15 (MSB) through D0 (LSB). Multiplexed between program, data, and I/O spaces.
A15-A0	O/Z	16-bit address bus A15 (MSB) through A0 (LSB)
$\overline{PS}$, $\overline{DS}$, $\overline{IS}$	O/Z	Program, data, and I/O space select signals
R/$\overline{W}$	O/Z	Read/write signal
$\overline{STRB}$	O/Z	Strobe signal
$\overline{RS}$	I	Reset input
$\overline{INT2}$-$\overline{INT0}$	I	External user interrupt inputs
MP/$\overline{MC}$	I	Microprocessor/microcomputer mode select pin
$\overline{MSC}$	O	Microstate complete signal
$\overline{IACK}$	O	Interrupt acknowledge signal
READY	I	Data ready input. Asserted by external logic when using slower devices to indicate that the current bus transaction is complete.
$\overline{BR}$	O	Bus request signal. Asserted when the TMS320C25 requires access to an external global data memory space.
XF	O	External flag output (latched software-programmable signal)
$\overline{HOLD}$	I	Hold input. When asserted, TMS320C25 goes into an idle mode and places the data, address, and control lines in the high-impedance state.
$\overline{HOLDA}$	O	Hold acknowledge signal
$\overline{SYNC}$	I	Synchronization input
$\overline{BIO}$	I	Branch control input. Polled by BIOZ instruction.
DR	I	Serial data receive input
CLKR	I	Clock for receive input for serial port
FSR	I	Frame synchronization pulse for receive input
DX	O/Z	Serial data transmit output
CLKX	I	Clock for transmit output for serial port
FSX	I/O/Z	Frame synchronization pulse for transmit. Configurable as either an input or an output.

†I/O/Z denotes input/output/high-impedance state.

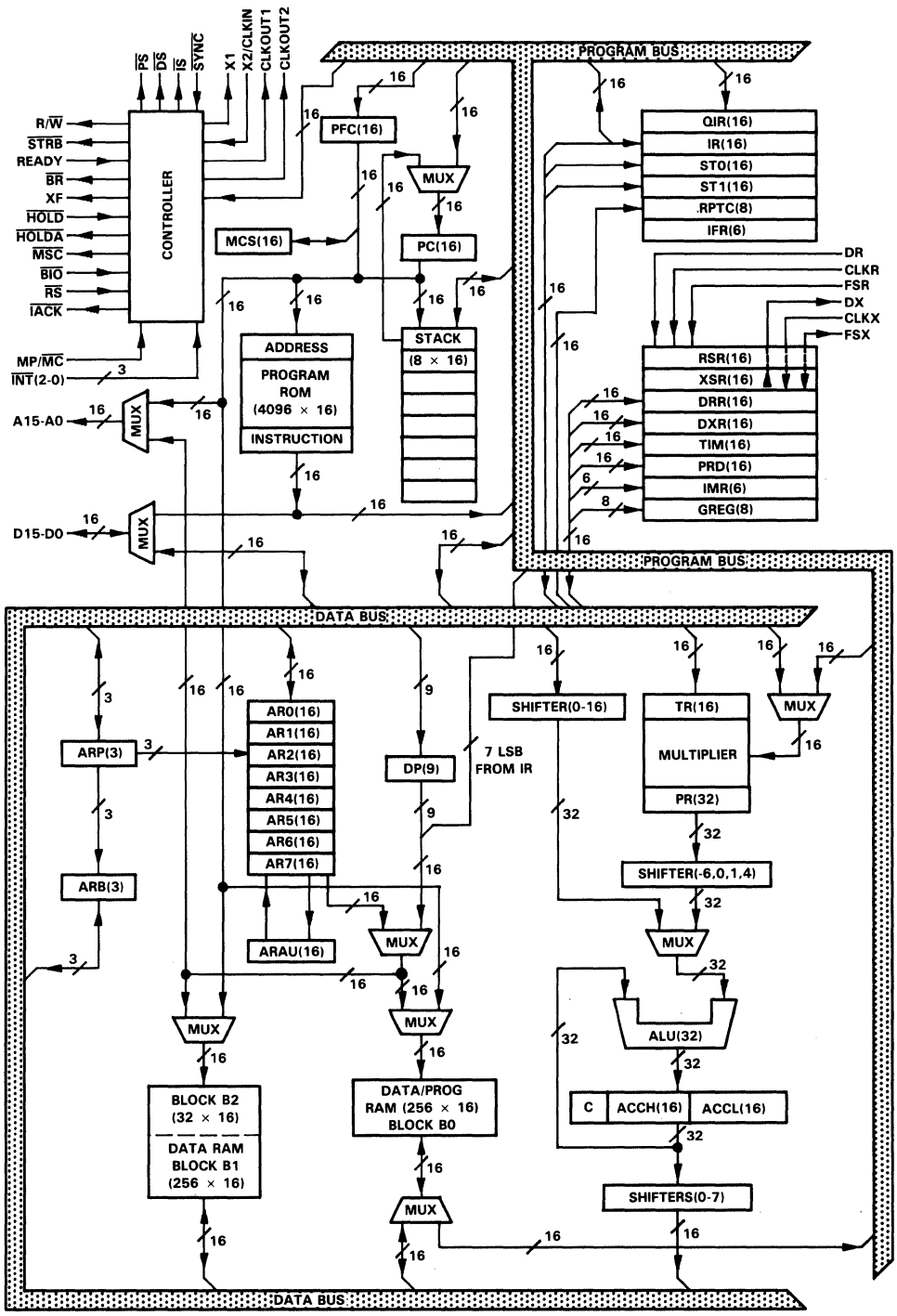
description

The TMS320C25 Digital Signal Processor is a member of the TMS320 family of VLSI digital signal processors and peripherals. The TMS320 family supports a wide range of digital signal processing applications, such as telecommunications, modems, image processing, speech processing, spectrum analysis, audio processing, digital filtering, high-speed control, graphics, and other computation-intensive applications.

With a 100-ns instruction cycle time and an innovative memory configuration, the TMS320C25 performs operations necessary for many real-time digital signal processing algorithms. Since most instructions require only one cycle, the TMS320C25 is capable of executing ten million instructions per second. On-chip data RAM of 544 16-bit words, on-chip program ROM of 4K words, direct addressing of up to 64K words of external data memory space and 64K words of external program memory space, and multiprocessor interface features for sharing global memory minimize unnecessary data transfers to take full advantage of the capabilities of the processor.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

functional block diagram



LEGEND:

ACCH	= Accumulator high	IFR	= Interrupt flag register	PC	= Program counter
ACCL	= Accumulator low	IMR	= Interrupt mask register	PFC	= Prefetch counter
ALU	= Arithmetic logic unit	IR	= Instruction register	RPTC	= Repeat instruction counter
ARAU	= Auxiliary register arithmetic unit	MCS	= Microcall stack	GREG	= Global memory allocation register
ARB	= Auxiliary register pointer buffer	QIR	= Queue instruction register	RSR	= Serial port receive shift register
ARP	= Auxiliary register pointer	PR	= Product register	XSR	= Serial port transmit shift register
DP	= Data memory page pointer	PRD	= Period register for timer	AR0-AR7	= Auxiliary registers
DRR	= Serial port data receive register	TIM	= Timer	ST0,ST1	= Status registers
DXR	= Serial port data transmit register	TR	= Temporary register		

FIGURE 3-1.

architecture

The TMS320C25 architecture is based upon that of the TMS32020, the second member of the TMS320 family. The TMS320C25 increases performance of DSP algorithms through innovative additions to the TMS320 family architecture. TMS32020 source code is upward-compatible with TMS320C25 source code and can be assembled using the TMS320C25 Macro Assembler. TMS32020 object code will run directly on the TMS320C25.

Increased throughput on the TMS320C25 for many DSP applications is accomplished by means of single-cycle multiply/accumulate instructions with a data move option, eight auxiliary registers with a dedicated arithmetic unit, and faster I/O necessary for data-intensive signal processing.

The architectural design of the TMS320C25 emphasizes overall speed, communication, and flexibility in processor configuration. Control signals and instructions provide floating-point support, block-memory transfers, communication to slower off-chip devices, and multiprocessing implementations.

Two large on-chip RAM blocks, configurable either as separate program and data spaces or as two contiguous data blocks, provide increased flexibility in system design. Programs of up to 4K words can be masked into the internal program ROM. The remainder of the 64K-word program memory space is located externally. Large programs can execute at full speed from this memory space. Programs can also be downloaded from slow external memory to high-speed on-chip RAM. A total of 64K data memory address space is included to facilitate implementation of DSP algorithms. The VLSI implementation of the TMS320C25 incorporates all of these features as well as many others, such as a hardware timer, serial port, and block data transfer capabilities.

32-bit ALU/accumulator

The TMS320C25 32-bit Arithmetic Logic Unit (ALU) and accumulator perform a wide range of arithmetic and logical instructions, the majority of which execute in a single clock cycle. The ALU executes a variety of branch instructions dependent on the status of the ALU or a single bit in a word. These instructions provide the following capabilities:

- Branch to an address specified by the accumulator
- Normalize fixed-point numbers contained in the accumulator
- Test a specified bit of a word in data memory.

One input to the ALU is always provided from the accumulator, and the other input may be provided from the Product Register (PR) of the multiplier or the input scaling shifter which has fetched data from the RAM on the data bus. After the ALU has performed the arithmetic or logical operations, the result is stored in the accumulator.

The 32-bit accumulator is split into two 16-bit segments for storage in data memory. Additional shifters at the output of the accumulator perform shifts while the data is being transferred to the data bus for storage. The contents of the accumulator remain unchanged.

scaling shifter

The TMS320C25 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. The LSBs of the output are filled with zeroes, and the MSBs may be either filled with zeroes or sign-extended, depending upon the status programmed into the SXM (sign-extension mode) bit of status register ST0.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

16 x 16-bit parallel multiplier

The TMS320C25 has a 16 x 16-bit hardware multiplier, which is capable of computing a signed or unsigned 32-bit product in a single machine cycle. The multiplier has the following two associated registers:

- A 16-bit Temporary Register (TR) that holds one of the operands for the multiplier, and
- A 32-bit Product Register (PR) that holds the product.

Incorporated into the TMS320C25 instruction set are single-cycle multiply/accumulate instructions that allow both operands to be processed simultaneously. The data for these operations may reside anywhere in internal or external memory, and can be transferred to the multiplier each cycle via the program and data buses.

Four product shift modes are available at the Product Register (PR) output that are useful when performing multiply/accumulate operations, fractional arithmetic, or justifying fractional products.

timer

The TMS320C25 provides a memory-mapped 16-bit timer for control operations. The on-chip timer (TIM) register is a down counter that is continuously clocked by CLKOUT1. A timer interrupt (TINT) is generated every time the timer decrements to zero. The timer is reloaded with the value contained in the period (PRD) register within the next cycle after it reaches zero so that interrupts may be programmed to occur at regular intervals of PRD + 1 cycles of CLKOUT1.

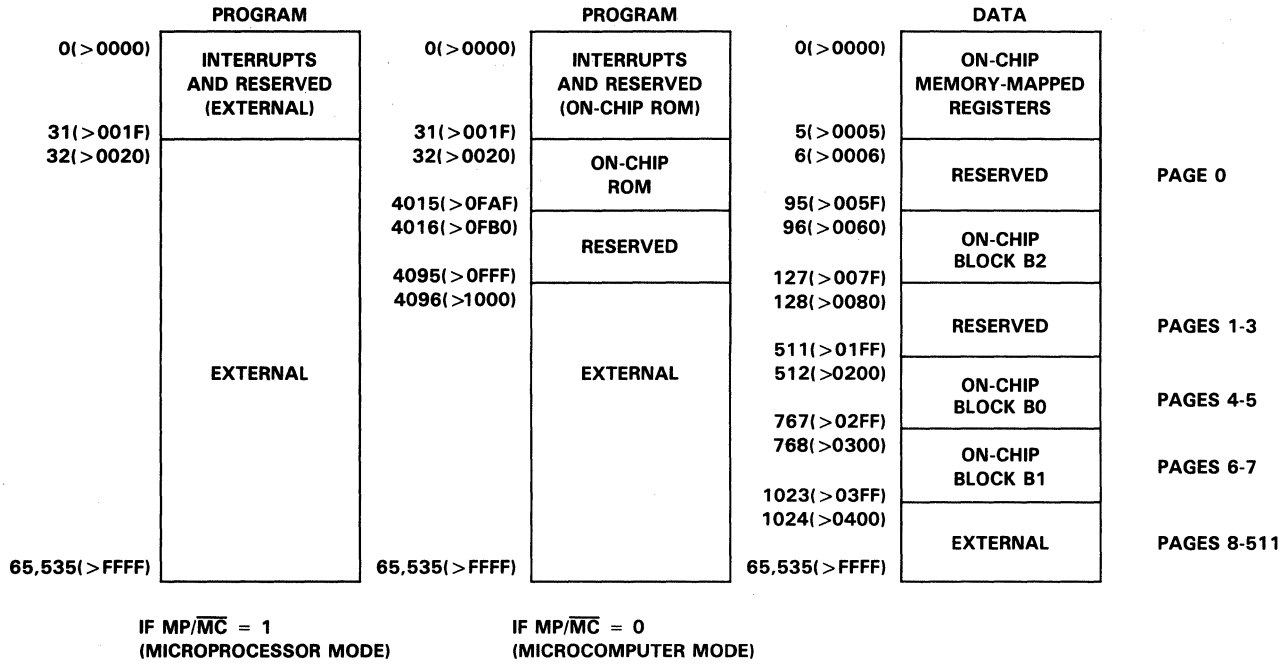
memory control

The TMS320C25 provides a total of 544 16-bit words of on-chip data RAM, divided into three separate blocks (B0, B1, and B2). Of the 544 words, 288 words (blocks B1 and B2) are always data memory, and 256 words (block B0) are programmable as either data or program memory. A data memory size of 544 words allows the TMS320C25 to handle a data array of 512 words (256 words if on-chip RAM is used for program memory), while still leaving 32 locations for intermediate storage. When using block B0 as program memory, instructions can be downloaded from external program memory into on-chip RAM and then executed.

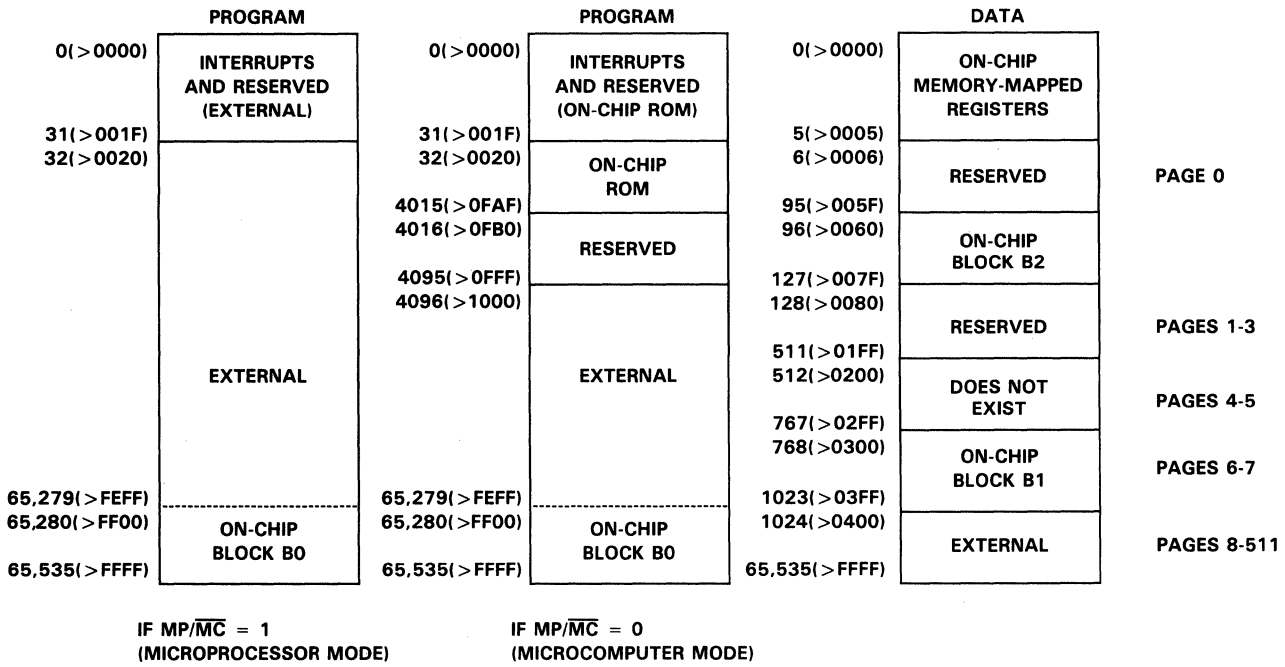
When using on-chip program RAM, ROM, or high-speed external program memory, the TMS320C25 runs at full speed without wait states. However, the READY line can be used to interface the TMS320C25 to slower, less-expensive external memory. Downloading programs from slow off-chip memory to on-chip program RAM speeds processing while cutting system costs.

The TMS320C25 provides three separate address spaces for program memory, data memory, and I/O. The on-chip memory is mapped into either the 64K-word data memory or program memory space, depending upon the memory configuration. The CNFD (configure block B0 as data memory) and CNFP (configure block B0 as program memory) instructions allow dynamic configuration of the memory maps through software. Regardless of the configuration, the user may still execute from external program memory.

The TMS320C25 has six registers that are mapped into the data memory space: a serial port data receive register, serial port data transmit register, timer register, period register, interrupt mask register, and global memory allocation register.



(a) MEMORY MAPS AFTER A CNFD INSTRUCTION



(b) MEMORY MAPS AFTER A CNFP INSTRUCTION

FIGURE 1. MEMORY MAPS

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

interrupts and subroutines

The TMS320C25 has three external maskable user interrupts $\overline{INT2}$-$\overline{INT0}$, available for external devices that interrupt the processor. Internal interrupts are generated by the serial port (RINT and XINT), by the timer (TINT), and by the software interrupt (TRAP) instruction. Interrupts are prioritized with reset ($\overline{RS}$) having the highest priority and the serial port transmit interrupt (XINT) having the lowest priority. All interrupt locations are on two-word boundaries so that branch instructions can be accommodated in those locations if desired.

A built-in mechanism protects multicycle instructions from interrupts. If an interrupt occurs during a multicycle instruction, the interrupt is not processed until the instruction is completed. This mechanism applies both to instructions that are repeated or become multicycle due to the READY signal.

external interface

The TMS320C25 supports a wide range of system interfacing requirements. Program, data, and I/O address spaces provide interface to memory and I/O, thus maximizing system throughput. I/O design is simplified by having I/O treated the same way as memory. I/O devices are mapped into the I/O address space using the processor's external address and data busses in the same manner as memory-mapped devices. Interface to memory and I/O devices of varying speeds is accomplished by using the READY line. When transactions are made with slower devices, the TMS320C25 processor waits until the other device completes its function and signals the processor via the READY line. Then, theTMS320C25 continues execution.

A serial port provides communication with serial devices, such as codecs, serial A/D converters, and other serial systems. The interface signals are compatible with codecs and many other serial devices with a minimum of external hardware. The serial port may also be used for intercommunication between processors in multiprocessing applications.

The serial port has two memory-mapped registers: the data transmit register (DXR) and the data receive register (DRR). Both registers operate in either the byte mode or 16-bit word mode, and may be accessed in the same manner as any other data memory location. Each register has an external clock, a framing synchronization pulse, and associated shift registers. One method of multiprocessing may be implemented by programming one device to transmit while the others are in the receive mode.

multiprocessing

The flexibility of the TMS320C25 allows configurations to satisfy a wide range of system requirements. The TMS320C25 can be used as follows:

- A standalone processor
- A multiprocessor with devices in parallel
- A slave/host multiprocessor with global memory space
- A peripheral processor interfaced via processor-controlled signals to another device.

For multiprocessing applications, the TMS320C25 has the capability of allocating global data memory space and communicating with that space via the $\overline{BR}$ (bus request) and READY control signals. Global memory is data memory shared by more than one processor. Global data memory access must be arbitrated. The 8-bit memory-mapped GREG (global memory allocation register) specifies part of the TMS320C25's data memory as global external memory. The contents of the register determine the size of the global memory space. If the current instruction addresses an operand within that space, $\overline{BR}$ is asserted to request control of the bus. The length of the memory cycle is controlled by the READY line.

The TMS320C25 supports DMA (direct memory access) to its external program/data memory using the $\overline{HOLD}$ and $\overline{HOLDA}$ signals. Another processor can take complete control of the TMS320C25's external memory by asserting $\overline{HOLD}$ low. This causes the TMS320C25 to place its address, data, and control lines in a high-impedance state, and assert $\overline{HOLDA}$.

instruction set

The TMS320C25 microprocessor implements a comprehensive instruction set that supports both numeric-intensive signal processing operations as well as general-purpose applications, such as multiprocessing and high-speed control. The TMS32020 source code is upward-compatible with TMS320C25 source code. TMS32020 object code runs directly on the TMS320C25.

For maximum throughput, the next instruction is prefetched while the current one is being executed. Since the same data lines are used to communicate to external data/program or I/O space, the number of cycles may vary depending upon whether the next data operand fetch is from internal or external program memory. Highest throughput is achieved by maintaining data memory on-chip and using either internal or fast external program memory.

addressing modes

The TMS320C25 instruction set provides three memory addressing modes: direct, indirect, and immediate addressing.

Both direct and indirect addressing can be used to access data memory. In direct addressing, seven bits of the instruction word are concatenated with the nine bits of the data memory page pointer to form the 16-bit data memory address. Indirect addressing accesses data memory through the eight auxiliary registers. In immediate addressing, the data is based on a portion of the instruction word(s).

In direct memory addressing, the instruction word contains the lower seven bits of the data memory address. This field is concatenated with the nine bits of the data memory page pointer to form the full 16-bit address. Thus, memory is paged in the direct addressing mode with a total of 512 pages, each page containing 128 words.

Eight auxiliary registers (AR0-AR7) provide flexible and powerful indirect addressing. To select a specific auxiliary register, the Auxiliary Register Pointer (ARP) is loaded with a value from 0 through 7 for AR0 through AR7, respectively.

There are seven types of indirect addressing: auto-increment or auto-decrement, post-indexing by either adding or subtracting the contents of AR0, single indirect addressing with no increment or decrement, and bit-reversal addressing (used in FFTs) with increment or decrement. All operations are performed on the current auxiliary register in the same cycle as the original instruction, followed by a new ARP value being loaded.

repeat feature

A repeat feature, used with instructions such as multiply/accumulates, block moves, I/O transfers, and table read/writes, allows a single instruction to be performed up to 256 times. The repeat counter (RPTC) is loaded with either a data memory value (RPT instruction) or an immediate value (RPTK instruction). The value of this operand is one less than the number of times that the next instruction is executed. Those instructions that are normally multicycle are pipelined when using the repeat feature, and effectively become single-cycle instructions.

instruction set summary

Table 1 lists the symbols and abbreviations used in Table 2, the instruction set summary. Table 2 consists primarily of single-cycle, single-word instructions. Infrequently used branch, I/O, and CALL instructions are multicycle. The instruction set summary is arranged according to function and alphabetized within each functional grouping. The symbol (†) indicates those instructions that are not included in the TMS32010 instruction set. The symbol (‡) indicates instructions that are not included in the TMS32020 instruction set.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TABLE 1. INSTRUCTION SYMBOLS

SYMBOL	MEANING
B	4-bit field specifying a bit code
CM	2-bit field specifying compare mode
D	Data memory address field
FO	Format status bit
I	Addressing mode bit
K	Immediate operand field
PA	Port address (PA0 through PA15 are predefined assembler symbols equal to 0 through 15, respectively.)
PM	2-bit field specifying P register output shift code
R	3-bit operand field specifying auxiliary register
S	4-bit left-shift code
X	3-bit accumulator left-shift field

TABLE 2. TMS320C25 INSTRUCTION SET SUMMARY

ACCUMULATOR MEMORY REFERENCE INSTRUCTIONS			
MNEMONIC	DESCRIPTION	NO. WORDS	INSTRUCTION BIT CODE
			15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ABS	Absolute value of accumulator	1	1 1 0 0 1 1 1 0 0 0 0 1 1 0 1 1
ADD	Add to accumulator with shift	1	0 0 0 0 ←S→ I ←D→
ADDC‡	Add to accumulator with carry	1	0 1 0 0 0 0 1 1 I ←D→
ADDH	Add to high accumulator	1	0 1 0 0 1 0 0 0 I ←D→
ADDK‡	Add to accumulator short immediate	1	1 1 0 0 1 1 0 0 ←K→
ADDS	Add to low accumulator with sign extension suppressed	1	0 1 0 0 1 0 0 1 I ←D→
ADDT†	Add to accumulator with shift specified by T register	1	0 1 0 0 1 0 1 0 I ←D→
ADLK†	Add to accumulator long immediate with shift	2	1 1 0 1 ←S→ 0 0 0 0 0 0 1 0
AND	AND with accumulator	1	0 1 0 0 1 1 1 0 I ←D→
ANDK†	AND immediate with accumulator with shift	2	1 1 0 1 ←S→ 0 0 0 0 0 1 0 0
CMPL†	Complement accumulator	1	1 1 0 0 1 1 1 0 0 0 1 0 0 1 1 1
LAC	Load accumulator with shift	1	0 0 1 0 ←S→ I ←D→
LACK	Load accumulator immediate short	1	1 1 0 0 1 0 1 0 ←K→
LACT†	Load accumulator with shift specified by T register	1	0 1 0 0 0 0 1 0 I ←D→
LALK†	Load accumulator long immediate with shift	2	1 1 0 1 ←S→ 0 0 0 0 0 0 0 1
NEG†	Negate accumulator	1	1 1 0 0 1 1 1 0 0 0 1 0 0 0 1 1
NORM†	Normalize contents of accumulator	1	1 1 0 0 1 1 1 0 1 ←D→
OR	OR with accumulator	1	0 1 0 0 1 1 0 1 I ←D→
ORK†	OR immediate with accumulator with shift	2	1 1 0 1 ←S→ 0 0 0 0 0 1 0 1
ROL‡	Rotate accumulator left	1	1 1 0 0 1 1 1 0 0 0 1 1 0 1 0 0
ROR‡	Rotate accumulator right	1	1 1 0 0 1 1 1 0 0 0 1 1 0 1 0 1
SACH	Store high accumulator with shift	1	0 1 1 0 1 ←X→ I ←D→
SACL	Store low accumulator with shift	1	0 1 1 0 0 ←X→ I ←D→
SBLK†	Subtract from accumulator long immediate with shift	2	1 1 0 1 ←S→ 0 0 0 0 0 0 1 1
SFL†	Shift accumulator left	1	1 1 0 0 1 1 1 0 0 0 0 1 1 0 0 0
SFR†	Shift accumulator right	1	1 1 0 0 1 1 1 0 0 0 0 1 1 0 0 1
SUB	Subtract from accumulator with shift	1	0 0 0 1 ←S→ I ←D→
SUBB‡	Subtract from accumulator with borrow	1	0 1 0 0 1 1 1 1 I ←D→
SUBC	Conditional subtract	1	0 1 0 0 0 1 1 1 I ←D→
SUBH	Subtract from high accumulator	1	0 1 0 0 0 1 0 0 I ←D→
SUBK‡	Subtract from accumulator short immediate	1	1 1 0 0 1 1 0 1 ←K→
SUBS	Subtract from low accumulator with sign extension suppressed	1	0 1 0 0 0 1 0 1 I ←D→
SUBT†	Subtract from accumulator with shift specified by T register	1	0 1 0 0 0 1 1 0 I ←D→
XOR	Exclusive-OR with accumulator	1	0 1 0 0 1 1 0 0 I ←D→
XORK†	Exclusive-OR immediate with accumulator with shift	2	1 1 0 1 ←S→ 0 0 0 0 0 1 1 0
ZAC	Zero accumulator	1	1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0
ZALH	Zero low accumulator and load high accumulator	1	0 1 0 0 0 0 0 0 I ←D→
ZALR‡	Zero low accumulator and load high accumulator with rounding	1	0 1 1 1 1 0 1 1 I ←D→
ZALS	Zero accumulator and load low accumulator with sign extension suppressed	1	0 1 0 0 0 0 0 1 I ←D→

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TABLE 2. TMS320C25 INSTRUCTION SET SUMMARY (CONTINUED)

AUXILIARY REGISTERS AND DATA PAGE POINTER INSTRUCTIONS																	
MNEMONIC	DESCRIPTION	NO. WORDS	INSTRUCTION BIT CODE														
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADRK‡	Add to auxiliary register short immediate	1	0	1	1	1	1	1	1	0	◄			K				►
CMPR†	Compare auxiliary register with auxiliary register AR0	1	1	1	0	0	1	1	1	0	0	1	0	1	0	0	◄	CM ►
LAR	Load auxiliary register	1	0	0	1	1	0	◄	R	►	I	◄			D			►
LARK	Load auxiliary register short immediate	1	1	1	0	0	0	◄	R	►	◄				K			►
LARP	Load auxiliary register pointer	1	0	1	0	1	0	1	0	1	1	0	0	0	1	◄	R	►
LDP	Load data memory page pointer	1	0	1	0	1	0	0	1	0	I	◄			D			►
LDPK	Load data memory page pointer immediate	1	1	1	0	0	1	0	0	◄				DP				►
LRLK†	Load auxiliary register long immediate	2	1	1	0	1	0	◄	R	►	0	0	0	0	0	0	0	0
MAR	Modify auxiliary register	1	0	1	0	1	0	1	0	1	I	◄			D			►
SAR	Store auxiliary register	1	0	1	1	1	0	◄	R	►	I	◄			D			►
SBRK‡	Subtract from auxiliary register short immediate	1	0	1	1	1	1	1	1	1	◄				K			►

T REGISTER, P REGISTER, AND MULTIPLY INSTRUCTIONS																	
MNEMONIC	DESCRIPTION	NO. WORDS	INSTRUCTION BIT CODE														
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
APAC	Add P register to accumulator	1	1	1	0	0	1	1	1	0	0	0	0	1	0	1	0	1
LPH†	Load high P register	1	0	1	0	1	0	0	1	1	I	◄			D			►
LT	Load T register	1	0	0	1	1	1	1	0	0	I	◄			D			►
LTA	Load T register and accumulate previous product	1	0	0	1	1	1	1	0	1	I	◄			D			►
LTD	Load T register, accumulate previous product, and move data	1	0	0	1	1	1	1	1	1	I	◄			D			►
LTP†	Load T register and store P register in accumulator	1	0	0	1	1	1	1	1	0	I	◄			D			►
LTS†	Load T register and subtract previous product	1	0	1	0	1	1	0	1	1	I	◄			D			►
MAC†	Multiply and accumulate	2	0	1	0	1	1	1	0	1	I	◄			D			►
MACD†	Multiply and accumulate with data move	2	0	1	0	1	1	1	0	0	I	◄			D			►
MPY	Multiply (with T register, store product in P register)	1	0	0	1	1	1	0	0	0	I	◄			D			►
MPYA‡	Multiply and accumulate previous product	1	0	0	1	1	1	0	1	0	I	◄			D			►
MPYK	Multiply immediate	1	1	0	1	◄									K			►
MPYS‡	Multiply and subtract previous product	1	0	0	1	1	1	0	1	1	I	◄			D			►
MPYU‡	Multiply unsigned	1	1	1	0	0	1	1	1	1	I	◄			D			►
PAC	Load accumulator with P register	1	1	1	0	0	1	1	1	0	0	0	0	1	0	1	0	0
SPAC	Subtract P register from accumulator	1	1	1	0	0	1	1	1	0	0	0	0	1	0	1	1	0
SPH‡	Store high P register	1	0	1	1	1	1	1	0	1	I	◄			D			►
SPL‡	Store low P register	1	0	1	1	1	1	1	0	0	I	◄			D			►
SPM†	Set P register output shift mode	1	1	1	0	0	1	1	1	0	0	0	0	0	1	0	◄	PM ►
SQRA†	Square and accumulate	1	0	0	1	1	1	0	0	1	I	◄			D			►
SQRS†	Square and subtract previous product	1	0	1	0	1	1	0	1	0	I	◄			D			►

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

TABLE 2. TMS320C25 INSTRUCTION SET SUMMARY (CONTINUED)

BRANCH/CALL INSTRUCTIONS

MNEMONIC	DESCRIPTION	NO. WORDS	INSTRUCTION BIT CODE 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
B	Branch unconditionally	2	1 1 1 1 1 1 1 1 1 ◄——D——►
BACC†	Branch to address specified by accumulator	1	1 1 0 0 1 1 1 0 0 0 1 0 0 1 0 1
BANZ	Branch on auxiliary register not zero	2	1 1 1 1 1 0 1 1 1 ◄——D——►
BBNZ†	Branch if TC bit ≠ 0	2	1 1 1 1 1 0 0 1 1 ◄——D——►
BBZ†	Branch if TC bit = 0	2	1 1 1 1 1 0 0 0 1 ◄——D——►
BC‡	Branch on carry	2	0 1 0 1 1 1 1 0 1 ◄——D——►
BGEZ	Branch if accumulator ≥ 0	2	1 1 1 1 0 1 0 0 1 ◄——D——►
BGZ	Branch if accumulator > 0	2	1 1 1 1 0 0 0 1 1 ◄——D——►
BIOZ	Branch on I/O status = 0	2	1 1 1 1 1 0 1 0 1 ◄——D——►
BLEZ	Branch if accumulator ≤ 0	2	1 1 1 1 0 0 1 0 1 ◄——D——►
BLZ	Branch if accumulator < 0	2	1 1 1 1 0 0 1 1 1 ◄——D——►
BNC‡	Branch on no carry	2	0 1 0 1 1 1 1 1 1 ◄——D——►
BNV†	Branch if no overflow	2	1 1 1 1 0 1 1 1 1 ◄——D——►
BNZ	Branch if accumulator ≠ 0	2	1 1 1 1 0 1 0 1 1 ◄——D——►
BV	Branch on overflow	2	1 1 1 1 0 0 0 0 1 ◄——D——►
BZ	Branch if accumulator = 0	2	1 1 1 1 0 1 1 0 1 ◄——D——►
CALA	Call subroutine indirect	1	1 1 0 0 1 1 1 0 0 0 1 0 0 1 0 0
CALL	Call subroutine	2	1 1 1 1 1 1 1 0 1 ◄——D——►
RET	Return from subroutine	1	1 1 0 0 1 1 1 0 0 0 1 0 0 1 1 0

I/O AND DATA MEMORY OPERATIONS

MNEMONIC	DESCRIPTION	NO. WORDS	INSTRUCTION BIT CODE 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
BLKD†	Block move from data memory to data memory	2	1 1 1 1 1 1 0 1 I ◄——D——►
BLKP†	Block move from program memory to data memory	2	1 1 1 1 1 1 0 0 I ◄——D——►
DMOV	Data move in data memory	1	0 1 0 1 0 1 1 0 I ◄——D——►
FORT†	Format serial port registers	1	1 1 0 0 1 1 1 0 0 0 0 0 1 1 1 FO
IN	Input data from port	1	1 0 0 0 ◄—PA—► I ◄——D——►
OUT	Output data to port	1	1 1 1 0 ◄—PA—► I ◄——D——►
RFSM‡	Reset serial port frame synchronization mode	1	1 1 0 0 1 1 1 0 0 0 1 1 0 1 1 0
RTXM†	Reset serial port transmit mode	1	1 1 0 0 1 1 1 0 0 0 1 0 0 0 0 0
RXF†	Reset external flag	1	1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 0
SFSM‡	Set serial port frame synchronization mode	1	1 1 0 0 1 1 1 0 0 0 1 1 0 1 1 1
STXM†	Set serial port transmit mode	1	1 1 0 0 1 1 1 0 0 0 1 0 0 0 0 1
SXF†	Set external flag	1	1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 1
TBLR	Table read	1	0 1 0 1 1 0 0 0 I ◄——D——►
TBLW	Table write	1	0 1 0 1 1 0 0 1 I ◄——D——►

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TABLE 2. TMS320C25 INSTRUCTION SET SUMMARY (CONCLUDED)

CONTROL INSTRUCTIONS																		
MNEMONIC	DESCRIPTION	NO. WORDS	INSTRUCTION BIT CODE															
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIT†	Test bit	1	1	0	0	1	◄	B	►		I	◄		D			►	
BITT†	Test bit specified by T register	1	0	1	0	1	0	1	1	1	I	◄		D			►	
CNFD†	Configure block as data memory	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	0	0
CNFP†	Configure block as program memory	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	0	1
DINT	Disable interrupt	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	1
EINT	Enable interrupt	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0
IDLE†	Idle until interrupt	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	1	1
LST	Load status register ST0	1	0	1	0	1	0	0	0	0	I	◄		D			►	
LST1†	Load status register ST1	1	0	1	0	1	0	0	0	1	I	◄		D			►	
NOP	No operation	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0
POP	Pop top of stack to low accumulator	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	0	1
POPD†	Pop top of stack to data memory	1	0	1	1	1	1	0	1	0	I	◄		D			►	
PSHD†	Push data memory value onto stack	1	0	1	0	1	0	1	0	0	I	◄		D			►	
PUSH	Push low accumulator onto stack	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	0	0
RC‡	Reset carry bit	1	1	1	0	0	1	1	1	0	0	0	1	1	0	0	0	0
RHM‡	Reset hold mode	1	1	1	0	0	1	1	1	0	0	0	1	1	1	0	0	0
ROVM	Reset overflow mode	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	1	0
RPT†	Repeat instruction as specified by data memory value	1	0	1	0	0	1	0	1	1	I	◄		D			►	
RPTK†	Repeat instruction as specified by immediate value	1	1	1	0	0	1	0	1	1	◄			K			►	
RSXM†	Reset sign-extension mode	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	1	0
RTC‡	Reset test/control flag	1	1	1	0	0	1	1	1	0	0	0	1	1	0	0	1	0
SC‡	Set carry bit	1	1	1	0	0	1	1	1	0	0	0	1	1	0	0	0	1
SHM‡	Set hold mode	1	1	1	0	0	1	1	1	0	0	0	1	1	1	0	0	1
SOVM	Set overflow mode	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	1	1
SST	Store status register ST0	1	0	1	1	1	1	0	0	0	I	◄		D			►	
SST1†	Store status register ST1	1	0	1	1	1	1	0	0	1	I	◄		D			►	
SSXM†	Set sign-extension mode	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	1	1
STC‡	Set test/control flag	1	1	1	0	0	1	1	1	0	0	0	1	1	0	0	1	1
TRAP†	Software interrupt	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	1	0

†These instructions are not included in the TMS32010 instruction set.
‡These instructions are not included in the TMS32020 instruction set.

development systems and software support

Texas Instruments offers concentrated development support and complete documentation for designing a TMS320C25-based microprocessor system. When developing an application, tools are provided to evaluate the performance of the processor, to develop the algorithm implementation, and to fully integrate the design's software and hardware modules. When questions arise, additional support can be obtained by calling the nearest Texas Instruments Regional Technology Center (RTC).

Sophisticated development operations are performed with the TMS320C25 Macro Assembler/Linker, Simulator, and Emulator (XDS). The macro assembler and linker are used to translate program modules into object code and link them together. This puts the program modules into a form which can be loaded into the TMS320C25 Simulator or Emulator. The simulator provides a quick means for initially debugging TMS320C25 software while the emulator provides the real-time in-circuit emulation necessary to perform system level debug efficiently.

Table 3 gives a complete list of TMS320C25 software and hardware development tools.

TABLE 3. TMS320C25 SOFTWARE AND HARDWARE SUPPORT

MACRO ASSEMBLERS/LINKERS		
Host Computer	**Operating System**	**Part Number**
DEC VAX	VMS	TMDS3242210-08
TI/IBM PC	MS/PC-DOS	TMDS3242810-02
SIMULATORS		
Host Computer	**Operating System**	**Part Number**
DEC VAX	VMS	TMDS3242211-08
TI/IBM PC	MS/PC-DOS	TMDS3242811-02
EMULATORS		
Model	**Power Supply**	**Part Number**
XDS/22	Included	TMDS3262221

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

absolute maximum ratings over specified temperature range (unless otherwise noted)†

Supply voltage range, V_{CC}‡ . . . . . −0.3 V to 7 V
Input voltage range . . . . . −0.3 V to 7 V
Output voltage range . . . . . −0.3 V to 7 V
Continuous power dissipation . . . . . 1.5 W
Operating free-air temperature range . . . . . 0°C to 70°C
Storage temperature range . . . . . −55°C to 150°C

†Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions beyond those indicated in the "Recommended Operating Conditions" section of this specification is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.
‡All voltage values are with respect to V_{SS}.

recommended operating conditions

			MIN	NOM	MAX	UNIT
V_{CC}	Supply voltage		4.5	5	5.5	V
V_{SS}	Supply voltage			0		V
V_{IH}	High-level input voltage	All inputs except CLKIN	2		V_{CC}+0.3	V
		CLKIN	2.4		V_{CC}+0.3	V
V_{IL}	Low-level input voltage	All inputs except CLKIN	−0.3		0.8	V
		CLKIN	−0.3		0.8	V
I_{OH}	High-level output current				300	μA
I_{OL}	Low-level output current				2	mA
T_A	Operating free-air temperature		0		70	°C

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_{CC} = MIN, I_{OH} = MAX	2.4	3		V
V_{OL}	Low-level output voltage	V_{CC} = MIN, I_{OL} = MAX		0.3	0.6	V
I_Z	Three-state current	V_{CC} = MAX	−20		20	μA
I_I	Input current	V_I = V_{SS} to V_{CC}	−10		10	μA
I_{CC}	Supply current	T_A = 0°C, V_{CC} = MAX, f_x = MAX			180	mA
C_I	Input capacitance			15		pF
C_O	Output capacitance			15		pF

†All typical values are at V_{CC} = 5 V, T_A = 25°C.

Caution. This device contains circuits to protect its inputs and outputs against damage due to high static voltages or electrostatic fields. These circuits have been qualified to protect this device against electrostatic discharges (ESD) of up to 2 kV according to MIL-STD-883C, Method 3015; however, it is advised that precautions be taken to avoid application of any voltage higher than maximum rated voltages to these high-impedance circuits. During storage or handling, the device leads should be shorted together or the device should be placed in conductive foam. In a circuit, unused inputs should always be connected to an appropriate logic voltage level, preferrably either V_{CC} or ground. Specific guidelines for handling devices of this type are contained in the publication "Guidelines for Handling Electrostatic-Discharge Sensitive (ESDS) Devices and Assemblies" available from Texas Instruments.

CLOCK CHARACTERISTICS AND TIMING

The TMS320C25 can use either its internal oscillator or an external frequency source for a clock.

internal clock option

The internal oscillator is enabled by connecting a crystal across X1 and X2/CLKIN (see Figure 2). The frequency of CLKOUT1 is one-fourth the crystal fundamental frequency. The crystal should be fundamental mode, and parallel resonant, with an effective series resistance of 30 ohms, a power dissipation of 1 mW, and be specified at a load capacitance of 20 pF.

PARAMETER		TEST CONDITIONS	MIN	TYP	MAX	UNIT
f_x	Input clock frequency	T_A = 0°C to 70°C	6.7		40	MHz
f_{sx}	Serial port frequency	T_A = 0°C to 70°C	0		5,000	kHz
C1, C2		T_A = 0°C to 70°C		10		pF

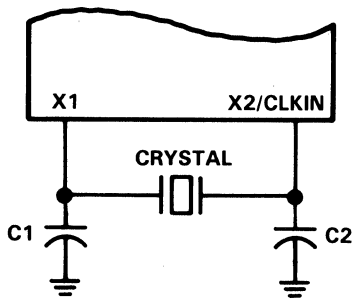


FIGURE 2. INTERNAL CLOCK OPTION

external clock option

An external frequency source can be used by injecting the frequency directly into X2/CLKIN with X1 left unconnected. The external frequency injected must conform to the specifications listed in the following table.

switching characteristics over recommended operating conditions (see Note 1)

PARAMETER		MIN	TYP	MAX	UNIT
$t_{c(C)}$	CLKOUT1/CLKOUT2 cycle time	100		597	ns
$t_{d(CIH-C)}$	CLKIN high to CLKOUT1/CLKOUT2/$\overline{STRB}$ high/low	12		25	ns
$t_{f(C)}$	CLKOUT1/CLKOUT2/$\overline{STRB}$ fall time			5	ns
$t_{r(C)}$	CLKOUT1/CLKOUT2/$\overline{STRB}$ rise time			5	ns
$t_{w(CL)}$	CLKOUT1/CLKOUT2 low pulse duration	2Q−8	2Q	2Q+8	ns
$t_{w(CH)}$	CLKOUT1/CLKOUT2 high pulse duration	2Q−8	2Q	2Q+8	ns
$t_{d(C1-C2)}$	CLKOUT1 high to CLKOUT2 low, CLKOUT2 high to CLKOUT1 high, etc.	Q−5	Q	Q+5	ns

NOTE 1: Q = $1/4t_{c(C)}$.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

timing requirements over recommended operating conditions (see Note 1)

		MIN	NOM	MAX	UNIT
$t_{c(CI)}$	CLKIN cycle time	25		150	ns
$t_{f(CI)}$	CLKIN fall time			5	ns
$t_{r(CI)}$	CLKIN rise time			5	ns
$t_{w(CIL)}$	CLKIN low pulse duration, $t_{c(CI)}$ = 50 ns (Note 2)	5		20	ns
$t_{w(CIH)}$	CLKIN high pulse duration, $t_{c(CI)}$ = 50 ns (Note 2)	5		20	ns
$t_{su(S)}$	$\overline{SYNC}$ setup time before CKLIN low	5		Q−5	ns
$t_{h(S)}$	$\overline{SYNC}$ hold time from CLKIN low	8			ns

NOTES: 1. Q = $1/4t_{c(C)}$.
2. CLKIN duty cycle $[t_{r(CI)} + t_{w(CIH)}]/t_{c(CI)}$ must be within 40-60%.

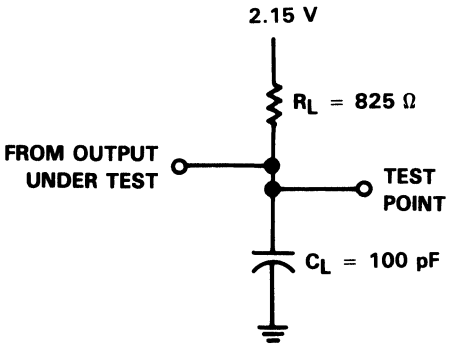
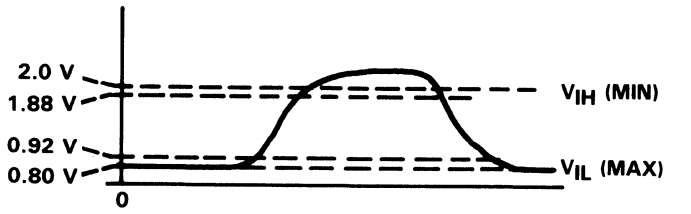
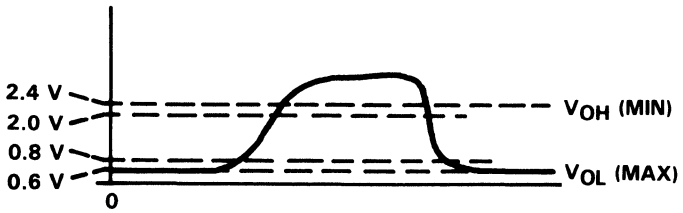


FIGURE 3. TEST LOAD CIRCUIT



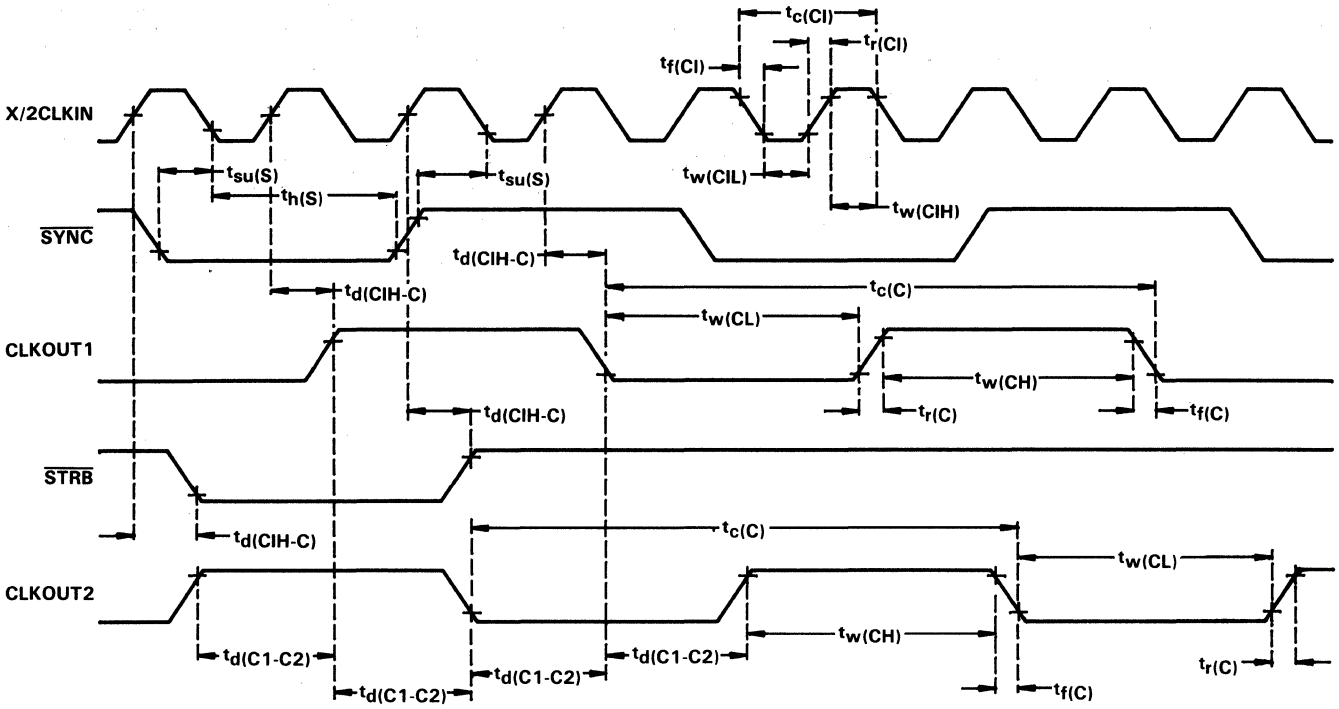
(a) INPUT



(b) OUTPUTS

FIGURE 4. VOLTAGE REFERENCE LEVELS

clock timing



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

MEMORY AND PERIPHERAL INTERFACE TIMING

switching characteristics over recommended operating conditions (see Note 1)

	PARAMETER	MIN	TYP	MAX	UNIT
$t_{d(C1-S)}$	$\overline{STRB}$ from CLKOUT1 (if $\overline{STRB}$ is present)	Q−8	Q	Q+8	ns
$t_{d(C2-S)}$	CLKOUT2 to $\overline{STRB}$ (if $\overline{STRB}$ is present)	−8	0	8	ns
$t_{su(A)}$	Address setup time before $\overline{STRB}$ low (Note 3)	Q−15			ns
$t_{h(A)}$	Address hold time after $\overline{STRB}$ high (Note 3)	Q−8			ns
$t_{w(SL)}$	$\overline{STRB}$ low pulse duration (no wait states, Note 4)		2Q		ns
$t_{w(SH)}$	$\overline{STRB}$ high pulse duration (between consecutive cycles, Note 4)		2Q		ns
$t_{su(D)W}$	Data write setup time before $\overline{STRB}$ high (no wait states)	2Q−22			ns
$t_{h(D)W}$	Data write hold time from $\overline{STRB}$ high	Q−8	Q		ns
$t_{en(D)}$	Data bus starts being driven after $\overline{STRB}$ low (write cycle)	0			ns
$t_{dis(D)}$	Data bus three-state after $\overline{STRB}$ high (write cycle)		Q	Q+15	ns
$t_{d(MSC)}$	$\overline{MSC}$ valid from CLKOUT1	−12	0	12	ns

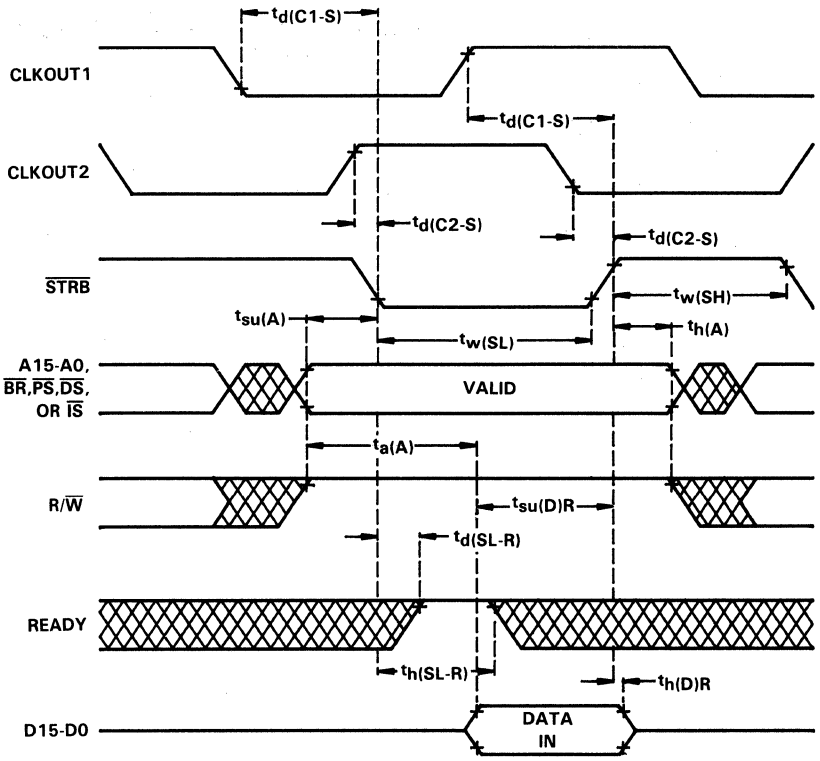
NOTES: 1. Q = $1/4t_{c(C)}$.
3. A15-A0, $\overline{PS}$, $\overline{DS}$, $\overline{IS}$, R/$\overline{W}$, and $\overline{BR}$ timings are all included in timings referenced as "address."
4. Delays between CLKOUT1/CLKOUT2 edges and $\overline{STRB}$ edges track each other, resulting in $t_{w(SL)}$ and $t_{w(SH)}$ being 2Q with no wait states.

timing requirements over recommended operating conditions (see Note 1)

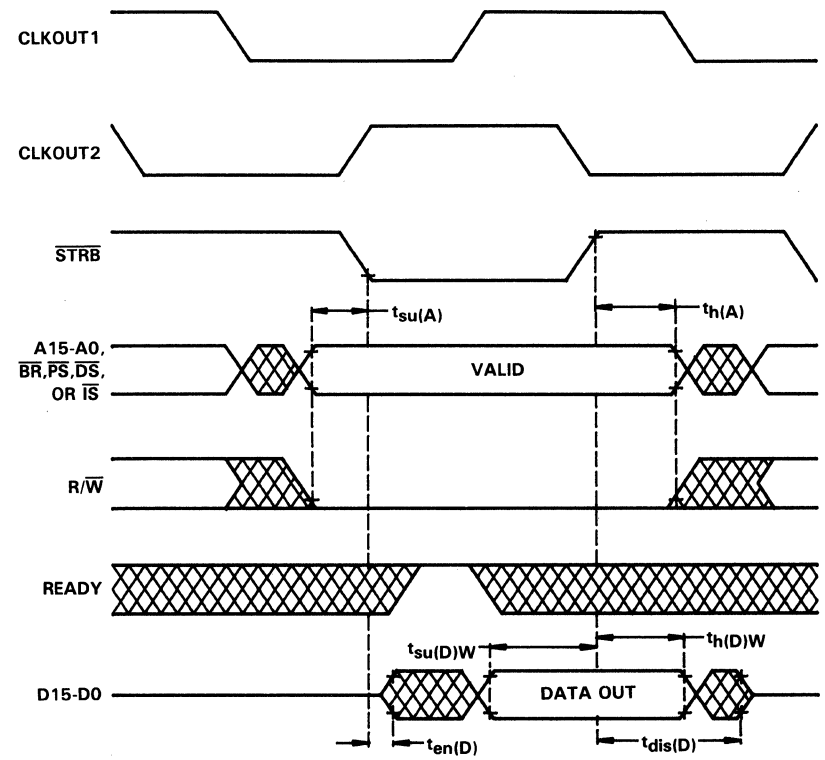
		MIN	NOM	MAX	UNIT
$t_{a(A)}$	Read data access time from address time (read cycle, Notes 3 and 5)			3Q−35	ns
$t_{su(D)R}$	Data read setup time before $\overline{STRB}$ high	20			ns
$t_{h(D)R}$	Data read hold time from $\overline{STRB}$ high	0			ns
$t_{d(SL-R)}$	READY valid after $\overline{STRB}$ low (no wait states)			Q−20	ns
$t_{d(C2H-R)}$	READY valid after CLKOUT2 high			Q−20	ns
$t_{h(SL-R)}$	READY hold time after $\overline{STRB}$ low (no wait states)	Q−2			ns
$t_{h(C2H-R)}$	READY hold after CLKOUT2 high	Q−2			ns
$t_{d(M-R)}$	READY valid after $\overline{MSC}$ valid			2Q−25	ns
$t_{h(M-R)}$	READY hold time after $\overline{MSC}$ valid	0			ns

NOTES: 1. Q = $1/4t_{c(C)}$.
3. A15-A0, $\overline{PS}$, $\overline{DS}$, $\overline{IS}$, R/$\overline{W}$, and $\overline{BR}$ timings are all included in timings referenced as "address."
5. Read data access time is defined as $t_{a(A)} = t_{su(A)} + t_{w(SL)} - t_{su(D)R}$.

memory read timing

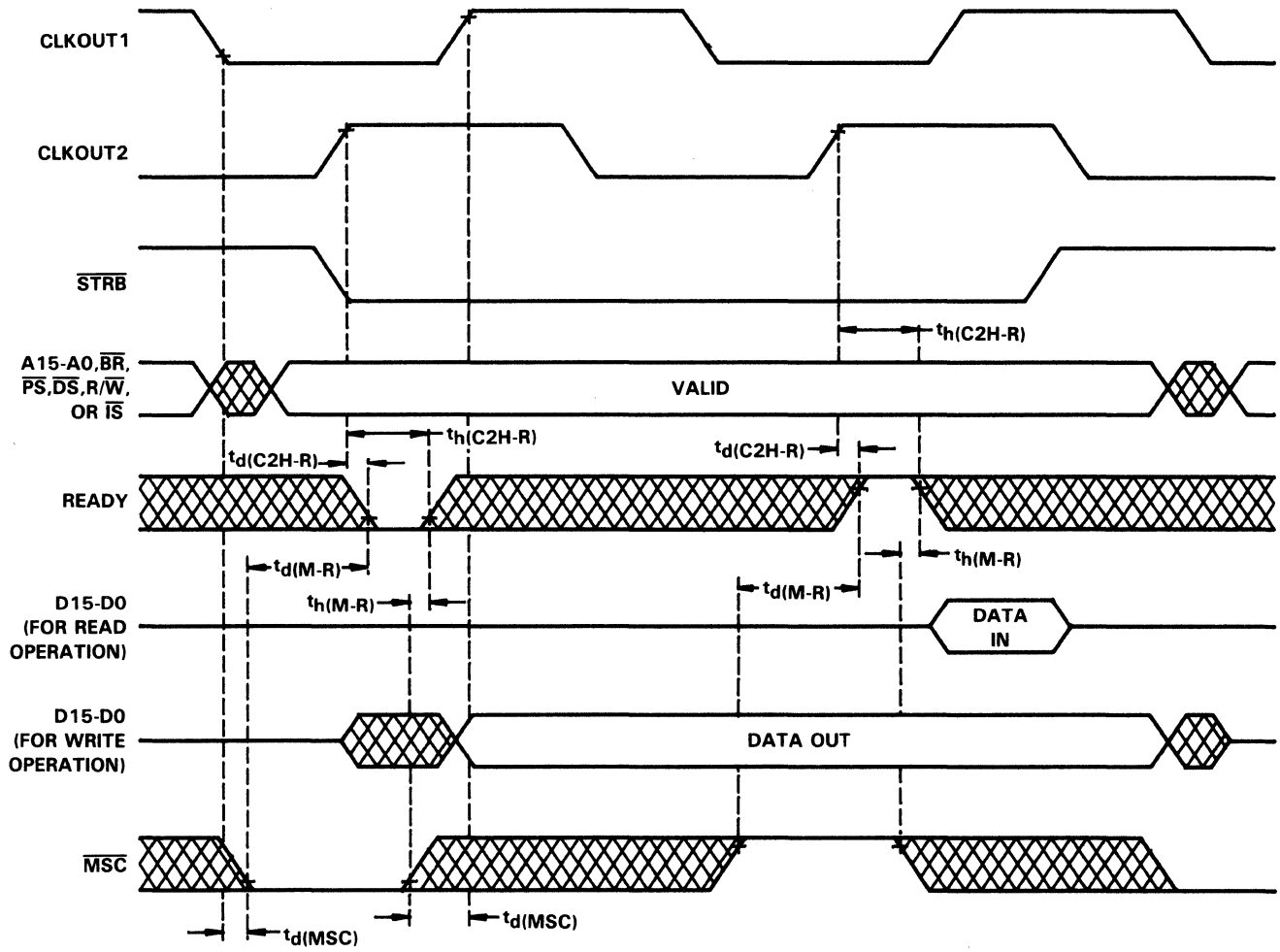


memory write timing



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

one wait-state memory access timing



$\overline{RS}$, $\overline{INT}$, $\overline{BIO}$, and XF TIMING

switching characteristics over recommended operating conditions (see Note 1)

	PARAMETER	MIN	TYP	MAX	UNIT
$t_{d(RS)}$	CLKOUT1 low to reset state entered			22	ns
$t_{d(IACK)}$	CLKOUT1 to $\overline{IACK}$ valid	−12	0	12	ns
$t_{d(XF)}$	XF valid before falling edge of $\overline{STRB}$	Q−15			ns

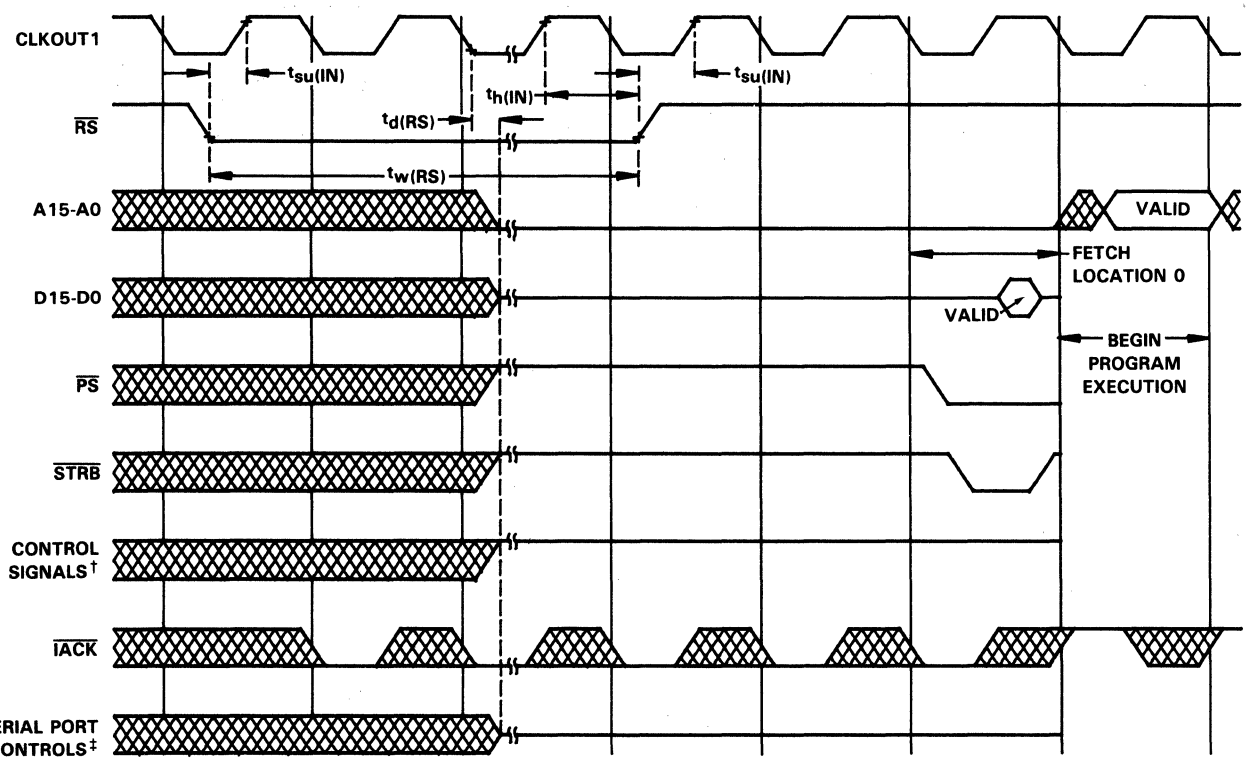
NOTES: 1. Q = $1/4t_{c(C)}$.
6. $\overline{RS}$, $\overline{INT}$, and $\overline{BIO}$ are asynchronous inputs and can occur at any time during a clock cycle. However, if the specified setup time is met, the exact sequence shown in the timing diagrams will occur.

timing requirements over recommended operating conditions (see Note 1)

		MIN	NOM	MAX	UNIT
$t_{su(IN)}$	$\overline{INT}/\overline{BIO}/\overline{RS}$ setup before CLKOUT1 high	25			ns
$t_{h(IN)}$	$\overline{INT}/\overline{BIO}/\overline{RS}$ hold after CLKOUT1 high	0			ns
$t_{f(IN)}$	$\overline{INT}/\overline{BIO}$ fall time			8	ns
$t_{w(IN)}$	$\overline{INT}/\overline{BIO}$ low pulse duration	$t_{c(C)}$			ns
$t_{w(RS)}$	$\overline{RS}$ low pulse duration	$3t_{c(C)}$			ns

NOTES: 1. Q = $1/4t_{c(C)}$.
6. $\overline{RS}$, $\overline{INT}$, and $\overline{BIO}$ are asynchronous inputs and can occur at any time during a clock cycle. However, if the specified setup time is met, the exact sequence shown in the timing diagrams will occur.

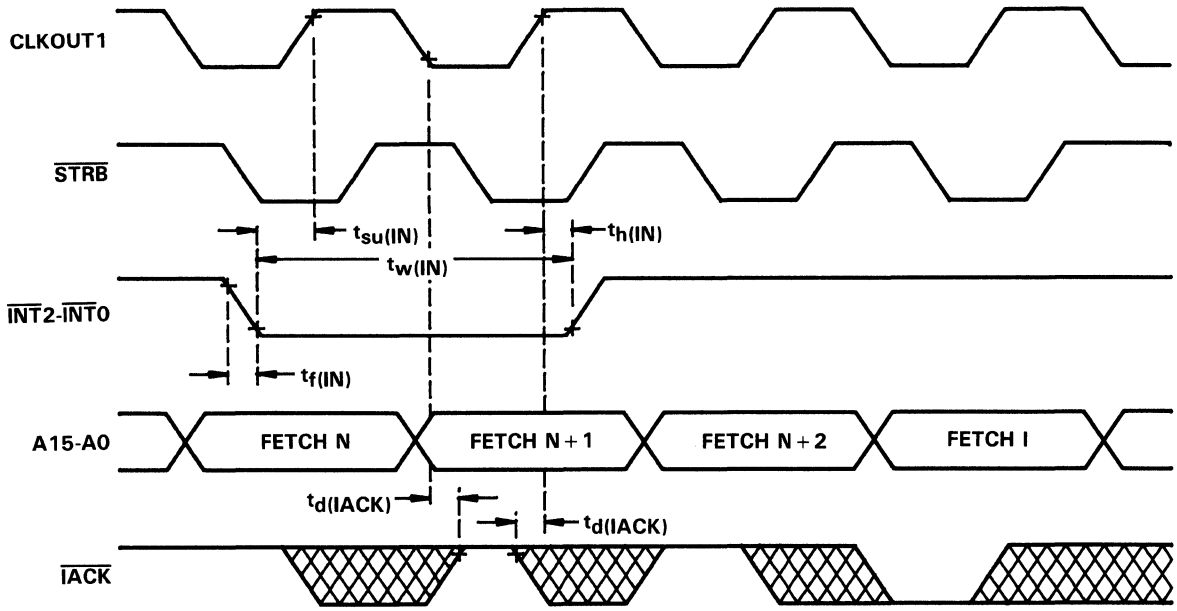
reset timing



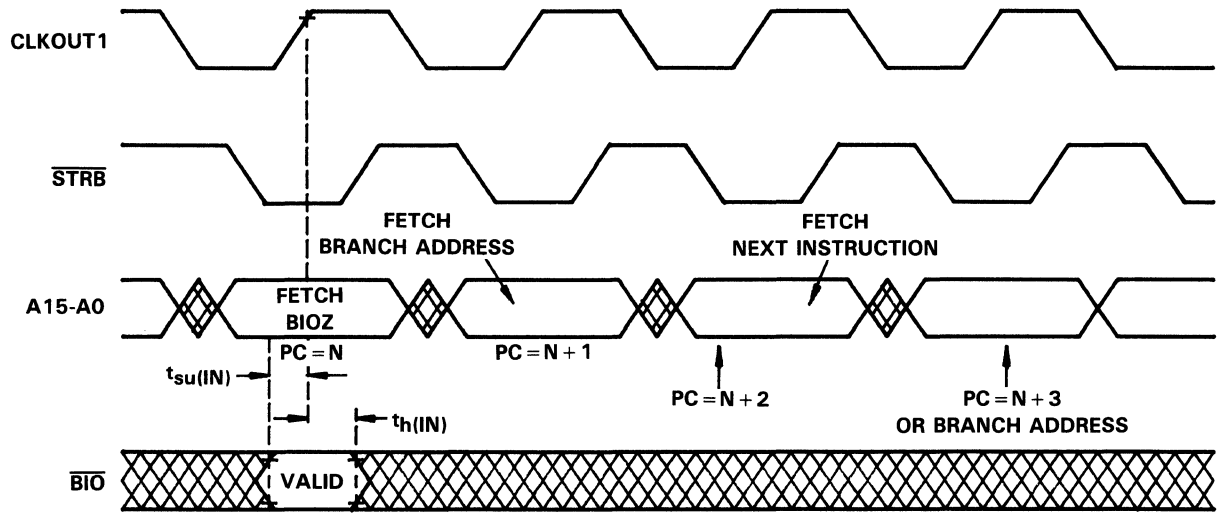
†Control signals are $\overline{DS}$, $\overline{IS}$, R/$\overline{W}$ and XF.
‡Serial port controls are $\overline{DX}$ and FSX.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

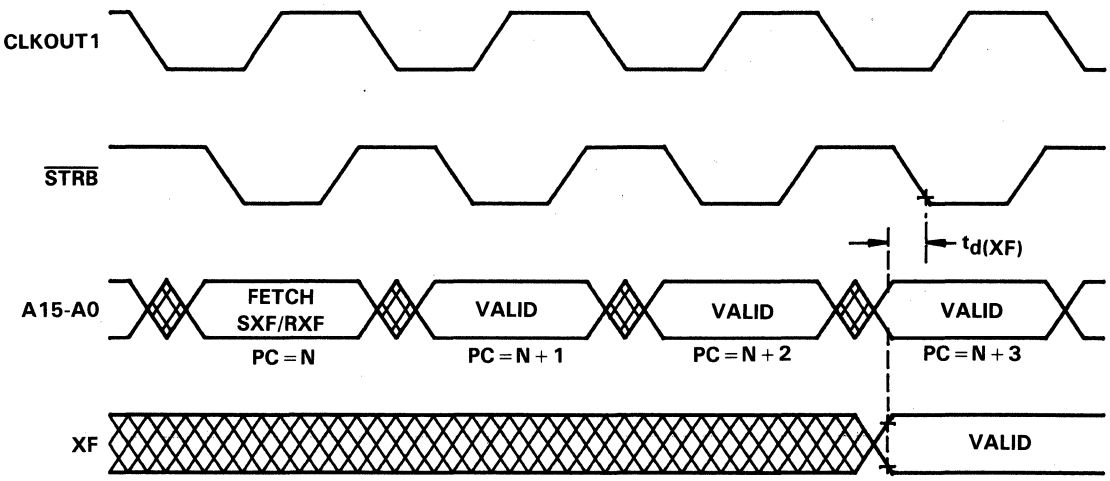
interrupt timing



BIO timing



external flag timing



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

$\overline{\text{HOLD}}$ TIMING

switching characteristics over recommended operating conditions (see Note 1)

	PARAMETER	MIN	TYP	MAX	UNIT
$t_{d(C1L-AL)}$	$\overline{\text{HOLDA}}$ low after CLKOUT1 low	−12		12	ns
$t_{dis(AL-A)}$	$\overline{\text{HOLDA}}$ low to address three-state		15		ns
$t_{dis(C1L-A)}$	Address three-state after CLKOUT1 low ($\overline{\text{HOLD}}$ mode, Note 3)			30	ns
$t_{d(HH-AH)}$	$\overline{\text{HOLD}}$ high to $\overline{\text{HOLDA}}$ high			25	ns
$t_{en(A-C1L)}$	Address driven before CLKOUT1 low ($\overline{\text{HOLD}}$ mode, Note 3)			5	ns

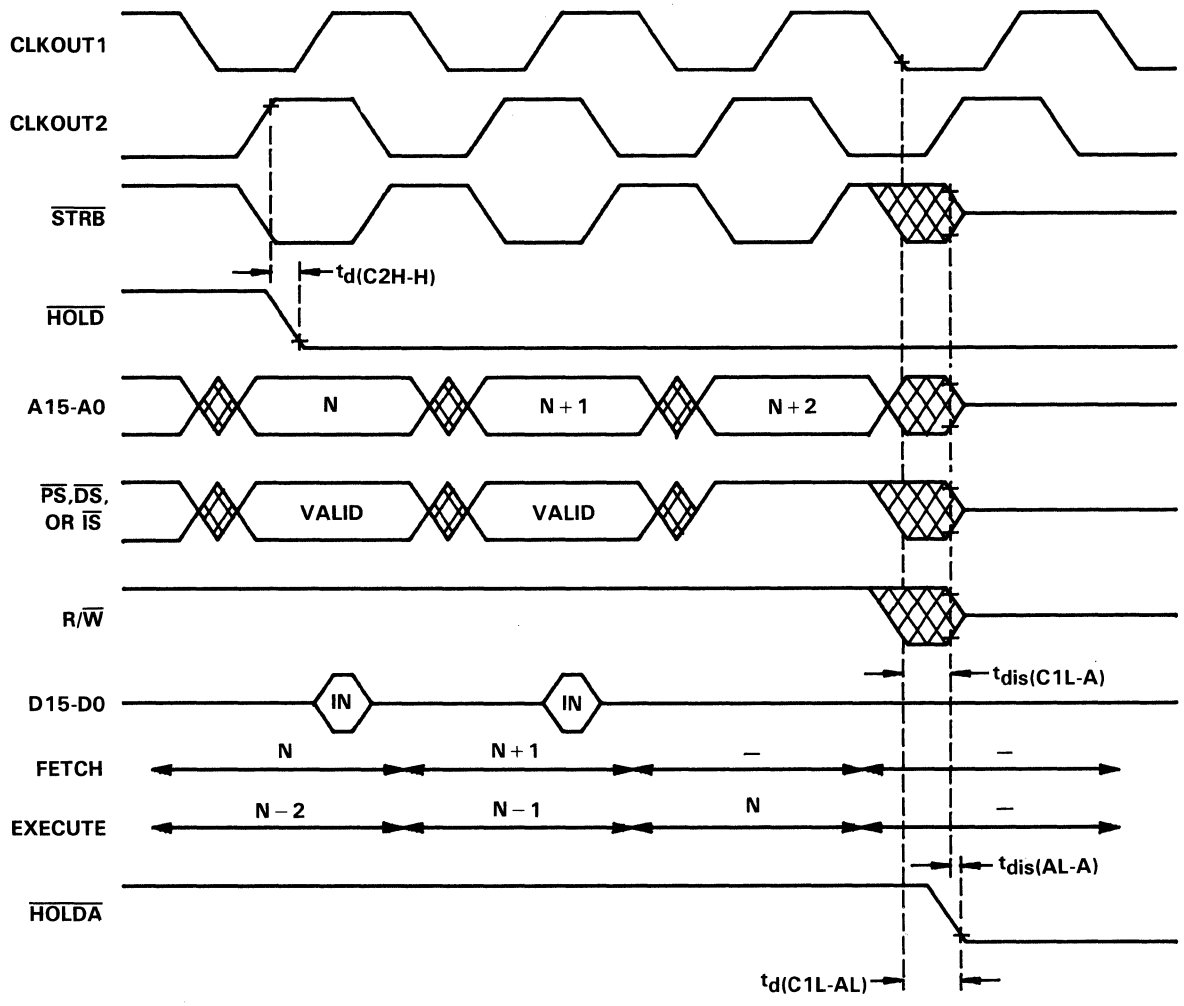
NOTES: 1. $Q = 1/4t_{c(C)}$.
3. A15-A0, $\overline{\text{PS}}$, $\overline{\text{DS}}$, $\overline{\text{IS}}$, R/$\overline{\text{W}}$, and $\overline{\text{BR}}$ timings are all included in timings referenced as "address."

timing requirements over recommended operating conditions (see Note 1)

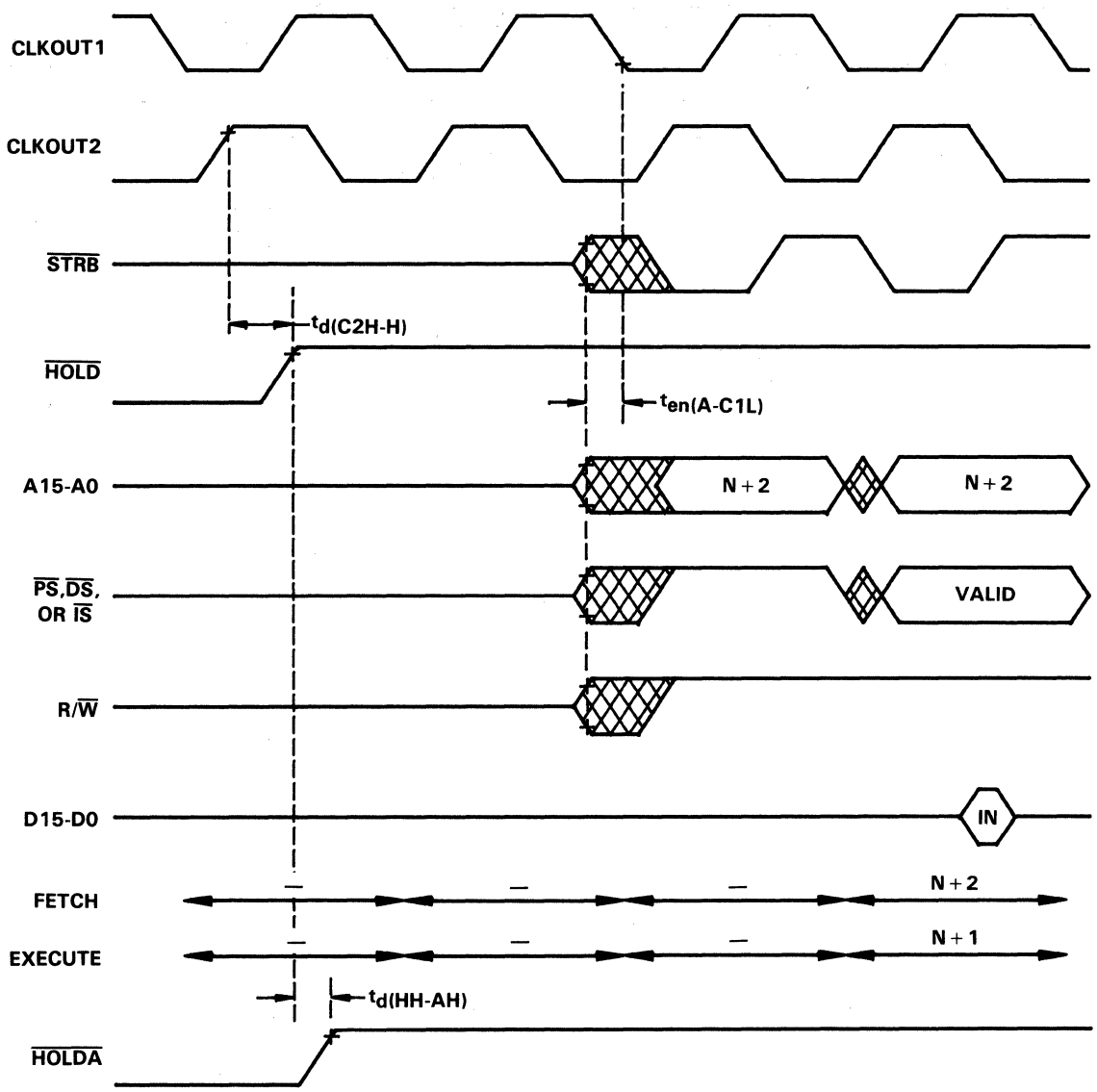
		MIN	NOM	MAX	UNIT
$t_{d(C2H-H)}$	$\overline{\text{HOLD}}$ valid after CLKOUT2 high			Q − 20	ns

NOTE: 1. $Q = 1/4t_{c(C)}$.

$\overline{\text{HOLD}}$ timing (part A)



$\overline{\text{HOLD}}$ timing (part B)



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

SERIAL PORT TIMING

switching characteristics over recommended operating conditions (see Note 1)

	PARAMETER	MIN	TYP	MAX	UNIT
$t_{d(CH-DX)}$	DX valid after CLKX rising edge (Note 7)			50	ns
$t_{d(FL-DX)}$	DX valid after FSX falling edge (TXM = 0, Note 7)			25	ns
$t_{d(CH-FS)}$	FSX valid after CLKX rising edge (TXM = 1)			30	ns

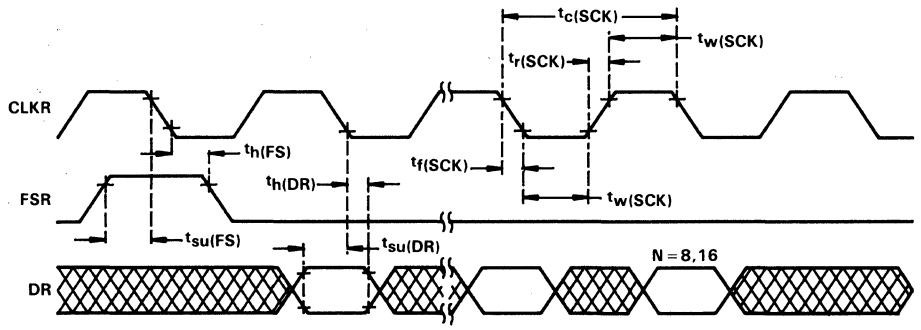
NOTES: 1. Q = $1/4t_{c(C)}$.
7. The last occurrence of FSX falling and CLKX rising.

timing requirements over recommended operating conditions (see Note 1)

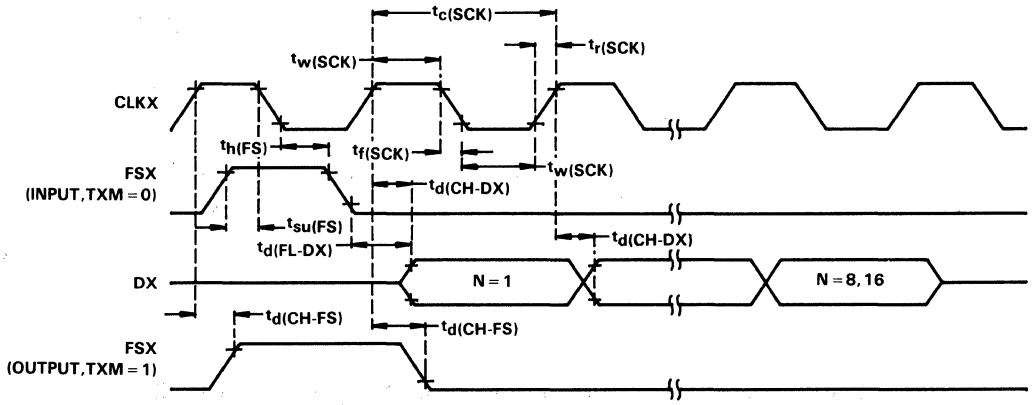
		MIN	NOM	MAX	UNIT
$t_{c(SCK)}$	Serial port clock (CLKX/CLKR) cycle time	200			ns
$t_{f(SCK)}$	Serial port clock (CLKX/CLKR) fall time			25	ns
$t_{r(SCK)}$	Serial port clock (CLKX/CLKR) rise time			25	ns
$t_{w(SCK)}$	Serial port clock (CLKX/CLKR) low pulse duration (see Note 8)	80			ns
$t_{w(SCK)}$	Serial port clock (CLKX/CLKR) high pulse duration (see Note 8)	80			ns
$t_{su(FS)}$	FSX/FSR setup time before (CLKX/CLKR) falling edge (TXM = 0)	10			ns
$t_{h(FS)}$	FSX/FSR hold time after (CLKX/CLKR) falling edge (TXM = 0)	10			ns
$t_{su(DR)}$	DR setup time before CLKR falling edge	10			ns
$t_{h(DR)}$	DR hold time after CLKR falling edge	10			ns

NOTES: 1. Q = $1/4t_{c(C)}$.
8. The duty cycle of the serial port clock must be within 40-60%.

serial port receive timing



serial port transmit timing

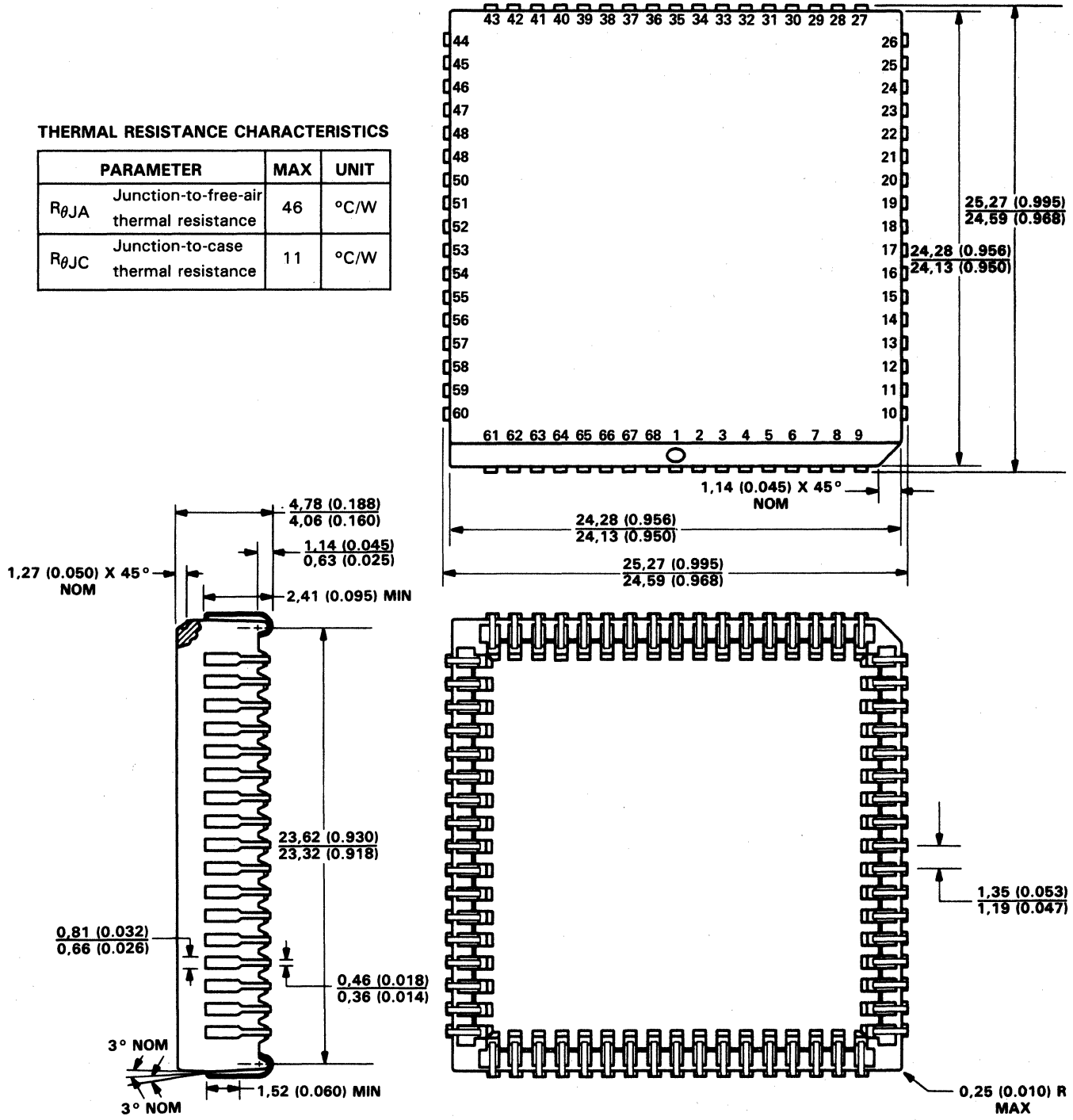


MECHANICAL DATA

68-pin plastic leaded chip carrier package

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



ALL LINEAR DIMENSIONS ARE IN MILLIMETERS AND PARENTHETICALLY IN INCHES

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TMS32020
DIGITAL SIGNAL PROCESSOR

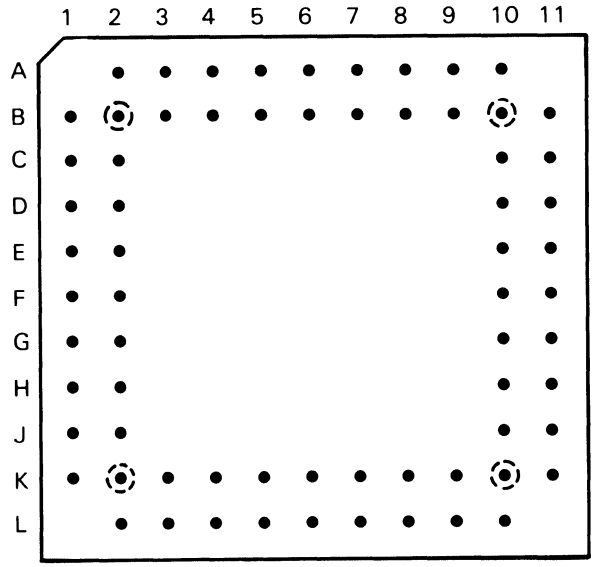
MARCH 1985—REVISED MAY 1986

- 200-ns Instruction Cycle Time
- 544 Words of Programmable On-Chip Data RAM
- 128K Words of Data/Program Space
- Sixteen Input and Sixteen Output Channels
- 16-Bit Parallel Interface
- Directly Accessible External Data Memory Space
- Global Data Memory Interface
- 16-Bit Instruction and Data Words
- 32-Bit ALU and Accumulator
- Single-Cycle Multiply/Accumulate Instructions
- 0 to 16-Bit Scaling Shifter
- Bit Manipulation and Logical Instructions
- Instruction Set Support for Floating-Point Operations
- Block Moves for Data/Program Management
- Repeat Instructions for Efficient Use of Program Space
- Five Auxiliary Registers and Dedicated Arithmetic Unit for Indirect Addressing
- Serial Port for Direct Codec Interface
- Synchronization Input for Synchronous Multiprocessor Configurations
- Wait States for Communication to Slow Off-Chip Memories/Peripherals
- On-Chip Timer for Control Operations
- Three External Maskable User Interrupts
- Input Pin Polled by Software Branch Instruction
- Programmable Output Pin for Signalling External Devices
- 2.4-Micron NMOS Technology
- Single 5-V Supply
- On-Chip Clock Generator

PIN ASSIGNMENTS

PIN	FUNCTION	PIN	FUNCTION	PIN	FUNCTION
A2	D8	C11	CLKOUT1	J10	$\overline{PS}$
A3	D10	D1	D4	J11	$\overline{IS}$
A4	D12	D2	D3	K1	A0
A5	D14	D10	CLKOUT2	K2	A1
A6	V_{CC}	D11	XF	K3	A3
A7	$\overline{HOLD}$	E1	D2	K4	A5
A8	$\overline{RS}$	E2	D1	K5	A7
A9	CLKX	E10	$\overline{HOLDA}$	K6	A8
A10	V_{CC}	E11	DX	K7	A10
B1	V_{SS}	F1	D0	K8	A12
B2	D7	F2	$\overline{SYNC}$	K9	A14
B3	D9	F10	FSX	K10	$\overline{DS}$
B4	D11	F11	X2/CLKIN	K11	V_{SS}
B5	D13	G1	$\overline{INT0}$	L2	V_{SS}
B6	D15	G2	$\overline{INT1}$	L3	A2
B7	$\overline{BIO}$	G10	X1	L4	A4
B8	READY	G11	$\overline{BR}$	L5	A6
B9	CLKR	H1	$\overline{INT2}$	L6	V_{CC}
B10	V_{CC}	H2	V_{CC}	L7	A9
B11	$\overline{IACK}$	H10	$\overline{STRB}$	L8	A11
C1	D6	H11	R/$\overline{W}$	L9	A13
C2	D5	J1	DR	L10	A15
C10	$\overline{MSC}$	J2	FSR		

68-PIN GB
PIN GRID ARRAY CERAMIC PACKAGE†
(TOP VIEW)



† See Pin Assignments Table (Page 1) and Pin Nomenclature Table (Page 2) for location and description of all pins.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

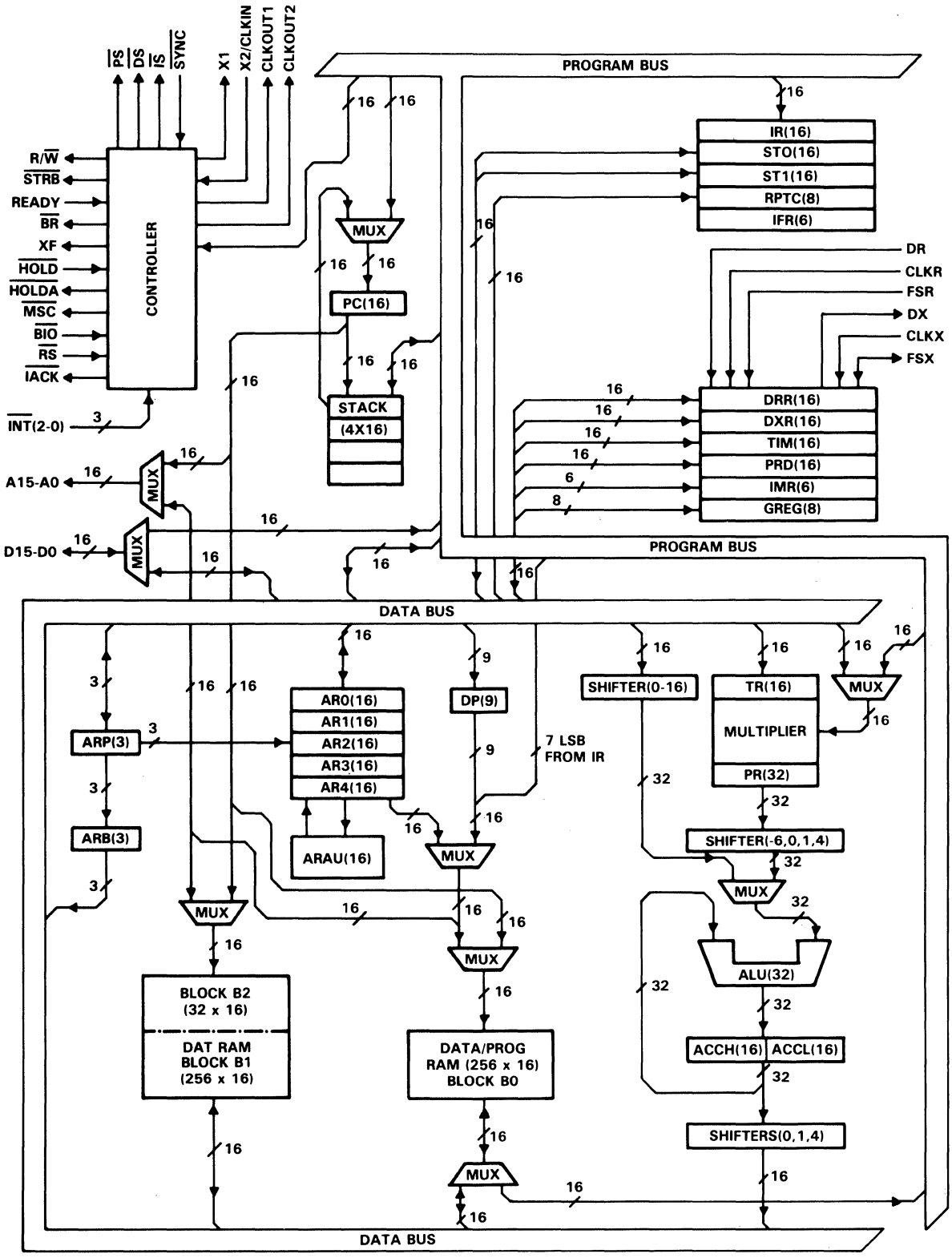
PIN NOMENCLATURE

NAME	I/O/Z†	DEFINITION
V_{CC}	I	5-V supply pins
V_{SS}	I	Ground pins
X1	O	Output from internal oscillator for crystal
X2/CLKIN	I	Input to internal oscillator from crystal or external clock
CLKOUT1	O	Master clock output (crystal or CLKIN frequency/4)
CLKOUT2	O	A second clock output signal
D15-D0	I/O/Z	16-bit data bus D15 (MSB) through D0 (LSB). Multiplexed between program, data, and I/O spaces.
A15-A0	O/Z	16-bit address bus A15 (MSB) through A0 (LSB)
$\overline{PS}$,$\overline{DS}$,$\overline{IS}$	O/Z	Program, data, and I/O space select signals
R/$\overline{W}$	O/Z	Read/write signal
$\overline{STRB}$	O/Z	Strobe signal
$\overline{RS}$	I	Reset input
$\overline{INT2}$-$\overline{INT0}$	I	External user interrupt inputs
$\overline{MSC}$	O	Microstate complete signal
$\overline{IACK}$	O	Interrupt acknowledge signal
READY	I	Data ready input. Asserted by external logic when using slower devices to indicate that the current bus transaction is complete.
$\overline{BR}$	O	Bus request signal. Asserted when the TMS32020 requires access to an external global data memory space.
XF	O	External flag output (latched software-programmable signal).
$\overline{HOLD}$	I	Hold input. When asserted, TMS32020 goes into an idle mode and puts the data, address, and control lines in the high-impedance state.
$\overline{HOLDA}$	O	Hold acknowledge signal
$\overline{SYNC}$	I	Clock synchronization input
$\overline{BIO}$	I	Branch control input. Polled by BIOZ instruction.
DR	I	Serial data receive input
CLKR	I	Clock for receive input for serial port
FSR	I	Frame synchronization pulse for receive input
DX	O/Z	Serial data transmit output
CLKX	I	Clock for transmit output for serial port
FSX	I/O/Z	Frame synchronization pulse for transmit. Configurable as either an input or an output.

†I/O/Z = Input/Output/High-impedance state.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

functional block diagram



description

The TMS32020 Digital Signal Processor is the second member of the TMS320 family of VLSI digital signal processors and peripherals. The TMS320 family supports a wide range of digital signal processing applications, such as telecommunications, modems, image processing, speech processing, spectrum analysis, audio processing, digital filtering, high-speed control, graphics, and other computation-intensive applications.

With a 200-ns instruction cycle time and an innovative memory configuration, the TMS32020 performs operations necessary for many real-time digital signal processing algorithms. Since most instructions require only one cycle, the TMS32020 is capable of executing five million instructions per second. On-chip data RAM of 544 16-bit words, direct addressing of up to 64K words of external data memory space and 64K words of external program memory space, and multiprocessor interface features for sharing global memory minimize unnecessary data transfers to take full advantage of the capabilities of the processor.

architecture

The TMS32020 architecture is based upon that of the TMS32010, the first member of the TMS320 family. The TMS32020 increases performance of DSP algorithms through innovative additions to the TMS320 family architecture. TMS32010 source code is upward-compatible with TMS32020 source code and can be assembled using the TMS32020 Macro Assembler.

Increased throughput on the TMS32020 for many DSP applications is accomplished by means of single-cycle multiply/accumulate instructions with a data move option, five auxiliary registers with a dedicated arithmetic unit, and faster I/O necessary for data-intensive signal processing.

The architectural design of the TMS32020 emphasizes overall speed, communication, and flexibility in processor configuration. Control signals and instructions provide floating-point support, block-memory transfers, communication to slower off-chip devices, and multiprocessing implementations.

Two large on-chip RAM blocks, configurable either as separate program and data spaces or as two contiguous data blocks, provide increased flexibility in system design. Maintaining program memory off-chip allows large address spaces from which large programs of up to 64K words can operate at full speed. Programs can also be downloaded from slow external memory to high-speed on-chip RAM. A total of 64K data memory address space is included to facilitate implementation of DSP algorithms. The VLSI implementation of the TMS32020 incorporates all of these features as well as many others, such as a hardware timer, serial port, and block data transfer capabilities.

32-bit ALU/accumulator

The TMS32020 32-bit Arithmetic Logic Unit (ALU) and accumulator perform a wide range of arithmetic and logical instructions, the majority of which execute in a single clock cycle. The ALU executes a variety of branch instructions dependent on the status of the ALU or a single bit in a word. These instructions provide the following capabilities:

- Branch to an address specified by the accumulator
- Normalize fixed-point numbers contained in the accumulator
- Test a specified bit of a word in data memory.

One input to the ALU is always provided from the accumulator, and the other input may be provided from the Product Register (PR) of the multiplier or the input scaling shifter which has fetched data from the RAM on the data bus. After the ALU has performed the arithmetic or logical operations, the result is stored in the accumulator.

The 32-bit accumulator is split into two 16-bit segments for storage in data memory. Additional shifters at the output of the accumulator perform shifts while the data is being transferred to the data bus for storage. The contents of the accumulator remain unchanged.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

scaling shifter

The TMS32020 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. The LSBs of the output are filled with zeroes, and the MSBs may be either filled with zeroes or sign-extended, depending upon the status programmed into the SXM (sign-extension mode) bit of status register ST0.

16 x 16-bit parallel multiplier

The TMS32020 has a two's complement 16 x 16-bit hardware multiplier, which is capable of computing a 32-bit product in a single machine cycle. The multiplier has the following two associated registers:

- A 16-bit Temporary Register (TR) that holds one of the operands for the multiplier, and
- A 32-bit Product Register (PR) that holds the product.

Incorporated into the TMS32020 instruction set are single-cycle multiply/accumulate instructions that allow both operands to be processed simultaneously. The data for these operations resides in the on-chip RAM blocks and can be transferred to the multiplier each cycle via the program and data buses.

Four product shift modes are available at the Product Register (PR) output that are useful when performing multiply/accumulate operations, fractional arithmetic, or justifying fractional products.

timer

The TMS32020 provides a memory-mapped 16-bit timer for control operations. The on-chip timer (TIM) register is a down counter that is continuously clocked by an internal clock. This clock is derived by dividing the CLKOUT1 frequency by 4. A timer interrupt (TINT) is generated every time the timer decrements to zero. The timer is reloaded with the value contained in the period (PRD) register within the same cycle that it reaches zero so that interrupts may be programmed to occur at regular intervals of 4 × (PRD) cycles of CLKOUT1.

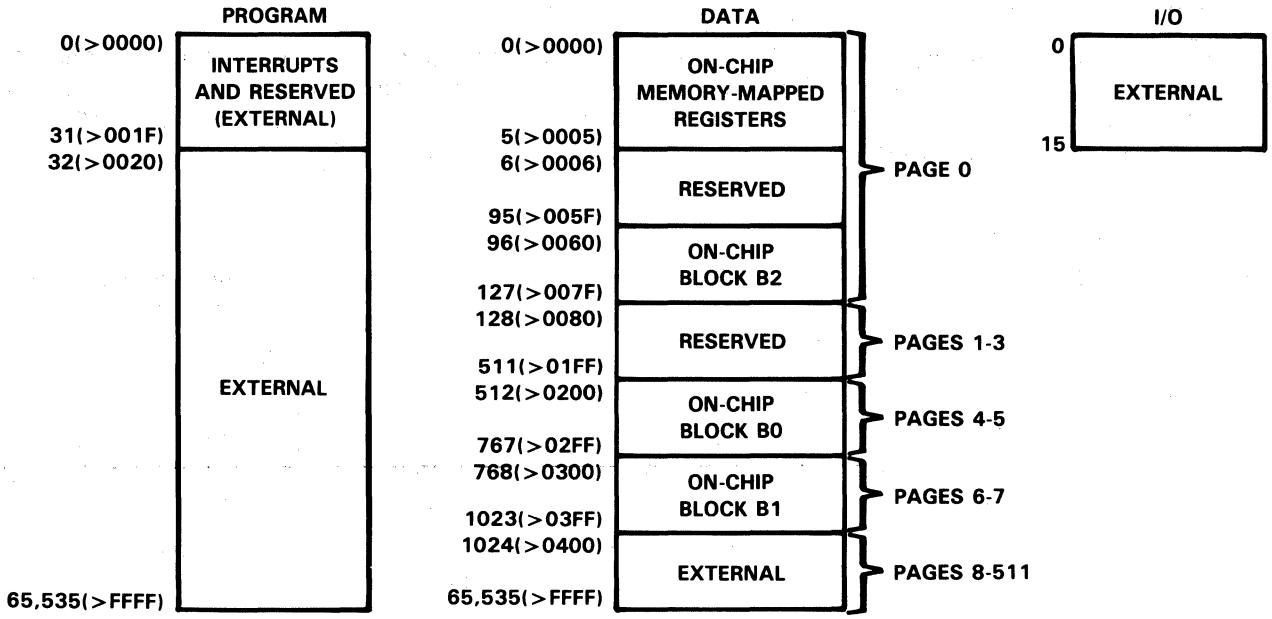
memory control

The TMS32020 provides a total of 544 16-bit words of on-chip data RAM, divided into three separate blocks (B0, B1, and B2). Of the 544 words, 288 words (blocks B1 and B2) are always data memory, and 256 words (block B0) are programmable as either data or program memory. A data memory size of 544 words allows the TMS32020 to handle a data array of 512 words (256 words if on-chip RAM is used for program memory), while still leaving 32 locations for intermediate storage. When using block B0 as program memory, instructions can be downloaded from external program memory into on-chip RAM and then executed.

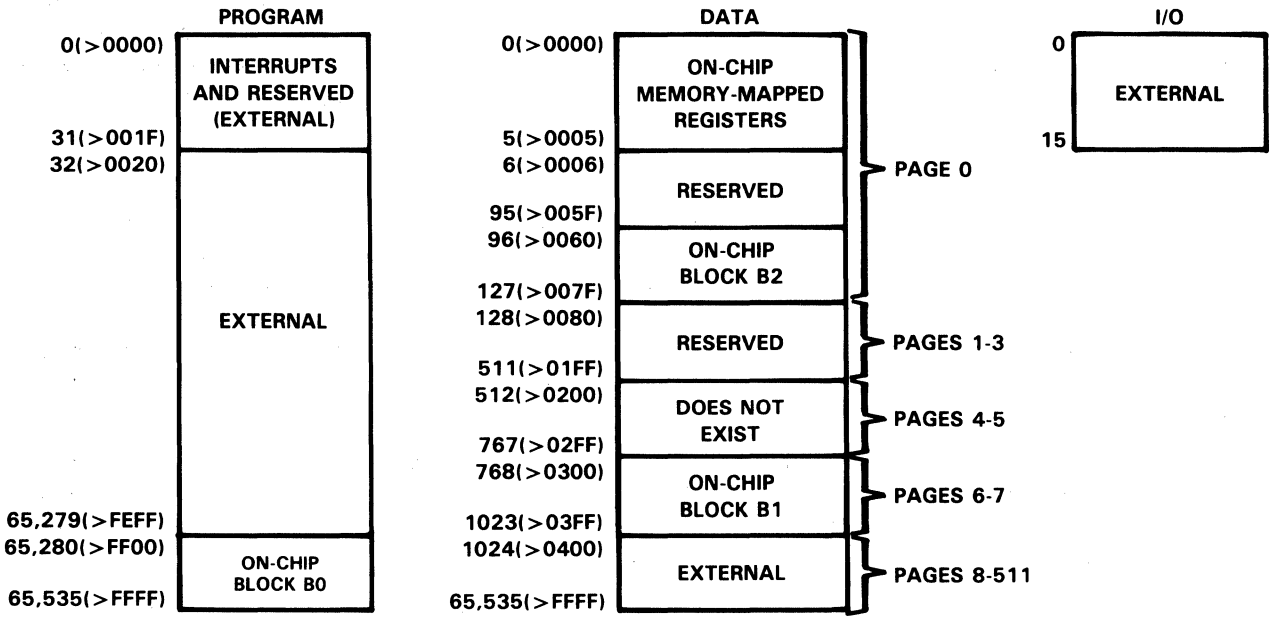
When using on-chip program RAM or high-speed external program memory, the TMS32020 runs at full speed without wait states. However, the READY line can be used to interface the TMS32020 to slower, less-expensive external memory. Downloading programs from slow off-chip memory to on-chip program RAM speeds processing while cutting system costs.

The TMS32020 provides three separate address spaces for program memory, data memory, and I/O. The on-chip memory is mapped into either the 64K-word data memory or program memory space, depending upon the memory configuration. The CNFD (configure block B0 as data memory) and CNFP (configure block B0 as program memory) instructions allow dynamic configuration of the memory maps through software. Regardless of the configuration, the user may still execute from external program memory.

The TMS32020 has six registers that are mapped into the data memory space: a serial port data receive register, serial port data transmit register, timer register, period register, interrupt mask register, and global memory allocation register.



(a) ADDRESS MAPS AFTER A CNFD INSTRUCTION



(b) ADDRESS MAPS AFTER A CNFP INSTRUCTION

FIGURE 1. MEMORY MAPS

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

interrupts and subroutines

The TMS32020 has three external maskable user interrupts $\overline{INT}2$-$\overline{INT}0$, available for external devices that interrupt the processor. Internal interrupts are generated by the serial port (RINT and XINT), by the timer (TINT), and by the software interrupt (TRAP) instruction. Interrupts are prioritized with reset ($\overline{RS}$) having the highest priority and the serial port transmit interrupt (XINT) having the lowest priority. All interrupt locations are on two-word boundaries so that branch instructions can be accommodated in those locations if desired.

A built-in mechanism protects multicycle instructions from interrupts. If an interrupt occurs during a multicycle instruction, the interrupt is not processed until the instruction is completed. This mechanism applies both to instructions that are repeated or become multicycle due to the READY signal.

external interface

The TMS32020 supports a wide range of system interfacing requirements. Program, data, and I/O address spaces provide interface to memory and I/O, thus maximizing system throughput. I/O design is simplified by having I/O treated the same way as memory. I/O devices are mapped into the I/O address space using the processor's external address and data busses in the same manner as memory-mapped devices. Interface to memory and I/O devices of varying speeds is accomplished by using the READY line. When transactions are made with slower devices, the TMS32020 processor waits until the other device completes its function and signals the processor via the READY line. Then, theTMS32020 continues execution.

A serial port provides communication with serial devices, such as codecs, serial A/D converters, and other serial systems. The interface signals are compatible with codecs and many other serial devices with a minimum of external hardware. The serial port may also be used for intercommunication between processors in multiprocessing applications.

The serial port has two memory-mapped registers: the data transmit register (DXR) and the data receive register (DRR). Both registers operate in either the byte mode or 16-bit word mode, and may be accessed in the same manner as any other data memory location. Each register has an external clock, a framing synchronization pulse, and associated shift registers. One method of multiprocessing may be implemented by programming one device to transmit while the others are in the receive mode.

multiprocessing

The flexibility of the TMS32020 allows configurations to satisfy a wide range of system requirements. The TMS32020 can be used as follows:

- A standalone processor
- A multiprocessor with devices in parallel
- A slave/host multiprocessor with global memory space
- A peripheral processor interfaced via processor-controlled signals to another device.

For multiprocessing applications, the TMS32020 has the capability of allocating global data memory space and communicating with that space via the $\overline{BR}$ (bus request) and READY control signals. Global memory is data memory shared by more than one processor. Global data memory access must be arbitrated. The 8-bit memory-mapped GREG (global memory allocation register) specifies part of the TMS32020's data memory as global external memory. The contents of the register determine the size of the global memory space. If the current instruction addresses an operand within that space, $\overline{BR}$ is asserted to request control of the bus. The length of the memory cycle is controlled by the READY line.

The TMS32020 supports DMA (direct memory access) to its external program/data memory using the $\overline{HOLD}$ and $\overline{HOLDA}$ signals. Another processor can take complete control of the TMS32020's external memory by asserting $\overline{HOLD}$ low. This causes the TMS32020 to place its address, data, and control lines in a high-impedance state and assert $\overline{HOLDA}$.

instruction set

The TMS32020 microprocessor implements a comprehensive instruction set that supports both numeric-intensive signal processing operations as well as general-purpose applications, such as multiprocessing and high-speed control. The TMS32010 source code is upward-compatible with TMS32020 source code.

For maximum throughput, the next instruction is prefetched while the current one is being executed. Since the same data lines are used to communicate to external data/program or I/O space, the number of cycles may vary depending upon whether the next data operand fetch is from internal or external program memory. Highest throughput is achieved by maintaining data memory on-chip and using either internal or fast external program memory.

addressing modes

The TMS32020 instruction set provides three memory addressing modes: direct, indirect, and immediate addressing.

Both direct and indirect addressing can be used to access data memory. In direct addressing, seven bits of the instruction word are concatenated with the nine bits of the data memory page pointer to form the 16-bit data memory address. Indirect addressing accesses data memory through the five auxiliary registers. In immediate addressing, the data is based on a portion of the instruction word(s).

In direct memory addressing, the instruction word contains the lower seven bits of the data memory address. This field is concatenated with the nine bits of the data memory page pointer to form the full 16-bit address. Thus, memory is paged in the direct addressing mode with a total of 512 pages, each page containing 128 words.

Five auxiliary registers (AR0-AR4) provide flexible and powerful indirect addressing. To select a specific auxiliary register, the Auxiliary Register Pointer (ARP) is loaded with either a 0, 1, 2, 3, or a 4 for AR0 through AR4, respectively.

There are five types of indirect addressing: auto-increment or auto-decrement, post-indexing by either adding or subtracting the contents of AR0, or single indirect addressing with no increment or decrement. All operations are performed on the current auxiliary register in the same cycle as the original instruction, followed by a new ARP value being loaded.

repeat feature

A repeat feature, used with instructions such as multiply/accumulates, block moves, I/O transfers, and table read/writes, allows a single instruction to be performed up to 256 times. The repeat counter (RPTC) is loaded with either a data memory value (RPT instruction) or an immediate value (RPTK instruction). The value of this operand is one less than the number of times that the next instruction is executed. Those instructions that are normally multicycle are pipelined when using the repeat feature, and effectively become single-cycle instructions.

instruction set summary

Table 1 lists the symbols and abbreviations used in Table 2, the instruction set summary. Table 2 consists primarily of single-cycle, single-word instructions. Infrequently used branch, I/O, and CALL instructions are multicycle. The instruction set summary is arranged according to function and alphabetized within each functional grouping. The symbol (†) indicates those instructions that are not included in the TMS32010 instruction set.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TABLE 1. INSTRUCTION SYMBOLS

SYMBOL	MEANING
B	4-bit field specifying a bit code
CM	2-bit field specifying compare mode
D	Data memory address field
FO	Format status bit
I	Addressing mode bit
K	Immediate operand field
PA	Port address (PA0 through PA15 are predefined assembler symbols equal to 0 through 15, respectively.)
PM	2-bit field specifying P register output shift code
R	3-bit operand field specifying auxiliary register
S	4-bit left-shift code
X	3-bit accumulator left-shift field

TABLE 2. INSTRUCTION SET SUMMARY

ACCUMULATOR MEMORY REFERENCE INSTRUCTIONS																		
Mnemonic	**Description**	**No. Words**	**Instruction Bit Code**															
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ABS	Absolute value of accumulator	1	1	1	0	0	1	1	1	0	0	0	0	1	1	0	1	1
ADD	Add to accumulator with shift	1	0	0	0	0	←	S	→		I	←		D	→			
ADDH	Add to high accumulator	1	0	1	0	0	1	0	0	0	I	←		D	→			
ADDS	Add to low accumulator with sign extension suppressed	1	0	1	0	0	1	0	0	1	I	←		D	→			
ADDT†	Add to accumulator with shift specified by T register	1	0	1	0	0	1	0	1	0	I	←		D	→			
ADLK†	Add to accumulator long immediate with shift	2	1	1	0	1	←	S	→		0	0	0	0	0	0	1	0
AND	AND with accumulator	1	0	1	0	0	1	1	1	0	I	←		D	→			
ANDK†	AND immediate with accumulator with shift	2	1	1	0	1	←	S	→		0	0	0	0	0	1	0	0
CMPL†	Complement accumulator	1	1	1	0	0	1	1	1	0	0	0	1	0	0	1	1	1
LAC	Load accumulator with shift	1	0	0	1	0	←	S	→		I	←		D	→			
LACK	Load accumulator immediate short	1	1	1	0	0	1	0	1	0	←			K	→			
LACT†	Load accumulator with shift specified by T register	1	0	1	0	0	0	0	1	0	I	←		D	→			
LALK†	Load accumulator long immediate with shift	2	1	1	0	1	←	S	→		0	0	0	0	0	0	0	1
NEG†	Negate accumulator	1	1	1	0	0	1	1	1	0	0	0	1	0	0	0	1	1
NORM†	Normalize contents of accumulator	1	1	1	0	0	1	1	1	0	1	0	1	0	0	0	1	0
OR	OR with accumulator	1	0	1	0	0	1	1	0	1	I	←		D	→			
ORK†	OR immediate with accumulator with shift	2	1	1	0	1	←	S	→		0	0	0	0	0	1	0	1
SACH	Store high accumulator with shift	1	0	1	1	0	1	←	X	→	I	←		D	→			
SACL	Store low accumulator with shift	1	0	1	1	0	0	←	X	→	I	←		D	→			
SBLK†	Subtract from accumulator long immediate with shift	2	1	1	0	1	←	S	→		0	0	0	0	0	0	1	1
SFL†	Shift accumulator left	1	1	1	0	0	1	1	1	0	0	0	0	1	1	0	0	0
SFR†	Shift accumulator right	1	1	1	0	0	1	1	1	0	0	0	0	1	1	0	0	1
SUB	Subtract from accumulator with shift	1	0	0	0	1	←	S	→		I	←		D	→			
SUBC	Conditional subtract	1	0	1	0	0	0	1	1	1	I	←		D	→			
SUBH	Subtract from high accumulator	1	0	1	0	0	0	1	0	0	I	←		D	→			
SUBS	Subtract from low accumulator with sign extension suppressed	1	0	1	0	0	0	1	0	1	I	←		D	→			
SUBT†	Subtract from accumulator with shift specified by T register	1	0	1	0	0	0	1	1	0	I	←		D	→			
XOR	Exclusive-OR with accumulator	1	0	1	0	0	1	1	0	0	I	←		D	→			
XORK†	Exclusive-OR immediate with accumulator with shift	2	1	1	0	1	←	S	→		0	0	0	0	0	1	1	0
ZAC	Zero accumulator	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0
ZALH	Zero low accumulator and load high accumulator	1	0	1	0	0	0	0	0	0	I	←		D	→			
ZALS	Zero accumulator and load low accumulator with sign extension suppressed	1	0	1	0	0	0	0	0	1	I	←		D	→			

AUXILIARY REGISTERS AND DATA PAGE POINTER INSTRUCTIONS																		
Mnemonic	**Description**	**No. Words**	**Instruction Bit Code**															
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMPR†	Compare auxiliary register with auxiliary register AR0	1	1	1	0	0	1	1	1	0	0	1	0	1	0	0<CM>		
LAR	Load auxiliary register	1	0	0	1	1	0	←	R	→	I	←		D	→			
LARK	Load auxiliary register immediate short	1	1	1	0	0	0	←	R	→	←			K	→			
LARP	Load auxiliary register pointer	1	0	1	0	1	0	1	0	1	1	0	0	0	1		R	
LDP	Load data memory page pointer	1	0	1	0	1	0	0	1	0	I	←		D	→			
LDPK	Load data memory page pointer immediate	1	1	1	0	0	1	0	0	←				K	→			
LRLK†	Load auxiliary register long immediate	2	1	1	0	1	0	←	R	→	0	0	0	0	0	0	0	0
MAR	Modify auxiliary register	1	0	1	0	1	0	1	0	1	I	←		D	→			
SAR	Store auxiliary register	1	0	1	1	1	0	←	R	→	I	←		D	→			

†These instructions not included in the TMS32010 instruction set.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TABLE 2. INSTRUCTION SET SUMMARY (CONTINUED)

T REGISTER, P REGISTER, AND MULTIPLY INSTRUCTIONS

Mnemonic	Description	No. Words	Instruction Bit Code															
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
APAC	Add P register to accumulator	1	1	1	0	0	1	1	1	0	0	0	0	1	0	1	0	1
LPH†	Load high P register	1	0	1	0	1	0	0	1	1	I ←				D →			
LT	Load T register	1	0	0	1	1	1	1	0	0	I ←				D →			
LTA	Load T register and accumulate previous product	1	0	0	1	1	1	1	0	1	I ←				D →			
LTD	Load T register, accumulate previous product, and move data	1	0	0	1	1	1	1	1	1	I ←				D →			
LTP†	Load T register and store P register in accumulator	1	0	0	1	1	1	1	1	0	I ←				D →			
LTS†	Load T register and subtract previous product	1	0	1	0	1	1	0	1	1	I ←				D →			
MAC†	Multiply and accumulate	2	0	1	0	1	1	1	0	1	I ←				D →			
MACD†	Multiply and accumulate with data move	2	0	1	0	1	1	1	0	0	I ←				D →			
MPY	Multiply (with T register, store product in P register)	1	0	0	1	1	1	0	0	0	I ←				D →			
MPYK	Multiply immediate	1	1	0	1 ←							K						→
PAC	Load accumulator with P register	1	1	1	0	0	1	1	1	0	0	0	0	1	0	1	0	0
SPAC	Subtract P register from accumulator	1	1	1	0	0	1	1	1	0	0	0	0	1	0	1	1	0
SPM†	Set P register output shift mode	1	1	1	0	0	1	1	1	0	0	0	0	0	1	0<PM>		
SQRA†	Square and accumulate	1	0	0	1	1	1	0	0	1	I ←				D →			
SQRS†	Square and subtract previous product	1	0	1	0	1	1	0	1	0	I ←				D →			

BRANCH/CALL INSTRUCTIONS

Mnemonic	Description	No. Words	Instruction Bit Code															
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	Branch unconditionally	2	1	1	1	1	1	1	1	1	1 ←				D →			
BACC†	Branch to address specified by accumulator	1	1	1	0	0	1	1	1	0	0	0	1	0	0	1	0	1
BANZ	Branch on auxiliary register not zero	2	1	1	1	1	1	0	1	1	1 ←				D →			
BBNZ†	Branch if TC bit ≠ 0	2	1	1	1	1	1	0	0	1	1 ←				D →			
BBZ†	Branch if TC bit = 0	2	1	1	1	1	1	0	0	0	1 ←				D →			
BGEZ	Branch if accumulator ≥ 0	2	1	1	1	1	0	1	0	0	1 ←				D →			
BGZ	Branch if accumulator > 0	2	1	1	1	1	0	0	0	1	1 ←				D →			
BIOZ	Branch on I/O status = 0	2	1	1	1	1	1	0	1	0	1 ←				D →			
BLEZ	Branch if accumulator ≤ 0	2	1	1	1	1	0	0	1	0	1 ←				D →			
BLZ	Branch if accumulator < 0	2	1	1	1	1	0	0	1	1	1 ←				D →			
BNV†	Branch if no overflow	2	1	1	1	1	0	1	1	1	1 ←				D →			
BNZ	Branch if accumulator ≠ 0	2	1	1	1	1	0	1	0	1	1 ←				D →			
BV	Branch on overflow	2	1	1	1	1	0	0	0	0	1 ←				D →			
BZ	Branch if accumulator = 0	2	1	1	1	1	0	1	1	0	1 ←				D →			
CALA	Call subroutine indirect	1	1	1	0	0	1	1	1	0	0	0	1	0	0	1	0	0
CALL	Call subroutine	2	1	1	1	1	1	1	1	0	1 ←				D →			
RET	Return from subroutine	1	1	1	0	0	1	1	1	0	0	0	1	0	0	1	1	0

†These instructions not included in the TMS32010 instruction set.

TABLE 2. INSTRUCTION SET SUMMARY (CONCLUDED)

CONTROL INSTRUCTIONS																		
Mnemonic	**Description**	**No. Words**	**Instruction Bit Code**															
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIT†	Test bit	1	1	0	0	1	←		B	→	I	←			D			→
BITT†	Test bit specified by T register	1	0	1	0	1	0	1	1	1	I	←			D			→
CNFD†	Configure block as data memory	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	0	0
CNFP†	Configure block as program memory	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	0	1
DINT	Disable interrupt	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	1
EINT	Enable interrupt	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0
IDLE†	Idle until interrupt	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	1	1
LST	Load status register ST0	1	0	1	0	1	0	0	0	0	I	←			D			→
LST1†	Load status register ST1	1	0	1	0	1	0	0	0	1	I	←			D			→
NOP	No operation	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0
POP	Pop top of stack to low accumulator	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	0	1
POPD†	Pop top of stack to data memory	1	0	1	1	1	1	0	1	0	I	←			D			→
PSHD†	Push data memory value onto stack	1	0	1	0	1	0	1	0	0	I	←			D			→
PUSH	Push low accumulator onto stack	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	0	0
ROVM	Reset overflow mode	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	1	0
RPT†	Repeat instruction as specified by data memory value	1	0	1	0	0	1	0	1	1	I	←			D			→
RPTK†	Repeat instruction as specified by immediate value	1	1	1	0	0	1	0	1	1	←				K			→
RSXM†	Reset sign-extension mode	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	1	0
SOVM	Set overflow mode	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	1	1
SST	Store status register ST0	1	0	1	1	1	1	0	0	0	I	←			D			→
SST1†	Store status register ST1	1	0	1	1	1	1	0	0	1	I	←			D			→
SSXM†	Set sign-extension mode	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	1	1
TRAP†	Software interrupt	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	1	0
I/O AND DATA MEMORY OPERATIONS																		
Mnemonic	**Description**	**No. Words**	**Instruction Bit Code**															
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BLKD†	Block move from data memory to data memory	2	1	1	1	1	1	1	0	1	I	←			D			→
BLKP†	Block move from program memory to data memory	2	1	1	1	1	1	1	0	0	I	←			D			→
DMOV	Data move in data memory	1	0	1	0	1	0	1	1	0	I	←			D			→
FORT†	Format serial port registers	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	1	FO
IN	Input data from port	1	1	0	0	0	←		PA	→	I	←			D			→
OUT	Output data to port	1	1	1	1	0	←		PA	→	I	←			D			→
RTXM†	Reset serial port transmit mode	1	1	1	0	0	1	1	1	0	0	0	1	0	0	0	0	0
RXF†	Reset external flag	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	0	0
STXM†	Set serial port transmit mode	1	1	1	0	0	1	1	1	0	0	0	1	0	0	0	0	1
SXF†	Set external flag	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	0	1
TBLR	Table read	1	0	1	0	1	1	0	0	0	I	←			D			→
TBLW	Table write	1	0	1	0	1	1	0	0	1	I	←			D			→

†These instructions not included in the TMS32010 instruction set.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

development systems and software support

Texas Instruments offers concentrated development support and complete documentation for designing a TMS32020-based microprocessor system. When developing an application, tools are provided to evaluate the performance of the processor, to develop the algorithm implementation, and to fully integrate the design's software and hardware modules. When questions arise, additional support can be obtained by calling the nearest Texas Instruments Regional Technology Center (RTC).

Sophisticated development operations are performed with the TMS32020 Macro Assembler/Linker, Simulator, and Emulator (XDS). The macro assembler and linker are used to translate program modules into object code and link them together. This puts the program modules into a form which can be loaded into the TMS32020 Simulator or Emulator. The simulator provides a quick means for initially debugging TMS32020 software while the emulator provides the real-time in-circuit emulation necessary to perform system level debug efficiently.

Table 3 gives a complete list of TMS32020 software and hardware development tools.

TABLE 3. TMS32020 SOFTWARE AND HARDWARE SUPPORT

MACRO ASSEMBLERS/LINKERS		
Host Computer	**Operating System**	**Part Number**
DEC VAX	VMS	TMDS3241210-08
TI/IBM PC	MS/PC-DOS	TMDS3241810-02
SIMULATORS		
Host Computer	**Operating System**	**Part Number**
DEC VAX	VMS	TMDS3241211-08
TI/IBM PC	MS/PC-DOS	TMDS3241811-02
EMULATORS		
Model	**Power Supply**	**Part Number**
XDS/11	5 V @ 5 A required	TMDS3261120
XDS/22	Included	TMDS3262220

absolute maximum ratings over specified temperature range (unless otherwise noted)†

Supply voltage range, V_{CC}‡ . . . . . −0.3 V to 7 V
Input voltage range . . . . . −0.3 V to 7 V
Output voltage range . . . . . −0.3 V to 7 V
Continuous power dissipation . . . . . 2.0 W
Operating free-air temperature range . . . . . 0°C to 70°C
Storage temperature range . . . . . −55°C to 150°C

†Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions beyond those indicated in the "Recommended Operating Conditions" section of this specification is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

‡All voltage values are with respect to V_{SS}.

recommended operating conditions

			MIN	NOM	MAX	UNIT
V_{CC}	Supply voltage		4.75	5	5.25	V
V_{SS}	Supply voltage			0		V
V_{IH}	High-level input voltage	All inputs except CLKIN	2		V_{CC}+0.3	V
		CLKIN	2.4		V_{CC}+0.3	V
V_{IL}	Low-level input voltage	All inputs except CLKIN	−0.3		0.8	V
		CLKIN	−0.3		0.8	V
I_{OH}	High-level output current				300	μA
I_{OL}	Low-level output current				2	mA
T_A	Operating free-air temperature (Notes 1 and 2)		0		70	°C

NOTES: 1. Case temperature (T_C) must be maintained below 90°C.
2. $R_{\theta JA}$ = 36°C/Watt; $R_{\theta JC}$ = 6°C/Watt.

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_{CC} = MIN, I_{OH} = MAX	2.4	3		V
V_{OL}	Low-level output voltage	V_{CC} = MIN, I_{OL} = MAX		0.3	0.6	V
I_Z	Three-state current	V_{CC} = MAX	−20		20	μA
I_I	Input current	V_I = V_{SS} to V_{CC}	−10		10	μA
I_{CC}	Supply current	T_A = 0°C, V_{CC} = MAX, f_x = MAX			360	mA
		T_A = 25°C, V_{CC} = 5 V, f_x = MAX		250		mA
		T_C = 90°C, V_{CC} = MAX, f_x = MAX			285	mA
C_I	Input capacitance				15	pF
C_O	Output capacitance				15	pF

†All typical values are at V_{CC} = 5 V, T_A = 25°C.

Caution. This device contains circuits to protect its inputs and outputs against damage due to high static voltages or electrostatic fields. These circuits have been qualified to protect this device against electrostatic discharges (ESD) of up to 2 kV according to MIL-STD-883C, Method 3015; however, it is advised that precautions be taken to avoid application of any voltage higher than maximum rated voltages to these high-impedance circuits. During storage or handling, the device leads should be shorted together or the device should be placed in conductive foam. In a circuit, unused inputs should always be connected to an appropriate logic voltage level, preferrably either V_{CC} or ground. Specific guidelines for handling devices of this type are contained in the publication "Guidelines for Handling Electrostatic-Discharge Sensitive (ESDS) Devices and Assemblies" available from Texas Instruments.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

CLOCK CHARACTERISTICS AND TIMING

The TMS32020 can use either its internal oscillator or an external frequency source for a clock.

internal clock option

The internal oscillator is enabled by connecting a crystal across X1 and X2/CLKIN (see Figure 2). The frequency of CLKOUT1 is one-fourth the crystal fundamental frequency. The crystal should be fundamental mode, and parallel resonant, with an effective series resistance of 30 ohms, a power dissipation of 1 mW, and be specified at a load capacitance of 20 pF.

PARAMETER		TEST CONDITIONS	MIN	TYP	MAX	UNIT
f_x	Input clock frequency	T_A = 0°C to 70°C	6.7		20.5	MHz
f_{sx}	Serial port frequency	T_A = 0°C to 70°C	50		2563	kHz
C1, C2		T_A = 0°C to 70°C		10		pF

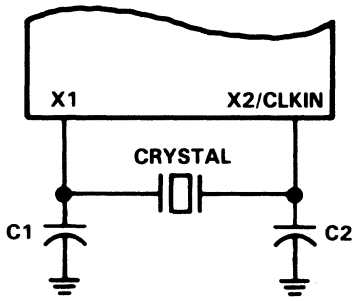


FIGURE 2. INTERNAL CLOCK OPTION

external clock option

An external frequency source can be used by injecting the frequency directly into X2/CLKIN with X1 left unconnected. The external frequency injected must conform to the specifications listed in the following table.

switching characteristics over recommended operating conditions (see Note 3)

PARAMETER		MIN	TYP	MAX	UNIT
$t_{c(C)}$	CLKOUT1/CLKOUT2 cycle time	195		597	ns
$t_{d(CIH-C)}$	CLKIN high to CLKOUT1/CLKOUT2/$\overline{STRB}$ high/low	25		50	ns
$t_{f(C)}$	CLKOUT1/CLKOUT2/$\overline{STRB}$ fall time			10	ns
$t_{r(C)}$	CLKOUT1/CLKOUT2/$\overline{STRB}$ rise time			10	ns
$t_{w(CL)}$	CLKOUT1/CLKOUT2 low pulse duration	2Q−15	2Q	2Q+15	ns
$t_{w(CH)}$	CLKOUT1/CLKOUT2 high pulse duration	2Q−15	2Q	2Q+15	ns
$t_{d(C1-C2)}$	CLKOUT1 high to CLKOUT2 low, CLKOUT2 high to CLKOUT1 high, etc.	Q−10	Q	Q+10	ns

NOTE 3: Q = $1/4t_{c(C)}$.

timing requirements over recommended operating conditions (see Note 3)

		MIN	NOM	MAX	UNIT
$t_{c(CI)}$	CLKIN cycle time	48.8		150	ns
$t_{f(CI)}$	CLKIN fall time			10	ns
$t_{r(CI)}$	CLKIN rise time			10	ns
$t_{w(CIL)}$	CLKIN low pulse duration, $t_{c(CI)}$ = 50 ns (Note 4)	10		40	ns
$t_{w(CIH)}$	CLKIN high pulse duration, $t_{c(CI)}$ = 50 ns (Note 4)	10		40	ns
$t_{su(S)}$	$\overline{SYNC}$ setup time before CLKIN low	10		Q−10	ns
$t_{h(S)}$	$\overline{SYNC}$ hold time from CLKIN low	15			ns

NOTES: 3. $Q = 1/4t_{c(C)}$.
4. CLKIN duty cycle $[t_{r(CI)} + t_{w(CIH)}]/t_{c(CI)}$ must be within 40-60%.

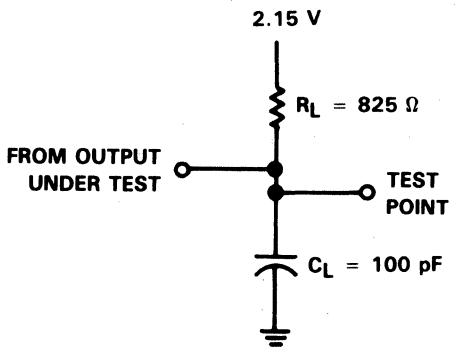
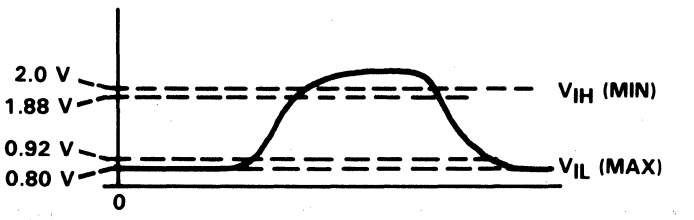
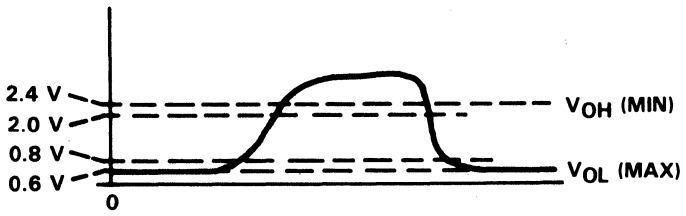


FIGURE 3. TEST LOAD CIRCUIT



(a) INPUT

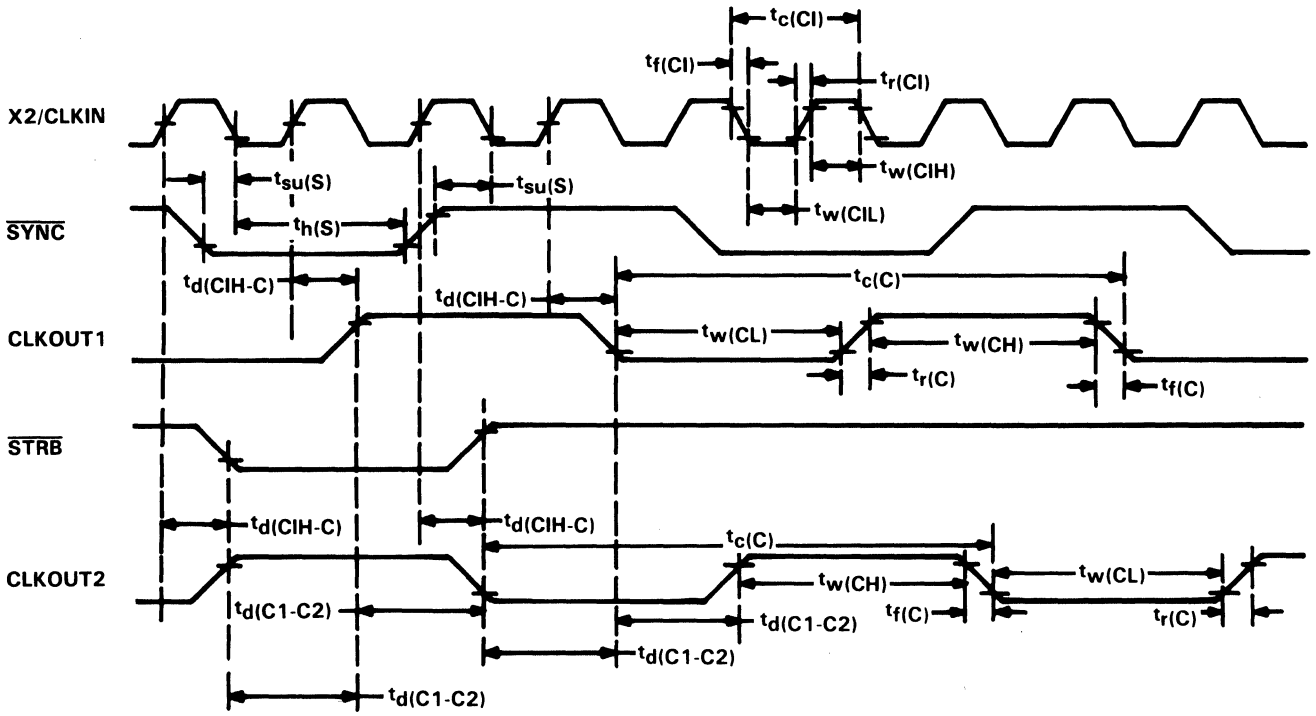


(b) OUTPUTS

FIGURE 4. VOLTAGE REFERENCE LEVELS

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

clock timing



MEMORY AND PERIPHERAL INTERFACE TIMING

switching characteristics over recommended operating conditions (see Note 3)

	PARAMETER	MIN	TYP	MAX	UNIT
$t_{d(C1\text{-}S)}$	$\overline{STRB}$ from CLKOUT1 (if $\overline{STRB}$ is present)	Q – 15	Q	Q + 15	ns
$t_{d(C2\text{-}S)}$	CLKOUT2 to $\overline{STRB}$ (if $\overline{STRB}$ is present)	– 15	0	15	ns
$t_{su(A)}$	Address setup time before $\overline{STRB}$ low (Note 5)	Q – 30			ns
$t_{h(A)}$	Address hold time after $\overline{STRB}$ high (Note 5)	Q – 15			ns
$t_{w(SL)}$	$\overline{STRB}$ low pulse duration (no wait states, Note 6)		2Q		ns
$t_{w(SH)}$	$\overline{STRB}$ high pulse duration (between consecutive cycles, Note 6)		2Q		ns
$t_{su(D)W}$	Data write setup time before $\overline{STRB}$ high (no wait states)	2Q – 45			ns
$t_{h(D)W}$	Data write hold time from $\overline{STRB}$ high	Q – 15	Q		ns
$t_{en(D)}$	Data bus starts being driven after $\overline{STRB}$ low (write cycle)	0			ns
$t_{dis(D)}$	Data bus three-state after $\overline{STRB}$ high (write cycle)		Q	Q + 30	ns
$t_{d(MSC)}$	$\overline{MSC}$ valid from CLKOUT1	– 25	0	25	ns

NOTES: 3. Q = $1/4t_{c(C)}$.
5. A15-A0, $\overline{PS}$, $\overline{DS}$, $\overline{IS}$, R/$\overline{W}$, and $\overline{BR}$ timings are all included in timings referenced as "address."
6. Delays between CLKOUT1/CLKOUT2 edges and $\overline{STRB}$ edges track each other, resulting in $t_{w(SL)}$ and $t_{w(SH)}$ being 2Q with no wait states.

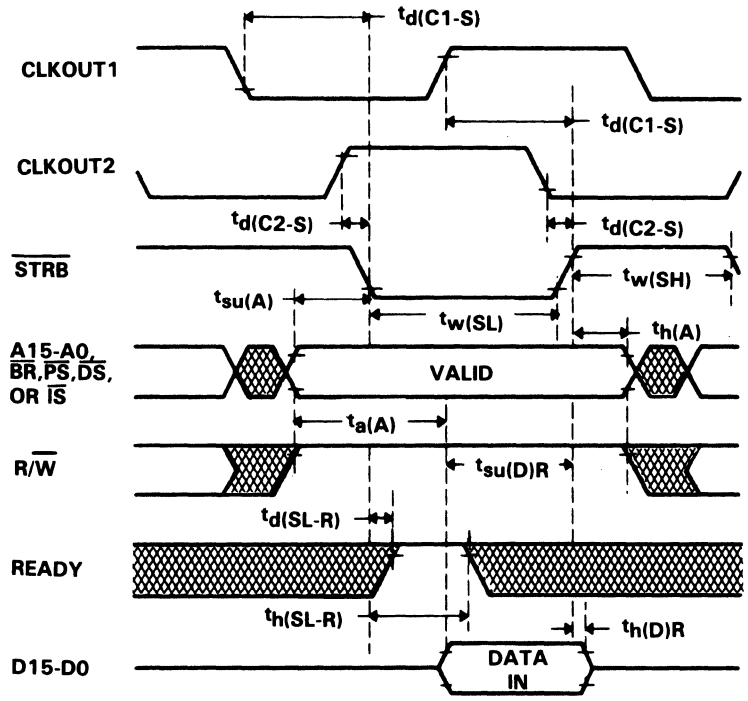
timing requirements over recommended operating conditions (see Note 3)

		MIN	NOM	MAX	UNIT
$t_{a(A)}$	Read data access time from address time (read cycle, Notes 5 and 7)			3Q – 70	ns
$t_{su(D)R}$	Data read setup time before $\overline{STRB}$ high	40			ns
$t_{h(D)R}$	Data read hold time from $\overline{STRB}$ high	0			ns
$t_{d(SL\text{-}R)}$	READY valid after $\overline{STRB}$ low (no wait states)			Q – 40	ns
$t_{d(C2H\text{-}R)}$	READY valid after CLKOUT2 high			Q – 40	ns
$t_{h(SL\text{-}R)}$	READY hold time after $\overline{STRB}$ low (no wait states)	Q – 5			ns
$t_{h(C2H\text{-}R)}$	READY hold after CLKOUT2 high	Q – 5			ns
$t_{d(M\text{-}R)}$	READY valid after $\overline{MSC}$ valid			2Q – 50	ns
$t_{h(M\text{-}R)}$	READY hold time after $\overline{MSC}$ valid	0			ns

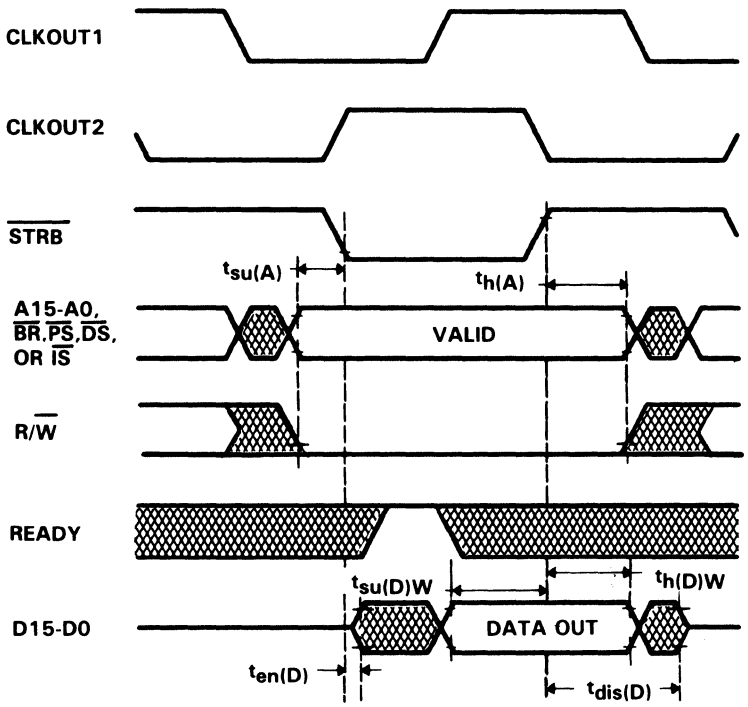
NOTES: 3. Q = $1/4t_{c(C)}$.
5. A15-A0, $\overline{PS}$, $\overline{DS}$, $\overline{IS}$, R/$\overline{W}$, and $\overline{BR}$ timings are all included in timings referenced as "address."
7. Read data access time is defined as $t_{a(A)} = t_{su(A)} + t_{w(SL)} - t_{su(D)R}$.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

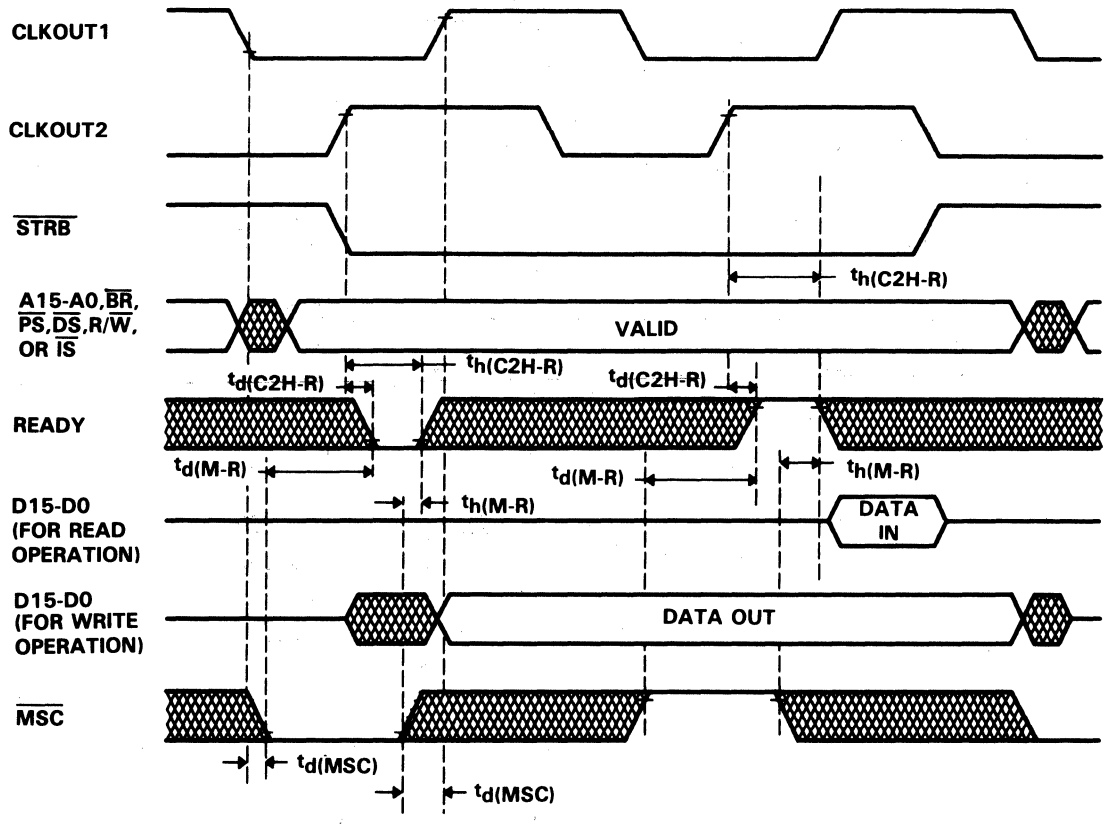
memory read timing



memory write timing



one wait-state memory access timing



$\overline{RS}$, $\overline{INT}$, $\overline{BIO}$, and XF TIMING

switching characteristics over recommended operating conditions (see Note 3)

PARAMETER		MIN	TYP	MAX	UNIT
$t_{d(RS)}$	CLKOUT1 low to reset state entered			45	ns
$t_{d(IACK)}$	CLKOUT1 to $\overline{IACK}$ valid	−25	0	25	ns
$t_{d(XF)}$	XF valid before falling edge of $\overline{STRB}$	Q − 30			ns

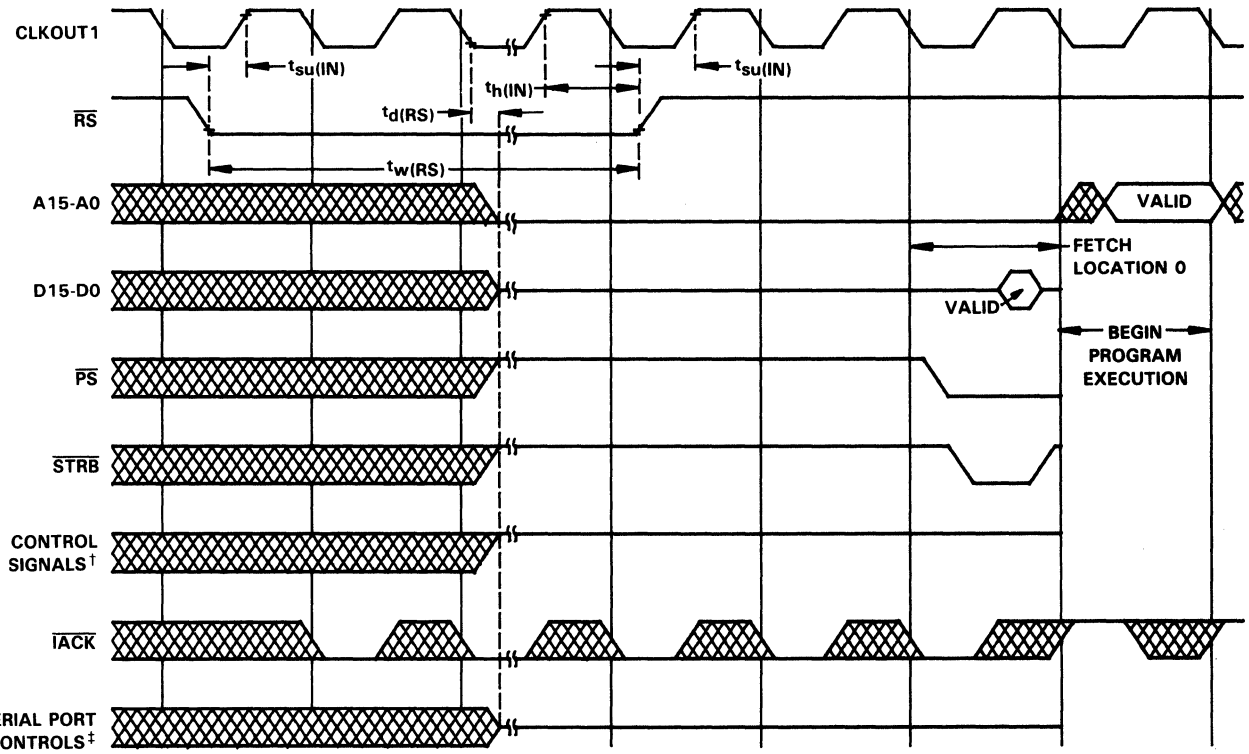
NOTE 3: Q = $1/4t_{c(C)}$.
8. $\overline{RS}$, $\overline{INT}$, and $\overline{BIO}$ are asynchronous inputs and can occur at any time during a clock cycle. However, if the specified setup time is met, the exact sequence shown in the timing diagrams will occur.

timing requirements over recommended operating conditions (see Note 3)

		MIN	NOM	MAX	UNIT
$t_{su(IN)}$	$\overline{INT}/\overline{BIO}/\overline{RS}$ setup before CLKOUT1 high	50			ns
$t_{h(IN)}$	$\overline{INT}/\overline{BIO}/\overline{RS}$ hold after CLKOUT1 high	0			ns
$t_{f(IN)}$	$\overline{INT}/\overline{BIO}$ fall time			15	ns
$t_{w(IN)}$	$\overline{INT}/\overline{BIO}$ low pulse duration	$t_{c(C)}$			ns
$t_{w(RS)}$	$\overline{RS}$ low pulse duration	$3t_{c(C)}$			ns

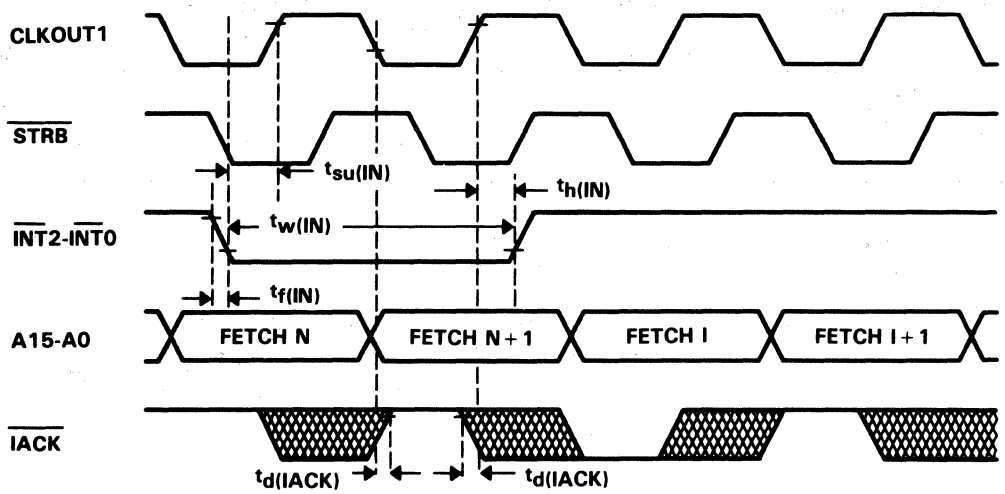
NOTE 3: Q = $1/4t_{c(C)}$.
8. $\overline{RS}$, $\overline{INT}$, and $\overline{BIO}$ are asynchronous inputs and can occur at any time during a clock cycle. However, if the specified setup time is met, the exact sequence shown in the timing diagrams will occur.

reset timing

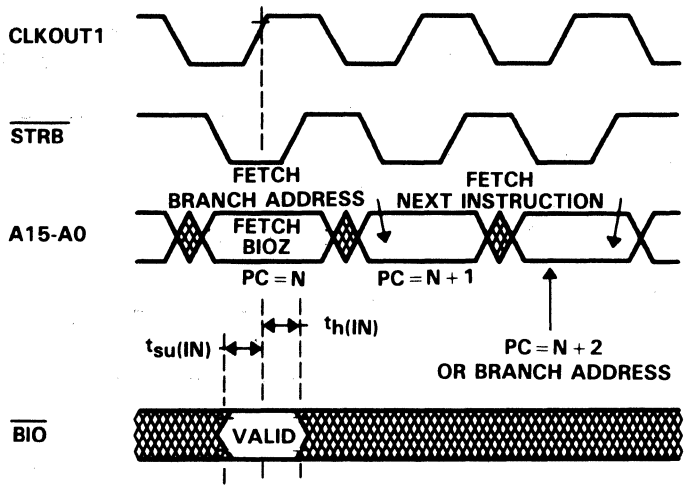


† Control signals are $\overline{DS}$, $\overline{IS}$, R/$\overline{W}$ and XF.
‡ Serial port controls are $\overline{DX}$ and FSX.

interrupt timing

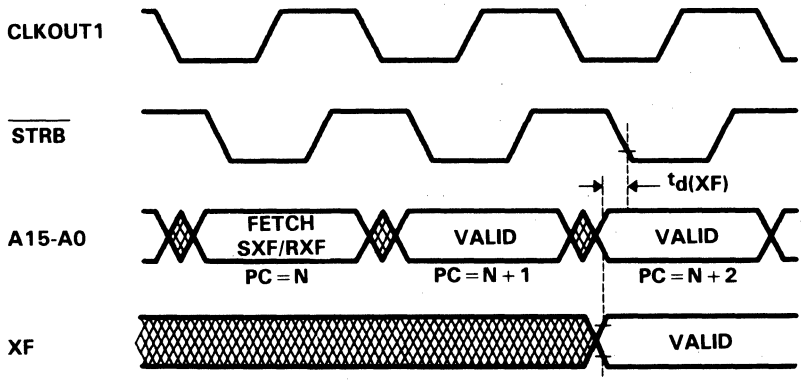


$\overline{BIO}$ timing



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

external flag timing



$\overline{\text{HOLD}}$ TIMING

switching characteristics over recommended operating conditions (see Note 3)

	PARAMETER	MIN	TYP	MAX	UNIT
$t_{d(C1L-AL)}$	$\overline{\text{HOLDA}}$ low after CLKOUT1 low	−25		25	ns
$t_{dis(AL-A)}$	$\overline{\text{HOLDA}}$ low to address three-state		15		ns
$t_{dis(C1L-A)}$	Address three-state after CLKOUT1 low ($\overline{\text{HOLD}}$ mode, Note 5)			30	ns
$t_{d(HH-AH)}$	$\overline{\text{HOLD}}$ high to $\overline{\text{HOLDA}}$ high			50	ns
$t_{en(A-C1L)}$	Address driven before CLKOUT1 low ($\overline{\text{HOLD}}$ mode, Note 5)			10	ns

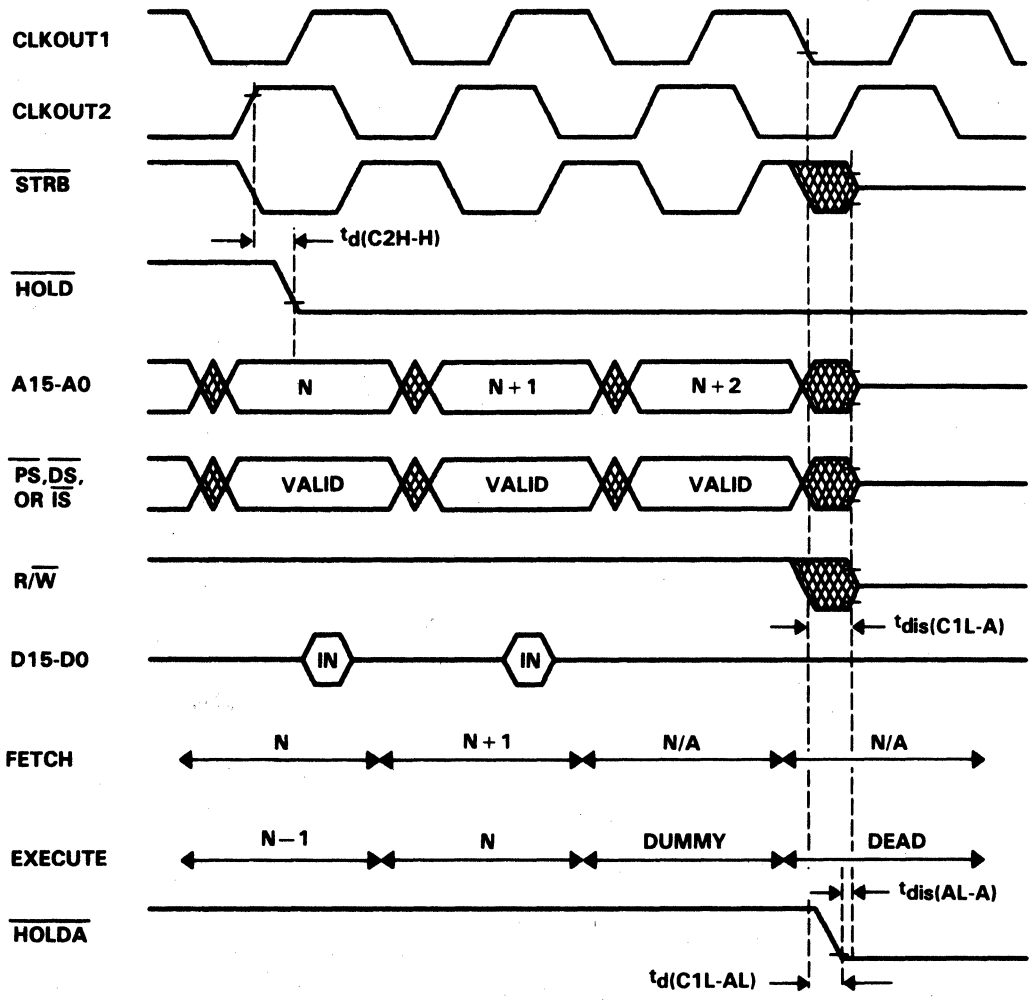
NOTES: 3. $Q = 1/4t_{c(C)}$.
5. A15-A0, $\overline{\text{PS}}$, $\overline{\text{DS}}$, $\overline{\text{IS}}$, R/$\overline{\text{W}}$, and $\overline{\text{BR}}$ timings are all included in timings referenced as "address."

timing requirements over recommended operating conditions (see Note 3)

		MIN	NOM	MAX	UNIT
$t_{d(C2H-H)}$	$\overline{\text{HOLD}}$ valid after CLKOUT2 high			Q-40	ns

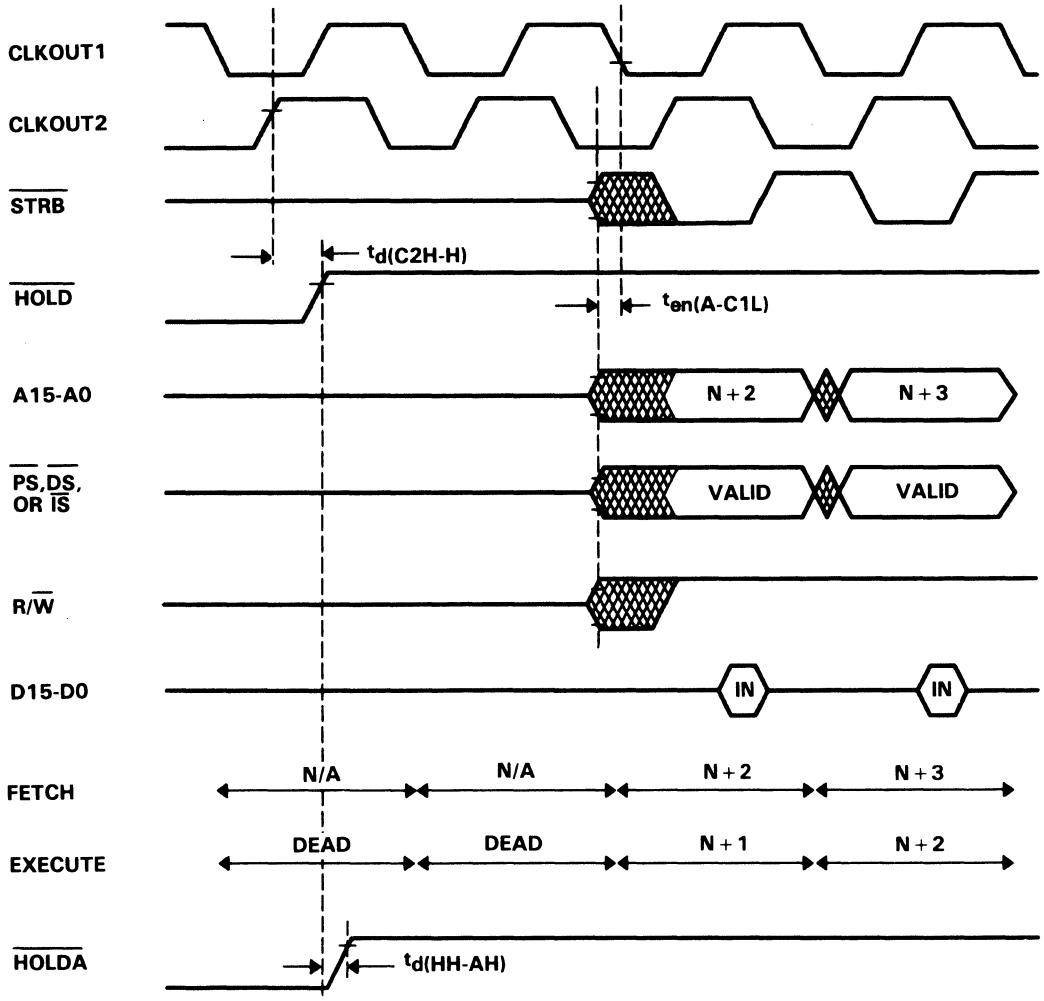
NOTE: 3. $Q = 1/4t_{c(C)}$.

$\overline{\text{HOLD}}$ timing (part A)



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

$\overline{\text{HOLD}}$ timing (part B)



SERIAL PORT TIMING

switching characteristics over recommended operating conditions (see Note 3)

PARAMETER		MIN	TYP	MAX	UNIT
$t_{d(CH-DX)}$	DX valid after CLKX rising edge (Note 9)			100	ns
$t_{d(FL-DX)}$	DX valid after FSX falling edge (TXM = 0, Note 9)			50	ns
$t_{d(CH-FS)}$	FSX valid after CLKX rising edge (TXM = 1)			60	ns

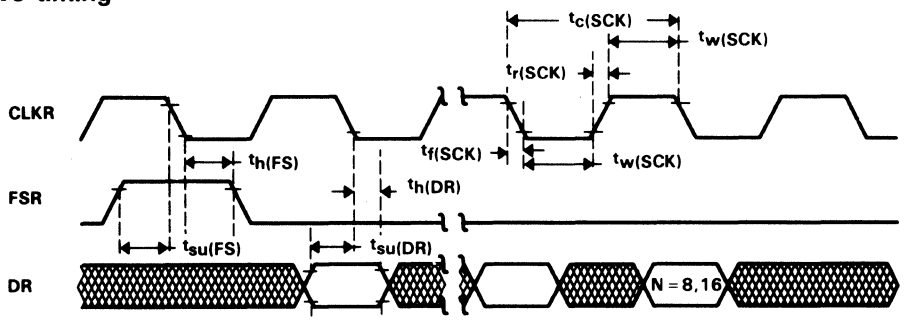
NOTES: 3. $Q = 1/4t_{c(C)}$.
9. The last occurrence of FSX falling and CLKX rising.

timing requirements over recommended operating conditions (see Note 3)

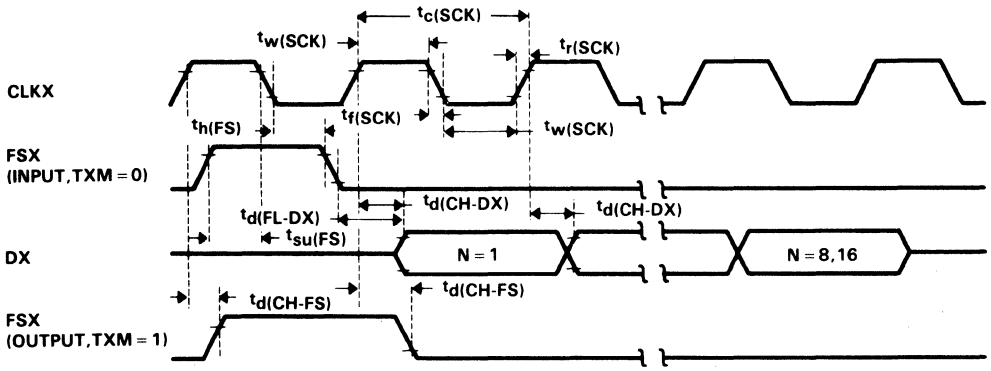
		MIN	NOM	MAX	UNIT
$t_{c(SCK)}$	Serial port clock (CLKX/CLKR) cycle time	390		20,000	ns
$t_{f(SCK)}$	Serial port clock (CLKX/CLKR) fall time			50	ns
$t_{r(SCK)}$	Serial port clock (CLKX/CLKR) rise time			50	ns
$t_{w(SCK)}$	Serial port clock (CLKX/CLKR) low pulse duration (see Note 10)	150		12,000	ns
$t_{w(SCK)}$	Serial port clock (CLKX/CLKR) high pulse duration (see Note 10)	150		12,000	ns
$t_{su(FS)}$	FSX/FSR setup time before (CLKX/CLKR) falling edge (TXM = 0)	20			ns
$t_{h(FS)}$	FSX/FSR hold time after (CLKX/CLKR) falling edge (TXM = 0)	20			ns
$t_{su(DR)}$	DR setup time before CLKR falling edge	20			ns
$t_{h(DR)}$	DR hold time after CLKR falling edge	20			ns

NOTES: 3. $Q = 1/4t_{c(C)}$.
10. The duty cycle of the serial port clock must be within 40-60%.

serial port receive timing



serial port transmit timing



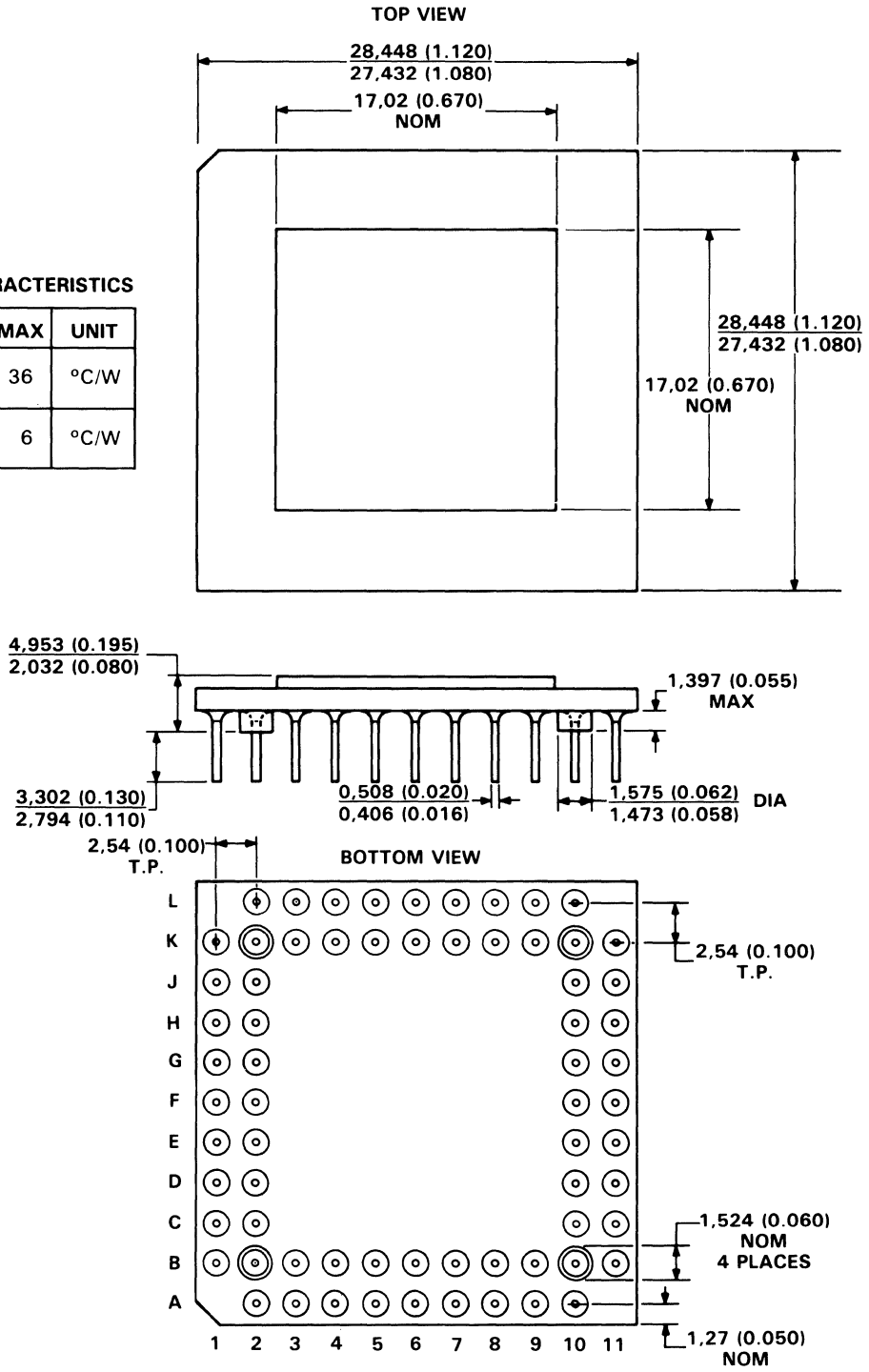
TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

MECHANICAL DATA

68-pin GB pin grid array ceramic package

THERMAL RESISTANCE CHARACTERISTICS

PARAMETER		MAX	UNIT
$R_{\theta JA}$	Junction-to-free-air thermal resistance	36	°C/W
$R_{\theta JC}$	Junction-to-case thermal resistance	6	°C/W



ALL LINEAR DIMENSIONS ARE IN MILLIMETERS AND PARENTHETICALLY IN INCHES

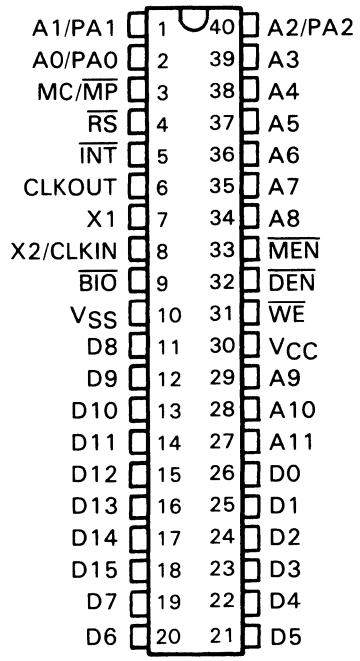
ADVANCE INFORMATION

TMS320C10
DIGITAL SIGNAL PROCESSOR

JANUARY 1986

- 200-ns Instruction Cycle
- 144-Word On-Chip Data RAM
- ROMless Version — TMS320C10
- 1.5K-Word On-Chip Program ROM — TMS320CM10
- External Memory Expansion to a Total of 4K Words at Full Speed
- 16-Bit Instruction/Data Word
- 32-Bit ALU/Accumulator
- 16 × 16-Bit Multiply in 200 ns
- 0 to 16-Bit Barrel Shifter
- Eight Input and Eight Output Channels
- 16-Bit Bidirectional Data Bus with 40-Megabits-per-Second Transfer Rate
- Interrupt with Full Context Save
- Signed Two's-Complement Fixed-Point Arithmetic
- CMOS Technology
- Single 5-V Supply

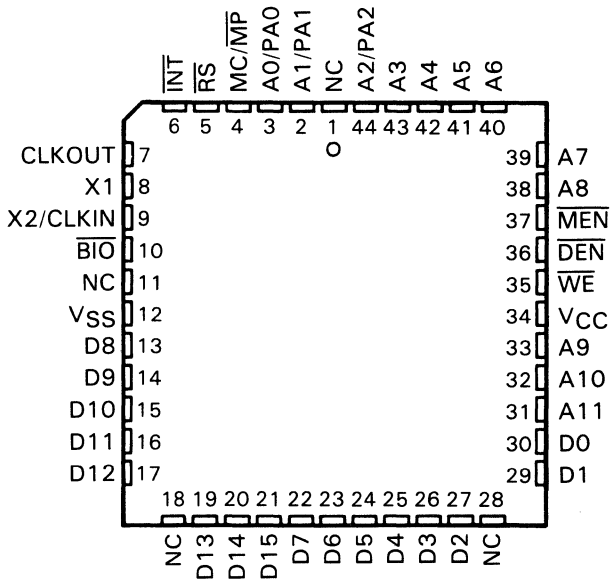
JN PACKAGE
(TOP VIEW)



description

The TMS320C10 is the first low-power CMOS member of the Texas Instruments TMS320 family of Digital Signal Processors. This device is a CMOS pin-for-pin compatible version of the industry-standard TMS32010 Digital Signal Processor. The 100-mW typical power dissipation of the TMS320C10 enables power-sensitive applications to take advantage of the TMS32010's high performance. The 16/32-bit microcomputer was designed to support a wide range of high-speed and numeric-intensive applications. The TMS320C10 combines the flexibility of a high-speed controller with the numerical capability of an array processor, thereby offering an inexpensive alternative to multichip bit-slice processors. The highly pipelined architecture and efficient instruction set of the TMS320C10 provides the capability of executing more than five million instructions per second. The instruction set is easily programmed and contains general-purpose as well as digital signal processing instructions.

FN PACKAGE
(TOP VIEW)



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

PIN NOMENCLATURE

NAME	I/O	DEFINITION
A11-A0/PA2-PA0	O	External address bus. I/O port address multiplexed over PA2-PA0.
$\overline{BIO}$	I	External polling input for bit test and jump operations.
CLKOUT	O	System clock output, ¼ crystal/CLKIN frequency.
D15-D0	I/O	16-bit data bus.
$\overline{DEN}$	O	Data enable indicates the processor accepting input data on D15-D0.
$\overline{INT}$	I	Interrupt.
MC/$\overline{MP}$	I	Memory mode select pin. High selects microcomputer mode. Low selects microprocessor mode.
$\overline{MEN}$	O	Memory enable indicates that D15-D0 will accept external memory instruction.
NC		No connection.
$\overline{RS}$	I	Reset used to initialize the device.
V_{CC}	I	Power.
V_{SS}	I	Ground.
$\overline{WE}$	O	Write enable indicates valid data on D15-D0.
X1	I	Crystal input.
X2/CLKIN	I	Crystal input or external clock input.

The TMS320 family's unique versatility and power give the design engineer a new approach to a variety of complex applications. In addition, these microcomputers are capable of providing the multiple functions often required for a single application. For example, the TMS320 family can enable an industrial robot to synthesize and recognize speech, sense objects with radar or optical intelligence, and perform mechanical operations through digital servo-loop computations.

architecture

The TMS320 family utilizes a modified Harvard architecture for speed and flexibility. In a strict Harvard architecture, program and data memory lie in two separate spaces, permitting a full overlap of the instruction fetch and execution. The TMS320 family's modification of the Harvard architecture allows transfers between program and data spaces, thereby increasing the flexibility of the device. This modification permits coefficients stored in program memory to be read into the RAM, eliminating the need for a separate coefficient ROM. It also makes available immediate instructions and subroutines based on computed values.

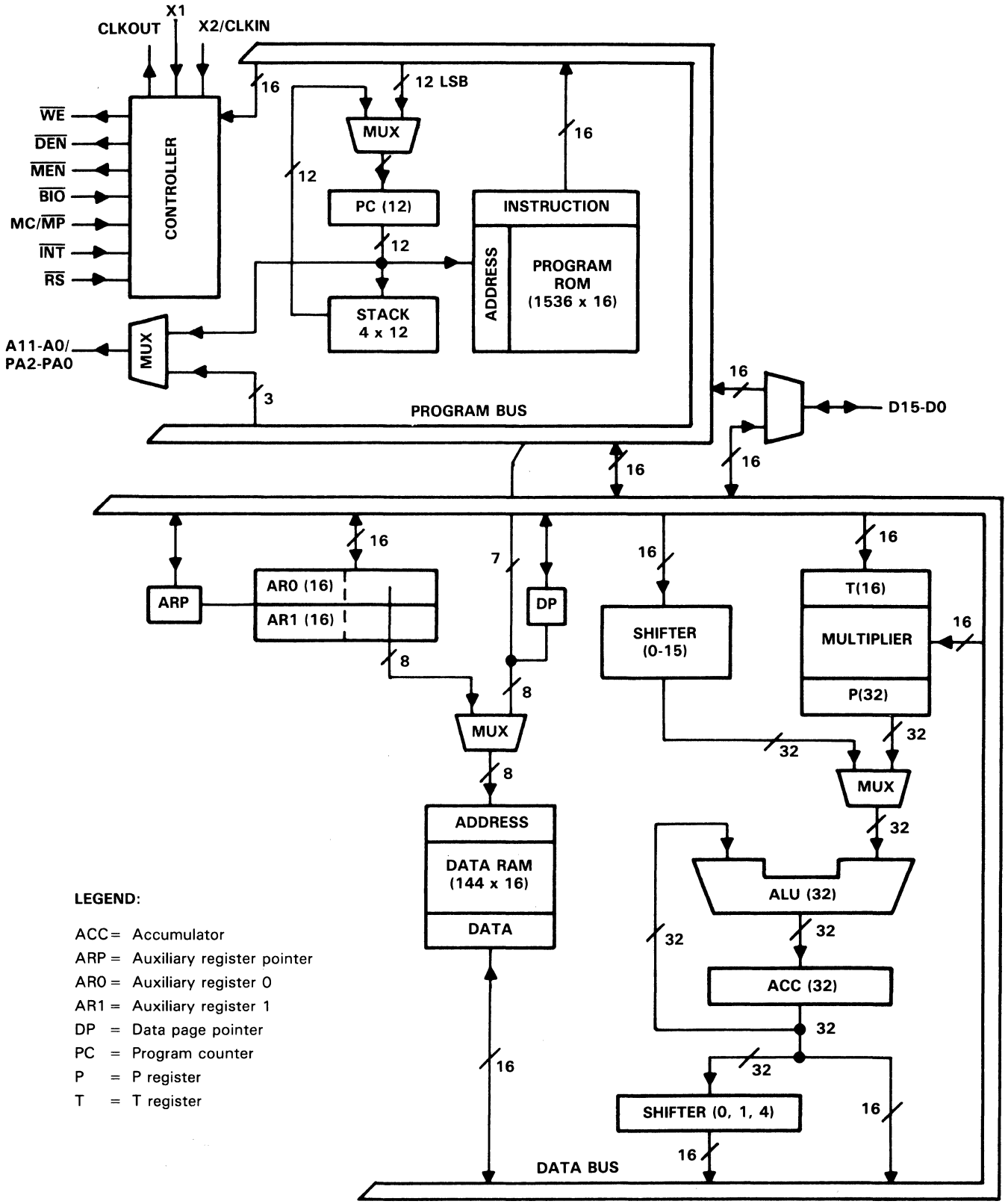
The TMS320C10 utilizes hardware to implement functions that other processors typically perform in software. For example, this device contains a hardware multiplier to perform a multiplication in a single 200-ns cycle. There is also a hardware barrel shifter for shifting data on its way into the ALU. Finally, extra hardware has been included so that auxiliary registers, which provide indirect data RAM addresses, can be configured in an autoincrement/decrement mode for single-cycle manipulation of data tables. This hardware-intensive approach gives the design engineer the type of power previously unavailable on a single chip.

32-bit ALU/accumulator

The TMS320C10 contains a 32-bit ALU and accumulator that support double-precision arithmetic. The ALU operates on 16-bit words taken from the data RAM or derived from immediate instructions. Besides the usual arithmetic instructions, the ALU can perform Boolean operations, providing the bit manipulation ability required of a high-speed controller.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

functional block diagram



shifters

A barrel shifter is available for left-shifting data 0 to 15 places before it is loaded into, subtracted from, or added to the accumulator. This shifter extends the high-order bit of the data word and zero-fills the low-order bits for two's-complement arithmetic. A second shifter left-shifts the upper half of the accumulator 0, 1, or 4 places while it is being stored in the data RAM. Both shifters are useful for scaling and bit extraction.

16 × 16-bit parallel multiplier

The TMS320C10's multiplier performs a 16 × 16-bit, two's-complement multiplication in one 200-ns instruction cycle. The 16-bit T Register temporarily stores the multiplicand; the P Register stores the 32-bit result. Multiplier values either come from the data memory or are derived immediately from the MPYK (multiply immediate) instruction word. The fast on-chip multiplier allows the TMS320C10 to perform such fundamental operations as convolution, correlation, and filtering at the rate of better than 3 million samples per second.

program memory expansion

The TMS320CM10 is equipped with a 1536-word ROM, which is mask-programmed at the factory with a customer's program. It can also execute from an additional 2560 words of off-chip program memory at full speed. This memory expansion capability is especially useful for those situations where a customer has a number of different applications that share the same subroutines. In this case, the common subroutines can be stored on-chip while the application specific code is stored off-chip.

The TMS320CM10 can operate in either of the following memory modes via the MC/$\overline{\text{MP}}$ pin:

Microcomputer Mode (MC) — Instruction addresses 0-1535 fetched from on-chip ROM; instruction addresses 1536-4095 fetched from off-chip memory at full speed.

Microprocessor Mode ($\overline{\text{MP}}$) — Full-speed execution from all 4096 off-chip instruction addresses.

The TMS320C10 is identical to the TMS320CM10, except that the TMS320C10 operates only in the microprocessor mode. Henceforth, TMS320C10 refers to both versions.

The ability of the TMS320C10 to execute at full speed from off-chip memory provides the following important benefits:

- Easier prototyping and development work than possible with a device that can address only on-chip ROM,
- Purchase of a standard off-the-shelf product rather than a semicustom mask-programmed device,
- Ease of updating code,
- Execution from external RAM,
- Downloading of code from another microprocessor, and
- Use of off-chip RAM to expand data storage capability.

input/output

The TMS320C10's 16-bit parallel data bus can be utilized to perform I/O functions at burst rates of 40 million bits per second. Available for interfacing to peripheral devices are 128 input and 128 output bits consisting of eight 16-bit multiplexed input ports and eight 16-bit multiplexed output ports. In addition, a polling input for bit test and jump operations ($\overline{\text{BIO}}$) and an interrupt pin ($\overline{\text{INT}}$) have been incorporated for multitasking.

interrupts and subroutines

The TMS320C10 contains a four-level hardware stack for saving the contents of the program counter during interrupts and subroutine calls. Instructions are available for saving the TMS320C10's complete context. The instructions, PUSH stack from accumulator and POP stack to accumulator, permit a level of nesting restricted only by the amount of available RAM. The interrupts used in the TMS320C10 are maskable.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

instruction set

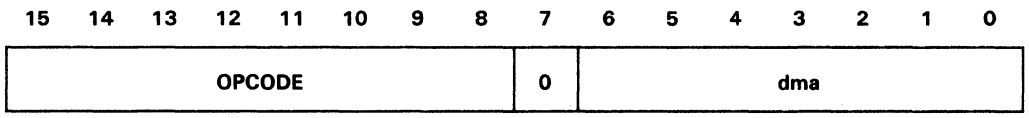
The TMS320C10's comprehensive instruction set supports both numeric-intensive operations, such as signal processing, and general-purpose operations, such as high-speed control. The instruction set, explained in Tables 1 and 2, consists primarily of single-cycle single-word instructions, permitting execution rates of better than 5 million instructions per second. Only infrequently used branch and I/O instructions are multicycle.

The TMS320C10 also contains a number of instructions that shift data as part of an arithmetic operation. These all execute in a single cycle and are useful for scaling data in parallel with other operations.

Three main addressing modes are available with the TMS320C10 instruction set: direct, indirect, and immediate addressing.

direct addressing

In direct addressing, seven bits of the instruction word concatenated with the data page pointer form the data memory address. This implements a paging scheme in which the first page contains 128 words and the second page contains 16 words. In a typical application, infrequently accessed variables, such as those used for servicing an interrupt, are stored on the second page. The instruction format for direct addressing is shown below.



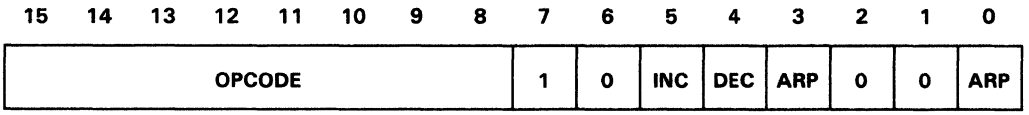
Bit 7 = 0 defines direct addressing mode. The opcode is contained in bits 15 through 8. Bits 6 through 0 contain data memory address.

The seven bits of the data memory address (dma) field can directly address up to 128 words (1 page) of data memory. Use of the data memory page pointer is required to address the full 144 words of data memory.

Direct addressing can be used with all instructions requiring data operands, except for the immediate operand instructions.

indirect addressing

Indirect addressing forms the data memory address from the least significant eight bits of one of two auxiliary registers, AR0 and AR1. The auxiliary register pointer (ARP) selects the current auxiliary register. The auxiliary registers can be automatically incremented or decremented in parallel with the execution of any indirect instruction to permit single-cycle manipulation of data tables. The instruction format for indirect addressing is as follows:



Bit 7 = 1 defines the indirect addressing mode. The opcode is contained in bits 15 through 8. Bits 7 through 0 contain indirect addressing control bits.

Bit 3 and bit 0 control the Auxiliary Register Pointer (ARP). If bit 3 = 0, then the content of bit 0 is loaded into the ARP. If bit 3 = 1, then the content of ARP remains unchanged. ARP = 0 defines the contents of AR0 as memory address. ARP = 1 defines the contents of AR1 as memory address.

Bit 5 and bit 4 control the auxiliary registers. If bit 5 = 1, then the ARP defines which auxiliary register is to be incremented by 1. If bit 4 = 1, then the ARP defines which auxiliary register is to be decremented by 1. If bit 5 and bit 4 are zero, then neither auxiliary register is incremented or decremented. Bits 6, 2, and 1 are reserved and should always be programmed to zero.

Indirect addressing can be used with all instructions requiring data operands, except for the immediate operand instructions.

immediate addressing

The TMS320C10 instruction set contains special "immediate" instructions. These instructions derive data from part of the instruction word rather than from the data RAM. Some useful immediate instructions are multiply immediate (MPYK), load accumulator immediate (LACK), and load auxiliary register immediate (LARK).

TABLE 1. INSTRUCTION SYMBOLS

SYMBOL	MEANING
ACC	Accumulator
D	Data memory address field
I	Addressing mode bit
K	Immediate operand field
PA	3-bit port address field
R	1-bit operand field specifying auxiliary register
S	4-bit left-shift code
X	3-bit accumulator left-shift field

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TABLE 2. TMS320C10 INSTRUCTION SET SUMMARY

ACCUMULATOR INSTRUCTIONS				
MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE INSTRUCTION REGISTER 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ABS	Absolute value of accumulator	1	1	0 1 1 1 1 1 1 1 1 0 0 0 1 0 0 0
ADD	Add to accumulator with shift	1	1	0 0 0 0 ← S → I ← D →
ADDH	Add to high-order accumulator bits	1	1	0 1 1 0 0 0 0 0 I ← D →
ADDS	Add to accumulator with no sign extension	1	1	0 1 1 0 0 0 0 1 I ← D →
AND	AND with accumulator	1	1	0 1 1 1 1 0 0 1 I ← D →
LAC	Load accumulator with shift	1	1	0 0 1 0 ← S → I ← D →
LACK	Load accumulator immediate	1	1	0 1 1 1 1 1 1 0 ← K →
OR	OR with accumulator	1	1	0 1 1 1 1 0 1 0 I ← D →
SACH	Store high-order accumulator bits with shift	1	1	0 1 0 1 1 ← X → I ← D →
SACL	Store low-order accumulator bits	1	1	0 1 0 1 0 0 0 0 I ← D →
SUB	Subtract from accumulator with shift	1	1	0 0 0 1 ← S → I ← D →
SUBC	Conditional subtract (for divide)	1	1	0 1 1 0 0 1 0 0 I ← D →
SUBH	Subtract from high-order accumulator bits	1	1	0 1 1 0 0 0 1 0 I ← D →
SUBS	Subtract from accumulator with no sign extension	1	1	0 1 1 0 0 0 1 1 I ← D →
XOR	Exclusive OR with accumulator	1	1	0 1 1 1 1 0 0 0 I ← D →
ZAC	Zero accumulator	1	1	0 1 1 1 1 1 1 1 1 0 0 0 1 0 0 1
ZALH	Zero accumulator and load high-order bits	1	1	0 1 1 0 0 1 0 1 I ← D →
ZALS	Zero accumulator and load low-order bits with no sign extension	1	1	0 1 1 0 0 1 1 0 I ← D →

AUXILIARY REGISTER AND DATA PAGE POINTER INSTRUCTIONS				
MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE INSTRUCTION REGISTER 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
LAR	Load auxiliary register	1	1	0 0 1 1 1 0 0 R I ← D →
LARK	Load auxiliary register immediate	1	1	0 1 1 1 0 0 0 R ← K →
LARP	Load auxiliary register pointer immediate	1	1	0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 K
LDP	Load data memory page pointer	1	1	0 1 1 0 1 1 1 1 I ← D →
LDPK	Load data memory page pointer immediate	1	1	0 1 1 0 1 1 1 0 0 0 0 0 0 0 0 K
MAR	Modify auxiliary register and pointer	1	1	0 1 1 0 1 0 0 0 I ← D →
SAR	Store auxiliary register	1	1	0 0 1 1 0 0 0 R I ← D →

TABLE 2. TMS320C10 INSTRUCTION SET SUMMARY (CONTINUED)

BRANCH INSTRUCTIONS

MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE INSTRUCTION REGISTER 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
B	Branch unconditionally	2	2	1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BANZ	Branch on auxiliary register not zero	2	2	1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BGEZ	Branch if accumulator ≥ 0	2	2	1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BGZ	Branch if accumulator > 0	2	2	1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BIOZ	Branch on $\overline{BIO}$ = 0	2	2	1 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BLEZ	Branch if accumulator ≤ 0	2	2	1 1 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BLZ	Branch if accumulator < 0	2	2	1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BNZ	Branch if accumulator ≠ 0	2	2	1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BV	Branch on overflow	2	2	1 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BZ	Branch if accumulator = 0	2	2	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
CALA	Call subroutine from accumulator	2	1	0 1 1 1 1 1 1 1 1 0 0 0 1 1 0 0
CALL	Call subroutine immediately	2	2	1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
RET	Return from subroutine or interrupt routine	2	1	0 1 1 1 1 1 1 1 1 0 0 0 1 1 0 1

T REGISTER, P REGISTER, AND MULTIPLY INSTRUCTIONS

MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE INSTRUCTION REGISTER 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
APAC	Add P register to accumulator	1	1	0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1
LT	Load T register	1	1	0 1 1 0 1 0 1 0 I ← D →
LTA	LTA combines LT and APAC into one instruction	1	1	0 1 1 0 1 1 0 0 I ← D →
LTD	LTD combines LT, APAC, and DMOV into one instruction	1	1	0 1 1 0 1 0 1 1 I ← D →
MPY	Multiply with T register, store product in P register	1	1	0 1 1 0 1 1 0 1 I ← D →
MPYK	Multiply T register with immediate operand; store product in P register	1	1	1 0 0 ← K →
PAC	Load accumulator from P register	1	1	0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0
SPAC	Subtract P register from accumulator	1	1	0 1 1 1 1 1 1 1 1 0 0 1 0 0 0 0

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TABLE 2. TMS320C10 INSTRUCTION SET SUMMARY (CONCLUDED)

CONTROL INSTRUCTIONS																		
MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE INSTRUCTION REGISTER														
				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DINT	Disable interrupt	1	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1
EINT	Enable interrupt	1	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	1	0
LST	Load status register	1	1	0	1	1	1	1	0	1	1	I	◄			D			►
NOP	No operation	1	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
POP	POP stack to accumulator	2	1	0	1	1	1	1	1	1	1	1	0	0	1	1	1	0	1
PUSH	PUSH stack from accumulator	2	1	0	1	1	1	1	1	1	1	1	0	0	1	1	1	0	0
ROVM	Reset overflow mode	1	1	0	1	1	1	1	1	1	1	1	0	0	0	1	0	1	0
SOVM	Set overflow mode	1	1	0	1	1	1	1	1	1	1	1	0	0	0	1	0	1	1
SST	Store status register	1	1	0	1	1	1	1	1	0	0	I	◄			D			►

I/O AND DATA MEMORY OPERATIONS																		
MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE INSTRUCTION REGISTER														
				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DMOV	Copy contents of data memory location into next location	1	1	0	1	1	0	1	0	0	1	I	◄			D			►
IN	Input data from port	2	1	0	1	0	0	0	◄	PA	►	I	◄			D			►
OUT	Output data to port	2	1	0	1	0	0	1	◄	PA	►	I	◄			D			►
TBLR	Table read from program memory to data RAM	3	1	0	1	1	0	0	1	1	1	I	◄			D			►
TBLW	Table write from data RAM to program	3	1	0	1	1	1	1	1	0	1	I	◄			D			►

development systems and software support

Texas Instruments offers concentrated development support and complete documentation for designing a TMS32010-based microprocessor system. When developing an application, tools are provided to evaluate the performance of the processor, to develop the algorithm implementation, and to fully integrate the design's software and hardware modules. When questions arise, additional support can be obtained by calling the nearest Texas Instruments Regional Technology Center (RTC).

Sophisticated development operations are performed with the TMS32010 Evaluation Module (EVM), Macro Assembler/Linker, Simulator, and Emulator (XDS). In the initial phase of developing an application, the evaluation module is used to characterize the performance of the TMS320C10. Once this evaluation phase is completed, the macro assembler and linker are used to translate program modules into object code and link them together. This puts the program modules into a form that can be loaded into the TMS32010 Evaluation Module, Simulator, or Emulator. The simulator provides a quick means for initially debugging TMS320C10 software while the emulator provides real-time in-circuit emulation necessary to perform system level debug efficiently.

A complete list of TMS320C10 software and hardware development tools is given in Table 3.

TABLE 3. TMS320C10 SOFTWARE AND HARDWARE SUPPORT

HOST COMPUTER	OPERATING SYSTEM	PART NUMBER
TMS32010 MACRO ASSEMBLERS/LINKERS		
DEC VAX	VMS	TMDS3240210-08
TI/IBM PC	MS/PC-DOS	TMDS3240810-02
TMS32010 SIMULATORS		
DEC VAX	VMS	TMDS3240211-08
TI/IBM PC	MS/PC-DOS	TMDS3240811-02
TMS32010 DIGITAL FILTER DESIGN PACKAGE (DFDP)		
TI PC	MS-DOS	DFDP-TI001
IBM PC	PC-DOS	DFDP-IBM001
TMS32010 HARDWARE		
Evaluation Module (EVM)		RTC/EVM320A-03
Analog Interface Board (AIB)		RTC/EVM320C-06
Emulator:		
XDS/22		TMDS3262210
Enhanced XDS/22 (available early 1986)		TMDS3262211

absolute maximum ratings over specified temperature range (unless otherwise noted)†

Supply voltage, V_{CC}‡ . . . . . −0.3 V to 7 V
All input voltages . . . . . −0.3 V to 15 V
Output voltage . . . . . −0.3 V to 15 V
Continuous power dissipation . . . . . 0.4 W
Air temperature range above operating device . . . . . 0°C to 70°C
Storage temperature range . . . . . −55°C to +150°C

†Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions beyond those indicated in the "Recommended Operating Conditions" section of this specification is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.
‡All voltage values are with respect to V_{SS}.

recommended operating conditions

			MIN	NOM	MAX	UNIT
V_{CC}	Supply voltage		4.5	5	5.5	V
V_{SS}	Supply voltage			0		V
V_{IH}	High-level input voltage	All inputs except CLKIN	2			V
		CLKIN	2.8			
V_{IL}	Low-level input voltage (all inputs)				0.8	V
I_{OH}	High-level output current (all outputs)				300	μA
I_{OL}	Low-level output current (all outputs)				2	mA
T_A	Operating free-air temperature		0		70	°C

NOTE 1. For dual-in-line package:
$R_{\theta JA}$ = 51.6°C/Watt
$R_{\theta JC}$ = 16.6°C/Watt.
For plastic chip-carrier package:
$R_{\theta JA}$ = 70°C/Watt
$R_{\theta JC}$ = 20°C/Watt.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

electrical characteristics over specified temperature range (unless otherwise noted)

PARAMETER			TEST CONDITIONS		MIN	TYP†	MAX	UNIT
V_{OH}	High-level output voltage		I_{OH} = MAX		2.4	3		V
V_{OL}	Low-level output voltage		I_{OL} = MAX			0.3	0.5	V
I_{OZ}	Off-state output current		V_{CC} = MAX	V_O = 2.4 V			20	μA
				V_O = 0.4 V			−20	
I_I	Input current		V_I = V_{SS} to V_{CC}				±50	μA
I_{CC}‡	Supply current		T_A = 0°C			20		mA
C_i	Input capacitance	Data bus	f = 1 MHz, All other pins 0 V			25		pF
		All others				15		
C_o	Output capacitance	Data bus				25		pF
		All others				10		

†All typical values except for I_{CC} are at V_{CC} = 5 V, T_A = 25°C.
‡I_{CC} characteristics are inversely proportional to temperature; i.e., I_{CC} decreases approximately linearly with temperature.
§Value derived from characterization data and not tested.

CLOCK CHARACTERISTICS AND TIMING

The TMS320C10 can use either its internal oscillator or an external frequency source for a clock.

internal clock option

The internal oscillator is enabled by connecting a crystal across X1 and X2/CLKIN (see Figure 1). The frequency of CLKOUT is one-fourth the crystal fundamental frequency. The crystal should be fundamental mode, and parallel resonant, with an effective series resistance of 30 ohms, a power dissipation of 1 mW, and be specified at a load capacitance of 20 pF.

PARAMETER	TEST CONDITIONS	MIN	NOM	MAX	UNIT
Crystal frequency f_X	0°C – 70°C	6.7		20.5	MHz
C1, C2	0°C – 70°C		10		pF

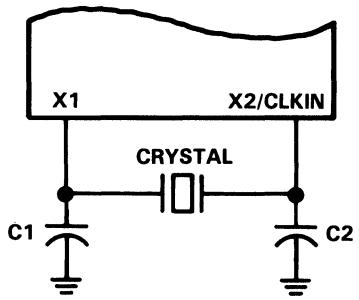


FIGURE 1. INTERNAL CLOCK OPTION

external clock option

An external frequency source can be used by injecting the frequency directly into X2/CLKIN with X1 left unconnected. The external frequency injected must conform to the specifications listed in the table below.

timing requirements over recommended operating conditions

		MIN	NOM	MAX	UNIT
$t_{c(MC)}$	Master clock cycle time	48.78		150	ns
$t_{r(MC)}$	Rise time master clock input		5	10	ns
$t_{f(MC)}$	Fall time master clock input		5	10	ns
$t_{w(MCP)}$	Pulse duration master clock	$0.475t_{c(C)}$		$0.525t_{c(C)}$	ns
$t_{w(MCL)}$	Pulse duration master clock low, $t_{c(MC)}$ = 50 ns		20		ns
$t_{w(MCH)}$	Pulse duration master clock high, $t_{c(MC)}$ = 50 ns		20		ns

switching characteristics over recommended operating conditions

PARAMETER		TEST CONDITIONS	MIN	NOM	MAX	UNIT
$t_{c(C)}$	CLKOUT cycle time†	R_L = 825 Ω, C_L = 100 pF, See Figure 2	195.12			ns
$t_{r(C)}$	CLKOUT rise time			10		ns
$t_{f(C)}$	CLKOUT fall time			8		ns
$t_{w(CL)}$	Pulse duration, CLKOUT low			92		ns
$t_{w(CH)}$	Pulse duration, CLKOUT high			90		ns
$t_{d(MCC)}$	Delay time CLKIN↑ to CLKOUT↓‡		25		60	ns

†$t_{c(C)}$ is the cycle time of CLKOUT, i.e., 4*$t_{c(MC)}$ (4 times CLKIN cycle time if an external oscillator is used).
‡Values given were derived from characterization data and are not tested.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

PARAMETER MEASUREMENT INFORMATION

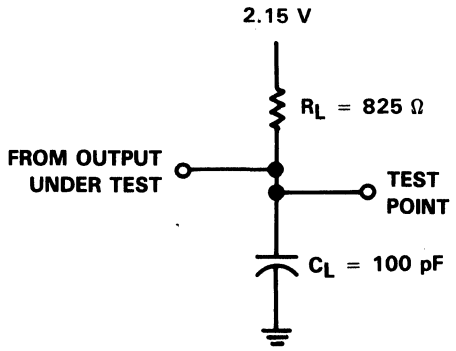
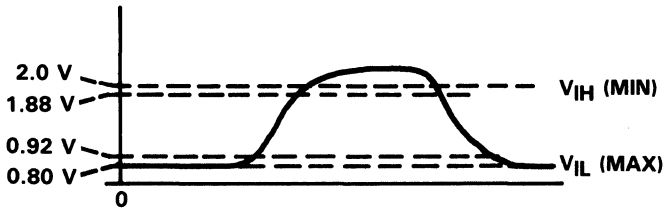
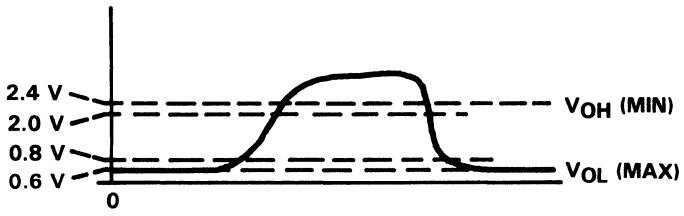


FIGURE 2. TEST LOAD CIRCUIT



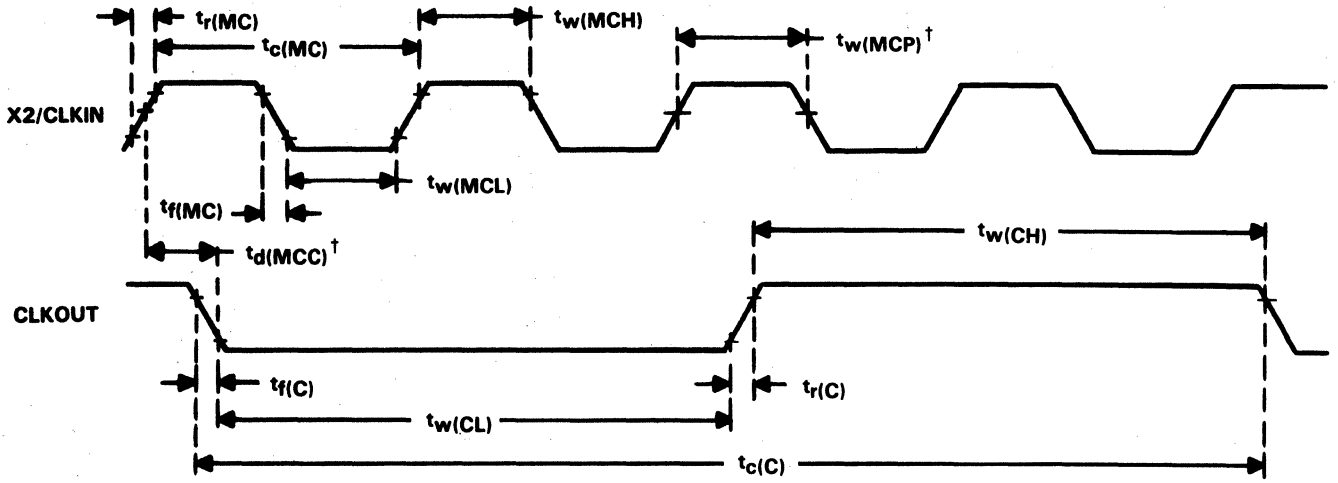
(a) INPUT



(b) OUTPUTS

FIGURE 3. VOLTAGE REFERENCE LEVELS

clock timing



NOTE 2: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted. †$t_{d(MCC)}$ and $t_{w(MCP)}$ are referenced to an intermediate level of 1.5 volts on the CLKIN waveform.

MEMORY AND PERIPHERAL INTERFACE TIMING

switching characteristics over recommended operating conditions

PARAMETER		TEST CONDITIONS	MIN	TYP	MAX	UNIT
t_{d1}	Delay time CLKOUT↓ to address bus valid (see Note 4)	R_L = 825 Ω, C_L = 100 pF, See Figure 2	10†		50	ns
t_{d2}	Delay time CLKOUT↓ to $\overline{MEN}$↓		$¼t_{c(C)}-5$†		$¼t_{c(C)}+15$	ns
t_{d3}	Delay time CLKOUT↓ to $\overline{MEN}$↑		−10†		15	ns
t_{d4}	Delay time CLKOUT↓ to $\overline{DEN}$↓		$¼t_{c(C)}-5$†		$¼t_{c(C)}+15$	ns
t_{d5}	Delay time CLKOUT↓ to $\overline{DEN}$↑		−10†		15	ns
t_{d6}	Delay time CLKOUT↓ to $\overline{WE}$↓		$½t_{c(C)}-5$†		$½t_{c(C)}+15$	ns
t_{d7}	Delay time CLKOUT↓ to $\overline{WE}$↑		−10†		15	ns
t_{d8}	Delay time CLKOUT↓ to data bus OUT valid				$¼t_{c(C)}+65$	ns
t_{d9}	Time after CLKOUT↓ that data bus starts to be driven		$¼t_{c(C)}-5$†			ns
t_{d10}	Time after CLKOUT↓ that data bus stops being driven				$¼t_{c(C)}+30$†	ns
t_v	Data bus OUT valid after CLKOUT↓		$¼t_{c(C)}-10$			ns

NOTE 3: Address bus will be valid upon $\overline{WE}$↑, $\overline{DEN}$↑, or $\overline{MEN}$↑.
†These values were derived from characterization data and are not tested.

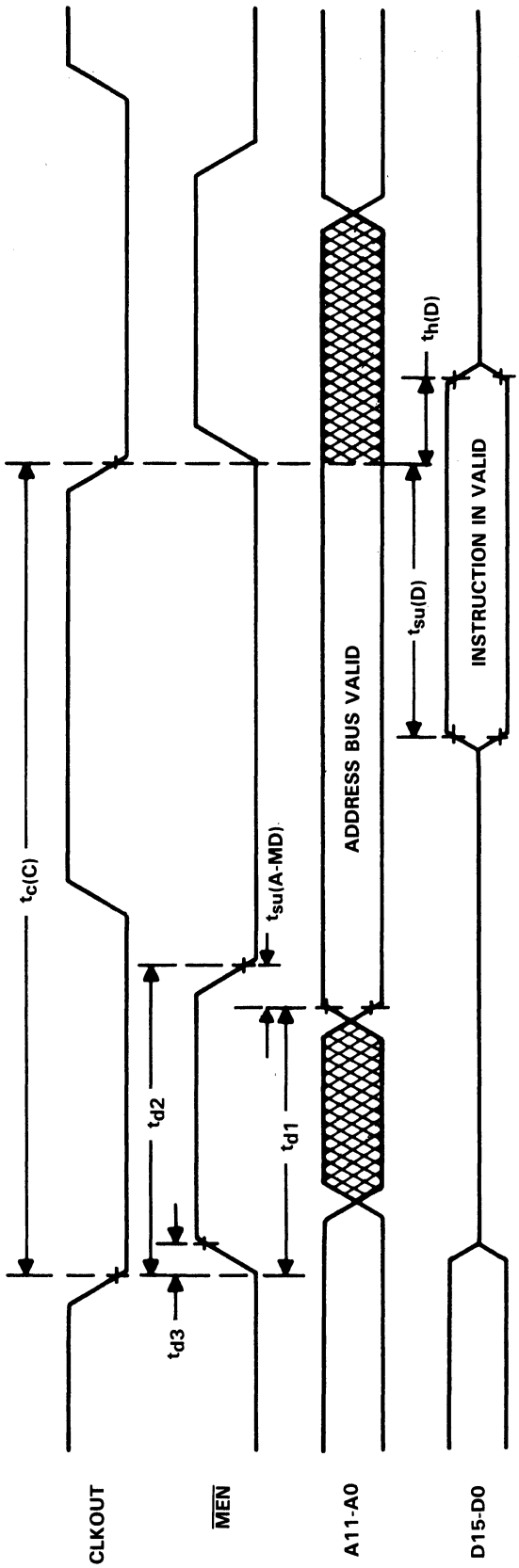
timing requirements over recommended operating conditions

		TEST CONDITIONS	MIN	NOM	MAX	UNIT
$t_{su(D)}$	Setup time data bus valid prior to CLKOUT↓	R_L = 825 Ω, C_L = 100 pF, See Figure 2	50			ns
$t_{su(A-MD)}$	Address bus setup time prior to $\overline{MEN}$↓ or $\overline{DEN}$↓		5			ns
$t_{h(D)}$	Hold time data bus held valid after CLKOUT↓		0			ns

NOTE 4: Data may be removed from the data bus upon $\overline{MEN}$↑ or $\overline{DEN}$↑ preceding CLKOUT↓.

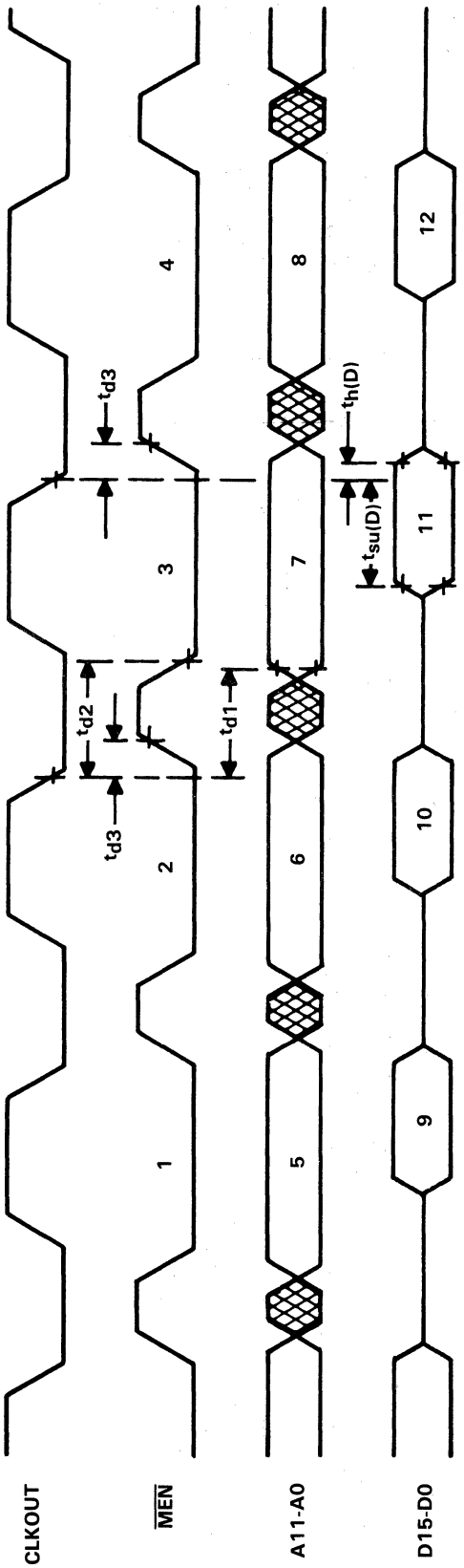
TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

memory read



NOTE 2: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

TBLR instruction timing



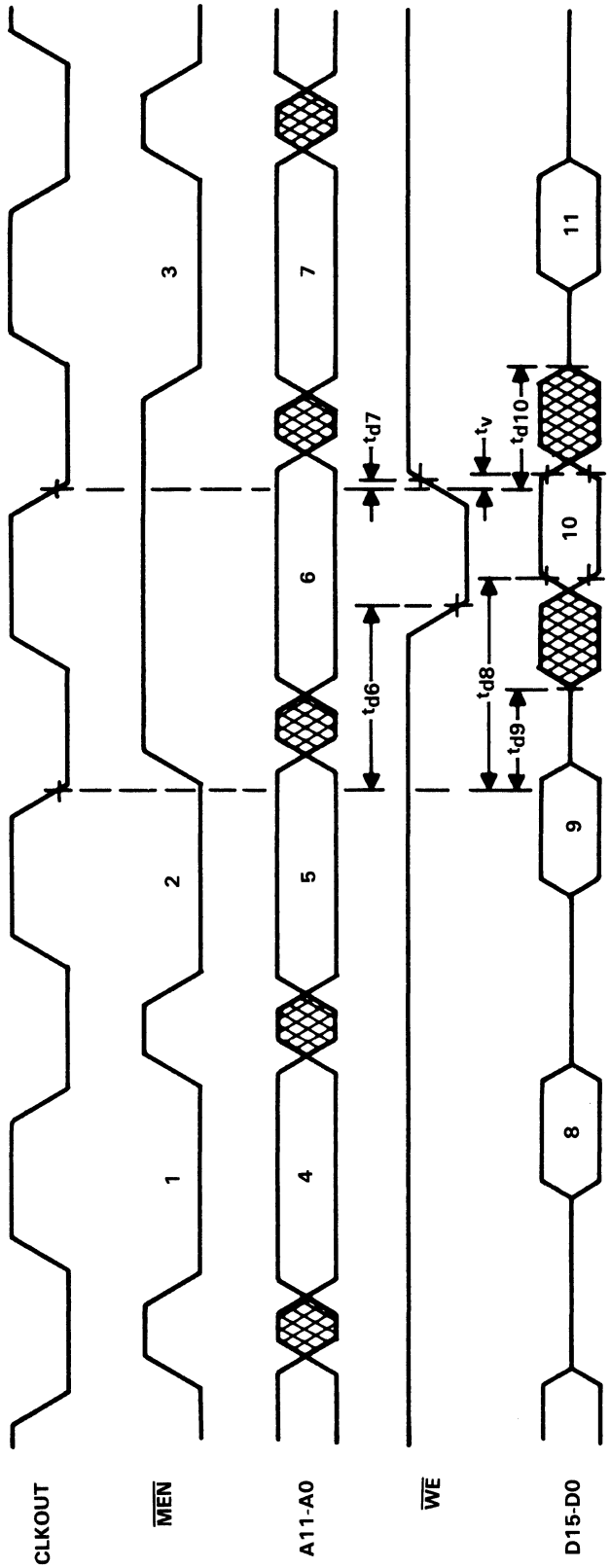
LEGEND:

1. TBLR INSTRUCTION PREFETCH
2. DUMMY PREFETCH
3. DATA FETCH
4. NEXT INSTRUCTION PREFETCH
5. ADDRESS BUS VALID
6. ADDRESS BUS VALID
7. ADDRESS BUS VALID
8. ADDRESS BUS VALID
9. INSTRUCTION IN VALID
10. INSTRUCTION IN VALID
11. DATA IN VALID
12. INSTRUCTION IN VALID

NOTE 2: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TBLW instruction timing

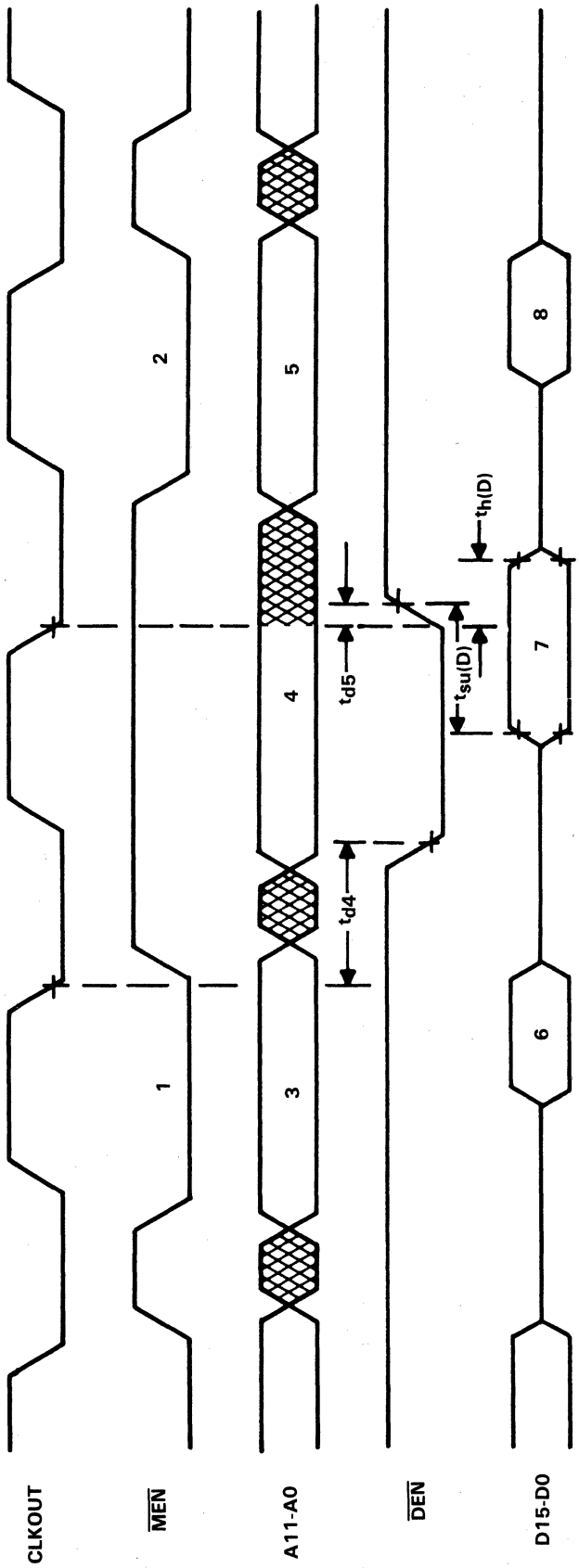


LEGEND:

1. TBLW INSTRUCTION PREFETCH
2. DUMMY PREFETCH
3. NEXT INSTRUCTION PREFETCH
4. ADDRESS BUS VALID
5. ADDRESS BUS VALID
6. ADDRESS BUS VALID
7. ADDRESS BUS VALID
8. INSTRUCTION IN VALID
9. INSTRUCTION IN VALID
10. DATA OUT VALID
11. INSTRUCTION IN VALID

NOTE 2: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

IN instruction timing



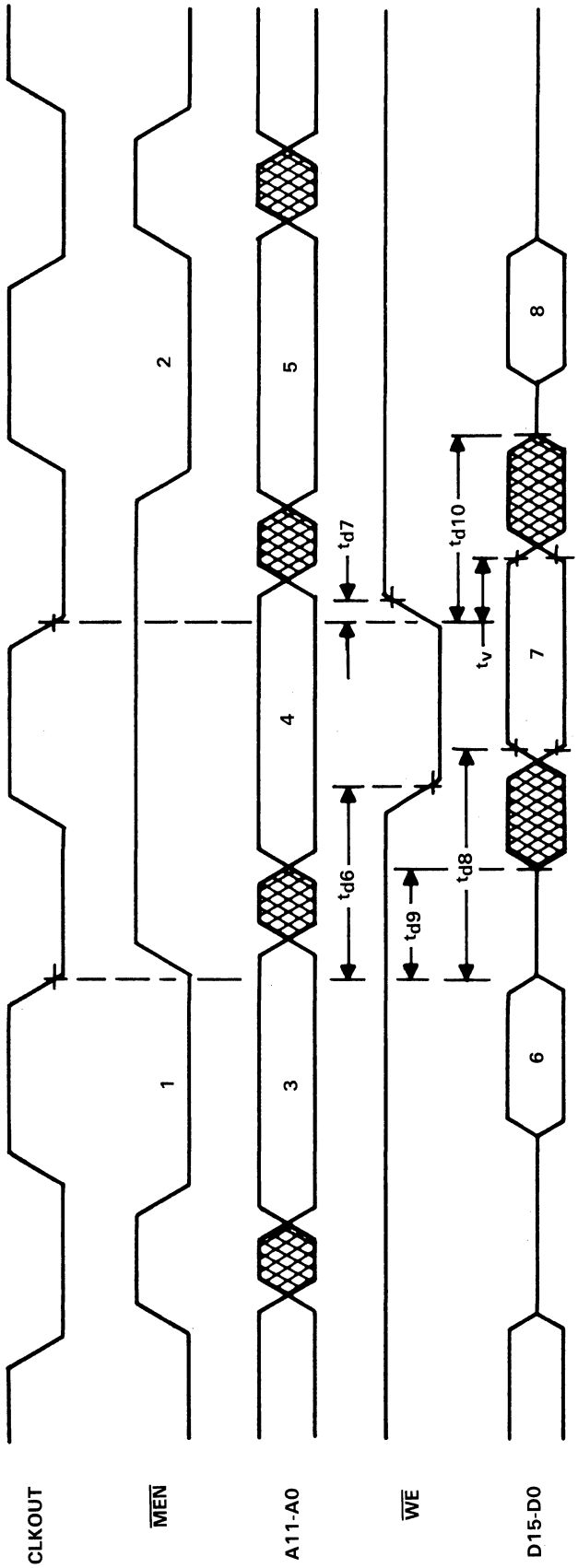
LEGEND:

1. IN INSTRUCTION PREFETCH
2. NEXT INSTRUCTION PREFETCH
3. ADDRESS BUS VALID
4. PERIPHERAL ADDRESS VALID
5. ADDRESS BUS VALID
6. INSTRUCTION IN VALID
7. DATA IN VALID
8. INSTRUCTION IN VALID

NOTE 2: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

OUT instruction timing



LEGEND:

1. OUT INSTRUCTION PREFETCH
2. NEXT INSTRUCTION PREFETCH
3. ADDRESS BUS VALID
4. PERIPHERAL ADDRESS VALID
5. ADDRESS BUS VALID
6. INSTRUCTION IN VALID
7. DATA OUT VALID
8. INSTRUCTION IN VALID

NOTE 2: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

RESET ($\overline{RS}$) TIMING

timing requirements over recommended operating conditions

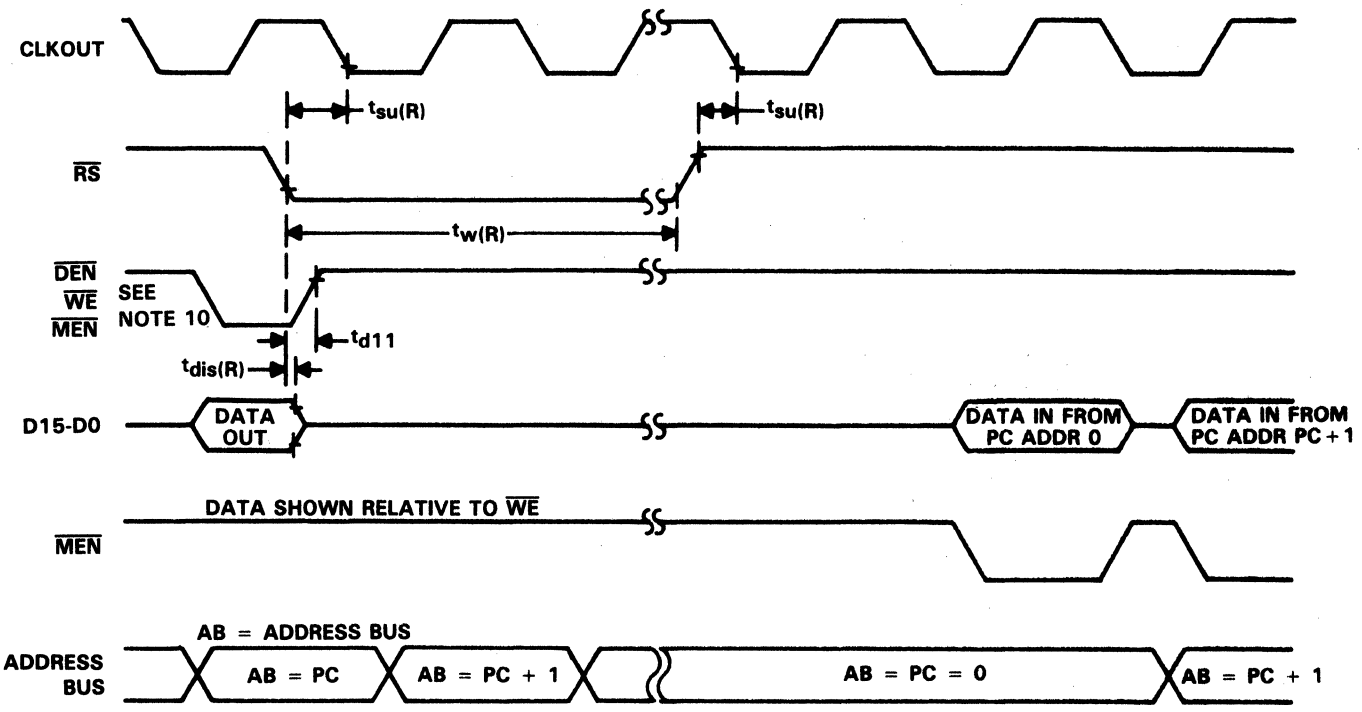
		MIN	NOM	MAX	UNIT
$t_{su(R)}$	Reset ($\overline{RS}$) setup time prior to CLKOUT. See Note 5.	50			ns
$t_{w(R)}$	$\overline{RS}$ pulse duration	$5t_{c(C)}$			ns

switching characteristics over recommended operating conditions

PARAMETER		TEST CONDITIONS	MIN	TYP	MAX	UNIT
t_{d11}	Delay time $\overline{DEN}$↑, $\overline{WE}$↑, and $\overline{MEN}$↑ from $\overline{RS}$	R_L = 825 Ω, C_L = 100 pF, See Figure 2			$\frac{1}{2}t_{c(C)}+50^{\dagger}$	ns
$t_{dis(R)}$	Data bus disable time after $\overline{RS}$				$\frac{1}{4}t_{c(C)}+50^{\dagger}$	ns

NOTE 5: $\overline{RS}$ can occur anytime during a clock cycle. Time given is minimum to ensure synchronous operation.
†These values were derived from characterization data and are not tested.

reset timing



NOTES:
2. Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.
6. $\overline{RS}$ forces $\overline{DEN}$, $\overline{WE}$, and $\overline{MEN}$ high and tristates data bus D0 through D15. AB outputs (and program counter) are synchronously cleared to zero after the next complete CLK cycle from ↓$\overline{RS}$.
7. $\overline{RS}$ must be maintained for a minimum of five clock cycles.
8. Resumption of normal program will commence after one complete CLK cycle from ↑$\overline{RS}$.
9. Due to the synchronizing action on $\overline{RS}$, time to execute the function can vary dependent upon when ↑$\overline{RS}$ or ↓$\overline{RS}$ occur in the CLK cycle.
10. Diagram shown is for definition purpose only. $\overline{DEN}$, $\overline{WE}$, and $\overline{MEN}$ are mutually exclusive.
11. During a write cycle, $\overline{RS}$ may produce an invalid write address.

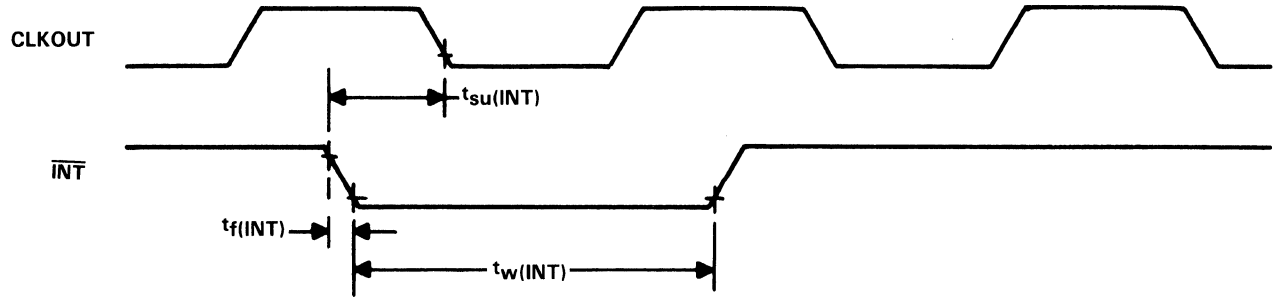
TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

INTERRUPT ($\overline{INT}$) TIMING

timing requirements over recommended operating conditions

		MIN	NOM	MAX	UNIT
$t_{f(INT)}$	Fall time $\overline{INT}$			15	ns
$t_{w(INT)}$	Pulse duration $\overline{INT}$	$t_{c(C)}$			ns
$t_{su(INT)}$	Setup time $\overline{INT}\downarrow$ before CLKOUT$\downarrow$	50			ns

interrupt timing



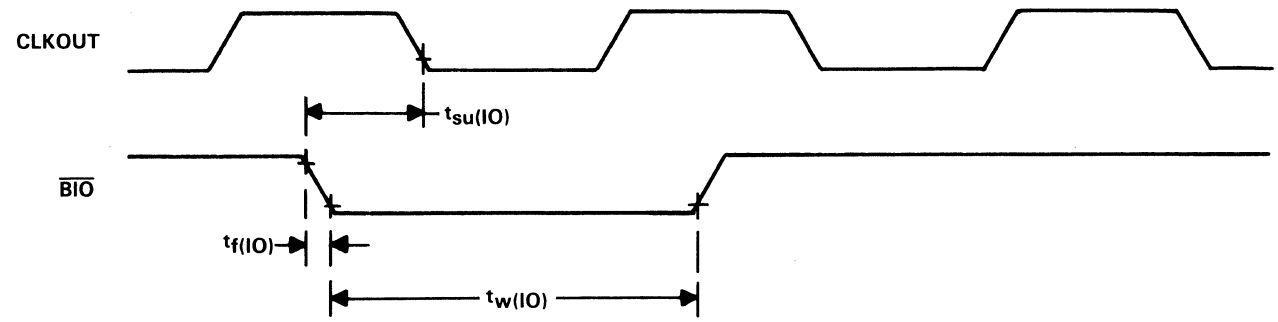
NOTE 2: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

I/O ($\overline{BIO}$) TIMING

timing requirements over recommended operating conditions

		MIN	NOM	MAX	UNIT
$t_{f(IO)}$	Fall time $\overline{BIO}$			15	ns
$t_{w(IO)}$	Pulse duration $\overline{BIO}$	$t_{c(C)}$			ns
$t_{su(IO)}$	Setup time $\overline{BIO}\downarrow$ before CLKOUT$\downarrow$	50			ns

$\overline{BIO}$ timing



NOTE 2: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

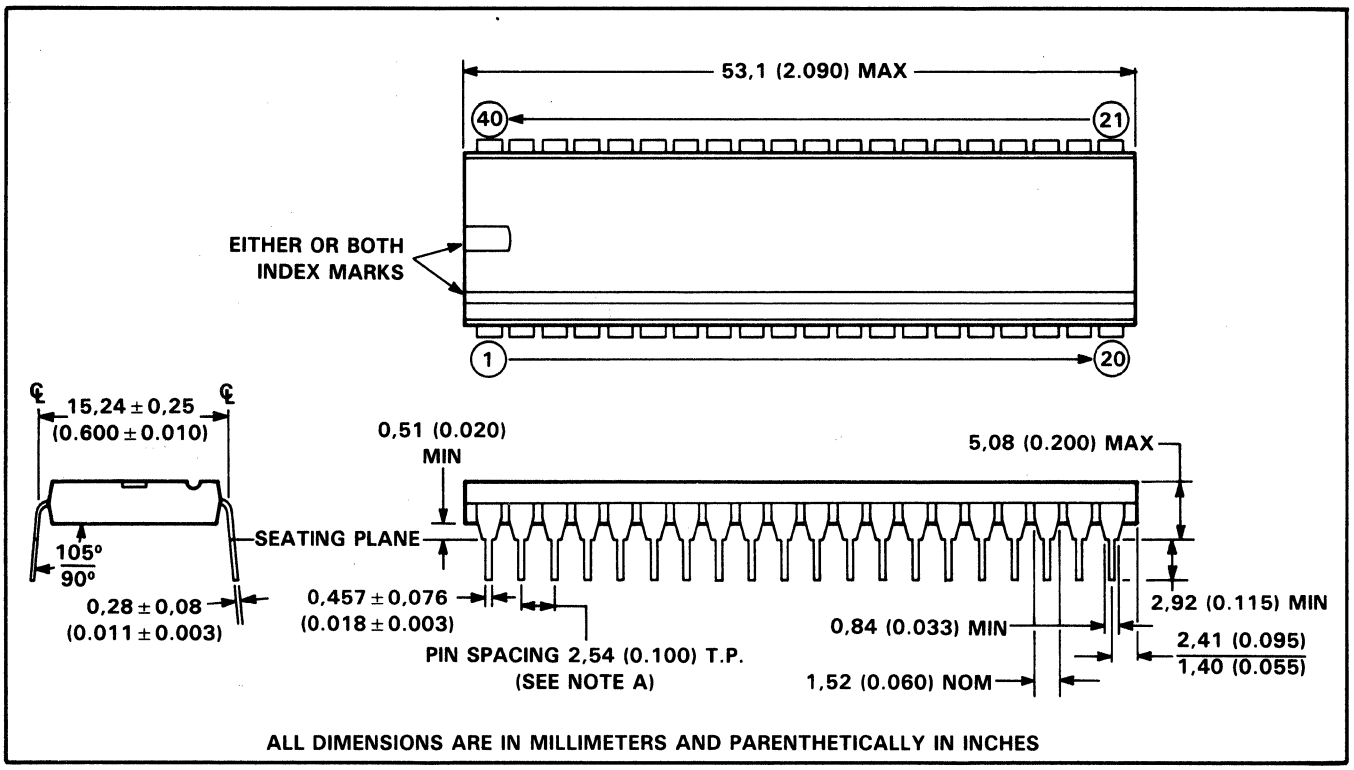
THERMAL DATA

thermal resistance characteristics

PACKAGE	$R_{\theta JA}$ (°C/W)	$R_{\theta JC}$ (°C/W)
40-pin plastic dual-in-line package	51.6	16.6
44-lead plastic chip carrier package	70	20

MECHANICAL DATA

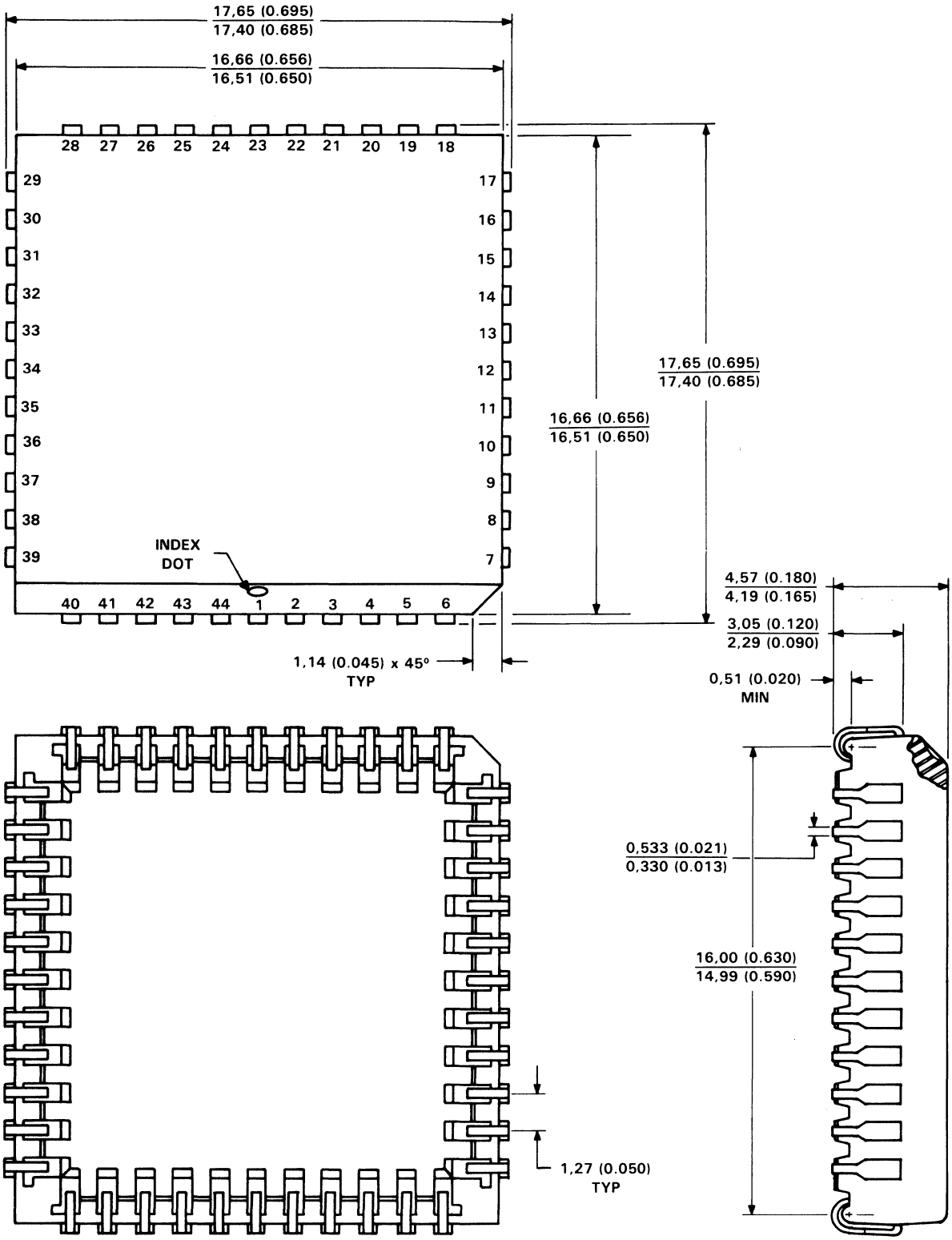
40-pin plastic dual-in-line package



NOTE A: Each pin centerline is located within 0,254 (0.010) of its true longitudinal position.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

44-lead plastic chip carrier package



ALL DIMENSIONS ARE IN MILLIMETERS AND PARENTHETICALLY IN INCHES

D. TMS32020/TMS320C25 System Migration

This appendix contains information necessary to upgrade a TMS32020 program to a TMS320C25-based system. The information consists of a detailed list of the programming differences and hardware and timing differences between the two processors. The following items should be considered in migrating from the TMS32020 to the TMS320C25:

1) Instructions are fully compatible at the object code level. TMS32020 object (memory image) code can be used directly on the TMS320C25 processor.

2) Instructions are compatible at the source code level. The NORM instruction that previously had no operands now has an optional operand to define the auxiliary register modification. Any comments on the same line in the source code file will be interpreted as the operand if no other operand is specified. NORM instructions should be modified to specify the default operand, *+.

3) Execution cycle timings of instructions have been modified. Most TMS320C25 instructions execute in a single machine cycle. The number of cycles for some multicycle instructions have been changed. Refer to Appendix E for detailed information on instruction cycle timings. By following the entries in this appendix, the key timing differences can be noted.

4) The IDLE instruction automatically sets the INTM bit in status register ST0 to a zero. This assures that an external interrupt will 'wake up' the processor. The instruction also requires three memory cycles to execute on the TMS320C25 rather than one as on the TMS32020.

5) In general, all branch, call, and return instructions that reload the program counter (PC) should be counted as three-cycle instructions when evaluating code execution timings on the TMS320C25.

6) The store instructions (SACH, SACL, etc.) execute in one less cycle on the TMS320C25 than on the TMS32020 when data is stored to external data memory.

7) The MAC and MACD instructions require one extra cycle, going from three to four cycles. The extra cycle is in the instruction read and setup overhead, and repeated execution will be one cycle per execution as on the TMS32020.

8) The delay for a new memory configuration to become effective when using the CNFD or CNFP instructions on the TMS320C25 is two instruction fetches (for single-cycle instructions) when executing from external memory or internal ROM, as compared to one instruction fetch for the TMS32020. Thus, on the TMS320C25, a CNFP instruction must be placed at location 65277 if execution is to continue from the first location in block B0. When execution is from internal RAM on the TMS320C25, however, this delay is one instruction fetch as on the TMS32020.

9) The timer on the TMS320C25 is clocked by CLKOUT1 and counts PRD + 1 CLKOUT1 cycles, whereas the timer on the TMS32020 is clocked by CLKOUT1/4 and counts 4 × PRD cycles. Therefore, to count an equivalent amount of time on the TMS320C25 using the same input clock frequency, PRD values from the TMS32020 must first be multiplied by four and then decremented by one. If different input clock frequencies are used, this must also be accounted for by multiplying the PRD value for the TMS320C25 obtained above by the ratio of the TMS320C25 input clock frequency to the TMS32020 input clock frequency.

10) On the TMS320C25, both the timer (TIM) and period (PRD) registers are initialized to >FFFF on reset, while on the TMS32020, only the TIM register is initialized.

11) Several bits (C, HM, and FSM) have been added to status register ST1 on the TMS320C25, as shown below.

TMS32020 Status Register ST1:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARB			CNF	TC	SXM	1	1	1	1	1	XF	FO	TXM	PM	

TMS320C25 Status Register ST1:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARB			CNF	TC	SXM	C	1	1	HM	FSM	XF	FO	TXM	PM	

The FSM, HM, and C status register bits are initialized by reset and are all set to one when reset occurs. Note that the new bits are assigned polarities in such a way that the values of the corresponding bits on the TMS32020 invoke a TMS32020-like operation on the TMS320C25.

The SXM and PM status register bits that were previously uninitialized on the TMS32020 are now initialized by reset on the TMS320C25. When the TMS320C25 is reset, SXM is set to one, and the PM bits are set to zero.

12) There are four differences between the serial ports on the TMS32020 and TMS320C25 that impact system migration. The two major differences are that the serial port on the TMS320C25 is double-buffered and is fully static in operation. Double-buffering greatly increases the amount of time available for processing serial port interrupts. Fully static operation effectively places no lower limit on serial port clock frequency. Neither of these features is present on the TMS32020.

Another difference in serial port operation between the two processors is that serial port interrupts are generated half of a CLKR or CLKX cycle later on the TMS320C25 than they are on the TMS32020. Specifically, on the TMS32020, RINT and XINT are generated on the falling edge of CLKR and CLKX, respectively, during transfer of the last bit. On the TMS320C25, RINT and XINT are generated on the rising edge of CLKR or CLKX after the last bit has been transferred. This should not be critical for TMS32020 programs running on the TMS320C25 since double-buffering of the serial port on the TMS320C25 allows more time for processing of serial port interrupts. Some modification of TMS32020 programs may, however, be required to take advantage of the double-buffering, depending on how serial port interrupt servicing is implemented.

Finally, when operating the TMS320C25 serial port in byte mode, DRR behaves differently than it does on the TMS32020. On the TMS32020, the contents of the most significant byte of DRR remain unchanged once byte mode is initiated by executing a FORT instruction. On the TMS320C25, however, each time a new byte is received, the previous contents of the least significant byte of DRR are transferred to the most significant byte of DRR. Figure D-1 illustrates the behavior of DRR on both the TMS32020 and the TMS320C25 processors.

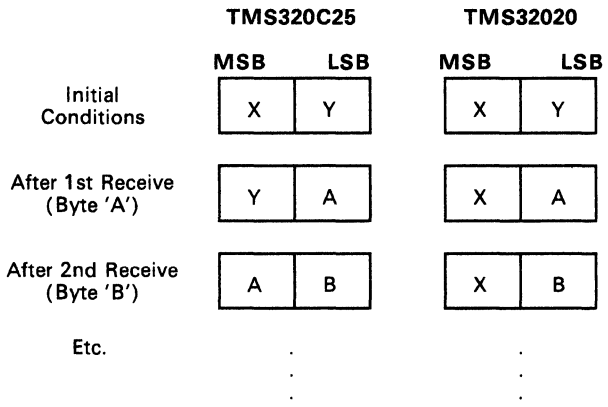


Figure D-1. Serial Port System Migration

E. TMS320C25 Instruction Cycle Timings

This appendix details the instruction cycle timings for the TMS320C25. Table E-1 lists the instructions according to cycle classification.

Table E-1. TMS320C25 Instructions by Cycle Class

CLASS	INSTRUCTION
I	ADD ADDC ADDH ADDS ADDT AND BIT BITT DMOV LAC LACT LPH LT LTA LTD LTP LTS MPY MPYA MPYS MPYU PSHD OR RPT SQRA SQRS SUB SUBB SUBC SUBH SUBS SUBT XOR ZALH ZALR ZALS (RPT not repeatable)
II	LAR LDP LST LST1
III	POPD SACH SACL SAR SPH SPL SST SST1
IV	ABS ADDK ADRK APAC CMPL CMPR CNFD CNFP DINT EINT FORT LACK LARK LARP LDPK MAR MPYK NEG NOP NORM PAC POP PUSH RC RFSM RHM ROL ROR ROVM RPTK RSXM RTC RTXM RXF SBRK SC SFL SFR SFSM SHM SOVM SPAC SPM SSXM STC STXM SUBK SXF ZAC (ADDK, ADRK, LACK, LARK, LDPK, MPYK, RPTK, SBRK, SPM, SUBK, and ZAC not repeatable)
V	ADLK ANDK LALK LRLK ORK SBLK XORK (All not repeatable)
VI	MAC MACD
VII	BANZ BBNZ BBZ BC BGEZ BGZ BIOZ BLEZ BLZ BNC BNV BNZ BV BZ (All not repeatable)
VIII	B BACC CALA CALL RET TRAP (All not repeatable)
IX	IN
X	OUT
XI	TBLR
XII	TBLW (Table in ROM not applicable)
XIII	BLKD
XIV	BLKP
XV	IDLE

Table E-2 and Table E-3 show the number of cycles required for a given TMS320C25 instruction to execute in a given memory configuration. The column headings in the tables indicate the program source location and data destination or source.

The number of cycles required for each instruction is given in terms of the program/data memory and I/O access times as defined in the following listing:

p - Program memory wait states. Represents the number of clock cycles the device waits for external program memory to respond to an access. T_{ac} is the access time, in nanoseconds, (maximum) required by the TMS320C25 for an external memory access to be made with no wait states. T_{mem} is the memory device access time, and T_p is the clock period (4/crystal frequency).

$p = 0$; If $T_{mem} \leq T_{ac}$
$p = 1$; If $T_{ac} < T_{mem} \leq (T_p + T_{ac})$
$p = 2$; If $(T_p + T_{ac}) < T_{mem} \leq (T_p \times 2 + T_{ac})$
$p = k$; If $[T_p \times (k-1) + T_{ac}] < T_{mem} \leq (T_p \times k + T_{ac})$

d - Data memory wait states. Represents the number of cycles the device must wait for external data memory to respond to an access. This number is calculated in the same way as the p number.

i - I/O memory wait states. Represents the number of cycles the device must wait for external I/O memory to respond to an access. This number is calculated in the same way as the p number.

The other abbreviations used in the tables and their meanings are as follows:

PI - The instruction executes from internal program memory (RAM).

PR - The instruction executes from internal program memory (ROM).

PE - The instruction executes from external program memory.

DI - The instruction executes using internal data memory.

DE - The instruction executes using external data memory.

INT - Interrupt.

n - The number of times an instruction is executed when using the RPT or RPTK instruction.

Table E-2. Cycle Timings for Cycle Classes When Not in Repeat Mode

CLASS	PI/DI	PI/DE	PE/DI	PE/DE	PR/DI	PR/DE
I	1	2+d	1+p	2+d+p	1	2+d
II	1	2+d	1+p	2+d+p	1	2+d
III	1	1+d	1+p	2+d+p	1	1+d
IV	1		1+p		1	
V	2		2+2p		2	

Table E-2. Cycle Timings for Cycle Classes When Not in Repeat Mode (Concluded)

CLASS	PI/DI	PI/DE	PE/DI	PE/DE	PR/DI	PR/DE
VI	Table is in on-chip RAM: 3	4+d	4+2p	5+d+2p	4	5+d
	Table is in on-chip ROM: 4	5+d	4+2p	5+d+2p	4	5+d
	Table is in external memory: 4+p	5+d+p	4+3p	5+d+3p	4+p	5+d+p
VII	True Conditions: Destination is on-chip RAM: 2		2+2p		2	
	Destination is on-chip ROM: 3		3+2p		3	
	Destination is external memory: 3+p		3+3p		3+p	
	False Condition: Destination is anywhere: 2		2+2p		2	
VIII	Destination is on-chip RAM: 2		2+p		2	
	Destination is on-chip ROM: 3		3+p		3	
	Destination is external memory: 3+p		3+2p		3+p	
IX	2+i	2+d+i	2+p+i	3+d+p+i	2+i	2+d+i
X	1+i	2+d+i	2+p+i	3+d+p+i	1+i	2+d+i
XI	Table is in on-chip RAM: 2	2+d	3+p	3+d+p	3	3+d
	Table is in on-chip ROM: 3	3+d	4+p	4+d+p	4	4+d
	Table is in external memory: 3+p	3+d+p	4+2p	4+d+2p	4+p	4+d+p
XII	Table is in on-chip RAM: 2	3+d	3+p	4+d+p	3	4+d
	Table is in on-chip ROM:		not applicable			
	Table is in external memory: 2+p	3+d+p	3+2p	4+d+2p	3+p	4+d+p
XIII	Source data is in on-chip RAM: 3	3+d	3+2p	3+d+2p	3	3+d
	Source data is in external memory: 4+d	4+2d	4+d+2p	4+2d+2p	4+d	4+2d
XIV	Table is in on-chip RAM: 3	3+d	4+2p	4+d+2p	4	4+d
	Table is in on-chip ROM: 4	4+d	4+2p	4+d+2p	4	4+d
	Table is in external memory: 4+p	4+d+p	4+3p	4+d+3p	4+p	4+d+p
XV	(Interrupt) destination is on-chip ROM 3 (minimum waits for INT) (Interrupt) destination is external memory 3+2p (minimum waits for INT)					

Table E-3. Cycle Timings for Cycle Classes When in Repeat Mode

CLASS	PI/DI	PI/DE	PE/DI	PE/DE	PR/DI	PR/DE
I	n	1+n+nd	n+p	1+n+nd+p	n	1+n+nd
II	n	2n+nd	n+p	2n+nd+p	n	2n+nd
III	n	n+nd	n+p	1+n+nd+p	n	n+nd
IV	n		n+p		n	
V	not repeatable					
VI	Table is in on-chip RAM: 2+n	2+2n+nd	3+n+2p	3+2n+nd+2p	3+n	3+2n+nd
	Table is in on-chip ROM: 3+n	3+2n+nd	3+n+2p	3+2n+nd+2p	3+n	3+2n+nd
	Table is in external memory: 3+n+np	3+2n+nd+np	3+n+np+2p	3+2n+nd+np+2p	3+n+np	3+2n+nd+np
VII	not repeatable					
VIII	not repeatable					
IX	1+n+ni	2n+nd+ni	1+n+p+ni	1+2n+nd+p+ni	1+n+ni	2n+nd+ni
X	n+ni	2n+nd+ni	1+n+p+ni	1+2n+nd+p+ni	n+ni+ni	2n+nd+ni
XI	Table is in on-chip RAM: 1+n	1+n+nd	2+n+p	2+n+nd+p	2+n	2+n+nd
	Table is in on-chip ROM: 2+n	2+n+nd	3+n+p	3+n+nd+p	3+n	3+n+nd
	Table is in external memory: 2+n+np	1+2n+nd+np	3+n+np+p	2+2n+nd+np+p	3+n+np	2+2n+nd+np
XII	Table is in on-chip RAM: 1+n	2+n+nd	2+n+p	3+n+nd+p	2+n	3+n+nd
	Table is in on-chip ROM: not applicable					
	Table is in external memory: 1+n+np	1+2n+nd+np	2+n+np+p	2+2n+nd+np+p	2+n+np	2+2n+nd+np
XIII	Source data is in on-chip RAM: 2+n	2+n+nd	2+n+2p	2+n+nd+2p	2+n	2+n+nd
	Source data is in external memory: 3+n+nd	2+2n+2nd	3+n+nd+2p	2+2n+2nd+2p	3+n+nd	2+2n+2nd
XIV	Table is in on-chip RAM: 2+n	2+n+nd	3+n+2p	3+n+nd+2p	3+n	3+n+nd
	Table is in on-chip ROM: 3+n	3+n+nd	3+n+2p	3+n+nd+2p	3+n	3+n+nd
	Table is in external memory: 3+n+np	2+2n+nd+np	3+n+np+2p	2+2n+nd+np+2p	3+n+np	2+2n+nd+np
XV	not repeatable					

F. TMS320C25 Development Support/Part Order Information

Texas Instruments offers extensive development support and complete documentation with the TMS320 family of digital signal processors (see Figure F-1). Tools are provided to evaluate the performance of the processor, develop algorithm implementations, and fully integrate the design's software and hardware modules.

The development support available for the TMS320C25 is listed below.

- Macro Assembler/Linker
- Simulator
- Emulator (XDS/22)

Key features, a description, and part order information for each TMS320C25 development support tool can be found in the following pages. Contact the nearest TI field sales office for availability or further details (see list of sales offices and distributors at end of book).

Figure F-1. TMS320 Family Development Support

F.1 TMS320C25 Macro Assembler/Linker

The TMS320C25 Macro Assembler translates TMS320C25 assembly language source code into executable object code. The assembler allows the programmer to work with mnemonics rather than hexadecimal machine instructions and to reference memory locations with symbolic addresses. The macro assembler supports macro calls and definitions along with conditional assembly.

The TMS320C25 Linker permits a program to be designed and implemented in separate modules that will later be linked together to form the complete program. The linker resolves external definitions and references for relocatable code, creating an object file that can be executed by the TMS320C25 Simulator, TMS320C25 Emulators, or TMS320C25 processor.

The following key features distinguish the TMS320C25 Macro Assembler/Linker:

- Macro Capabilities and Library Functions
- Conditional Assembly
- Relocatable Modules
- Complete Error Diagnostics
- Symbol Table and Cross Reference

The TMS320C25 Macro Assembler/Linker is currently available for the VAX/VMS, TI PC/MS-DOS, and IBM PC/PC-DOS operating systems.

HOST	OPERATING SYSTEM	PART NUMBER	MEDIUM
DEC VAX TI/IBM PC	VMS MS/PC-DOS	TMDS3242210-08 TMDS3242810-02	1600 BPI MAG TAPE 5 1/4" FLOPPY

F.2 TMS320C25 Simulator

The TMS320C25 Simulator is a software program that simulates operation of the TMS320C25 to allow program verification. The debug mode enables the user to monitor the state of the simulated TMS320C25 while the program is executing. The simulator uses the TMS320C25 object code produced by the TMS320C25 Macro Assembler/Linker. During program execution, the internal registers and memory of the simulated TMS320C25 are modified as each instruction is interpreted by the host computer. Once program execution is suspended, the internal registers and both program and data memories can be inspected and/or modified. In addition, files can be associated with the I/O ports.

The following features highlight simulator capability for effective TMS320C25 software development:

- Program Debug/Verification
- Single-Step Option
- Trace/Breakpoint Capabilities
- Full Access to Simulated Registers and Memories
- I/O Device Simulation

The TMS320C25 Simulator is currently available for the VAX/VMS, TI PC/MS-DOS, and IBM PC/PC-DOS operating systems.

HOST	OPERATING SYSTEM	PART NUMBER	MEDIUM
DEC VAX TI/IBM PC	VMS MS/PC-DOS	TMDS3242211-08 TMDS3242811-02	1600 BPI MAG TAPE 5 1/4" FLOPPY

F.3 TMS320C25 Emulator

The TMS320C25 Emulator (XDS/22) is a user-friendly system that has all the features necessary for realtime in-circuit emulation. This allows integration of hardware and software modules in the debug mode. By setting breakpoints based on internal conditions or external events, execution of the program can be suspended and control given to the debug mode. In the debug mode, all registers and memory locations can be inspected and modified. Single-step execution is available. Full-trace capabilities at full speed and a reverse assembler that translates machine code back into assembly instructions also increase debugging productivity. Using a standard RS-232-C port, the object file produced by the TMS320C25 Macro Assembler/Linker can be downloaded into the emulator, which then can be controlled through a terminal.

The XDS/22 provides 4K x 16 words of high-speed static RAM (zero wait states) for program memory and sockets for 4K x 16 words of high-speed static RAM for user-supplied data memory. It also has the capability of executing out of target memory to utilize the full TMS320C25 program/data address range. For multiprocessing configurations, up to nine emulators can be daisy-chained together.

The XDS/22 emulator is a completely self-contained system with power supply. This model also includes memory expansion with 64K x 16 words of DRAM (two wait states). This slower memory is configurable by the user as either all program memory, all data memory, or 32K words of each. With three RS-232-C ports, the XDS/22 Emulator can be interfaced to a terminal, host computer for source or object downloading/uploading capabilities, and printer or PROM programmer.

The key features of the XDS/22 Emulator are as follows:

- Full-Speed In-Circuit Emulation
- 4K Words of Program Memory for User Code
- Program/Data DRAM Memory Expansion to 64K Words
- Hardware Breakpoint on Program, Data, or I/O Conditions
- 2K Words of Full-Speed Hardware Trace
- Use of Target System Crystal, Internal Crystal, or External Clock Signal
- Up to Ten Software Breakpoints
- Single-Step Option
- Assembler/Reverse Assembler
- Host-Independent Upload/Download Capabilities to/from Program or Data Memory
- Ability to Inspect and Modify Registers and Program/Data Memory
- Multiprocessor System Development

MODEL	PART NUMBER	POWER SUPPLY
XDS/22	TMDS3262221	(Included)

Figure F-2 shows a block diagram of a typical system configuration using the TMS320C25 XDS/22 Emulator.

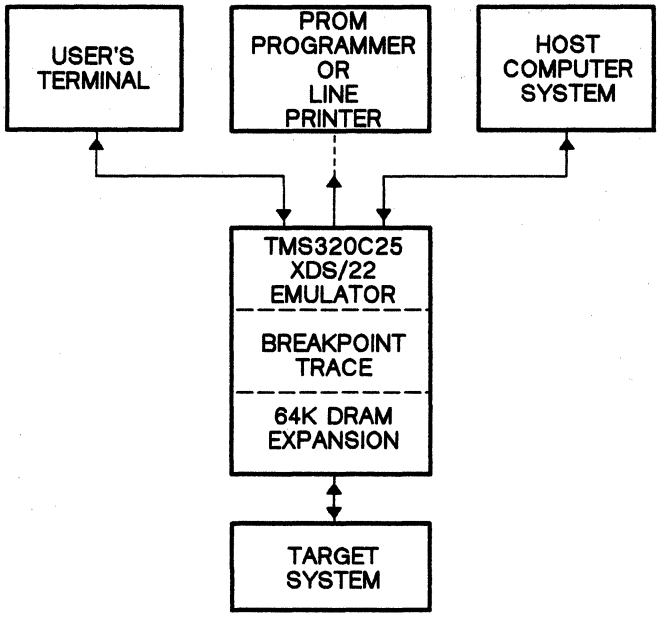


Figure F-2. TMS320C25 XDS/22 Emulator System Configuration

F.4 Device Prefix Designators

To provide expeditious system evaluations by customers during the product development cycle, Texas Instruments assigns a prefix designator with three options: TMX, TMP, and TMS. These prefixes are representative of the evolutionary stages of product development from engineering prototypes (TMX) through fully qualified production devices (TMS). This evolutionary development flow is defined below.

TMX Experimental devices that are not representative of the final device's electrical specifications.

TMP Final silicon die that conforms to the device's electrical specifications but has not completed quality and reliability verification.

TMS Fully qualified production devices.

TMX devices are shipped against the following disclaimer:

1) Product is experimental and its reliability has not been characterized.

2) Product is sold "as is."

3) Product is not warranted to be exemplary of final production version if or when released by Texas Instruments.

TMP devices are shipped against the following disclaimer:

1) Customer understands that the product purchased hereunder has not been fully characterized and the expectation of quality and reliability cannot be defined; therefore, Texas Instruments standard warranty refers only to the device's specifications.

2) No warranty of merchantability or fitness is expressed or implied.

> **Note:**
>
> Texas Instruments recommends that prototype devices (TMX or TMP) not be used in production systems since their expected end-use failure rate is undefined but predicted to be greater than standard qualified production devices.

TMS devices have been fully characterized and the quality and reliability of the device has been fully demonstrated. Texas Instruments standard warranty applies.

F.5 TMS320 Nomenclature

In addition to the prefix, the device family name, the specific device name, package type, and temperature range are designated in the device nomenclature. Figure F-3 provides a legend for reading the complete device name.

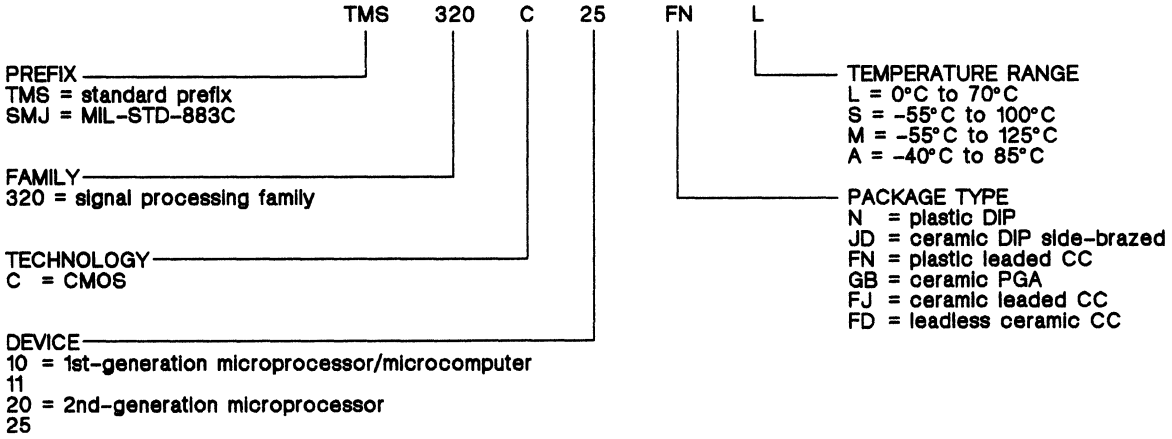


Figure F-3. TMS320 Nomenclature

G. TMS320C25 Macro Assembler and Link Editor Installation

This appendix contains step-by-step instructions for installing, verifying, and relinking the TMS320C25 Macro Assembler and Link Editor. This software can be installed on two operating systems:

- VAX/VMS (Digital Equipment Corporation VAX-11)[1]
- MS/PC-DOS (MS-DOS for the TI PC and PC-DOS for the IBM PC)[2]

The following style and symbol conventions are used to present information clearly and concisely:

- The symbol **<CR>** indicates that a carriage return should be entered.
- Screen displays are shown in a `special font`.
- Portions of a display that are user responses are underscored.

---

1 VAX-11 and VMS are trademarks of Digital Equipment Corporation.

2 PC-DOS is a trademark of International Business Machines.

G.1 TMS320C25 VAX/VMS CrossWare Installation

The TMS320C25 CrossWare tape was created with the VMS BACKUP utility. The package is contained in two directories, shipped in two save-sets.

In the examples, replace **<directory>** with the name of the directory in which this package resides, e.g., DUA2:[DSP.ASM25]. Note the use of brackets in this section to indicate a directory.

The following subsections include the sequence of steps used for restoring the directories of the Macro Assembler and Link Editor, installing command files, providing transparent access, verifying the installation procedure, and relinking the product components. A list of the product directories is also provided.

G.1.1 Restoring the Distribution Tape to Disk

In the following examples, **MFA0** is the tape drive name and **DUA2** is the hard disk drive name. The tape drive and disk drive may have other names, dependent on a particular system.

- **Mount the Tape**

  Place the tape on a tape drive. Mount it by entering:

  ```
  ALLOC MFA0: <CR>
  MOUNT MFA0:/OVER=ID/FOR/DEN=1600 <CR>
  ```

  If the mount is successful, the screen displays:

  ```
  ASM25 MOUNTED ON MFA0
  ```

- **Restore the Macro Assembler**

  Use the BACKUP utility to read the C25ASM save-set from the tape:

  ```
  BACKUP/LOG/VERIFY MFA0:C25ASM.BCK DUA2:[<directory>]*.* <CR>
  ```

  The CrossWare package can reside in either the user directory or a system directory. The examples copy the package into the user directory, copying the C25ASM.BCK directory structure on the tape into [<directory>] on disk DUA2.

  A README file explaining the Macro Assembler validation procedure is contained in this directory:

  ```
  [<directory>.C25ASM]README.DAT
  ```

  If not installing the Link Editor, skip the next step and unload the tape.

- **Restore the Link Editor**

  Use the BACKUP utility to copy the LINKER save-set from the tape:

  ```
  BACKUP/LOG/VERIFY MFA0:LINKER.BCK DUA2:[<directory>]*.* <CR>
  ```

  The LINKER.BCK directory structure on the tape is copied into [<directory>] on disk DUA2.

  A README file explaining the Link Editor validation procedure is contained in this directory:

  ```
  [<directory>.LINKER]README.DAT
  ```

- **Dismount the Tape**

  Dismount the tape by entering:

  DISMOUNT MFA0: <CR>

  Remove the tape from the drive. Deallocate the tape drive by entering:

  DEALLOCATE MFA0: <CR>

G.1.2 Installing Command Files

Two command procedures are provided to ensure correct system-dependent parse features. If the VAX/VMS system runs under Version 2.5, use the PARSE.C25 command procedure by renaming it PARSE.COM. If the system runs under Version 3.0, use the default PARSE.COM.

Set the default directory to the directory to which the Assembler and Linker have been restored. Edit the Assembler and Linker command files, replacing existing pathnames with the pathnames to which they have been restored:

- Edit the file: [<directory>.C25ASM]XASM.COM

  Insert the appropriate file pathnames in three places:

  - For the two calls to the PARSE command, which appear within the first 20 lines, insert the appropriate file pathname after @ and before PARSE:

    ```
    $ @DUA2:[<directory>.C25ASM]PARSE 'P1' ...
    ```

  - For the one RUN statement, which appears near the bottom of the file, insert the appropriate file pathname after RUN and before ASM32020:

    ```
    $ RUN DUA2:[<directory>.C25ASM]ASM32020
    ```

- Edit the file: [<directory>.LINKER]XLINKER.COM

  Substitute the appropriate file pathnames in three places:

  - Two calls to the PARSE command, marked in the file by a preceding line '******...'.

    ```
    $ @[MOORE.LINKER]PARSE 'P1' ...
    ```

    Change them to:

    ```
    $ @DUA2:[<directory>.LINKER]PARSE 'P1' ...
    ```

  - One RUN statement near the end of the file.

    ```
    $ RUN[MOORE.LINKER]LINKER
    ```

    Change it to:

    ```
    $ RUN DUA2:[<directory>.LINKER]LINKER
    ```

G.1.3 Providing Transparent Access

Use the following procedure to provide transparent access to the Assembler and Link Editor for all users. After the directories are on disk, make the following assignments into the LOGIN.COM file:

```
$ X320 :== @DUA2:[<directory>.C25ASM]XASM.COM
$ XLINK :== @DUA2:[<directory>.LINKER]XLINKER.COM
```

This defines the X320 and XLINK commands, which execute the Macro Assembler and Link Editor. Execute the Macro Assembler by entering **X320** at the terminal in System Mode. Similarly, execute the Link Editor by entering **XLINK**.

G.1.4 Verifying Installation

This verification procedure is not designed to perform an exhaustive test. It simply verifies that the installation procedures were executed correctly. It also provides familiarity with the basic operation and data flow of this package.

The test procedure consists of creating a test directory, assembling three source files, and linking the assembler output files.

1) Create a test directory. Copy the TEST.ASM, TEST1.ASM, TEST2.ASM, and TEST1.CON files from [.C25ASM] and [.LINKER] into the directory by entering these commands:

```
$ CREATE/DIR [<userid>.TEST] <CR>
$ SET DEF [<userid>.TEST] <CR>
$ COPY [<directory>.C25ASM]TEST.ASM * <CR>
$ COPY [<directory>.LINKER]TEST1.ASM * <CR>
$ COPY [<directory>.LINKER]TEST2.ASM * <CR>
$ COPY [<directory>.LINKER]TEST1.CON * <CR>
```

2) In System Mode, enter: X320 <CR>

For the first input parameter in each of the three assembler runs, enter TEST.ASM, TEST1.ASM, and TEST2.ASM, respectively (ASM is the default). The command procedure parses the pathname and generates defaults for the output listing and object files. Press the carriage return to accept the defaults, or specify user file pathnames as follows:

```
$ X320 TEST <CR>
Object file (TEST.MPO): <CR>
Listing file (TEST.LIS): <CR>
Messages (__TXE1:): <CR>

$ X320 TEST1
Object file (TEST1.MPO): <CR>
Listing file (TEST1.LIS): <CR>
Messages (__TXE1:): <CR>

$ X320 TEST2
Object file (TEST2.MPO): <CR>
Listing file (TEST2.LIS): <CR>
Messages (__TXE1:): <CR>
```

This creates the TEST.MPO, TEST.LIS, TEST1.MPO, TEST1.LIS, TEST2.MPO and TEST2.LIS files in the directory [<userid>.TEST].

3) In System Mode, enter: XLINK <CR>

As the first input parameter, enter TEST1.CON. For the second and third parameters, the command procedure parses the pathname and generates

defaults for the output, load, and map files. This procedure links the object files for TEST1 and TEST2 into a single executable object file in TEST1.LOD (CON is the default for the first parameter):

```
$ XLINK TEST1 <CR>
Linked object file (TEST1.LOD): <CR>
Map file (TEST1.MAP): <CR>
```

This creates the files TEST1.LOD and TEST1.MAP. These files should agree with the precompiled versions in the product directories for the Macro Assembler and Link Editor.

G.1.5 Relinking the Macro Assembler and Link Editor

It should not be necessary to relink the Macro Assembler or Link Editor, but command files have been provided to allow for this contingency.

To relink the Macro Assembler, edit the LINKASM.COM procedure file to put the correct pathname for the runtime library in the logical assignment statement. In System Mode, execute LINKASM.COM to relink the ASM32020.EXE file:

```
$ SET DEF [<directory>.C25ASM] <CR>
$ @LINKASM <CR>
```

Similarly, to relink the Link Editor, edit the LINKLINK.COM procedure file to put the correct pathname for the runtime library in the logical assignment statement. In System Mode, execute LINKLINK.COM to relink the LINKER.EXE file:

```
$ SET DEF [<directory>.LINKER] <CR>
$ @LINKLINK <CR>
```

G.1.6 Product Directories

The following listing contains the product directories found in the CrossWare package. These two directories contain a total of 40 files.

```
SET DEF [<userid>.<directory>] <CR>
DIR <CR>

Directory [<directory>]
C25ASM.DIR;1   LINKER.DIR;1
Total: 2 files

DIR [<directory>.C25ASM] <CR>

Directory [<directory>.C25ASM]
ASM.OBJ;1        ASM32020.EXE;1  ASMRTS.OLB;1     FORM0.ASM;1
FORM0.LIS;1      FORM0.MPO;1     FORM1.ASM;1      FORM1.LIS;1
FORM1.MPO;1      FORM2.ASM;1     FORM2.LIS;1      FORM2.MPO;1
FORMREST.ASM;1   FORMREST.LIS;1  FORMREST.MPO;1   LINKASM.COM;1
PARSE.C25;1      PARSE.COM;1     README.LIS;1     TEST.ASM;1
TEST.LIS;1       TEST.MPO;1      XASM.COM;1
Total: 23 files

DIR [<directory>.LINKER] <CR>

Directory [<directory>.LINKER.]
XLINKER.COM;1   LINKER.EXE;1    LINKER.OBJ;1     LINKLINK.COM;1
LINKRTS.OLB;1   PARSE.C25;1     PARSE.COM;1      README.LIS;1
TEST1.ASM;1     TEST1.CON;1     TEST1.LIS;1      TEST1.LOD;1
TEST1.MAP;1     TEST1.MPO;1     TEST2.ASM;1      TEST2.LIS;1
TEST2.MPO;1
Total: 17 files
```

G.2 TMS320C25 MS/PC-DOS CrossWare

The TMS320C25 CrossWare installation package is contained on a dual-density double-sided floppy diskette. The Macro Assembler and Link Editor execute in batch mode on MS-DOS (TI PC) and PC-DOS (IBM PC) systems. At least 256K bytes of memory space must be available.

Instructions are included for both hard-disk systems and dual floppy-drive systems. The examples use these symbols for drive names:

A: Floppy-disk drive for hard-disk systems or source drive for dual floppy-drive systems.

B: Destination or system disk drive for dual floppy-drive systems.

E: Winchester (hard disk) for hard-disk systems.

The following subsections include a list of the files on the diskette and the sequence of steps used for restoring, executing, and testing the directories of the Macro Assembler and Link Editor.

G.2.1 Diskette Files

- The TMS320C25 Assembler portion of the diskette contains four files: an executable module and three test files.

  Executable Module:

XASM.EXE	Executes the Macro Assembler

  Test Files:

FFT.ASM	Source file for Assembler test program
FFT.LST	Correct output listing file for Assembler test program
FFT.MPO	Correct output object file for Assembler test program

- The TMS320C25 Linker portion of the diskette contains ten files: one executable module and nine test files.

  Executable Module:

LINKER.EXE	Executes the Link Editor

  Test Files:

TST1.ASM	Source file for test program #1
TST1.LST	Correct output listing file for test program #1
TST1.MPO	Correct output object file for test program #1
TST2.ASM	Source file for test program #2
TST2.LST	Correct output listing file for test program #2
TST2.MPO	Correct output object file for test program #2
LNKTST.CTL	Linker test program (link control file)
LNKTST.MAP	Correct output listing file for the Linker test program
LNKTST.LOD	Correct output object file for the Linker test program

G.2.2 Restoring the Macro Assembler and Linker

Instructions are provided for hard-disk systems and dual floppy-drive systems. If using a dual floppy-drive system, the MS/PC-DOS system diskette should be in drive B.

1) Make a backup diskette of the Macro Assembler and Linker.

   - On MS-DOS, insert the source diskette in drive A. Enter:

     DISKCOPY A: A:/F/V <CR>

     The /F (format) switch tells MS-DOS to format the new (destination) diskette before copying begins. The /V (verify) switch tells MS-DOS to verify that the source and destination diskettes are identical after the copy has completed.

   - On PC-DOS, insert a blank diskette in drive A. Enter:

     FORMAT A: <CR>

     Then enter:

     DISKCOPY A: A: <CR>

   When MS/PC-DOS first prompts for the destination diskette, remove the source diskette and insert a blank diskette. Follow the prompts, removing and inserting the source and destination diskettes as directed. When MS/PC-DOS prompts:

   ```
   COPY ANOTHER  (Y/N)?
   ```

   respond with N.

2) Copy the Macro Assembler onto the hard disk or the system disk.

   On hard-disk systems, enter:

   COPY A:XASM.EXE E:*.*/V <CR>

   On dual floppy-drive systems, enter:

   COPY A:XASM.EXE B:*.*/V <CR>

3) Copy the Link Editor onto the hard disk or the system disk:

   On hard-disk systems, enter:

   COPY A:LINKER.EXE E:*.*/V <CR>

   On dual floppy-drive systems, enter:

   COPY A:LINKER.EXE B:*.*/V <CR>

G.2.3 Executing the Macro Assembler

To execute the Macro Assembler, enter:

XASM <CR>

The command line parser prompts for the source, listing, and object file names:

Source File Enter the source file name (if the source file does not have an extension, then type the file name with an explicit '.').
Listing File Enter the output listing file name.
Object File Enter the output object file name.

MS/PC-DOS creates defaults for the listing and object files and/or their extensions. The default extensions are:

.ASM Source file
.LST Listing file
.MPO Object file

In the following examples, two special command characters are used: semicolon (;) and comma (,). Using a semicolon (;) followed immediately by a carriage return at any time after the first filename in a command line selects the default responses to the remaining prompts. The comma (,) separates responses to successive prompts.

Examples:

```
XASM <filename>.SRC;
```
- Uses <filename> with extension SRC.
- Generates defaults for the listing file <filename.LST> and object file <filename>.MPO.

```
XASM <filename>;
```
- Uses <filename> with default extension ASM.
- Generates defaults for the listing and object files as indicated above.

```
XASM <filename>,<newname>;
```
- Uses <filename> with default extension ASM.
- Generates listing file <newname>.LST and object file <newname>.MPO.

```
XASM <filename>,<newname>
```
- Uses <filename> with default extension ASM.
- Generates listing file <newname>.LST and prompts for object file name.

G.2.4 Executing the Link Editor

To execute the Linker, enter:

LINKER <CR>

The command line parser prompts for the control, linkmap, and load file names.

Control File Enter the control file name with extension (if the control file does not have an extension, type the file name with an explicit '.').
Map File Enter the linkmap file name with extension.
Load File Enter the load module file name with extension.

MS/PC-DOS generates defaults for the linkmap and load files and/or their extensions. The default extensions are:

.CTL Control file
.MAP Linkmap file
.LOD Load file

In the following examples, two special command characters are used: semicolon (;) and comma (,). Using a semicolon (;) followed immediately by a carriage return at any time after the first filename in a command line selects the default responses to the remaining prompts. The comma (,) separates responses to successive prompts.

Examples:

```
LINKER <filename>.SRC;
```
- Uses <filename> with extension SRC.
- Generates defaults for the linkmap and load files as indicated above.

```
LINKER <filename>;
```
- Uses <filename> with default extension CTL.
- Generates defaults for the linkmap and load files as indicated above.

```
LINKER <filename>,<newname>;
```
- Uses <filename> with default extension CTL.
- Generates linkmap file <newname>.MAP and load file <newname>.LOD.

```
LINKER <filename>,<newname>
```
- Uses <filename> with default extension CTL.
- Generates linkmap file <newname>.MAP and prompts for the load file name.

G.2.5 Testing the Macro Assembler

- Hard-Disk Systems

  1) Copy the FFT.ASM test file from the backup diskette onto the hard disk using the MS/PC-DOS COPY utility:

     ```
     COPY A:FFT.ASM E:*.*/V <CR>
     ```

  2) Execute the Macro Assembler using FFT.ASM as the source file. By entering:

     ```
     XASM FFT; <CR>
     ```

     in response to the system prompt, the Assembler generates the default object file FFT.MPO and default listing file FFT.LST.

  3) Compare the listing and object files just created to those on the backup diskette:

     - On MS-DOS, use the FILCOM utility to make the comparison:

       ```
       FILCOM FFT.MPO A:FFT.MPO <CR>
       FILCOM FFT.LST A:FFT.LST <CR>
       ```

       The contents of each file can be viewed with the TYPE utility.

     - On PC-DOS, use the TYPE utility to show the contents of the two files and to visually check the contents for verification:

       ```
       TYPE FFT.MPO <CR>
       TYPE A:FFT.MPO <CR>

       TYPE FFT.LST <CR>
       TYPE A:FFT.LST <CR>
       ```

     Only lines containing dates or times should differ.

- Floppy-Drive Systems:

  1) Insert the backup diskette into the default floppy drive.

  2) Execute the Macro Assembler using FFT.ASM as the source test file. It is important to use a different name for the object and listing files. Otherwise, the Assembler will write over these files on the backup diskette, and there will be no correct files with which to compare the created files. By entering:

     XASM FFT,MYFFT; <CR>

     in response to the system prompt, the Assembler generates object file MYFFT.MPO and listing file MYFFT.LST.

  3) Compare the listing and object files just created to those shipped on the backup diskette:

     - On MS-DOS, use the FILCOM utility to make the comparison:

       FILCOM FFT.MPO MYFFT.MPO <CR>
       FILCOM FFT.LST MYFFT.LST <CR>

       The contents of each file can be viewed with the TYPE utility.

     - On PC-DOS, use the TYPE utility to show the contents of the two files and to visually check the contents for verification:

       TYPE FFT.MPO <CR>
       TYPE MYFFT.MPO <CR>

       TYPE FFT.LST <CR>
       TYPE MYFFT.LST <CR>

     Only lines containing dates or times should differ.

> **Note:**
>
> The files TEST1.ASM and TEST2.ASM, contained in the Linker portion of the diskette, can also be used to test the Macro Assembler.

G.2.6 Testing the Link Editor

- Hard-Disk Systems

  1) Copy the LNKTST.CTL, TST1.MPO, and TST2.MPO files from the backup diskette onto the hard disk using the MS/PC-DOS COPY utility:

     COPY A:LNKTST.CTL E:*.*/V <CR>

     COPY A:TST*.MPO E:*.*/V <CR>

  2) Execute the Link Editor using LNKTST.CTL as the control file. By entering:

     LINKER LNKTST; <CR>

     in response to the system prompt, the Linker generates the default linkmap file LNKTST.MAP and default load file LNKTST.LOD.

  3) Compare the map and load files just created to those on the backup diskette:

- On MS-DOS, use the FILCOM utility to make the comparison:

```
FILCOM LNKTST.MAP A:LNKTST.MAP <CR>
FILCOM LNKTST.LOD A:LNKTST.LOD <CR>
```

The contents of each file can be viewed with the TYPE utility.

- On PC-DOS, use the TYPE utility to show the contents of the two files and to visually check the contents for verification:

```
TYPE LNKTST.MAP <CR>
TYPE A:LNKTST.MAP <CR>

TYPE LNKTST.LOD <CR>
TYPE A:LNKTST.LOD <CR>
```

Only lines containing dates or times should differ.

- Floppy-Drive Systems:

1) Insert the backup diskette into the default floppy drive.

2) Execute the Link Editor using LNKTST.CTL as the control file. It is important to use a different name for the map and load files. Otherwise, the Linker will write over these files on the backup diskette, and there will be no correct files with which to compare the created files. By entering:

```
LINKER LNKTST,MYLNK; <CR>
```

in response to the system prompt, the Linker generates the linkmap file MYLNK.MAP and load file MYLNK.LOD.

3) Compare the map and load files just created to those shipped on the backup diskette:

- On MS-DOS, use the FILCOM utility to make the comparison:

```
FILCOM LNKTST.MAP MYLNK.MAP <CR>
FILCOM LNKTST.LOD MYLNK.LOD <CR>
```

The contents of each file can be viewed with the TYPE utility.

- On PC-DOS, use the TYPE utility to show the contents of the two files and to visually check the contents for verification:

```
TYPE LNKTST.MAP <CR>
TYPE MYLNK.MAP <CR>

TYPE LNKTST.LOD <CR>
TYPE MYLNK.LOD <CR>
```

Only lines containing dates or times should differ.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

V

W

X

Z

TI Sales Offices

ALABAMA: Huntsville (205) 837-7530.

ARIZONA: Phoenix (602) 995-1007;
Tucson (602) 624-3276.

CALIFORNIA: Irvine (714) 660-8187;
Sacramento (916) 929-1521;
San Diego (619) 278-9601;
Santa Clara (408) 980-9000;
Torrance (213) 217-7010;
Woodland Hills (818) 704-7759.

COLORADO: Aurora (303) 368-8000.

CONNECTICUT: Wallingford (203) 269-0074.

FLORIDA: Ft. Lauderdale (305) 973-8502;
Maitland (305) 660-4600; **Tampa** (813) 870-6420.

GEORGIA: Norcross (404) 662-7900.

ILLINOIS: Arlington Heights (312) 640-2925.

INDIANA: Ft. Wayne (219) 424-5174;
Indianapolis (317) 248-8555.

IOWA: Cedar Rapids (319) 395-9550.

MARYLAND: Baltimore (301) 944-8600.

MASSACHUSETTS: Waltham (617) 895-9100.

MICHIGAN: Farmington Hills (313) 553-1500;
Grand Rapids (616) 957-4200.

MINNESOTA: Eden Prairie (612) 828-9300.

MISSOURI: Kansas City (816) 523-2500;
St. Louis (314) 569-7600.

NEW JERSEY: Iselin (201) 750-1050.

NEW MEXICO: Albuquerque (505) 345-2555.

NEW YORK: East Syracuse (315) 463-9291;
Endicott (607) 754-3900; **Melville** (516) 454-6600;
Pittsford (716) 385-6770;
Poughkeepsie (914) 473-2900.

NORTH CAROLINA: Charlotte (704) 527-0930;
Raleigh (919) 876-2725.

OHIO: Beachwood (216) 464-6100;
Dayton (513) 258-3877.

OREGON: Beaverton (503) 643-6758.

PENNSYLVANIA: Ft. Washington (215) 643-6450;
Coraopolis (412) 771-8550.

PUERTO RICO: Hato Rey (809) 753-8700

TEXAS: Austin (512) 250-7655;
Houston (713) 778-6592; **Richardson** (214) 680-5082;
San Antonio (512) 496-1779.

UTAH: Murray (801) 266-8972.

VIRGINIA: Fairfax (703) 849-1400.

WASHINGTON: Redmond (206) 881-3080.

WISCONSIN: Brookfield (414) 785-7140.

CANADA: Nepean, Ontario (613) 726-1970;
Richmond Hill, Ontario (416) 884-9181;
St. Laurent, Quebec (514) 335-8392.

TI Regional Technology Centers

CALIFORNIA: Irvine (714) 660-8140,
Santa Clara (408) 748-2220.

GEORGIA: Norcross (404) 662-7945.

ILLINOIS: Arlington Heights (312) 640-2909.

MASSACHUSETTS: Waltham (617) 895-9197.

TEXAS: Richardson (214) 680-5066.

CANADA: Nepean, Ontario (613) 726-1970

Customer Response Center

TOLL FREE: (800) 232-3200

OUTSIDE USA: (214) 995-6611
(8:00 a.m. — 5:00 p.m. CST)

TI Distributors

TI AUTHORIZED DISTRIBUTORS IN USA

Arrow Electronics
Diplomat Electronics
General Radio Supply Company
Graham Electronics
Hall-Mark Electronics
Kierulff Electronics
Marshall Industries
Milgray Electronics
Newark Electronics
Time Electronics
R.V. Weatherford Co.
Wyle Laboratories
Zeus Component, Inc. (Military)

TI AUTHORIZED DISTRIBUTORS IN CANADA

Arrow Electronics Canada
Future Electronics

TI AUTHORIZED DISTRIBUTORS IN USA

—OBSOLETE PRODUCT ONLY—
Rochester Electronics, Inc.
Newburyport, Massachusetts
(617) 462-9332

ALABAMA: Arrow (205) 837-6955;
Hall-Mark (205) 837-8700; Kierulff (205) 883-6070;
Marshall (205) 881-9235.

ARIZONA: Arrow (602) 968-4800;
Hall-Mark (602) 437-1200; Kierulff (602) 437-0750;
Marshall (602) 968-6181; Wyle (602) 866-2888.

CALIFORNIA: Los Angeles/Orange County:
Arrow (818) 701-7500, (714) 838-5422;
Hall-Mark (818) 716-7300, (714) 669-4700,
(213) 217-8400; Kierulff (213) 725-0325, (714) 731-5711,
(714) 220-6300; Marshall (818) 407-0101,
(818) 442-7204, (714) 660-0951;
R.V. Weatherford (714) 966-1447, (213) 849-3451,
Wyle (213) 322-8100, (818) 880-9001, (714) 863-9953;
Zeus (714) 632-6880;
Sacramento: Arrow (916) 925-7456;
Hall-Mark (916) 722-8600; Marshall (916) 635-9700;
Wyle (916) 638-5282;
San Diego: Arrow (619) 565-4800;
Hall-Mark (619) 268-1201; Kierulff (619) 278-2112;
Marshall (619) 578-9600; Wyle (619) 565-9171;
San Francisco Bay Area: Arrow (408) 745-6600;
(415) 487-4600; Hall-Mark (408) 946-0900;
Kierulff (408) 971-2600; Marshall (408) 943-4600;
Wyle (408) 727-2500; Zeus (408) 998-5121.

COLORADO: Arrow (303) 696-1111;
Hall-Mark (303) 790-1662; Kierulff (303) 790-4444;
Wyle (303) 457-9953.

CONNECTICUT: Arrow (203) 265-7741;
Diplomat (203) 797-9674; Hall-Mark (203) 269-0100;
Kierulff (203) 265-1115; Marshall (203) 265-3822;
Milgray (203) 795-0714.

FLORIDA: Ft. Lauderdale: Arrow (305) 429-8200;
Diplomat (305) 974-8700; Hall-Mark (305) 971-9280;
Kierulff (305) 486-4004; Marshall (305) 928-0661;
Orlando: Arrow (305) 725-1480;
Hall-Mark (305) 855-4020; Marshall (305) 841-1878;
Milgray (305) 647-5747; Zeus (305) 365-3000;
Tampa: Arrow (813) 576-8995;
Diplomat (813) 443-4514; Hall-Mark (813) 530-4543;
Kierulff (813) 576-1966.

GEORGIA: Arrow (404) 449-8252;
Hall-Mark (404) 447-8000; Kierulff (404) 447-5252;
Marshall (404) 923-5750.

TEXAS INSTRUMENTS

ILLINOIS: Arrow (312) 397-3440;
Diplomat (312) 595-1000; Hall-Mark (312) 860-3800;
Kierulff (312) 250-0500; Marshall (312) 490-0155;
Newark (312) 784-5100.

INDIANA: Indianapolis: Arrow (317) 243-9353;
Graham (317) 634-8202; Hall-Mark (317) 872-8875;
Marshall (317) 297-0483;
Ft. Wayne: Graham (219) 423-3422.

IOWA: Arrow (319) 395-7230.

KANSAS: Kansas City: Arrow (913) 541-9542;
Hall-Mark (913) 888-4747; Marshall (913) 492-3121.

MARYLAND: Arrow (301) 995-0003;
Diplomat (301) 995-1226; Hall-Mark (301) 988-9800;
Kierulff (301) 636-5800; Milgray (301) 995-6169;
Marshall (301) 840-9450; Zeus (301) 997-1118.

MASSACHUSETTS: Arrow (617) 933-8130;
Diplomat (617) 667-4670; Hall-Mark (617) 667-0902;
Kierulff (617) 667-8331; Marshall (617) 272-8200;
Time (617) 532-6200; Zeus (617) 863-8800.

MICHIGAN: Detroit: Arrow (313) 971-8220;
Marshall (313) 525-5850; Newark (313) 967-0600;
Grand Rapids: Arrow (616) 243-0912.

MINNESOTA: Arrow (612) 830-1800;
Hall-Mark (612) 941-2600; Kierulff (612) 941-7500;
Marshall (612) 559-2211.

MISSOURI: St. Louis: Arrow (314) 567-6888;
Hall-Mark (314) 291-5350; Kierulff (314) 739-0855.

NEW HAMPSHIRE: Arrow (603) 668-6968.

NEW JERSEY: Arrow (201) 575-5300, (609)
596-8000; Diplomat (201) 785-1830;
General Radio (609) 964-8560;
Hall-Mark (201) 575-4415, (609) 235-1900;
Kierulff (201) 575-6750, (609) 235-1444;
Marshall (201) 882-0320, (609) 234-9100;
Milgray (609) 983-5010.

NEW MEXICO: Arrow (505) 243-4566.

NEW YORK: Long Island: Arrow (516) 231-1000;
Diplomat (516) 454-6400;
Hall-Mark (516) 737-0600; Marshall (516) 273-2053;
Milgray (516) 420-9800; Zeus (914) 937-7400;
Rochester: Arrow (716) 427-0300;
Marshall (716) 235-7620; Diplomat (716) 359-4400;
Syracuse: Arrow (315) 652-1000;
Marshall (607) 798-1611.

NORTH CAROLINA: Arrow (919) 876-3132,
(919) 725-8711; Hall-Mark (919) 872-0712;
Kierulff (919) 872-8410; Marshall (919) 878-9882.

OHIO: Cleveland: Arrow (216) 248-3990;
Hall-Mark (216) 349-4632; Kierulff (216) 831-5222;
Marshall (216) 248-1788.
Columbus: Arrow (614) 885-8362;
Hall-Mark (614) 888-3313;
Dayton: Arrow (513) 435-5563;
Graham (513) 435-8660; Kierulff (513) 439-0045;
Marshall (513) 236-8088.

OKLAHOMA: Arrow (918) 665-7700;
Kierulff (918) 252-7537.

OREGON: Arrow (503) 684-1690;
Kierulff (503) 641-9153; Wyle (503) 640-6000;
Marshall (503) 644-5050.

PENNSYLVANIA: Arrow (412) 856-7000,
(215) 928-1800; General Radio (215) 922-7037.

RHODE ISLAND: Arrow (401) 431-0980

TEXAS: Austin: Arrow (512) 835-4180;
Hall-Mark (512) 258-8848; Kierulff (512) 835-2090;
Marshall (512) 837-1991; Wyle (512) 834-9957;
Dallas: Arrow (214) 380-6464;
Hall-Mark (214) 553-4300; Kierulff (214) 343-2400;
Marshall (214) 233-5200; Wyle (214) 235-9953;
Zeus (214) 783-7010;
Houston: Arrow (713) 530-4700;
Hall-Mark (713) 781-6100; Kierulff (713) 530-7030;
Marshall (713) 895-9200; Wyle (713) 879-9953.

UTAH: Arrow (801) 972-0404;
Diplomat (801) 486-4134;
Hall-Mark (801) 268-3779; Kierulff (801) 973-6913;
Wyle (801) 974-9953.

WASHINGTON: Arrow (206) 643-4800;
Kierulff (206) 575-4420; Wyle (206) 453-8300;
Marshall (206) 747-9100.

WISCONSIN: Arrow (414) 792-0150;
Hall-Mark (414) 797-7844; Kierulff (414) 784-8160;
Marshall (414) 797-8400.

CANADA: Calgary: Future (403) 235-5325;
Edmonton: Future (403) 438-2858;
Montreal: Arrow Canada (514) 735-5511;
Future (514) 694-7710;
Ottawa: Arrow Canada (613) 226-6903;
Future (613) 820-8313;
Quebec City: Arrow Canada (418) 687-4231;
Toronto: Arrow Canada (416) 661-0220;
Future (416) 638-4771;
Vancouver: Future (604) 294-1166
Winnipeg: Future (204) 339-0554

BQ

TI Worldwide Sales Offices

ALABAMA: Huntsville: 500 Wynn Drive, Suite 514, Huntsville, AL 35805, (205) 837-7530.

ARIZONA: Phoenix: 8825 N. 23rd Ave., Phoenix, AZ 85021, (602) 995-1007.

CALIFORNIA: Irvine: 17891 Cartwright Rd., Irvine, CA 92714, (714) 660-8187; **Sacramento:** 1900 Point West Way, Suite 171, Sacramento, CA 95815, (916) 929-1521; **San Diego:** 4333 View Ridge Ave., Suite B., San Diego, CA 92123, (619) 278-9601; **Santa Clara:** 5353 Betsy Ross Dr., Santa Clara, CA 95054, (408) 980-9000; **Torrance:** 690 Knox St., Torrance, CA 90502, (213) 217-7010; **Woodland Hills:** 21220 Erwin St., Woodland Hills, CA 91367, (818) 704-7759.

COLORADO: Aurora: 1400 S. Potomac Ave., Suite 101, Aurora, CO 80012, (303) 368-8000.

CONNECTICUT: Wallingford: 9 Barnes Industrial Park Rd., Barnes Industrial Park, Wallingford, CT 06492, (203) 269-0074.

FLORIDA: Ft. Lauderdale: 2765 N.W. 62nd St., Ft. Lauderdale, FL 33309, (305) 973-8502; **Maitland:** 2601 Maitland Center Parkway, Maitland, FL 32751, (305) 660-4600; **Tampa:** 5010 W. Kennedy Blvd., Suite 101, Tampa, FL 33609, (813) 870-6420.

GEORGIA: Norcross: 5515 Spalding Drive, Norcross, GA 30092, (404) 662-7900

ILLINOIS: Arlington Heights: 515 W. Algonquin, Arlington Heights, IL 60005, (312) 640-2925.

INDIANA: Ft. Wayne: 2020 Inwood Dr., Ft. Wayne, IN 46815, (219) 424-5174; **Indianapolis:** 2346 S. Lynhurst, Suite J-400, Indianapolis, IN 46241, (317) 248-8555.

IOWA: Cedar Rapids: 373 Collins Rd. NE, Suite 200, Cedar Rapids, IA 52402, (319) 395-9550.

MARYLAND: Baltimore: 1 Rutherford Pl., 7133 Rutherford Rd., Baltimore, MD 21207, (301) 944-8600.

MASSACHUSETTS: Waltham: 504 Totten Pond Rd., Waltham, MA 02154, (617) 895-9100.

MICHIGAN: Farmington Hills: 33737 W. 12 Mile Rd., Farmington Hills, MI 48018, (313) 553-1500.

MINNESOTA: Eden Prairie: 11000 W. 78th St., Eden Prairie, MN 55344 (612) 828-9300.

MISSOURI: Kansas City: 8080 Ward Pkwy., Kansas City, MO 64114, (816) 523-2500; **St. Louis:** 11816 Borman Drive, St. Louis, MO 63146, (314) 569-7600.

NEW JERSEY: Iselin: 485E U.S. Route 1 South, Parkway Towers, Iselin, NJ 08830 (201) 750-1050

NEW MEXICO: Albuquerque: 2820-D Broadbent Pkwy NE, Albuquerque, NM 87107, (505) 345-2555.

NEW YORK: East Syracuse: 6365 Collamer Dr., East Syracuse, NY 13057, (315) 463-9291; **Endicott:** 112 Nanticoke Ave., P.O. Box 618, Endicott, NY 13760, (607) 754-3900; **Melville:** 1 Huntington Quadrangle, Suite 3C10, P.O. Box 2936, Melville, NY 11747, (516) 454-6600; **Pittsford:** 2851 Clover St., Pittsford, NY 14534, (716) 385-6770; **Poughkeepsie:** 385 South Rd., Poughkeepsie, NY 12601, (914) 473-2900.

NORTH CAROLINA: Charlotte: 8 Woodlawn Green, Woodlawn Rd., Charlotte, NC 28210, (704) 527-0930; **Raleigh:** 2809 Highwoods Blvd., Suite 100, Raleigh, NC 27625, (919) 876-2725.

OHIO: Beachwood: 23408 Commerce Park Rd., Beachwood, OH 44122, (216) 464-6100; **Dayton:** Kingsley Bldg., 4124 Linden Ave., Dayton, OH 45432, (513) 258-3877.

OREGON: Beaverton: 6700 SW 105th St., Suite 110, Beaverton, OR 97005, (503) 643-6758.

PENNSYLVANIA: Ft. Washington: 260 New York Dr., Ft. Washington, PA 19034, (215) 643-6450; **Coraopolis:** 420 Rouser Rd., 3 Airport Office Park, Coraopolis, PA 15108, (412) 771-8550.

PUERTO RICO: Hato Rey: Mercantil Plaza Bldg., Suite 505, Hato Rey, PR 00919, (809) 753-8700.

TEXAS: Austin: P.O. Box 2909, Austin, TX 78769, (512) 250-7655; **Richardson:** 1001 E. Campbell Rd., Richardson, TX 75080, (214) 680-5082; **Houston:** 9100 Southwest Frwy., Suite 237, Houston, TX 77036, (713) 778-6592; **San Antonio:** 1000 Central Parkway South, San Antonio, TX 78232, (512) 496-1779.

UTAH: Murray: 5201 South Green SE, Suite 200, Murray, UT 84107, (801) 266-8972.

VIRGINIA: Fairfax: 2750 Prosperity, Fairfax, VA 22031, (703) 849-1400.

WASHINGTON: Redmond: 5010 148th NE, Bldg B, Suite 107, Redmond, WA 98052, (206) 881-3080.

WISCONSIN: Brookfield: 450 N. Sunny Slope, Suite 150, Brookfield, WI 53005, (414) 785-7140.

CANADA: Nepean: 301 Moodie Drive, Mallorn Center, Nepean, Ontario, Canada, K2H9C4, (613) 726-1970. **Richmond Hill:** 280 Centre St. E., Richmond Hill L4C1B1, Ontario, Canada (416) 884-9181; **St. Laurent:** Ville St. Laurent Quebec, 9460 Trans Canada Hwy., St. Laurent, Quebec, Canada H4S1R7, (514) 335-8392.

ARGENTINA: Texas Instruments Argentina S.A.I.C.F.: Esmeralda 130, 15th Floor, 1035 Buenos Aires, Argentina, 1+394-3008.

AUSTRALIA (& NEW ZEALAND): Texas Instruments Australia Ltd.: 6-10 Talavera Rd., North Ryde (Sydney), New South Wales, Australia 2113, 2 + 887-1122; 5th Floor, 418 St. Kilda Road, Melbourne, Victoria, Australia 3004, 3 + 267-4677; 171 Philip Highway, Elizabeth, South Australia 5112, 8 + 255-2066.

AUSTRIA: Texas Instruments Ges.m.b.H.: Industriestrabe B/16, A-2345 Brunn/Gebirge, 2236-846210.

BELGIUM: Texas Instruments N.V. Belgium S.A.: Mercure Centre, Raketstraat 100, Rue de la Fusee, 1130 Brussels, Belgium, 2/720.80.00.

BRAZIL: Texas Instruments Electronicos do Brasil Ltda.: Rua Paes Leme, 524-7 Andar Pinheiros, 05424 Sao Paulo, Brazil, 0815-6166.

DENMARK: Texas Instruments A/S, Mairelundvej 46E, DK-2730 Herlev, Denmark, 2 - 91 74 00.

FINLAND: Texas Instruments Finland OY: Teollisuuskatu 19D 00511 Helsinki 51, Finland, (90) 701-3133.

FRANCE: Texas Instruments France: Headquarters and Prod. Plant, BP 05, 06270 Villeneuve-Loubet, (93) 20-01-01; Paris Office, BP 67 8-10 Avenue Morane-Saulnier, 78141 Velizy-Villacoublay, (3) 946-97-12; Lyon Sales Office, L'Oree D'Ecully, Batiment B, Chemin de la Forestiere, 69130 Ecully, (7) 833-04-40; Strasbourg Sales Office, Le Sebastopol 3, Quai Kleber, 67055 Strasbourg Cedex, (88) 22-12-66; Rennes, 23-25 Rue du Puits Mauger, 35100 Rennes, (99) 31-54-86; Toulouse Sales Office, Le Peripole—2, Chemin du Pigeonnier de la Cepiere, 31100 Toulouse, (61) 44-18-19; Marseille Sales Office, Noilly Paradis—146 Rue Paradis, 13006 Marseille, (91) 37-25-30.

TEXAS INSTRUMENTS

Creating useful products and services for you

GERMANY (Fed. Republic of Germany): Texas Instruments Deutschland GmbH: Haggertystrasse 1, D-8050 Freising, 8161+80-4591; Kurfuerstendamm 195/196, D-1000 Berlin 15, 30+882-7365; III, Hagen 43/Kibbelstrasse, .19, D-4300 Essen, 201-24250; Frankfurter Allee 6-8, D-6236 Eschborm 1, 06196+8070; Hamburgerstrasse 11, D-2000 Hamburg 76, 040+220-1154, Kirchhorsterstrasse 2, D-3000 Hannover 51, 511+648021; Maybachstrabe 11, D-7302 Ostfildern 2-Nelingen, 711+547001; Mixikoring 19, D-2000 Hamburg 60, 40+637+0061; Postfach 1309, Roonstrasse 16, D-5400 Koblenz, 261+35044.

HONG KONG (+ PEOPLES REPUBLIC OF CHINA): Texas Instruments Asia Ltd., 8th Floor, World Shipping Ctr., Harbour City, 7 Canton Rd., Kowloon, Hong Kong, 3 + 722-1223.

IRELAND: Texas Instruments (Ireland) Limited: Brewery Rd., Stillorgan, County Dublin, Eire, 1 831311.

ITALY: Texas Instruments Semiconduttori Italia Spa: Viale Delle Scienze, 1, 02015 Cittaducale (Rieti), Italy, 746 694.1; Via Salaria KM 24 (Palazzo Cosma), Monterotondo Scalo (Rome), Italy, 6+9003241; Viale Europa, 38-44, 20093 Cologno Monzese (Milano), 2 2532541; Corso Svizzera, 185, 10100 Torino, Italy, 11 774545; Via J. Barozzi 6, 40100 Bologna, Italy, 51 355851.

JAPAN: Texas Instruments Asia Ltd.: 4F Aoyama Fuji Bldg., 6-12, Kita Aoyama 3-Chome, Minato-ku, Tokyo, Japan 107, 3-498-2111; Osaka Branch, 5F, Nissho Iwai Bldg., 30 Imabashi 3- Chome, Higashi-ku, Osaka, Japan 541, 06-204-1881; Nagoya Branch, 7F Daini Toyota West Bldg., 10-27, Meieki 4-Chome, Nakamura-ku Nagoya, Japan 450, 52-583-8691.

KOREA: Texas Instruments Supply Co.: 3rd Floor, Samon Bldg., Yuksam-Dong, Gangnam-ku, 135 Seoul, Korea, 2+462-8001.

MEXICO: Texas Instruments de Mexico S.A.: Mexico City, AV Reforma No. 450 — 10th Floor, Mexico, D.F., 06600, 5+514-3003.

MIDDLE EAST: Texas Instruments: No. 13, 1st Floor Mannai Bldg., Diplomatic Area, P.O. Box 26335, Manama Bahrain, Arabian Gulf, 973+274681.

NETHERLANDS: Texas Instruments Holland B.V., P.O. Box 12995, (Bullewijk) 1100 CB Amsterdam, Zuid-Oost, Holland 20+5602911.

NORWAY: Texas Instruments Norway A/S: PB106, Refstad 131, Oslo 1, Norway, (2) 155090.

PHILIPPINES: Texas Instruments Asia Ltd.: 14th Floor, Ba- Lepanto Bldg., 8747 Paseo de Roxas, Makati, Metro Manila, Philippines, 2+8188987.

PORTUGAL: Texas Instruments Equipamento Electronico (Portugal), Lda.: Rua Eng. Frederico Ulrich, 2650 Moreira Da Maia, 4470 Maia, Portugal, 2-948-1003.

SINGAPORE (+ INDIA, INDONESIA, MALAYSIA, THAILAND): Texas Instruments Asia Ltd.: 12 Lorong Bakar Batu, Unit 01-02, Kolam Ayer Industrial Estate, Republic of Singapore, 747-2255.

SPAIN: Texas Instruments Espana, S.A.: C/Jose Lazaro Galdiano No. 6, Madrid 16, 1/458.14.58.

SWEDEN: Texas Instruments International Trade Corporation (Sverigefilialen): Box 39103, 10054 Stockholm, Sweden, 8 - 235480.

SWITZERLAND: Texas Instruments, Inc., Reidstrasse 6, CH-8953 Dietikon (Zuerich) Switzerland, 1-740 2220.

TAIWAN: Texas Instruments Supply Co.: Room 903, 205 Tun Hwan Rd., 71 Sung-Kiang Road, Taipei, Taiwan, Republic of China, 2 + 521-9321.

UNITED KINGDOM: Texas Instruments Limited: Manton Lane, Bedford, MK41 7PA, England, 0234 67466; St. James House, Wellington Road North, Stockport, SK4 2RT, England, 61+442-7162.

BM

TMS320C25 User's Guide

Please use this form to communicate your comments about this document, its organization and subject matter, for the purpose of improving technical documentation.

1) What do you feel are the best features of this document? _____

_____

_____

2) How does this document meet your digital signal processing needs? _____

_____

_____

3) Do you find the organization of this document easy to follow? If not, why?

_____

_____

_____

4) What additions do you think would enhance the structure and subject matter?

_____

_____

_____

5) What deletions could be made without affecting overall usefulness? _____

_____

_____

6) Is there any incorrect or misleading information? _____

_____

_____

7) How would you improve this document? _____

_____

_____

If you would like a reply, please give your name and address below.

Name _____

Company _____ Title _____

Address _____

City/State/Zip _____ Telephone _____

Thank you for your cooperation.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

Texas Instruments Incorporated
M/S 640
P.O. Box 1443
Houston, Texas 77001

Instruction Set Summary (Concluded)

Mnemonic	Description	# Words	Instruction Bit Code
			F E D C B A 9 8 7 6 5 4 3 2 1 0
STXM	Set serial port transmit mode	1	1 1 0 0 1 1 1 0 0 0 1 0 0 0 0 1
SUB	Subtract from accumulator with shift	1	0 0 0 1 ← S → I ← D →
SUBB	Subtract from accumulator with borrow	1	0 1 0 0 1 1 1 1 I ← D →
SUBC	Conditional subtract	1	0 1 0 0 0 1 1 1 I ← D →
SUBH	Subtract from high accumulator	1	0 1 0 0 0 1 0 0 I ← D →
SUBK	Subtract from accumulator short immediate	1	1 1 0 0 1 1 0 1 ← K →
SUBS	Subtract from low accumulator with sign-extension suppressed	1	0 1 0 0 0 1 0 1 I ← D →
SUBT	Subtract from accumulator with shift specified by T register	1	0 1 0 0 0 1 1 0 I ← D →
SXF	Set external flag	1	1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 1
TBLR	Table read	1	0 1 0 1 1 0 0 0 I ← D →
TBLW	Table write	1	0 1 0 1 1 0 0 1 I ← D →
TRAP	Software interrupt	1	1 1 0 0 1 1 1 0 0 0 0 1 1 1 1 0
XOR	Exclusive-OR with accumulator	1	0 1 0 0 1 1 0 0 I ← D →
XORK	Exclusive-OR immediate with accumulator with shift	2	1 1 0 1 ← S → 0 0 0 0 0 1 1 0
ZAC	Zero accumulator	1	1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0
ZALH	Zero low accumulator and load high accumulator	1	0 1 0 0 0 0 0 0 I ← D →
ZALR	Zero low accumulator and load high accumulator with rounding	1	0 1 1 1 1 0 1 1 I ← D →
ZALS	Zero accumulator and load low accumulator with sign-extension suppressed	1	0 1 0 0 0 0 0 1 I ← D →

Assembler Directives (Concluded)

LOAD (Force Load): Defines symbols for other programs.
[<label>] LOAD <symbol>[,<symbol>] [<comment>]

MLIB (Define MACRO Library): Specifies the library containing macro definitions.
[<label>] MLIB '<pathname>' [<comment>]

OPTION (Output Options): Selects several options for the assembler listing output.
[<label>] OPTION <option list> [<comment>]

PAGE (Eject Page): Continues source listing on new page.
[<label>] PAGE [<comment>]

PEND (Program Segment End): Terminates definition of a block of program-relocatable code.
[<label>] PEND [<comment>]

PSEG (Program Segment): Defines succeeding locations as program-relocatable.
[<label>] PSEG [<comment>]

REF (External Reference): Provides access to symbols defined in other programs.
[<label>] REF <symbol>[,<symbol>] [<comment>]

RORG (Relocatable Origin): Defines succeeding locations as program-relocatable and initializes location counter.
[<label>] RORG [[<exp>] /<comment>/]

SREF (Secondary External Reference): Provides secondary access to symbols defined in other programs.
[<label>] SREF <symbol>[,<symbol>] [<comment>]

TEXT (Initialize Text): Places a character string in successive program memory words.
[<label>] TEXT [-]'<string>' [<comment>]

TITL (Page Title): Supplies source listing page titles.
[<label>] TITL '<string>' [<comment>]

UNL (Stop Source Listing): Halts source listing output until the occurrence of a LIST directive.
[<label>] UNL [<comment>]

XEND (Independent Segment End): Terminates definition of an independently stored program segment, definedby EXEC.
[<label>] XEND [<comment>]

TMS320C25
DIGITAL SIGNAL PROCESSOR
Programmer's Reference Card

TI Customer Response Center (CRC) Hotline Number

For help with the TMS320C25, call 1-800-232-3200.

Symbols for Instruction Set Summary

SYMBOL	MEANING
B	4-bit field specifying a bit code
CM	2-bit field specifying compare mode
D	Data memory address field
FO	Format status bit
I	Addressing mode bit
K	Immediate operand field
PA	Port address (PA0 through PA15 are predefined assembler symbols equal to 0 through 15, respectively.)
PM	2-bit field specifying P register output shift code
R	3-bit operand field specifying auxiliary register
S	4-bit left-shift code
X	3-bit accumulator left-shift field

Assembler Directives

AORG (Absolute Origin): Defines succeeding locations as absolute and places a value in the location counter.
[<label>] AORG [<exp> /<comment>/]

BES (Block Ending with Symbol): Advances location counter and assigns a label the value of location following block.
[<label>] BES <exp> [<comment>]

BSS (Block Starting with Symbol): Advances location counter and assigns a label the value of location of first word in block.
[<label>] BSS <exp> [<comment>]

CEND (Common Segment End): Terminates definition of a block of common-relocatable code.
[<label>] CEND [<comment>]

COPY (Copy Source File): Causes source statements to be read from a different file.
[<label>] COPY <file-name> [<comment>]

CSEG (Common Segment): Defines succeeding locations as common-relocatable.
[<label>] CSEG ['<string>' /<comment>/]

DATA (Initialize Word): Places values in successive program memory words.
[<label>] DATA <exp>[,<exp>] [<comment>]

DEF (External Definition): Defines symbols for other programs.
[<label>] DEF <symbol>[,<symbol>] [<comment>]

DEND (Data Segment End): Terminates definition of a block of data-relocatable code.
[<label>] DEND [<comment>]

DORG (Dummy Origin): Defines succeeding locations as a dummy block.
[<label>] DORG <exp> [<comment>]

DSEG (Data Segment): Defines succeeding locations as data-relocatable.
[<label>] DSEG [<comment>]

END (Program End): Terminates the assembly.
[<label>] END [<symbol>/<comment/]

EQU (Define Assembly-Time Constant): Assigns symbol value.
[<label>] EQU <exp> [<comment>]

EXEC (Independent Program Segment): Defines independently stored program segment and loads location counter.
[<label>] EXEC <pma> [<comment>]

IDT (Program Identifier): Names the object module produced.
[<label>] IDT '<string>' [<comment>]

LIST (Restart Source Listing): Resumes source listing.
[<label>] LIST [<comment>]

TEXAS INSTRUMENTS

Instruction Set Summary

Mnemonic	Description	# Words	Instruction Bit Code
			FEDCBA9876543210
ABS	Absolute value of accumulator	1	1100111000011011
ADD	Add to accumulator with shift	1	0000←S→I←D→
ADDC	Add to accumulator with carry	1	01000011 I←D→
ADDH	Add to high accumulator	1	01001000 I←D→
ADDK	Add to accumulator short immediate	1	11001100←K→
ADDS	Add to low accumulator with sign-extension suppressed	1	01001001 I←D→
ADDT	Add to accumulator with shift specified by T register	1	01001010 I←D→
ADLK	Add to accumulator long immediate with shift	2	1101←S→00000010
ADRK	Add to auxiliary register short immediate	1	01111110←K→
AND	AND with accumulator	1	01001110 I←D→
ANDK	AND immediate with accumulator with shift	2	1101←S→00000100
APAC	Add P register to accumulator	1	1100111000010101
B	Branch unconditionally	2	11111111←D→
BACC	Branch to address specified by accumulator	1	1100111000100101
BANZ	Branch on auxiliary register not zero	2	11111011←D→
BBNZ	Branch if TC bit ≠ 0	2	11111001←D→
BBZ	Branch if TC bit = 0	2	11111000←D→
BC	Branch on carry	2	01011101←D→
BGEZ	Branch if accumulator ≥ 0	2	11110100←D→
BGZ	Branch if accumulator > 0	2	11110011←D→
BIT	Test bit	1	1001←B→I←D→
BITT	Test bit specified by T register	1	01010111 I←D→
BIOZ	Branch on I/O status = 0	2	11111010←D→
BLEZ	Branch if accumulator ≤ 0	2	11110010←D→
BLKD	Block move from data memory to data memory	2	11111101 I←D→
BLKP	Block move from program memory to data memory	2	11111100 I←D→
BLZ	Branch if accumulator < 0	2	11110011←D→
BNC	Branch on no carry	2	01011111←D→
BNV	Branch if no overflow	2	11110111←D→
BNZ	Branch if accumulator ≠ 0	2	11110101←D→
BV	Branch on overflow	2	11110001←D→
BZ	Branch if accumulator = 0	2	11110110←D→
CALA	Call subroutine indirect	1	1100111000100100
CALL	Call subroutine	2	11111101←D→
CMPL	Complement accumulator	1	1100111000100111
CMPR	Compare auxiliary register with auxiliary register AR0	1	1100111001010 0 CM
CNFD	Configure block as data memory	1	1100111000000100
CNFP	Configure block as program memory	1	1100111000000101
DINT	Disable interrupt	1	1100111000000001
DMOV	Data move in data memory	1	01010110 I←D→
EINT	Enable interrupt	1	1100111000000000
FORT	Format serial port registers	1	1100111000001111FO
IDLE	Idle until interrupt	1	1100111000011111
IN	Input data from port	1	1000←PA→I←D→
LAC	Load accumulator with	1	0010←S→I←D→
LACK	Load accumulator immediate short	1	11001010←K→
LACT	Load accumulator with shift specified by T register	1	01000010 I←D→
LALK	Load accumulator long immediate with shift	2	1101←S→00000001
LAR	Load auxiliary register	1	00110←R→I←D→
LARK	Load auxiliary register immediate short	1	11000←R→I←K→
LARP	Load auxiliary register pointer	1	0101010110001←R→
LDP	Load data memory page pointer	1	01010010 I←D→
LDPK	Load data memory page pointer immediate	1	1100100←K→
LPH	Load high P register	1	01010011 I←D→
LST	Load status register ST0	1	01010000 I←D→
LST1	Load status register ST1	1	01010001 I←D→
LRLK	Load auxiliary register long immediate	2	11010←R→00000000
LT	Load T register	1	00111100 I←D→

Instruction Set Summary (Continued)

Mnemonic	Description	# Words	Instruction Bit Code
			FEDCBA9876543210
LTA	Load T register and accumulate previous product	1	00111101 I←D→
LTD	Load T register, accumulate previous product, and move data	1	00111111 I←D→
LTP	Load T register and store P register in accumulator	1	00111110 I←D→
LTS	Load T register and subtract previous product	1	01011011 I←D→
MAC	Multiply and accumulate	2	01011101 I←D→
MACD	Multiply and accumulate with data move	2	01011100 I←D→
MAR	Modify auxiliary register	1	01010101 I←D→
MPY	Multiply (with T register, store product in P register)	1	00111000 I←D→
MPYA	Multiply and accumulate previous product	1	00111010 I←D→
MPYK	Multiply immediate	1	101←K→
MPYS	Multiply and subtract previous product	1	00111011 I←D→
MPYU	Multiply unsigned	1	11001111 I←D→
NEG	Negate accumulator	1	1100111000100011
NOP	No operation	1	0101010100000000
NORM	Normalize contents of accumulator	1	11001110 1←D→
OR	OR with accumulator	1	01001101 I←D→
ORK	OR immediate with accumulator with shift	2	1101←S→00000101
OUT	Output data to port	1	1110←PA→I←D→
PAC	Load accumulator with P register	1	1100111000010100
POP	Pop top of stack to low accumulator	1	1100111000011101
POPD	Pop top of stack to data memory	1	01111010 I←D→
PSHD	Push data memory value onto stack	1	01010100 I←D→
PUSH	Push low accumulator onto stack	1	1100111000011100
RC	Reset carry bit	1	1100111000110000
RET	Return from subroutine	1	1100111000100110
RFSM	Reset serial port frame synchronization mode	1	1100111000110110
RHM	Reset hold mode	1	1100111000111000
ROL	Rotate accumulator left	1	1100111000110100
ROR	Rotate accumulator right	1	1100111000110101
ROVM	Reset overflow mode	1	1100111000000010
RPT	Repeat instruction as specified by data memory value	1	01001011 I←D→
RPTK	Repeat instruction as specified by immediate value	1	11001011←K→
RSXM	Reset sign-extension mode	1	1100111000000110
RTC	Reset test/control flag	1	1100111000110010
RTXM	Reset serial port transmit mode	1	1100111000100000
RXF	Reset external flag	1	1100111000001100
SACH	Store high accumulator with shift	1	01101←X→I←D→
SACL	Store low accumulator with shift	1	01100←X→I←D→
SAR	Store auxiliary register	1	01110←R→I←D→
SBLK	Subtract from accumulator long immediate with shift	2	1101←S→00000011
SBRK	Subtract from auxiliary register short immediate	1	01111111←K→
SC	Set carry bit	1	1100111000110001
SFL	Shift accumulator left	1	1100111000011000
SFR	Shift accumulator right	1	1100111000011001
SFSM	Set serial port frame synchronization mode	1	1100111000110111
SHM	Set hold mode	1	1100111000111001
SOVM	Set overflow mode	1	1100111000000011
SPAC	Subtract P register from accumulator	1	1100111000010110
SPH	Store high P register	1	01111101 I←D→
SPL	Store low P register	1	01111100 I←D→
SPM	Set P register output shift mode	1	11001110000010 PM
SQRA	Square and accumulate	1	00111001 I←D→
SQRS	Square and subtract previous product	1	01011010 I←D→
SST	Store status register ST0	1	01111000 I←D→
SST1	Store status register ST1	1	01111001 I←D→
SSXM	Set sign-extension mode	1	1100111000000111
STC	Set test/control flag	1	1100111000110011

July 1986
Printed in U.S.A.

TEXAS INSTRUMENTS

SPRU012