SPRU001B

TMS32010 User's Guide

Digital Signal Processor Products

TEXAS INSTRUMENTS

TMS32010 User's Guide

Digital Signal Processor Products

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments reserves the right to make changes at any time in order to improve design and to supply the best product possible.

Texas Instruments assumes no responsibility for infringement of patents or rights of others based on Texas Instruments applications assistance or product specifications, since TI does not possess full access to data concerning the use or applications of customer's products. TI also assumes no responsibility for customer product designs.

TABLE OF CONTENTS

LIST OF APPENDICES

LIST OF ILLUSTRATIONS

LIST OF TABLES

FOREWORD

Digital Signal Processing (DSP) is concerned with the representation of signals (and the information that they contain) by sequences of numbers, and the transformation or processing of such signal representations by numerical computation procedures.

Since the late 1950's, scientists and engineers in research labs have been touting the virtues of digital signal processing, but practical considerations have prevented widespread application. Now, with the availability of integrated circuits, such as Texas Instruments' TMS320, digital signal processing is leaving the laboratory and entering the world of application. The reasons for this are numerous and compelling. Perhaps the most important reason is that extremely sophisticated signal processing functions can be implemented using digital techniques. Indeed, many of the important DSP techniques are difficult or impossible to implement using analog (continuous-time) methods. It is almost equally important that VLSI technology is best suited to the implementation of digital systems, which are inherently more reliable, more compact, and less sensitive to environmental conditions and component aging than analog systems. Another advantage of the discrete-time approach is the possibility of time sharing a single processing unit among a number of different signal processing functions. This is particularly efficient and cost effective in large systems having many input and output channels. Indeed, until recently, digital processing was only cost effective where it could be applied in large systems. Now, however, with VLSI techniques, low-cost processors such as the TMS32010 are available and a wealth of opportunities exist for the application of DSP techniques.

The potential applications will be found in any area where signals arise as representations of information. In many cases, the signals represent information about the state of some physical system (including human beings). Often, the objective in processing the signal is to prepare the signal for digital transmission to a remote location or for digital storage of the information for later reference. On the other hand, the signal may be processed to remove distortions introduced by transducers, the signal generation environment, or by a transmission system. Still another important class of applications arises when information is automatically extracted from the signal so as to control another system or to infer something about the properties of the system which generated the signal. Some of the more important areas where the above types of processing are of interest include speech communication, geophysical exploration, instrumentation for chemical analysis, image processing for television, audio recording and reproduction, biomedical instrumentation, acoustical noise measurements, sonar, radar, automatic testing of systems, and consumer electronics.

In areas such as speech communication research and geophysical exploration, digital signal processing techniques already have been widely applied using general-purpose digital computers. In other areas, economic factors or processing speed have had limited applications up to recent times. Now, however, these limitations are subsiding rapidly and digital signal processing will soon be widely used in all the above mentioned areas and many more.

Ronald W. Schafer
Russell M. Mersereau
Thomas P. Barnwell, III

Atlanta Signal Processors, Inc.

and

Georgia Institute of Technology
School of Electrical Engineering

INTRODUCTION

1

1. INTRODUCTION

1.1 GENERAL DESCRIPTION

The TMS32010 is the first member of the new TMS320 digital signal processing family, designed to support a wide range of high-speed or numeric-intensive applications. This 16/32-bit single-chip microcomputer 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 TMS320 family contains the first MOS microcomputers capable of executing five million instructions per second. This high throughput is the result of the comprehensive, efficient, and easily programmed instruction set and of the highly pipelined architecture. Special instructions have been incorporated to speed the execution of digital signal processing (DSP) algorithms.

Development support is available for a variety of host computers. This includes a macro assembler, linker, simulator, emulator, and evaluation module.

1.2 TYPICAL APPLICATIONS

The TMS320 family's unique versatility and power give the design engineer a new approach to a variety of complicated applications. In addition, these digital signal processors 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.

Some typical applications of the TMS320 family are listed below.

SIGNAL PROCESSING

- Digital filtering
- Correlation
- Hilbert transforms
- Windowing
- Fast Fourier transforms
- Adaptive filtering
- Waveform generation
- Speech processing
- Radar and sonar processing
- Electronic counter measures
- Seismic processing

TELECOMMUNICATIONS

- Adaptive equalizers
- μ/A law conversion
- Time generators
- High-speed modems
- Multiple-bit-rate modems
- Amplitude, frequency, and phase modulation/demodulation
- Data encryption
- Data scrambling
- Digital filtering
- Data compression
- Spread-spectrum communications

IMAGE PROCESSING

- Pattern recognition
- Image enhancement
- Image compression
- Homomorphic processing
- Radar and sonar processing

HIGH-SPEED CONTROL

- Servo links
- Position and rate control
- Motor control
- Missile guidance
- Remote feedback control
- Robotics

INSTRUMENTATION

- Spectrum analysis
- Digital filtering
- Phase-locked loops
- Averaging
- Arbitrary waveform generation
- Transient analysis

NUMERIC PROCESSING

- Fast multiply/divide
- Double-precision operations
- Fast scaling
- Non-linear function computation (i.e., sin x, e^x)

SPEECH PROCESSING

- Speech analysis
- Speech synthesis
- Speech recognition
- Voice store and forward
- Vocoders
- Speaker authentification

1.3 KEY FEATURES

With an excellent combination of features, the TMS320 family of high-peformance digital signal processors is a cost-effective alternative to custom VLSI devices and bit-slice systems.

- 200-ns instruction cycle
- 288-byte on-chip data RAM
- Microprocessor version — TMS32010
- Microcomputer version — TMS320M10 — (3K-byte on-chip program ROM)
- External program memory expansion to a total of 8K bytes at full speed
- 16-bit instruction/data word
- 32-bit ALU/accumulator
- 16 × 16-bit multiply in 200 ns
- 0 to 15-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
- 2.7-micron NMOS technology
- Single 5-V supply
- 40-pin DIP

The TMS320M10 and the TMS32010 are exactly the same with one exception: the TMS320M10 contains an on-chip masked ROM while the TMS32010 utilizes off-chip program memory.

NOTE

Throughout this document, TMS32010 will refer to both the TMS32010 and the TMS320M10 except where otherwise indicated.

1.4 HOW TO USE THE TMS32010 MANUAL

It is the intent in the design of this user's guide that it be an effective reference book that provides information for both the hardware and the software engineer about the TMS32010 digital signal processor, its architecture, instruction set, electrical specifications, interface methods, and applications.

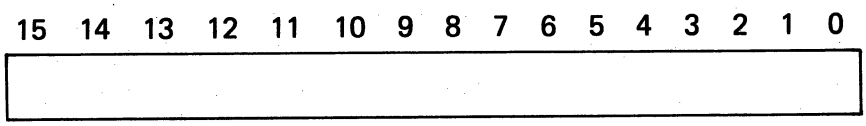
(mnemonic) (title of instruction) (mnemonic)

Addressing:

Operands:

Operation:

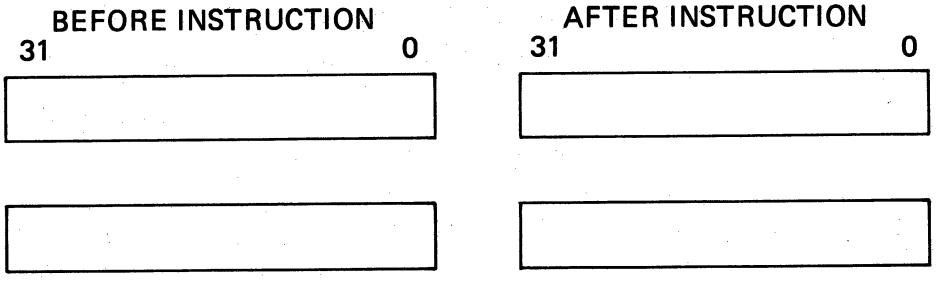
Encoding:



Description:

Words:
Cycles:

Example:



In the architecture section (Section 2), the design of the device and its hardware features are described. The instruction section (Section 3) explains individual instructions in detail. The following format is used for the instruction descriptions in Section 3.4.3 to provide ease of reading and application.

Section 4 on methodology for application development describes the tools, such as an emulator or evaluation module, that are available for developing an individual system and gives an example of TMS32010 software development. In the processor resource management section (Section 5), the engineer finds a description of the common algorithms or practices to be used for any application. He becomes familiar with interface techniques in the input/output design techniques section (Section 6).

The set of macros in the macro language extensions section (Section 7) aids the engineer in programming and in providing templates for further software development. Another special format is used for the macro descriptions in Section 7.2. Each macro instruction is named, followed by a summary table. A flowchart serves to clarify the macro source which is given. Examples of macro use are also presented. This macro description format is as follows:

(mnemonic) (title of macro) (mnemonic)

TITLE: (macro)

NAME: (mnemonic)

OBJECTIVE:

ALGORITHM:

CALLING SEQUENCE:

ENTRY CONDITIONS:

EXIT CONDITIONS:

PROGRAM MEMORY REQUIRED:	(# words)	**DATA MEMORY REQUIRED:**	(# words)
STACK REQUIRED:	(# levels)	**EXECUTION TIME:**	(# cycles)

FLOWCHART:

SOURCE:

EXAMPLE 1:

EXAMPLE 2:

Section 8 on digital signal processing contains an overview of signal processing theory, algorithms, and potential applications. The TMS32010 data sheet appears as Appendix A and the SMJ32010 data sheet as Appendix B. Data descriptions of the evaluation module, macro assembler/linker, simulator, and emulator are presented in Appendix C.

1.4.1 Glossary of Basic TMS32010 Hardware Terms

Table 1-1 lists in alphabetical order the TMS32010 basic hardware units, the symbol for the unit (if any), and the function of that particular unit.

TABLE 1-1 — TMS32010 HARDWARE TERMINOLOGY

UNIT	SYMBOL	FUNCTION
Accumulator	ACC	32-bit accumulator
Arithmetic Logic Unit	ALU	Two-port 32-bit arithmetic logic unit
Auxiliary Registers	AR0, AR1	Two 16-bit registers for indirect addressing of data memory and loop counting control. Nine LSBs of each register are configured as bidirectional counters
Auxiliary Register Pointer	ARP	Single-bit register containing address of current auxiliary register
Data Bus	D Bus	16-bit bus routing data from random access memory
Data Memory Page Pointer	DP	Single-bit register containing page address of data RAM (1 page = 128 words)
Data RAM	–	144 X 16 bit word on-chip random access memory containing data
Interrupt Flag Register	INTF	Single-bit flag register that indicates an interrupt request has occurred (is pending)
Interrupt Mode Register	INTM	Single-bit mode register that masks the interrupt flag
Multiplier	–	16 X 16-bit parallel hardware multiplier
Overflow Flag Register	OV	Single-bit flag register that indicates an overflow in arithmetic operations
Overflow Mode Register	OVM	Single-bit mode register that defines a saturated or unsaturated mode in arithmetic operations
P Register	P	32-bit register containing product of multiply operations
Program Bus	P Bus	16-bit bus routing instructions from program memory
Program Counter	PC	12-bit register containing address of program memory
Program ROM	–	1536 X 16-bit word read only memory containing program code (TMS320M10 only)
Shifter	–	Two shifters: one is a variable 0-15-bit left-shift barrel shifter that moves data from the RAM into the ALU. The other shifter acts on the accumulator when it is being stored in data RAM; it can left-shift by 0, 1, or 4 bits.
Stack	–	4 X 12-bit registers for saving program counter contents in subroutine and interrupt calls
T Register	T	16-bit register containing multiplicand during multiply operations

1.4.2 References

The following list of references, including textbooks, contains useful information regarding functions, operations, and applications of digital processing. These books, in turn, list other references to many useful technical papers.

Andrews, H.C., Hunt, B. R., DIGITAL IMAGE RESTORATION. Englewood Cliffs, N.J.: Prentice-Hall, Inc.,1977.

Brigham, E. Oran, THE FAST FOURIER TRANSFORM. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1974.

Hamming, R.W., DIGITAL FILTERS. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1977.

Morris, L. Robert, DIGITAL SIGNAL PROCESSING SOFTWARE. Ottawa, Canada: Carleton University, 1983.

Oppenheim, Alan V. (Editor), APPLICATIONS OF DIGITAL SIGNAL PROCESSING. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1978.

Oppenheim, Alan V., Schafer, R.W., DIGITAL SIGNAL PROCESSING. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1975.

Rabiner, Lawrence R., Gold, Bernard, THEORY AND APPLICATION OF DIGITAL SIGNAL PROCESSING. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1975.

Rabiner, Lawrence R., Schafer, R.W., DIGITAL PROCESSING OF SPEECH SIGNALS. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1978.

ARCHITECTURE

2. ARCHITECTURE

The TMS320 family utilizes a modified Harvard architecture for speed and flexibility (see Figure 2-1). 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.

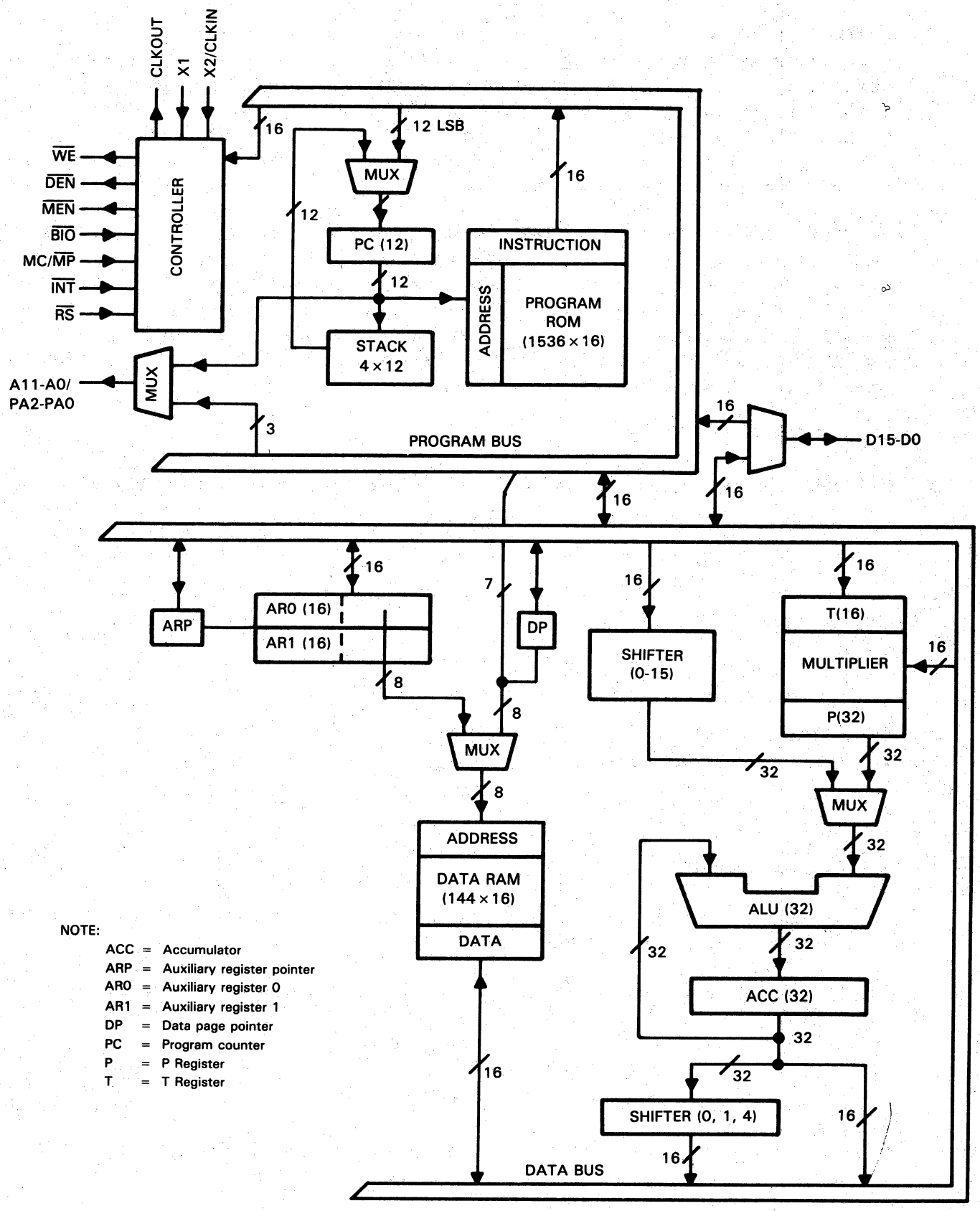
The TMS32010 utilizes hardware to implement functions that other processors typically perform in software. For example, the TMS32010 contains a hardware multiplier to perform a multiplication in a single 200-ns cycle. There is also a hardware barrel shifter for shifting data on its way into the ALU. Finally, extra hardware has been included so that the 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.

2.1 ARCHITECTURAL OVERVIEW

The TMS32010 microcomputers combine the following elements onto a single chip:

- Volatile 144 × 16-word read/write data memory
- Non-volatile 1536 X 16-word program memory (TMS320M10 only)
- Double-precision 32-bit ALU/accumulator
- Fast 200-ns multiplier
- Barrel shifter for shifting data memory words into the ALU
- Shifter that shifts the accumulator into the data RAM
- 16-bit data bus for fetching instruction words from off-chip at full speed
- 4 X 12-bit stack that allows context switching
- Autoincrementing/decrementing registers for indirect data addressing and loop counting
- Single-vectored interrupt
- On-chip oscillator

This section provides a description of these elements. The generic term 'TMS32010' is used to refer collectively to the TMS32010 and TMS320M10.



NOTE:

ACC = Accumulator
ARP = Auxiliary register pointer
AR0 = Auxiliary register 0
AR1 = Auxiliary register 1
DP = Data page pointer
PC = Program counter
P = P Register
T = T Register

FIGURE 2-1 — BLOCK DIAGRAM OF THE TMS320M10

2.1.1 Harvard Architecture

The TMS32010 utilizes a modified Harvard architecture in which program memory and data memory lie in two separate spaces. This permits a full overlap of instruction fetch and execution.

Program memory can lie both on-chip (in the form of the 1536 X 16-word ROM) and off-chip. The maximum amount of program memory that can be directly addressed is 4K X 16-bit words.

Instructions in off-chip program memory are executed at full speed. Fast memories with access times of under 100 ns are required.

Data memory is the 144 X 16-bit on-chip data RAM. Instruction operands are fetched from this RAM; no instruction operands can be directly fetched from off-chip. However, data can be written into the data RAM from a peripheral by using the IN instruction or read from program memory by using the TBLR (table read) instruction. The OUT instruction will write a word from the data RAM to a peripheral, while a TBLW instruction will write a data RAM word to program memory (presumably, off-chip).

Figure 2-2 outlines the overlap of the instruction prefetch and execution. On the falling edge of CLKOUT, the program counter (PC) is loaded with the instruction (load PC2) to be prefetched while the current instruction (execute 1) is decoded and is started to be executed. The next instruction is then fetched (fetch 2) while the current instruction continues to execute (execute 1). Even as another prefetch occurs (fetch 3), both the current instruction (execute 2) and the previous instruction are still executing. This is possible because of a highly pipelined internal operation.

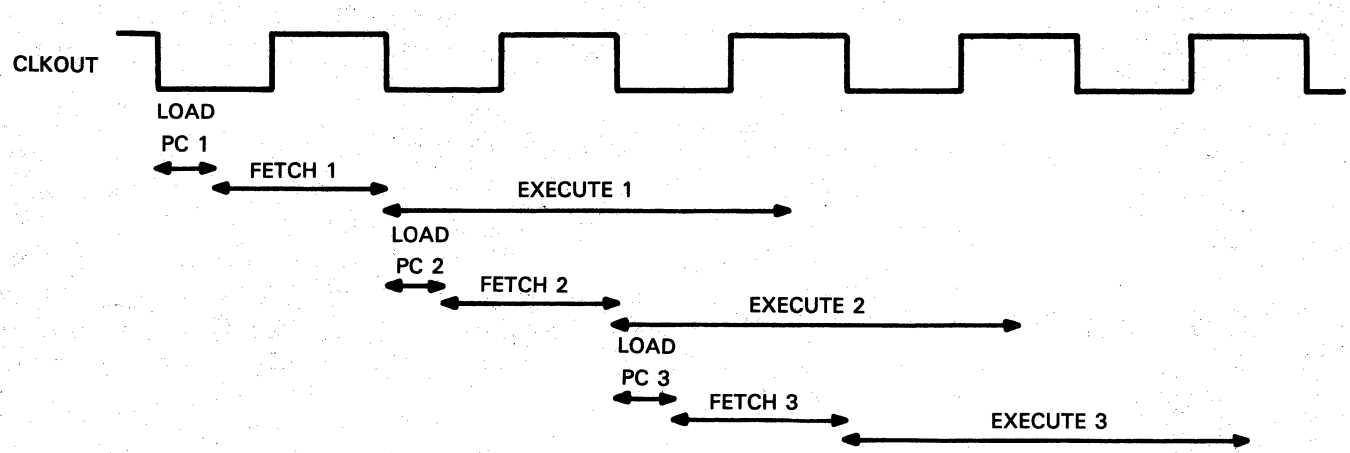


FIGURE 2-2 — HARVARD ARCHITECTURE

2.2 ARITHMETIC ELEMENTS

There are four basic arithmetic elements: the ALU, the accumulator, the multiplier, and the shifters. All arithmetic operations are performed using two's complement arithmetic (see Section 5.1.3).

Most arithmetic instructions will access a word in the data RAM, either directly or indirectly, and pass it through the barrel shifter. This shifter can left-shift a word 0 to 15 bits, depending on the value specified by the instruction. The data word then enters the ALU where it is loaded into or added/subtracted from the accumulator. After a result is obtained in the accumulator, it can be stored in the data RAM. Since the accumulator is 32 bits, both halves must be stored separately. A parallel left-shifter is present at the accumulator output to aid in scaling results as they are being moved to the data RAM.

2.2.1 ALU

The ALU is a general-purpose arithmetic logic unit that operates with a 32-bit data word. The unit will add, subtract, and perform logical operations. The accumulator is always the destination and the primary operand. The result of a logical operation is shown in Table 2-1. A data memory value is the operand for the lower half of the accumulator (bits 15 through 0). Zero is the operand for the upper half of the accumulator.

TABLE 2-1 — ACCUMULATOR RESULTS

FUNCTION	ACCUMULATOR RESULT	
	ACC BITS 31 THROUGH 16	ACC BITS 15 THROUGH 0
XOR	(zero) ⊕ (ACC bits 31-16)	(data memory value) ⊕ (ACC bits 15-0)
AND	(zero) . (ACC bits 31-16)	(data memory value) . (ACC bits 15-0)
OR	(zero) + (ACC bits 31-16)	(data memory value) + (ACC bits 15-0)

2.2.1.1 *Overflow Mode (OVM)*

The OVM register is directly under program control, i.e., it is set by the SOVM instruction and reset by the ROVM instruction. If an overflow occurs when set, the most positive or the most negative representable value of the ALU will be loaded into the accumulator. Whether it is the most positive or the most negative value is determined by the overflow sign. If an overflow occurs when reset, the accumulator is unmodified. (See the SOVM instruction in Section 3.4.3 for further information and an example.)

In signal processing, arithmetic overflows can create special problems. Since overflows can cause swings between very large and very small numbers, they will often result in erratic system behavior. The TMS32010 has been designed with a special overflow mode to compensate for this behavior. When the overflow mode register (OVM) is set by the SOVM instruction (i.e., 1 → OVM), an overflow will cause the largest/smallest representable value of the ALU to be loaded into the accumulator. This models the saturation processes inherent in analog systems. When the overflow mode register (OVM) is reset by the ROVM instructions (i.e., 0 → OVM), overflow results are loaded into the accumulator without modification.

The OVM register can be stored in data memory as a single-bit register that is part of the status register (see Section 2.7). It should not be confused with the overflow flag (OV), explained in Section 2.2.2.1.

2.2.2 Accumulator

The accumulator stores the output from the ALU and is also often an input to the ALU. It operates with a 32-bit word length. The accumulator is divided into a high-order word (bits 31 through 16) and a low-order word (bits 15 through 0). Instructions are provided for storing the high and low-order accumulator words in data memory (SACH and SACL).

2.2.2.1 *Accumulator Status*

Accumulator overflow status can be read from the accumulator overflow flag register (OV). This register will be set if an overflow occurs in the accumulator. Since the OV register is part of the status register (see Section 2.7), OV status can be stored in data memory. Once the overflow flag register is set, only the execution of the branch on overflow (BV) instruction or direct modification of the status register can clear it. This feature permits the examination of overflow results outside of time-critical loops.

A variety of other accumulator conditions can be tested by the branch instructions given in Table 2-2. These instructions will cause a branch to be executed if the condition is met.

TABLE 2-2 — ACCUMULATOR TEST CONDITIONS

INSTRUCTION	ACCUMULATOR CONDITION TESTED
BLZ	< 0
BLEZ	$\leqslant 0$
BGZ	> 0
BGEZ	$\geqslant 0$
BNZ	$<> 0$
BZ	$= 0$

2.2.3 Multiplier

The 16 X 16-bit parallel multiplier consists of three units: the T register, the P register, and the multiplier array. The T register is a 16-bit register that stores the multiplicand, while the P register is a 32-bit register that stores the product.

In order to use the multiplier, the multiplicand must first be loaded into the T register from the data RAM by using one of the following instructions: LT, LTA, or LTD. Then the MPY (multiply) or the MPYK (multiply immediate) instruction is executed. If the MPY instruction is used, the multiplier value is a 16-bit number from the data RAM. If the MPYK instruction is used, the multiplier value is a 13-bit immediate constant derived from the MPYK instruction word; this 13-bit constant is right justified and sign extended. After execution of the MPY or MPYK instruction, the product will be found in the P register. The product can then be added to, subtracted from, or loaded into the accumulator by executing one of the following instructions: APAC, SPAC, LTA, LTD, or PAC.

Pipelined multiply and accumulate operations at 400-ns rates can be accomplished with the LTA/LTD and MPY/MPYK instructions (see Section 3.4.3 for greater detail).

There is no convenient way to restore the contents of the P register without altering other registers. For this reason, special hardware has been incorporated in the TMS32010 to inhibit an interrupt from occurring until the instruction following the MPY or MPYK instruction has been executed. Thus, the MPY or MPYK instruction should always be followed by instructions that combine the P register with the accumulator: PAC, APAC, SPAC, LTA, or LTD. This is almost always done as a logical consequence of the TMS32010 instruction set.

2.2.4 Shifters

There are two shifters available for manipulating data: a barrel shifter for shifting data from the data RAM into the ALU and a parallel shifter for shifting the accumulator into the data RAM.

2.2.4.1 *Barrel Shifter*

The barrel shifter performs a left-shift of 0 to 15 places on all data memory words that are to be loaded into, subtracted from, or added to the accumulator by the LAC, SUB, and ADD instructions.

The barrel shifter zero-fills the low-order bits and sign-extends the 16-bit data memory word to 32 bits by what is called an arithmetic left-shift. An arithmetic left-shift means that the bits to the left of the MSB of the data word are filled with ones if the MSB is a one or with zeros if the MSB is a zero. This is different from a logical left-shift where the bits to the left of the MSB are always filled with zeros. A small amount of code is required to perform an arithmetic right-shift or a logical right-shift (see Section 5.1.2).

The following examples illustrate the barrel shifter's function:

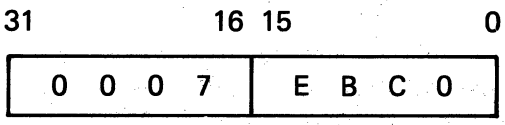
EXAMPLE 1:

Data memory location 20 holds the two's complement number: > 7EBC

The load accumulator (LAC) instruction is executed, specifying a left-shift of 4:

```
LAC 20,4
```

The accumulator would then hold the following 32-bit signed two's complement number:



Since the MSB of > 7EBC is a zero, the upper accumulator was zero-filled.

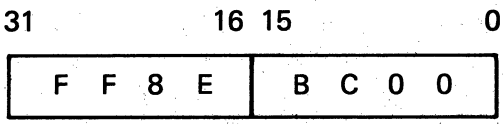
EXAMPLE 2:

Data memory location 30 holds the two's complement number: > 8EBC

The LAC instruction is executed, specifying a left-shift of 8:

```
LAC 30,8
```

The accumulator would then hold the following 32-bit signed two's complement number:



Since the MSB of > 8EBC is a one, the upper accumulator was filled with ones.

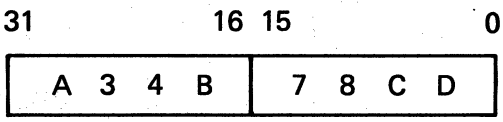
There are also instructions that perform operations with the lower half of the accumulator and a data word without first sign-extending the data word (i.e., treating it as a 16-bit rather than a 32-bit word). The mnemonics of these instructions typically end with an "S," indicating that sign-extension is suppressed (e.g., ADDS, SUBS). Along with the instructions that operate on the upper half of the accumulator, these instructions allow the manipulation of 32-bit precision numbers.

2.2.4.2 *Parallel Shifter*

The parallel shifter is activated only by the store high-order accumulator word (SACH) instruction. This shifter left-shifts the entire 32-bit accumulator and places 16 bits into the data RAM, resulting in a loss of the accumulator's high-order bits. This shifter can execute a shift of only 0, 1, or 4. Shifts of 1 and 4 were chosen to be used with multiplication operations (see Section 5.1.3.1). No right-shift is directly implemented. The following example illustrates the accumulator shifter's function:

EXAMPLE:

The accumulator holds the 32-bit two's complement number:



The SACH instruction is executed, specifying that a left-shift of four be performed on the high-order accumulator word before it is stored in data memory location 40:

```
SACH 40,4
```

Data memory location 40 then contains the following number: > 34B7. The accumulator still retains >A34B78CD.

2.3 DATA MEMORY

Data memory consists of the 144 words of 16-bit width of RAM present on-chip. All non-immediate data operands reside within this RAM.

Sometimes it is convenient to store data operands off-chip and then read them into the on-chip RAM as they are needed. Two means are available for doing this. First, there are the table read (TBLR) and the table write (TBLW) instructions. The table read (TBLR) instruction can transfer values from program memory, either on-chip ROM or off-chip PROM/RAM, to the on-chip data RAM. The table write (TBLW) instruction transfers values from the data RAM to program memory, presumably in the form of off-chip RAM. These instructions take three cycles to execute. The IN and OUT instructions provide another method. The IN instruction reads data from a peripheral and transfers it to the data RAM. With some extra hardware, the IN instruction, together with the OUT instruction, can be used to read and write from the data RAM to large amounts of external storage addressed as a peripheral (see Section 3.4.3). This method is faster since IN and OUT take only two cycles to execute.

2.3.1 Data Memory Addressing

There are three forms of data memory addressing: indirect, direct, and immediate.

2.3.1.1 *Indirect Addressing*

Indirect addressing uses the lower eight bits of the auxiliary registers as the data memory address (see Section 2.4.1). This is sufficient to address all 144 data words; no paging is necessary with indirect addressing. The current auxiliary register is selected by the auxiliary register pointer (ARP). In addition, the auxiliary registers can be made to autoincrement/decrement during any given indirect instruction. The increment/decrement occurs AFTER the current instruction is finished executing.

Some examples of indirect addressing are given below. AR0 and AR1 are predefined assembler constants with values of 0 and 1, respectively.

Each of the following examples should be viewed as a complete program sequence, rather than separate isolated statements. Indirect addressing is indicated by an asterisk (*) in these examples and in the TMS32010 assembler.

EXAMPLE 1:

LARP AR0	Load ARP with a zero. This sets AR0 as the current auxiliary register.
LARK AR0,5	Load AR0 with a 5.
ADD *	Add contents of data memory location 5 to accumulator.
ADD *+	Add contents of data memory location 5 to accumulator and increment AR0. AR0 now equals 6.
ADD *–	Add contents of data memory location 6 to accumulator and decrement AR0. AR0 now equals 5.
ADD *	Add contents of data memory location 5 to accumulator.

EXAMPLE 2:

LARK AR0,10	Load AR0 with the value 10.
LARK AR1,20	Load AR1 with the value 20.
LARP 1	Set ARP to one. This selects AR1 as the current auxiliary register.
ADD *,0,AR0	Add contents of data memory location 20 to accumulator with no shift, then load ARP with 0, selecting AR0 as the current auxiliary register.
ADD *+,0,AR1	Add contents of data memory location 10 to accumulator with no shift, then increment AR0 to have value 11, and load ARP with 1, selecting AR1 as the current auxiliary register.

2.3.1.2 *Direct Addressing*

In direct addressing, seven bits of the instruction word are concatenated with the data page pointer (DP) to form the data memory address. Thus, direct addressing uses the following paging scheme:

DP	MEMORY LOCATIONS
0	0 — 127
1	128 — 144

Usually the second page of data memory contains infrequently accessed system variables, such as those used by the interrupt routine.

DP is part of the status register and thus can be stored in data memory (see Section 2.7).

2.3.1.3 *Immediate Addressing*

The TMS32010 instruction set contains special "immediate" instructions, such as MPYK, LACK, and LARK. These instructions derive data from part of the instruction word rather than from the data RAM.

2.4 REGISTERS

2.4.1 Auxiliary Registers

There are two 16-bit hardware registers, the auxiliary registers, that are not part of the 144 X 16-bit data RAM. These auxiliary registers can be used for three functions: temporary storage, indirect addressing of data memory, and loop control.

Indirect addressing utilizes the least significant eight bits of an auxiliary register as the data memory address (see Section 2.3.1.1).

The branch on auxiliary register not zero (BANZ) instruction permits these registers to also be used as loop counters. BANZ checks if an auxiliary register is zero. If not, it decrements and branches. Thus, loops can be implemented as follows:

```
        LARP    AR0     Load ARP with 0, selecting AR0 as the current auxiliary
                        register.
        LARK    AR0,5   Load AR0 with 5.

LOOP    ADD     *       Indirectly add data memory to accumulator.
        BANZ    LOOP
```

The above program segment adds data memory locations 5 through 0 to the accumulator.

When the auxiliary registers are autoincremented/decremented by an indirect addressing instruction or by BANZ, the lowest nine bits are affected, one more than the lowest eight bits used for indirect addressing (see Figure 2-3A). This counter portion of an auxiliary register is a circular counter, as shown in Figures 2-3B and 2-3C.

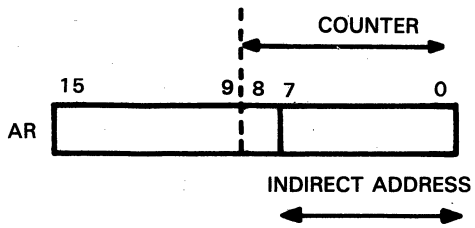


FIGURE 2-3A — AUXILIARY REGISTER COUNTER

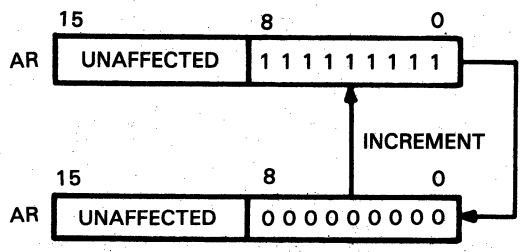


FIGURE 2-3B — AUTOINCREMENT

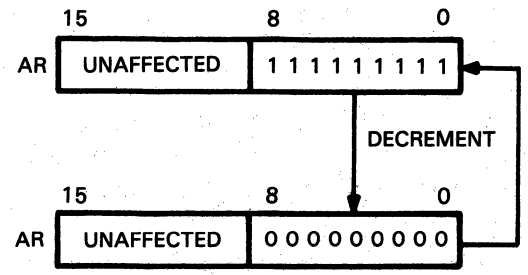


FIGURE 2-3C — AUTODECREMENT

FIGURE 2-3 — INDIRECT ADDRESSING AUTOINCREMENT/DECREMENT

The upper seven bits of an auxiliary register (i.e., bits 9 through 15) are unaffected by any autoincrement/decrement operation. This includes autoincrement of 111111111 (the lowest nine bits go to 0) and autodecrement of 000000000 (the lowest nine bits go to 111111111) ; in each case, bits 9 through 15 are unaffected.

The auxiliary registers can be saved in and loaded from the data RAM with the SAR (store auxiliary register) and LAR (load auxiliary register) instructions. This is useful for performing context saves. SAR and LAR transfer entire 16-bit values to and from the auxiliary registers even though indirect addressing and loop counting utilize only a portion of the auxiliary register.

2.4.2 Auxiliary Register Pointer

The auxiliary register pointer (ARP) is a single bit which is part of the status register. It indicates which auxiliary register is current as follows:

ARP	CURRENT AUXILIARY REGISTER
0	AR0
1	AR1

As part of the status register, the ARP can be stored in memory.

2.5 PROGRAM MEMORY

Program memory consists of up to 4K words of 16-bit width. The TMS320M10 has 1536 words of on-chip ROM, while the TMS32010 is ROMless. Program memory mode of operation is controlled by the MC/$\overline{MP}$ pin.

2.5.1 Modes of Operation

There are two modes of operation defined by the state of the MC/$\overline{MP}$ pin: the microcomputer mode and the microprocessor mode. A one (high) level on this pin places the device in the microcomputer mode, and a zero (low) level places a device in the microprocessor mode.

Table 2-3 illustrates the program memory capability of the TMS32010 microcomputers for each of the two modes of operation enabled by the MC/$\overline{MP}$ pin. Figure 2-4 shows the memory map for each setting of the MC/$\overline{MP}$ pin.

2.5.1.1 *Microcomputer Mode (TMS320M10)*

The microcomputer mode is defined by a one level on the MC/$\overline{MP}$ pin. Even though the TMS320M10 has a 1536 X 16-bit on-chip ROM, only locations 0 through 1523 are available for the user's program. Locations 1524-1535 are reserved by Texas Instruments for testing purposes. The device architecture allows for an additional 2560 words of program memory to reside off-chip.

2.5.1.2 *Microprocessor Mode (TMS320M10 and TMS32010)*

The microprocessor mode is defined by a zero level on the MC/$\overline{MP}$ pin. All 4K words of memory are external in this mode.

TABLE 2-3 — PROGRAM MEMORY FOR THE TMS320 FAMILY

MODEL	PROGRAM MEMORY OPTIONS	MICROCOMPUTER MODE MEMORY	MICROPROCESSOR MODE MEMORY
		MC/$\overline{MP}$ = 1	MC/$\overline{MP}$ = 0
TMS320M10	Microcomputer and microprocessor modes	1536 words on-chip ROM and 2560 words of external memory	4096 words of external memory
TMS32010	Microprocessor mode only	Not available	4096 words of external memory

After reset, the TMS32010 microcomputers will begin execution at location 0. Usually a branch instruction to the reset routine is contained in locations 0 and 1. Upon interrupt, the TMS32010 microcomputers will begin execution at location 2.

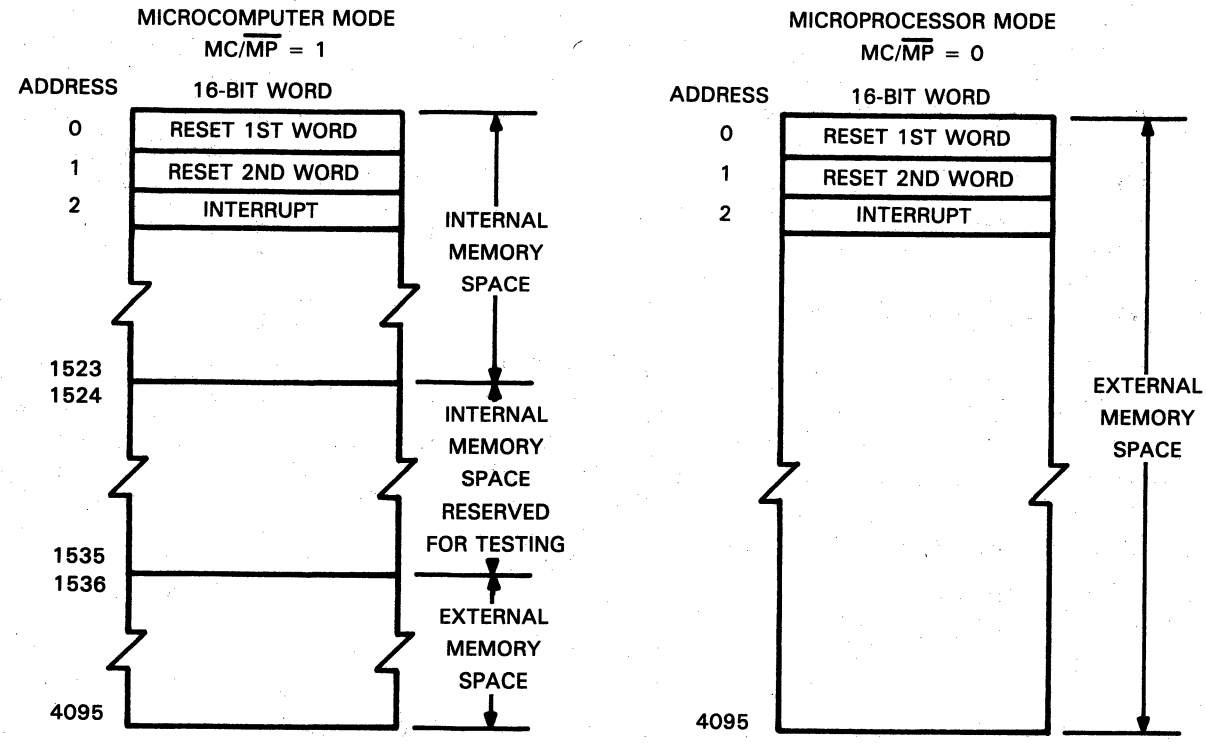


FIGURE 2-4 — TMS320 FAMILY MEMORY MAP

2.5.2 Using External Program Memory

Twelve output pins are available for addressing external memory. These pins are coded A11 (MSB) through A0 (LSB) and contain the buffered outputs of the program counter or the I/O port address. When an instruction is fetched from off-chip, the $\overline{\text{MEN}}$ (memory enable) strobe will be generated to enable the external memory. The instruction word is then transferred to the TMS32010 by means of the data bus. (See Section 2.8.)

When in the microcomputer mode, the TMS320M10 will internally select address locations 1535 and below from the on-chip program memory. The $\overline{\text{MEN}}$ strobe will still become active in this mode, and the address lines A11 through A0 will still output the current value of the program counter although the instruction word will be read from internal program memory.

Figure 2-5 gives an example of external program memory expansion. Even when executing from external memory, the TMS32010 performs at its full 200-ns instruction cycle. Fast memories under 100-ns access time must be used.

$\overline{\text{MEN}}$ is never active at the same time as the $\overline{\text{WE}}$ or $\overline{\text{DEN}}$ signals. In effect, $\overline{\text{MEN}}$ will go low every clock cycle except when an I/O function is being performed by the IN, OUT, or TBLW instructions.

In these multicycle instructions, $\overline{\text{MEN}}$ goes low during the clock cycles in which $\overline{\text{WE}}$ or $\overline{\text{DEN}}$ do not go low.

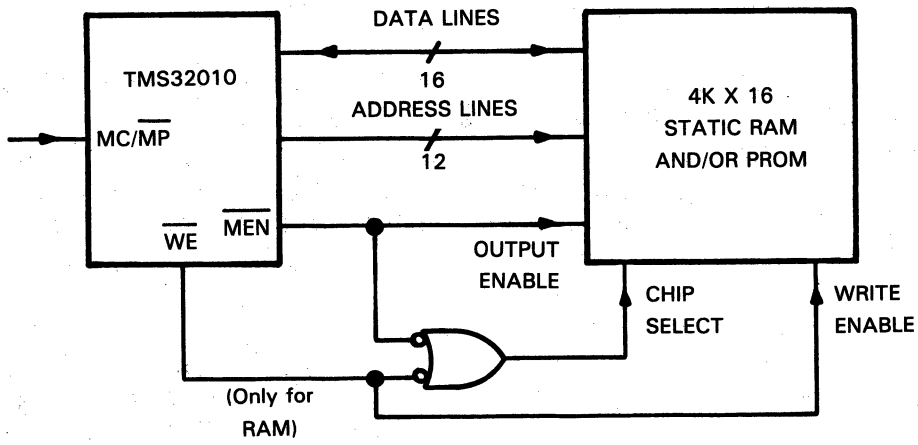


FIGURE 2-5 — EXTERNAL PROGRAM MEMORY EXPANSION EXAMPLE

2.6 PROGRAM COUNTER AND STACK

The program counter (PC) and stack enable the user to perform branches, subroutine calls, and interrupts, and to execute the table read (TBLR) and table write (TBLW) instructions (see Section 3.4.3).

2.6.1 Program Counter

The program counter (PC) is a 12-bit register that contains the program memory address of the next instruction to be executed. The device reads the instruction from the program memory location addressed by the PC and increments the PC in preparation for the next instruction prefetch. The PC is initialized to zero by activating the reset ($\overline{RS}$) line.

In order to permit the use of external program memory, the PC outputs are buffered to the output pins, A11 through A0. The PC outputs appear on the address bus during all modes of operation. The nine MSBs (A11 through A3) of the PC have unique outputs assigned to them, while the three LSBs are multiplexed with the port address field, PA2 through PA0. The port address field is used by the I/O instructions, IN and OUT.

Program memory is always addressed by the contents of the PC. The contents of the PC can be changed by a branch instruction if the particular branch condition being tested is true. Otherwise, the branch instruction simply increments the PC. All branches are absolute, rather than relative, i.e., a 12-bit value derived from the branch instruction word is loaded directly into the PC in order to accomplish the branch.

2.6.2 Stack

The stack is 12 bits wide and four layers deep. The PUSH instruction pushes the twelve LSBs of the accumulator onto the top of stack (TOS). The POP instruction pops the TOS into the twelve LSBs of the accumulator. Following the POP instruction, the TOS can be moved into data memory by storing the low-order accumulator word (SACL instruction). This allows expansion of the stack into the data RAM. From the data RAM, it can easily be copied into program RAM off-chip by using the TBLW instruction. In this way, the stack can be expanded to very large levels.

If the XDS/320 Emulator is used, one level of the stack is reserved by the emulator, reducing the number of available stack levels to three.

2.6.2.1 Stack Overlow

Up to four nested subroutines or interrupts can be accommodated by the device without a stack overflow if the TBLR and TBLW instructions are not executed. Since TBLR and TBLW utilize one level of the stack, only three nested subroutines or interrupts can be accommodated without stack overflow occurring if TBLR or TBLW are executed. If there is a stack overflow, the deepest level of stack will be lost. If the stack is overpopped, the value at the bottom of the stack will become copied into higher levels until it fills the stack.

To handle subroutines and interrupts of much higher nesting levels, part of the data RAM or external RAM can be allocated to stack management. In this case, the top of the stack (TOS) is popped immediately at the start of a subroutine or interrupt routine and stored in RAM. At the end of the subroutine or interrupt routine, the stack value stored in RAM is pushed back onto the TOS before returning to the main routine.

2.7 STATUS REGISTER

The status register, shown in Figure 2-6, consists of five status bits. These status bits can be individually altered through dedicated instructions. In addition, the entire status register can be saved in data memory through the SST instruction. New values can be reloaded into the status register using the LST instruction, with the execption of the INTM bit. The INTM bit cannot be changed through the LST instruction. It can only be changed by the instructions, EINT and DINT (enable, disable interrupts).

FIGURE 2-6 — TMS32010 STATUS REGISTER

Accumulator Oveflow Flag Register (OV) - Zero indicates that the accumulator has not overflowed. One indicates that an overflow in the accumulator has occurred. (See Section 2.2.2.1). The BV (branch on overflow) instruction will clear this bit and cause a branch if it is set.

Overflow Mode Bit (OVM) - Zero means the overflow mode is disabled. One means the overflow mode is enabled (see Section 2.2.1.1). The SOVM instruction loads the OVM bit with a one; the ROVM instruction loads the OVM bit instruction with a zero.

Interrupt Mask Bit (INTM) - Zero means an interrupt is enabled. One means an interrupt is disabled. The EINT instruction loads the INTM bit with a zero; DINT loads the INTM bit with a one. When an interrupt is executed, the INTM register is automatically set to one before the interrupt service routine begins. (See Section 2.10.) Note that the INTM bit can only be altered by executing the EINT and DINT instructions. Unlike the rest of the status bits, the INTM bit cannot be loaded with a new value by the LST instruction.

Auxiliary Register Pointer (ARP) - Zero selects AR0. One selects AR1. The ARP also can be changed by executing the MAR or LARP instruction, or by instructions that permit the indirect addressing option.

Data Memory Page Pointer (DP) - Zero selects first 128 words of data memory, i.e., page zero. One selects last 16 words of data memory, i.e., page one. The DP can also be changed by executing either the LDP or the LDPK instruction.

2.7.1 Saving Status Register

The contents of the status register can be stored in data memory by executing the SST instruction. If the SST instruction is executed using the direct addressing mode, the device automatically stores this information on page one of data memory at the location specified by the instruction. Thus, an SST instruction using the direct addressing mode can only specify an address less than 16, since the second page of memory contains only 16 words. If the indirect addressing mode is selected, then the contents of the status register may be stored in any RAM location selected by the auxiliary register.

The SST instruction does not modify the contents of the status register. Figure 2-7 shows the position of the status bits as they appear in the appropriate data RAM location after execution of the SST instruction.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OV	OVM	INTM	1	1	1	1	ARP	1	1	1	1	1	1	///	DP

/// = don't care

FIGURE 2-7 — STATUS WORD AS STORED BY SST INSTRUCTION

The LST instruction may be executed to load the status register. LST does not assume status bits are on page one, so the DP must be set to one for the LST instruction to access status bits stored on page one. The interrupt mask bit cannot be changed by the LST instruction. However, all other status bits can be changed by this instruction.

2.8 INPUT/OUTPUT FUNCTIONS

2.8.1 IN and OUT

Input and output of data to and from a peripheral is accomplished by the IN and OUT instructions. Data is transferred over the 16-bit data bus to and from the data memory by two independent strobes: data enable ($\overline{DEN}$) and write enable ($\overline{WE}$).

The bidirectional external data bus is always in a high-impedance mode, except when $\overline{WE}$ goes low. $\overline{WE}$ will go low during the first cycle of the OUT instruction and the second cycle of the TBLW instruction.

As shown in Figure 2-8, 128 I/O bits are available for interfacing to peripheral devices: eight 16-bit multiplexed input ports and eight 16-bit multiplexed output ports.

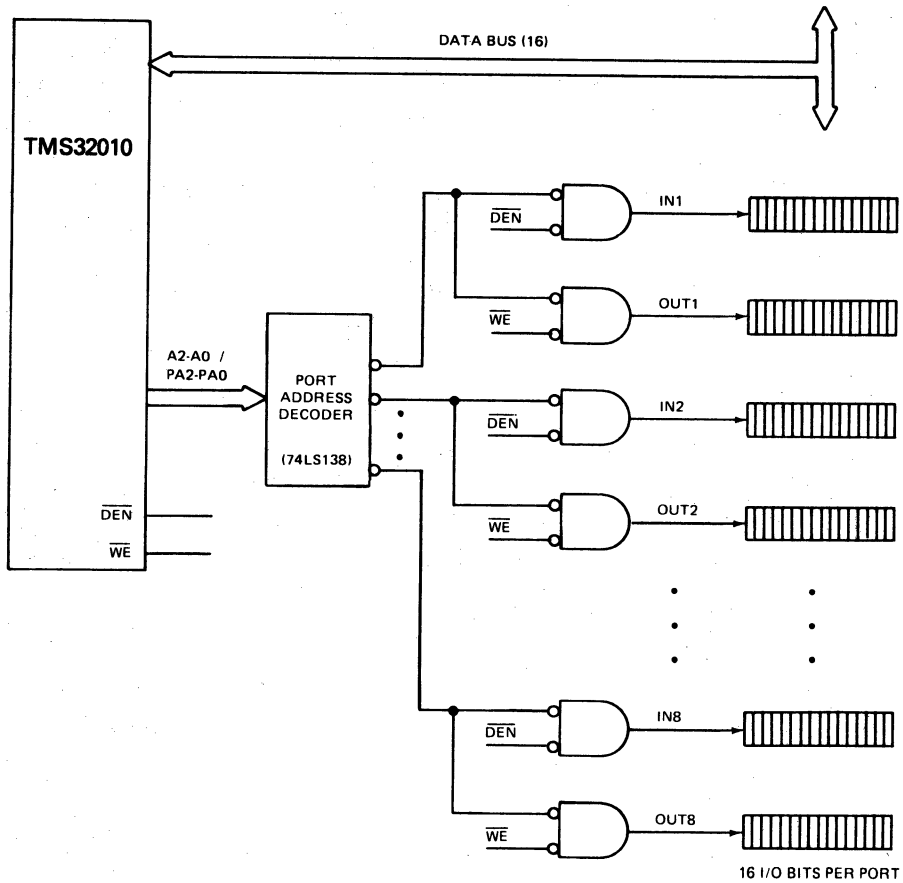


FIGURE 2-8 — EXTERNAL DEVICE INTERFACE

Execution of an IN instruction generates the $\overline{\text{DEN}}$ strobe for transferring data from a peripheral device to the data RAM (see Figure 2-9A). The IN instruction is the only instruction for which $\overline{\text{DEN}}$ will become active. Execution of an OUT instruction generates the $\overline{\text{WE}}$ strobe for transferring data from the data RAM to a peripheral device (see Figure 2-9B). $\overline{\text{WE}}$ becomes active only during the OUT instruction and the table write (TBLW) instruction. See Appendix A, the TMS32010 Data Sheet, for further timing information.

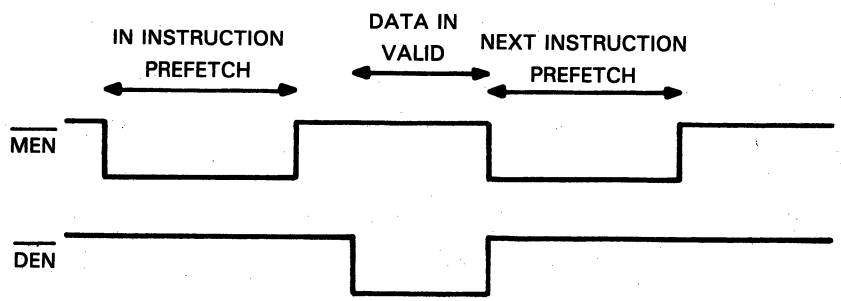


FIGURE 2-9A — INPUT INSTRUCTION TIMING

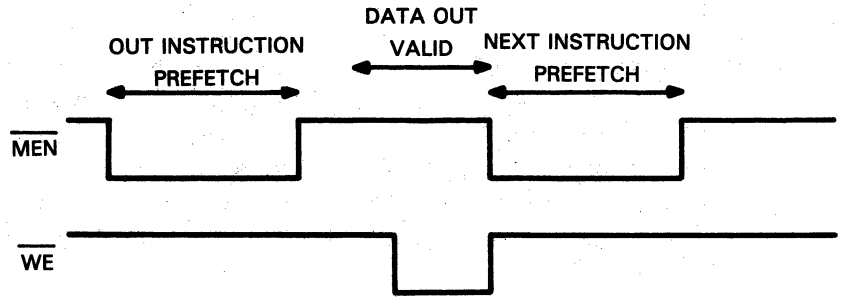


FIGURE 2-9B — OUTPUT INSTRUCTION TIMING

FIGURE 2-9 — INPUT/OUTPUT INSTRUCTION TIMING

The three multiplexed LSBs of the address bus, PA2 through PA0, are used as a port address by the IN and OUT instructions. The remaining higher order bits of the address bus, A11 through A3, are held at logic zero during execution of these instructions.

2.8.2 Table Read (TBLR) and Table Write (TBLW)

The TBLR and the TBLW instructions allow words to be transferred between program and data spaces. TBLR is used to read words from on-chip program ROM or off-chip program ROM/RAM into the data RAM. TBLW is used to write words from on-chip data RAM to off-chip program RAM.

Execution of the TBLR instruction generates $\overline{\text{MEN}}$ strobes to read the word from program memory (see Figure 2-10A). Execution of a TBLW instruction generates a $\overline{\text{WE}}$ strobe (see Figure 2-10B). Note that the $\overline{\text{WE}}$ strobe will be generated and the appropriate data transferred even if the TMS320M10 is in the microcomputer mode and a TBLW is performed to a program location less than 1535.

The dummy prefetch is a prefetch of the instruction following the TBLR or TBLW instructions and is discarded. The instruction following TBLR or TBLW is prefetched again at the end of the execution of the TBLR or TBLW instructions.

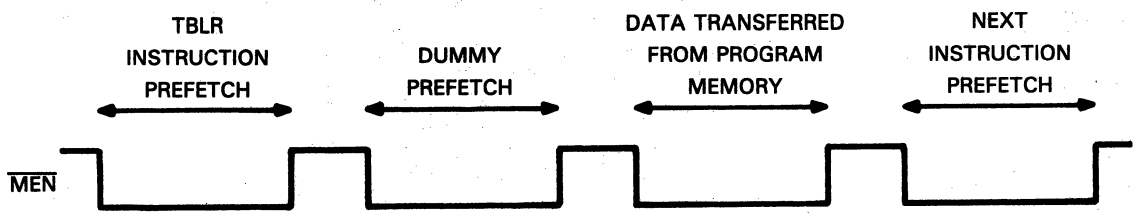


FIGURE 2-10A — TABLE READ INSTRUCTION TIMING

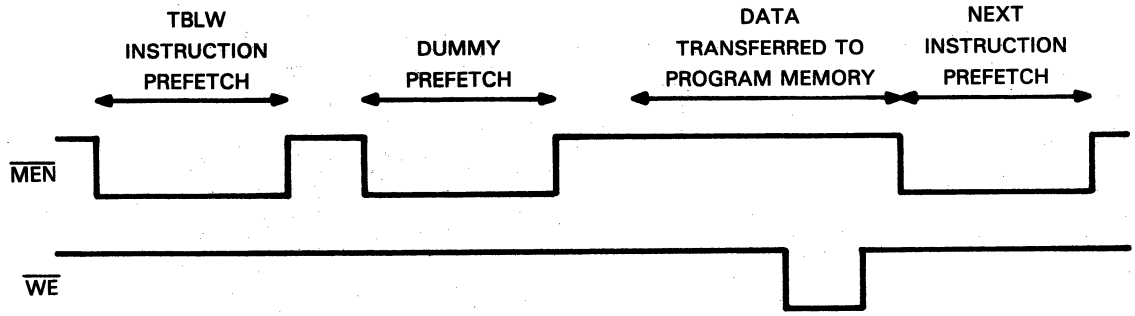


FIGURE 2-10B — TABLE WRITE INSTRUCTION TIMING

FIGURE 2-10 — TABLE READ AND TABLE WRITE INSTRUCTION TIMING

2.8.3 Address Bus Decoding

Since all three interface strobes, $\overline{MEN}$, $\overline{WE}$, and $\overline{DEN}$, are mutually exclusive, there are some very important considerations for those designs that utilize external program memory. Since the OUT and TBLW instructions use only the $\overline{WE}$ signal to indicate valid data, these instructions cannot be distinguished from one another on the basis of the interface strobes. Unless the address bus is decoded, execution of TBLW instructions will write data to peripherals and execution of OUT instructions will overwrite program memory locations 0 through 7. See Section 5-4 for an example of this decoding logic.

No matter what decoding logic is used, it will not be possible to use TBLW to uniquely write to program memory locations 0 through 7. This is because the address bus will be identical for OUT and TBLW, and there will be no way to distinguish between the two instructions.

2

2.9 $\overline{BIO}$ PIN

The $\overline{BIO}$ pin is an external pin which supports bit test and jump operations. When a low is present on this pin, execution of the BIOZ instruction will cause a branch to occur. This pin is sampled every clock cycle and is not latched.

The $\overline{BIO}$ 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.

2.10 INTERRUPTS

The TMS32010's interrupt is generated either by applying a negative-going edge to the interrupt ($\overline{INT}$) pin or by holding the $\overline{INT}$ pin low. A diagrammatic explanation of the TMS32010's internal interrupt circuitry is presented in Figure 2-11.

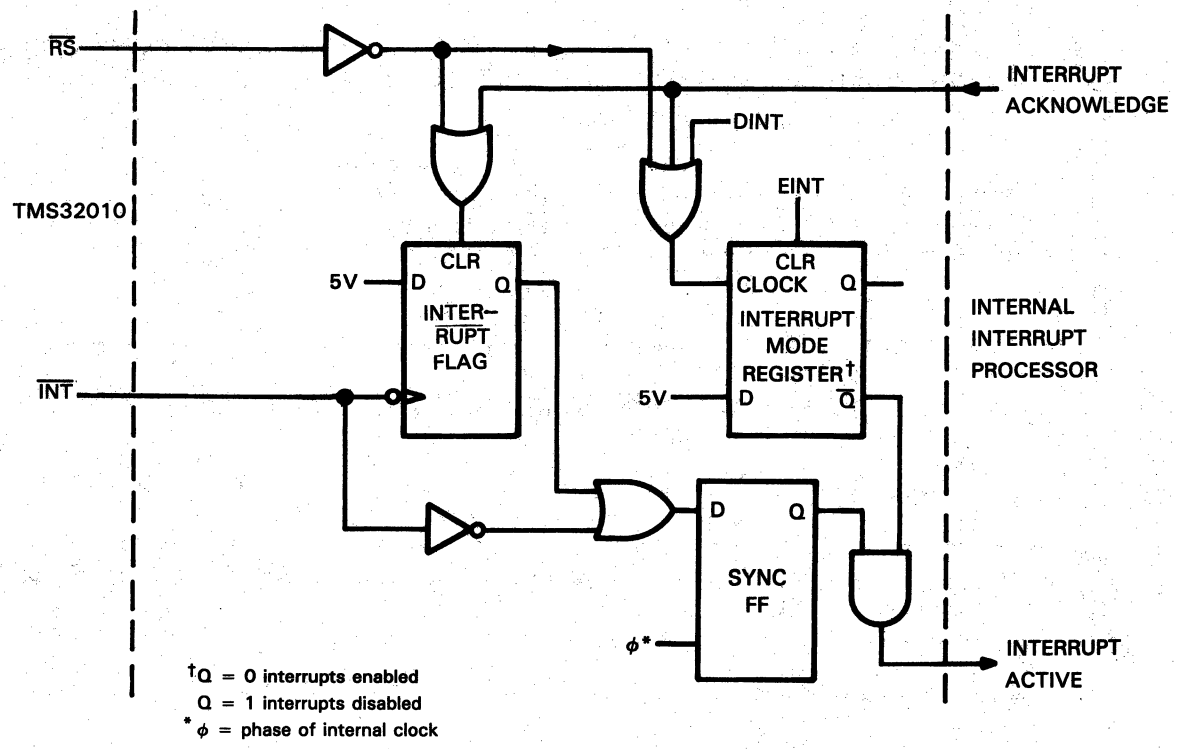


FIGURE 2-11 – SIMPLIFIED INTERRUPT LOGIC DIAGRAM

The Sync FF is a synchronizing flip-flop used to synchronize the external interrupt signal to the TMS32010's internal interrupt circuitry. When interrupts are enabled, an interrupt becomes active either due to a low voltage input on the INT pin or when a negative-edge has been latched into the interrupt flag.

If the interrupt mode register (INTM) is set, then an interrupt active signal to the internal interrupt processor (IIP) becomes valid. The IIP begins interrupt servicing by causing a branch to location 2 in program memory. It will delay interrupt servicing in each of the following cases:

1) Until the end of all cycles of a multicycle instruction,

2) Until the instruction following the MPY or MPYK has completed execution,

3) Until the instruction following EINT has been executed (when interrupts have been previously disabled). This allows the RET instruction to be executed after interrupts become enabled at the end of an interrupt routine.

When the interrupt service routine begins, the IIP sends out an internal interrupt acknowledge signal. This presets the INTM register (disabling interrupts) and clears the interrupt flag.

Figure 2-11 also shows that DINT or a hardware reset will set the INTM register, disabling interrupts, while EINT will clear the INTM register. Interrupts will continue to be latched while they are disabled. Note that DINT or EINT do not affect the interrupt flag.

Figure 2-12 shows the instruction sequence that occurs once an interrupt becomes active. The dummy fetch is an instruction that is fetched but not executed. This instruction will be fetched and executed after the interrupt routine is completed.

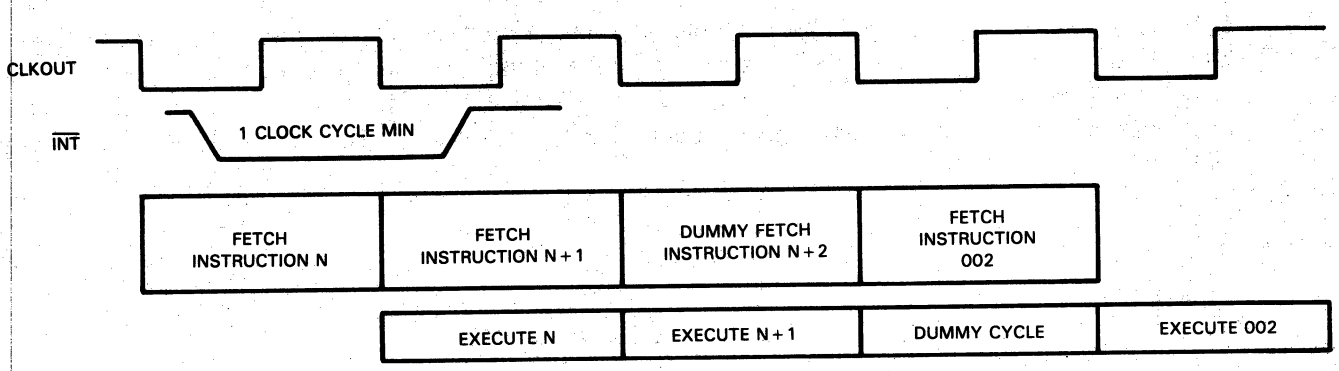


FIGURE 2-12 — INTERRUPT TIMING

See Section 2.14 for interrupt system design recommendations.

2.11 RESET

The reset function is enabled when an active low is placed on the RS pin for a minimum of five clock cycles (see Figure 2-13). The control lines for DEN, WE, and MEN are then forced high, and the data bus (D15 through D0) is tristated. The PC and the address bus (A11 through A0) are then synchronously cleared after the next complete clock cycle from the falling edge of RS. The RS pin also disables the interrupt, clears the interrupt flag register, and leaves the overflow mode register unchanged. The TMS32010 can be held in the reset state indefinitely.

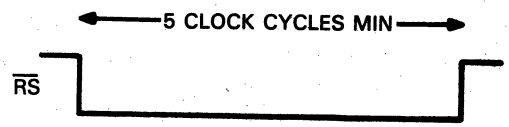


FIGURE 2-13 — RESET TIMING

2.12 CLOCK/OSCILLATOR

The TMS32010 can use either its internal oscillator or an external frequency source for a clock.

Use of the internal oscillator is achieved by connecting a crystal across X1 and X2/CLKIN. The frequency of CLKOUT and the cycle time of the TMS32010 is one-fourth of the crystal fundamental frequency (see Figure 2-14).

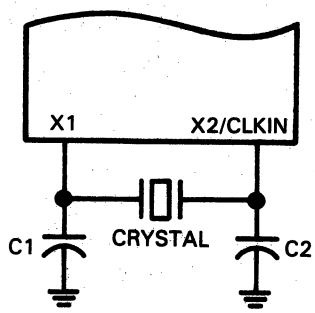


FIGURE 2-14 — INTERNAL CLOCK

An external frequency source can be used by injecting the frequency directly into X2/CLKIN with X1 left unconnected. If an external frequency source is used, a pull-up resistor may be necessary (see Figure 2-15). This is because the high-level voltage of the CLKIN input must be a minimum of 2.8 V while a standard TTL gate, for example, can have a high-level output voltage as low as 2.4 V. The size of the pull-up resistor will depend on such things as the frequency source's high-level output voltage and current and the number of other devices the frequency source will be driving. The resistor should be made as large as possible while still having the CLKIN input specification met.

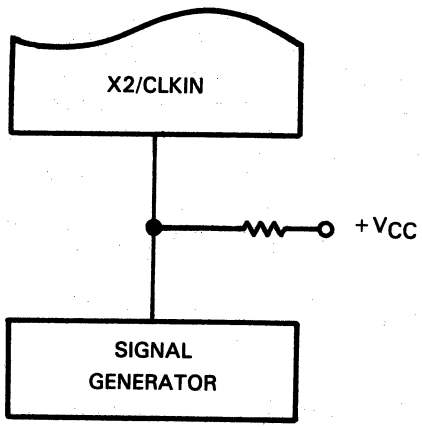


FIGURE 2-15 — EXTERNAL FREQUENCY SOURCE

The delay time between CLKIN and CLKOUT is not specified. This delay time can vary by as much a one CLKOUT cycle and is very temperature dependent. Hardware designs which depend upon this delay time should not be used.

2.13 PIN DESCRIPTIONS

Definitions of the TMS32010 pin assignments and descriptions of the function of each pin are presented in Table 2-4. Figure 2-16 illustrates the TMS32010 pin assignments.

TABLE 2-4 — TMS32010 PIN DESCRIPTIONS

SIGNAL	PIN	I/O	DESCRIPTION
			POWER SUPPLIES
V_{CC}	30		Supply voltage (+ 5 V NOM)
V_{SS}	10		Ground reference
			CLOCKS
X2/CLKIN	8	IN	Crystal input pin for internal oscillator (X2). Also input pin for external oscillator (CLKIN).
X1	7	OUT	Crystal input pin for internal oscillator
CLKOUT	6	OUT	Clock output signal. The frequency of CLKOUT is one-fourth of the oscillator input (external oscillator) or crystal frequency (internal oscillator). Duty cycle is 50 percent.
			CONTROL
$\overline{WE}$	31	OUT	Write Enable. When active (low), $\overline{WE}$ indicates that valid output data from the TMS32010 is available on the data bus. $\overline{WE}$ is only active during the first cycle of the OUT instruction and the second cycle of the TBLW instruction (see Section 3.4.3). $\overline{MEN}$ and $\overline{DEN}$ will always be inactive (high) when $\overline{WE}$ is active.
$\overline{DEN}$	32	OUT	Data Enable. When active (low), $\overline{DEN}$ indicates that the TMS32010 is accepting data from the data bus. $\overline{DEN}$ is only active during the first cycle of the IN instruction (see Section 3.4.3). $\overline{MEN}$ and $\overline{WE}$ will always be inactive (high) when $\overline{DEN}$ is active.
$\overline{MEN}$	33	OUT	Memory Enable. $\overline{MEN}$ will be active low on every machine cycle except when $\overline{WE}$ and $\overline{DEN}$ are active. $\overline{MEN}$ is a control signal generated by the TMS32010 to enable instruction fetches from program memory. $\overline{MEN}$ will be active on instructions fetched from both internal and external memory.

TABLE 2-4 — TMS32010 PIN DESCRIPTIONS (CONTINUED)

SIGNAL	PIN	I/O	DESCRIPTION
			INTERRUPTS
$\overline{RS}$	4	IN	Reset. When an active low is placed on the $\overline{RS}$ pin for a minimum of five clock cycles, $\overline{DEN}$, $\overline{WE}$, and $\overline{MEN}$ are forced high, and the data bus (D15 through D0) is tristated. The program counter (PC) and the address bus (A11 through A0) are then synchronously cleared after the next complete clock cycle from the falling edge of $\overline{RS}$. $\overline{RS}$ also disables the interrupt, clears the interrupt flag register, and leaves the overflow mode register unchanged. The TMS32010 can be held in the reset state indefinitely.
$\overline{INT}$	5	IN	Interrupt. The interrupt signal is generated by applying a negative-going edge to the $\overline{INT}$ pin. The edge is used to latch the interrupt flag register (INTF) until an interrupt is granted by the device. An active low level will also be sensed. (See Section 2.10.)
$\overline{BIO}$	9	IN	I/O Branch Control. If $\overline{BIO}$ is active (low) upon execution of the BIOZ instruction, the device will branch to the address specified by the instruction (see Section 2.9).
			PROGRAM MEMORY MODES
MC/$\overline{MP}$	3	IN	Microcomputer/Microprocessor Mode. A high on the MC/$\overline{MP}$ pin enables the microcomputer mode. In this mode, the user has available 1524 words of on-chip program memory. (Program memory locations 1524 through 1535 are reserved.) The microcomputer mode also allows an additional 2560 words of program memory to reside off-chip. A low on the MC/$\overline{MP}$ pin enables the microprocessor mode. In this mode, the entire memory space is external, i.e., addresses 0 through 4095. (See Section 2.3.1.)
			BIDIRECTIONAL DATA BUS
D15	18	I/O	D15 (MSB) through D0 (LSB). The data bus is always in the high-impedance state except when $\overline{WE}$ is active (low).
D14	17	I/O	
D13	16	I/O	
D12	15	I/O	
D11	14	I/O	
D10	13	I/O	
D9	12	I/O	
D8	11	I/O	
D7	19	I/O	
D6	20	I/O	
D5	21	I/O	
D4	22	I/O	
D3	23	I/O	
D2	24	I/O	
D1	25	I/O	
D0	26	I/O	

TABLE 2-4 — TMS32010 PIN DESCRIPTIONS (CONCLUDED)

SIGNAL	PIN	I/O	DESCRIPTION
			PROGRAM MEMORY ADDRESS BUS AND PORT ADDRESS BUS
A11	27	OUT	Program memory A11 (MSB) through A0 (LSB) and port addresses PA2 (MSB) through PA0 (LSB). Addresses A11 through A0 are always active and never go to high impedance. During execution of the IN and OUT instructions, pins A2 through A0 carry the port addresses PA2 through PA0.
A10	28	OUT	
A9	29	OUT	
A8	34	OUT	
A7	35	OUT	
A6	36	OUT	
A5	37	OUT	
A4	38	OUT	
A3	39	OUT	
A2/PA2	40	OUT	
A1/PA1	1	OUT	
A0/PA0	2	OUT	

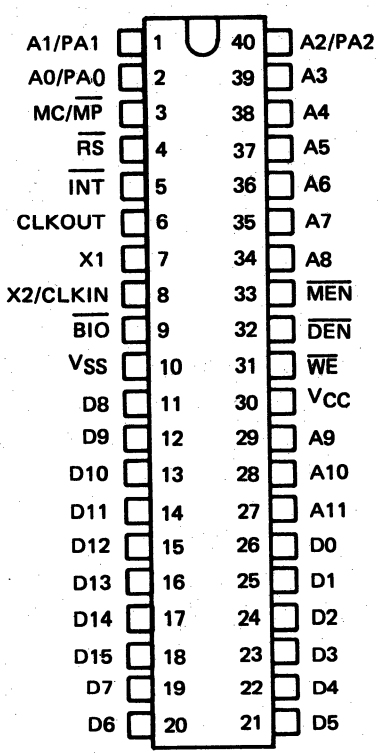


FIGURE 2-16 — TMS32010 PIN ASSIGNMENTS

2.14 INTERRUPT SYSTEM DESIGN

For systems using asynchronous interrupts on the TMS32010, the external hardware shown in Figure 2-17 is recommended to ensure proper execution of interrupts. This hardware synchronizes interrupt input signals with the rising edge of CLKOUT on the TMS32010. The pulse width required for the interrupt input signal is $t_{c(C)}$, which is one TMS32010 clock cycle, plus sufficient setup time for the flip-flop (dependent upon the flip-flop used).

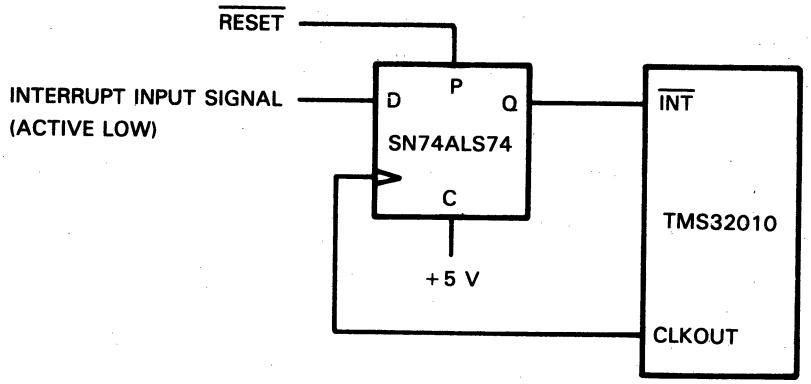


FIGURE 2-17 — INTERRUPT HARDWARE DESIGN

INSTRUCTIONS

3. INSTRUCTIONS

The TMS32010'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, shown in Table 3-2, consists primarily of single-cycle single-word instructions, permitting execution rates of up to five million instructions per second. Only infrequently used branch and I/O instructions are multicycle.

The TMS32010 also contains a number of instructions that shift data as part of an arithmetic operation. These all execute in a single cycle and are very useful for scaling data in parallel with other operations.

3.1 INTRODUCTION

The instruction set contains a full set of branch instructions. Combined with the Boolean operations and shifters, these instructions permit the bit manipulation and bit test capability needed for high-speed control operations. Double-precision operations are also supported by the instruction set. Some examples are ADDH (add to high-order accumulator) and ADDS (add to accumulator with sign extension suppressed), which allow easy manipulation of 32-bit numbers.

The TMS32010's hardware multiplier allows the MPY instruction to be executed in a single cycle. The SUBC (conditional subtract for divide) instruction performs the shifting and conditional branching necessary to implement a divide efficiently and quickly.

Two special instructions, TBLR (table read) and TBLW (table write), allow crossover between data memory and program memory. The TBLR instruction transfers words stored in program memory to the data RAM. This eliminates the need for a coefficient ROM separate from the program ROM, thus permitting the user to make efficient trade-offs as to the amount of ROM dedicated to program or coefficient store. The accompanying instruction, TBLW, transfers words in internal data RAM to an external RAM. In conjunction with TBLR, this instruction allows the use of external RAM to expand the amount of data storage.

When a very large amount of external data must be addressed (i.e., >4K words), TBLR and TBLW can no longer serve as a means of expanding the data RAM. Then it becomes necessary to address external data RAM as a peripheral by using the IN and OUT instructions; these instructions permit a data word to be read into the on-chip RAM in only two cycles. This procedure requires a minimal amount of external logic and permits the accessing of almost unlimited amounts of data RAM. This is very useful for pattern recognition applications, such as speech recognition or image processing.

3.2 ADDRESSING MODES

Three main addressing modes are available with the TMS32010 instruction set direct, indirect, and immediate addressing.

3.2.1 Direct Addressing Mode

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 when performing an interrupt service routine, are stored on the second page.

3.2.2 Indirect Addressing Mode

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.

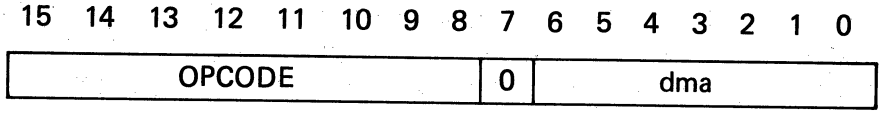
3.2.3. Immediate Addressing Mode

The TMS32010 instruction set contains special "immediate" instructions. These instructions derive data from part of the instruction word rather than from the data RAM. The constant in all immediate instructions may refer to values supplied by an external reference symbol. Some very useful immediate instructions are multiply immediate (MPYK), load accumulator immediate (LACK), and load auxiliary register immediate (LARK).

3.3 INSTRUCTION ADDRESSING FORMAT

The following sections describe the opcode format for the various addressing modes of the TMS32010.

3.3.1 Direct Addressing Format

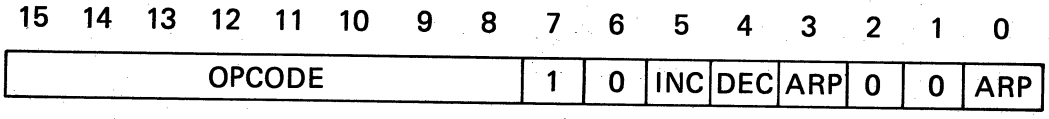


Bit 7 = 0 defines direct addressing mode. The opcode is contained in bits 15 through 8. Bits 6 through 0 contain data memory address.

The 7 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.

3.3.2. Indirect Addressing Format



Bit 7 = 1 defines indirect addressing mode. The opcode is contained in bits 15 through 8. Bits 6 through 0 contain indirect addressing control bits.

Bit 3 and bit 0 control the Auxiliary Register Pointer (ARP). If bit 3 = 0, then the contents of bit 0 are loaded into the ARP after execution of the current instruction. If bit 3 = 1, then the contents of the ARP remain unchanged. ARP = 0 defines the contents of ARO as a memory address. ARP = 1 defines the contents of AR1 as a memory address.

Bit 5 and bit 4 control the auxiliary registers. If bit 5 = 1, then ARP defines which auxiliary register is to be incremented by 1 after execution. If bit 4 = 1, then the ARP defines which auxiliary register is to be decremented by 1 after execution. 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.

3.3.3 Immediate Addressing Format

Included in the TMS32010's instruction set are five immediate operand instructions (LDPK, LARK, MPYK, LACK, and LARP). In these instructions, the operand is contained within the instruction word.

3.3.4 Examples of Opcode Format

1) ADD 9,5 — Add to accumulator the contents of 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

Note: Opcode of the ADD instruction is 0000 and appears in bits 15 through 12. Shift code of 5 appears in bits 11 through 8. Data memory address 9 appears in bits 6 through 0.

2) ADD *+,8 — Add to accumulator the contents of data memory address defined by contents of current auxiliary register. This data is left-shifted 8 bits before being added. The current auxiliary register is auto-incremented by 1.

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	1	0	0	0

Other variations of indirect addressing are as follows:

3) ADD *,8 — As in example 2, but with no auto-increment; opcode would be >0888

4) ADD *−,8 — As in example 2, except that current auxiliary register is decremented by 1; opcode would be >0898

5) ADD *+,8,1 — As in example 2, except that the auxiliary register pointer is loaded with the value 1 after execution; opcode would be >08A1

6) ADD *+,8,0 — As in example 2, except that the auxiliary register pointer is loaded with the value 0 after execution; opcode would be >08A0

3.4 INSTRUCTION SET

The following sections include the symbols and abbreviations that are used in the instruction set summary and in the instruction descriptions, the complete instruction set summary, and a description of each instruction.

All numbers are assumed to be decimal unless otherwise indicated. Hexidecimal numbers are specified by the symbol ">" before the number.

3.4.1. Symbols and Abbreviations

DATn and PRGn are assumed to have the symbolic value of n. They are used to represent any symbol with the value n.

TABLE 3-1 – INSTRUCTION SYMBOLS

SYMBOL	MEANING
ACC	Accumulator
AR	Auxiliary register (AR0 and AR1 are predefined assembler symbols equal to 0 and 1, respectively.)
ARP	Auxiliary register pointer
D	Data memory address field
DATn	Label assigned to data memory location n
dma	Data memory address
DP	Data page pointer
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.
OVM	Overflow (saturation) mode flag bit
P	Product (P) register
PA	Port address (PA0 through PA7 are predefined assembler symbols equal to 0 through 7, respectively)
PC	Program counter
pma	Program memory address
PRGn	Label assigned to program memory location n
R	1-bit operand field specifying auxiliary register
S	4-bit left-shift code
T	T register
TOS	Top of stack
X	3-bit accumulator left-shift field
→	Is assigned to
| |	Indicates an absolute value
< >	Items within angle brackets are defined by user.
[ ]	Items within brackets are optional.
( )	Indicates "contents of"
{ }	Items within braces are alternative items; one of them must be entered.
<>	Angle brackets back-to-back indicate "not equal".
	Blanks or spaces are significant.

3.4.2 Instruction Set Summary

The instruction set summary in the following table consists primarily of single-cycle single-word instructions. Only infrequently used branch and I/O instructions are multicycle.

TABLE 3-2 — 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	→		

TABLE 3-2 – INSTRUCTION SET SUMMARY (CONTINUED)

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 →

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	2	1	0 1 1 1 1 1 1 1 1 0 0 0 1 1 0 1

TABLE 3-2 – INSTRUCTION SET SUMMARY (CONCLUDED)

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

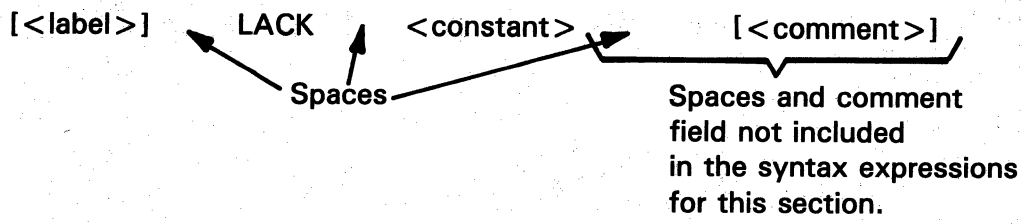
CONTROL INSTRUCTIONS				
MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE INSTRUCTION REGISTER
				15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
DINT	Disable interrupt	1	1	0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1
EINT	Enable interrupt	1	1	0 1 1 1 1 1 1 1 1 0 0 0 0 0 1 0
LST	Load status register	1	1	0 1 1 1 1 0 1 1 I ← 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 memory	3	1	0 1 1 1 1 1 0 1 I ← D →

3.4.3 Instruction Descriptions

Each instruction in the instruction set summary is described in the following pages. The instructions are listed in alphabetical order. An example is provided with each instruction.

Each instruction begins with an assembler syntax expression. Since the comment field which concludes the syntax is optional, it is not included in the syntax expression. A syntax example is given below that shows the spaces that are included and required in the syntax expression, and the optional comment field along with its preceding spaces that has been omitted.



Assembler Syntax: [<label>] ABS

Operands: None

Operation: If (ACC) < 0
Then – (ACC) → ACC

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	1	0	0	0	1	0	0	0

Description: If accumulator is greater than zero, then the accumulator is unchanged by the execution of this instruction. If the accumulator is less than zero, then the accumulator will be replaced by its two's complement value. Note that the hexadecimal number >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.

Words: 1
Cycles: 1

Example: ABS

	BEFORE INSTRUCTION (31 … 0)	AFTER INSTRUCTION (31 … 0)
ACC	>0 0 0 0 1 2 3 4	>0 0 0 0 1 2 3 4
and		
ACC	>F F F F F F F F	>0 0 0 0 0 0 0 1

Assembler Syntax:

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

Operands: $0 \leq shift \leq 15$
$0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (ACC) + (dma) × 2^{shift} → ACC

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 3.3						

Description: Contents of data memory address are left-shifted and added to accumulator. During shifting, low-order bits are zero-filled, and high-order bits are sign-extended. The result is stored in the accumulator.

Words: 1
Cycles: 1

Example:
```
ADD  DAT1,3
or
ADD  *,3        If current auxiliary register contains the value 1.
```

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 1	2	2
ACC	7	23

Note: If the contents of data memory address DAT2 is >8B0E, then the following instruction sequence will leave accumulator with the value > FFF8B0E0.

```
ZAC              Zero accumulator
ADD  DAT2,4      ACC = > FFF8B0E0
```

Assembler Syntax:

Direct Addressing:	[<label>]	ADDH	<dma>
Indirect Addressing:	[<label>]	ADDH	{*|*+|*−}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (ACC) + (dma) × 2^{16} → ACC

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	0	0	0	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	0	0	0	0	1	SEE SECTION 3.3						

Description: Add contents of data memory address to upper half of the accumulator (bits 31 through 16).

Words: 1
Cycles: 1

Example: ADDH DAT5
or
ADDH * If current auxiliary register contains the value 5.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 5	>4	>4
ACC	>0 0 0 0 0 0 1 3	>0 0 0 4 0 0 1 3

Note: This instruction can be used in performing 32-bit arithmetic.

Assembler Syntax:

Direct Addressing:	[<label>]	ADDS	<dma>
Indirect Addressing:	[<label>]	ADDS	{*|*+|*−}[,<ARP>]

Operands: $0 \le dma \le 127$
ARP = 0 or 1

Operation: (ACC) + (dma) → ACC

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	0	0	1	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	0	0	0	1	1	SEE SECTION 3.3						

Description: Add contents of specified data memory location with sign-extension suppressed. The data is treated as a 16-bit positive integer rather than a two's complement integer. Therefore, there is no sign-extension as there is with the ADD instruction.

Words: 1
Cycles: 1

Example: ADDS DAT11
or
ADDS * If current auxiliary register contains the value 11.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 11	>F 0 0 6	>F 0 0 6
ACC	>0 0 0 0 0 0 0 3	>0 0 0 0 F 0 0 9

Notes: The following routines illustrate the difference between the ADD and ADDS instructions. Data memory location DAT1 contains >E007.

```
ZAC             Zero ACC
ADDS  DAT1      ACC = >0000E007

ZAC             Zero ACC
ADD   DAT1,0    ACC = >FFFFE007
```

The ADDS instruction can be used in implementing 32-bit arithmetic.

Assembler Syntax:

Direct Addressing:	[<label>]	AND	<dma>
Indirect Addressing:	[<label>]	AND	{*|*+|*−}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: Zero. AND. high-order ACC bits:. (dma). AND. low-order ACC bits → ACC

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 3.3						

Description: The low-order bits of the accumulator are ANDed with the contents of the specified data memory address and concatenated with all zeroes ANDed with the high-order bits of the accumulator. The AND operation follows the truth table below.

DATA MEMORY BIT	ACC BIT (BEFORE)	ACC BIT (AFTER)
0	0	0
0	1	0
1	0	0
1	1	1

Words: 1
Cycles: 1

Example: AND DAT16
or
AND * If current auxiliary register contains the value 16.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 16	>0 0 F F	>0 0 F F
ACC	>1 2 3 4 5 6 7 8	>0 0 0 0 0 0 7 8

Note: This instruction is useful for examining bits of a word for high-speed control applications.

Assembler Syntax: [<label>] APAC

Operands: None

Operation: (ACC) + (P)→ ACC

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	1	0	0	0	1	1	1	1

Description: The contents of the P register, the result of a multiply, are added to the contents of the accumulator and the result is stored in the accumulator.

Words: 1
Cycles: 1

Example: APAC

	BEFORE INSTRUCTION	AFTER INSTRUCTION
P	64	64
ACC	32	96

Note: This instruction is a subset of the LTA and LTD instructions.

Assembler Syntax: [<label>] B <pma>

Operands: $0 \leqslant pma < 2^{12}$

Operation: pma → PC

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	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or a numeric address.

Words: 2
Cycles: 2

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

BANZ

Branch on Auxiliary Register Not Zero

BANZ

Assembler Syntax: [<label>] BANZ <pma>

Operands: $0 \leq pma < 2^{12}$

Operation:
If (AR bits 8 through 0) <> 0
Then (AR) − 1 → AR and pma → PC
Else (PC) + 2 → PC
(AR) − 1 → AR

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	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the lower nine bits of the current auxiliary register are not equal to zero, then the auxiliary register is decremented, and the address contained in the following word is loaded into the program counter. If these bits equal zero, the current program counter is incremented and AR also is decremented. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or numeric address.

Words: 2
Cycles: 2

Example: BANZ PRG35

	BEFORE INSTRUCTION	AFTER INSTRUCTION
AR	1	0
PC	46	35
or		
AR	0	>1 F F
PC	46	48

Note: This instruction can be used for loop control with the auxiliary register as loop counter. The auxiliary register is decremented after testing for zero. The auxiliary registers also behave as modulo 512 counters.

BGEZ

Branch if Accumulator Greater Than or Equal to Zero

BGEZ

Assembler Syntax: [<label>] BGEZ <pma>

Operands: $0 \leq pma < 2^{12}$

Operation: If (ACC) ≥ 0
Then pma → PC
Else (PC) + 2 → PC

Encoding:

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	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the contents of the accumulator are greater than or equal to zero, branch to the specified program memory location. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or a numeric address.

Words: 2
Cycles: 2

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>

Operands: $0 \leq pma < 2^{12}$

Operation: If (ACC) > 0
Then pma → PC
Else (PC) + 2 → PC

Encoding:

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	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the contents of the accumulator are greater than zero, branch to the specified program memory location. Branch to location in program specified by the program memory address (pma). Pma can be either a symbolic or a numeric address.

Words: 2
Cycles: 2

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

Assembler Syntax: [<label>] BIOZ <pma>

Operands: $0 \leqslant pma < 2^{12}$

Operation: If $\overline{BIO} = 0$
Then pma → PC
Else (PC) + 2 → PC

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	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the $\overline{BIO}$ pin is active low, then branch to specified memory location. Otherwise, the program counter is incremented to the next instruction. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or a numeric address.

Words: 2
Cycles: 2

Example: BIOZ PRG64 If the $\overline{BIO}$ pin is active low, then a branch to location 64 occurs. Otherwise, the program counter is incremented.

Note: This instruction can be used in conjunction with the $\overline{BIO}$ pin to test if peripheral is ready to deliver an input. This type of interrupt is preferable when performing time-critical loops.

Branch if Accumulator Less Than or Equal to Zero

BLEZ

Assembler Syntax: [<label>] BLEZ <pma>

Operands: $0 \leq pma < 2^{12}$

Operation: If (ACC) $\leq$ 0
Then pma → PC
Else (PC) + 2 → PC

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	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the contents of the accumulator are less than or equal to zero, branch to the specified program memory location. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or a numeric address.

Words: 2
Cycles: 2

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

Assembler Syntax: [<label>] BLZ <pma>

Operands: $0 \leq pma < 2^{12}$

Operation:
If (ACC) < 0
Then pma → PC
Else (PC) + 2 → PC

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	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the contents of the accumulator are less than zero, branch to the specified program memory location. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or numeric address.

Words: 2
Cycles: 2

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

Assembler Syntax: [<label>] BNZ <pma>

Operands: $0 \leq pma < 2^{12}$

Operation: If (ACC) <> 0
Then pma → PC
Else (PC) + 2 → 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	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the contents of the accumulator are not equal to zero, branch to the specified program memory location. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or numeric address.

Words: 2
Cycles: 2

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

Assembler Syntax: [<label>] BV <pma>

Operands: $0 \leq pma < 2^{12}$

Operation: If overflow flag = 1
Then pma→PC and 0→overflow flag
Else (PC) + 2 → PC

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	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the overflow flag has been set, then a branch to the program address occurs and the overflow flag is cleared. Otherwise, the program counter is incremented to the next instruction. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or a numeric address.

Words: 2
Cycles: 2

Example: BV PRG610 If an overflow has occurred since the overflow flag was last cleared, then 610 is loaded into the program counter. Otherwise, the program counter is incremented.

Assembler Syntax: [<label>] BZ <pma>

Operands: $0 \leq pma < 2^{12}$

Operation: If (ACC) = 0
Then pma → PC
Else (PC) + 2 → 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	1	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the contents of the accumulator are equal to zero, branch to the specified program memory location. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or numeric address.

Words: 2
Cycles: 2

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

Assembler Syntax: [<label>] CALA

Operands: None

Operation: (PC) + 1 → TOS
(ACC bits 11 through 0) → PC

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	1	0	0	0	1	1	0	0

Description: The current program counter is incremented and pushed onto the top of the stack. Then, the contents of the 12 least significant bits of the accumulator are loaded into the PC.

Words: 1
Cycles: 2

Example: CALA

	BEFORE INSTRUCTION	AFTER INSTRUCTION
PC	25	83
ACC	83	83
STACK	32 75 84 49	26 32 75 84

Note: This instruction is used to perform computed subroutine calls.

Assembler Syntax: [<label>] CALL <pma>

Operands: $0 \leq pma < 2^{12}$

Operation: (PC) + 2 → TOS
pma → PC

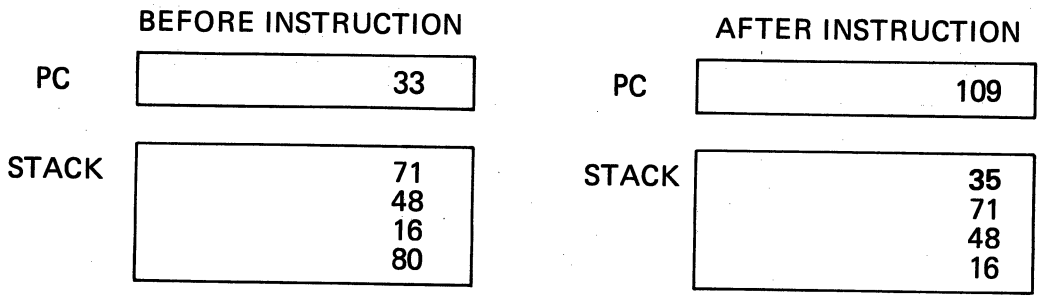
Encoding:

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

Description: The current program counter is incremented and pushed onto the top of the stack. Then, the program memory address is loaded into the PC.

Words: 2
Cycles: 2

Example: CALL PRG109



Assembler Syntax: [<label>] DINT

Operands: None

Operation: 1→INTM

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	1	0	0	0	0	0	0	1

Description: The interrupt-mode flag (INTM) bit is set to logic 1. When this flag is set, any further maskable interrupts are disabled.

Words: 1
Cycles: 1

Example: DINT

Assembler Syntax:

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

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (dma) → dma + 1

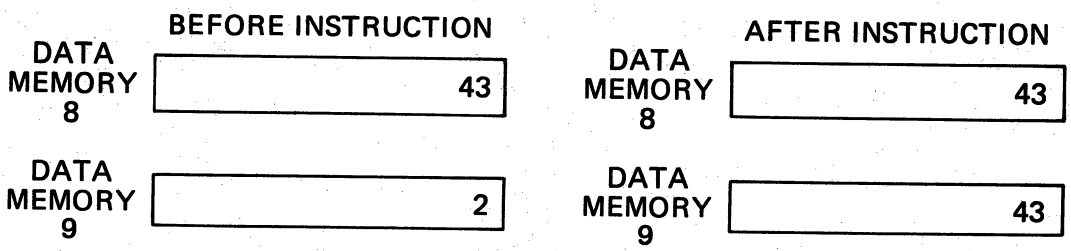
Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	0	0	1	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	1	0	0	1	1	SEE SECTION 3.3						

Description: The contents of the specified data memory address are copied into the contents of the next higher address.

Words: 1
Cycles: 1

Example: DMOV DAT8
or
DMOV * If current auxiliary register contains the value 8.



Note: DMOV is an instruction that can be associated with Z^{-1} in signal flow graphs. It is a subset of the LTD instruction. See LTD for more information.

Assembler Syntax: [<label>] EINT

Operands: None

Operation: 0→INTM

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	1	0	0	0	0	0	1	0

7F82

Description: The interrupt-mode flag (INTM) in the status register is cleared to logic 0. When this flag is not set, maskable interrupts are enabled.

3

Words: 1
Cycles: 1

Example: EINT

Assembler Syntax:

Direct Addressing:	[<label>]	IN	<dma>,<PA>
Indirect Addressing:	[<label>]	IN	{*|*+|*−},<PA>[,<ARP>]

Operands: $0 \leq dma \leq 127$
$0 \leq PA \leq 7$
ARP = 0 or 1

Operation: PA → address lines PA2-PA0
Data bus D15-D0 → dma

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	0	0	PORT ADDRESS			0	DATA MEMORY ADDRESS						
Indirect:	0	1	0	0	0	PORT ADDRESS			1	SEE SECTION 3.3						

Description: The IN instruction reads data from a peripheral and places it in data memory. It is a two-cycle instruction. During the first cycle, the port address is sent to address lines A2/PA2-A0/PA0. $\overline{DEN}$ goes low during the same cycle, strobing in the data which the addressed peripheral places on the data bus, D15-D0.

Words: 1
Cycles: 2

Example:

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

LARK    1,20        Load AR1 with decimal 20.
LARP    1           Load ARP with 1.
IN      *-,PA1,0    Read in word from peripheral on port address 1.
                    Store in data memory location 20. Decrement
                    AR1 to 19. Load the ARP with 0.
```

Notes: When the TMS32010 outputs address onto the three LSBs of address lines, the nine MSBs are zeroed.

Instruction causes the $\overline{DEN}$ line to go low during the first clock cycle of this instruction's execution. $\overline{MEN}$ remains high when $\overline{DEN}$ is active.

Assembler Syntax:

Direct Addressing:	[<label>]	LAC	<dma>[,<shift>]
Indirect Addressing:	[<label>]	LAC	{*|*+|*−}[,<shift>[,<ARP>]]

Operands: $0 \leq shift \leq 15$
$0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (dma) × 2^{shift} → ACC

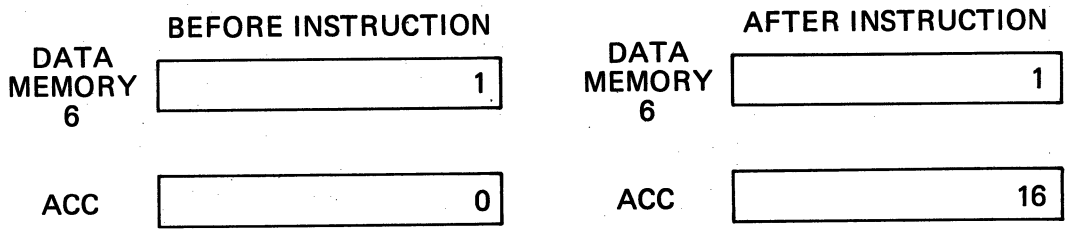
Encoding:

	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 3.3						

Description: Contents of data memory address are left-shifted and loaded into the accumulator. During shifting, low-order bits are zero-filled and high-order bits are sign-extended.

Words: 1
Cycles: 1

Example:
LAC DAT6,4
or
LAC *,4 If current auxiliary register contains the value 6.



Assembler Syntax: [<label>] LACK <constant>

Operands: $0 \leq \text{constant} \leq 255$

Operation: constant→ACC

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

Words: 1
Cycles: 1

Example: LACK 15

	BEFORE INSTRUCTION		AFTER INSTRUCTION
ACC	31	ACC	15

Note: If a constant longer than eight bits is used, the XDS/320 assembler will truncate it to eight bits. No error message will be given.

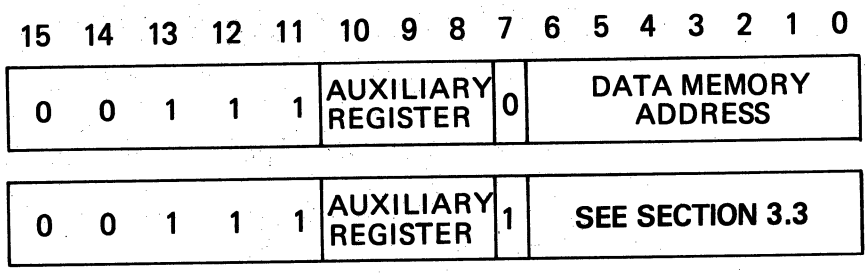
Assembler Syntax:

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

Operands: $0 \leq dma \leq 127$
AR = 0 or 1
ARP = 0 or 1

Operation: (dma)→AR

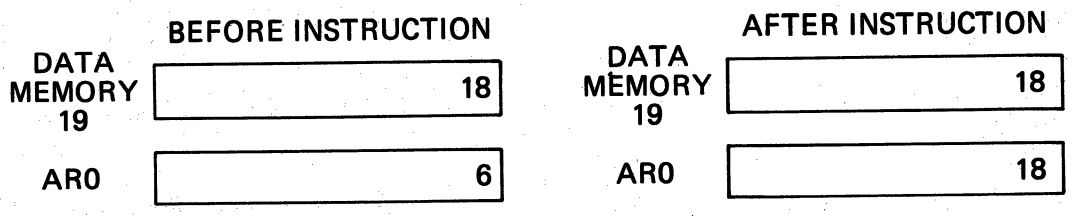
Encoding:



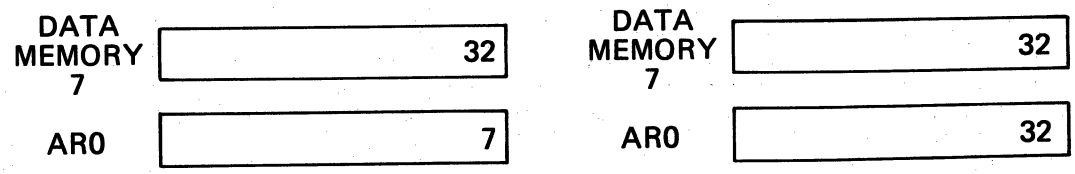
Description: The contents of the specified data memory address are loaded into the designated auxiliary register.

Words: 1
Cycles: 1

Example: LAR AR0,DAT19



also, LARP 0
LAR AR0,* —



Notes: AR0 is not decremented after the LAR instruction. Generally as in the above case, if indirect addressing with autodecrement is used with LAR to load the current auxiliary register, the new value of the auxiliary register is not decremented as a result of instruction execution. The analagous case is true with autoincrement.

LAR and its companion instruction SAR (store auxiliary registers) should be used to store and load the auxiliary during subroutine calls and interrupts.

If an auxiliary register is not being used for indirect addressing, LAR and SAR enable it to be used as an additional storage register, especially for swapping values between data memory locations.

LARK — Load Auxiliary Register with Eight-Bit Constant

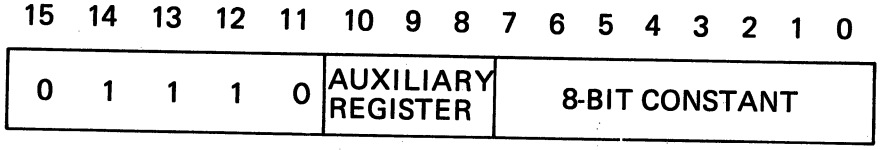
LARK

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

Operands: 0 ⩽ constant ⩽ 255
AR = 0 or 1

Operation: constant→AR

Encoding:



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

Words: 1
Cycles: 1

Example: LARK AR0,21

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

If a constant longer than eight bits is used, the XDS/320 assembler will truncate it to eight bits. No error message will be given.

Assembler Syntax: [<label>] LARP <constant>

Operands: $0 \leq$ constant ≤ 1

Operation: constant → ARP

Encoding:

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

Description: Load a one-bit constant identifying the desired auxiliary register into the auxiliary register pointer.

Words: 1
Cycles: 1

Example: LARP 1 Any succeeding instructions will use auxiliary register 1 for indirect addressing.

Note: This instruction is a subset of MAR.

Assembler Syntax:

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

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: LSB of (dma) → DP (DP = 0 or 1)

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	1	1	1	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	1	1	1	1	1	SEE SECTION 3.3						

Description: The least significant bit of the contents of the specified data memory address is loaded into the data memory page pointer register (DP). All higher-order bits are ignored in the data word. DP = 0 defines page 0 which contains words 0-127. DP = 1 defines page 1 which contains words 128-143.

Words: 1
Cycles: 1

Example:

LDP	DAT1	LSB of location DAT1 is loaded into data page pointer.
or		
LDP	*,1	LSB of location currently addressed by auxiliary register is loaded into data page pointer. ARP is set to one.

Assembler Syntax: [<label>] LDPK <constant>

Operands: $0 \leq \text{constant} \leq 1$

Operation: constant→DP

Encoding:

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

Description: The one-bit constant is loaded into the data memory page pointer register (DP). DP = 0 defines page 0 which contains words 0-127. DP = 1 defines page 1 which contains words 128-143.

Words: 1
Cycles: 1

Example: LDPK 0 Data page pointer is set to zero.

Assembler Syntax:

Direct Addressing:	[<label>]	LST	<dma>
Indirect Addressing:	[<label>]	LST	{*|*+|*−}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (dma) → status bits

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 3.3						

Description: Restores the contents of the status register as saved by the store status (SST) instruction from a data memory word.

Words: 1
Cycles: 1

Example:

```
LARP 0
LST  *,1
```

The data memory word addressed by the contents of auxiliary register 0 replaces the status bits. The auxiliary register pointer becomes 1.

Note: This instruction is used to load the TMS32010's status bits after interrupts and subroutine calls. These status bits include the Overflow Flag (OV) bit, Overflow Mode (OVM) bit, Auxiliary Register Pointer (ARP) bit, and the Data Memory Page Pointer (DP) bit. The Interrupt Mask bit cannot be changed by the LST instruction. 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
OV	OVM	INTM	1	1	1	1	ARP	1	1	1	1	1	1	1	DP

See SST.

Assembler Syntax:

Direct Addressing:	[<label>]	LT	<dma>
Indirect Addressing:	[<label>]	LT	{*|*+|*-}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (dma) → T

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	0	1	0	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	1	0	1	0	1	SEE SECTION 3.3						

Description: LT loads the T register with the contents of the specified data memory location.

Words: 1
Cycles: 1

Example:

```
LT      DAT24
or
LT      *
```

If current auxiliary register contains the value 24.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 24	62	62
T	3	62

Note: LT is used to load the T register in preparation for a multiplication. See MPY, LTA, and LTD.

Assembler Syntax:

Direct Addressing: [<label>] LTA <dma>
Indirect Addressing: [<label>] LTA {*|*+|*-}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (dma) → T
(ACC) + (P) → ACC

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	1	0	0	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	1	1	0	0	1	SEE SECTION 3.3						

Description: The contents of the specified data memory address are loaded into the T register. Then, the P register, containing the previous product of the multiply operation, is added to the accumulator, and the result is stored in the accumulator.

Words: 1
Cycles: 1

Example:
```
LTA  DAT24
or
LTA  *
```
If current auxiliary register contains the value 24.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 24	62	62
T	3	62
P	15	15
ACC	5	20

Note: This instruction is a subset of the LTD instruction.

Assembler Syntax:

Direct Addressing:	[<label>]	LTD	<dma>
Indirect Addressing:	[<label>]	LTD	{*|*+|*-}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (dma) → T
(ACC) + (P) → ACC
(dma) → dma + 1

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	0	1	1	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	1	0	1	1	1	SEE SECTION 3.3						

Description: The T register is loaded with the contents of the specified data memory address. Then, the contents of the P register are added to the accumulator. Next, the contents of the specified data memory address are transferred to the next higher data memory address.

Words: 1
Cycles: 1

Example: LTD DAT24
or
LTD *

IF current auxiliary register contains the value 24.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 24	62	62
DATA MEMORY 25	0	62
T	3	62
P	15	15
ACC	5	20

Assembler Syntax: [<label>] MAR {*|*+|*−}[,<ARP>]

Operands: ARP=0 or 1

Operation: Current auxiliary register is incremented, decremented, or remains the same. Auxiliary register pointer is loaded with the next ARP.

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	0	0	0	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	1	0	0	0	1	SEE SECTION 3.3						

Description: This instruction utilizes the indirect addressing mode to increment/decrement the auxiliary registers and to change the auxiliary register pointer. It has no other effect.

Words: 1
Cycles: 1

Example:

```
MAR *,1      Load ARP with 1.
MAR *-       Decrement current auxiliary register (in this case, AR1)
MAR *+,0     Increment current auxiliary register (AR1), load ARP with 0.
```

Note: In the direct addressing mode, MAR is a NOP. Also,the instruction LARP is a subset of MAR (i.e., MAR *,0 performs the same function as LARP 0).

Assembler Syntax:

Direct Addressing:	[<label>]	MPY	<dma>
Indirect Addressing:	[<label>]	MPY	{*|*+|*−}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (T) x (dma) → P

Encoding:

	15	14	13	12	11	10	9	8	7	6 5 4 3 2 1 0
Direct:	0	1	1	0	1	1	0	1	0	DATA MEMORY ADDRESS
Indirect:	0	1	1	0	1	1	0	1	1	SEE SECTION 3.3

Description: The contents of the T register are multiplied by the contents of the specified data memory address, and the result is stored in the P register.

Words: 1
Cycles: 1

Example: MPY DAT13
or
MPY * If current auxiliary register contains the value 13.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 13	7	7
T	6	6
P	36	42

Note: During an interrupt, all registers except the P register can be saved. However, the TMS32010 has hardware protection against servicing an interrupt between an MPY or MPYK instruction and the following instruction. For this reason, it is advisable to follow MPY and MPYK with LTA, LTD, PAC, APAC, or SPAC.

No provisions are made for the condition of >8000 X >8000. If this condition arises, the product will be >C0000000.

Assembler Syntax: [<label>] MPYK <constant>

Operands: $(-2^{12}) \leq$ constant $< 2^{12}$

Operation: (T) x constant→P

Encoding:

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

Description: The contents of the T register are multiplied by the signed 13-bit constant and the result loaded into the P register.

Words: 1
Cycles: 1

Example: MPYK −9

	BEFORE INSTRUCTION	AFTER INSTRUCTION
T	7	7
P	42	−63

Note: No provision is made to save the contents of the P register during an interrupt. Therefore, this instruction should be followed by one of the following instructions: PAC, APAC, SPAC, LTA, or LTD. Provision is made in hardware to inhibit interrupt during MPYK until the next instruction is executed.

NOP — No Operation — NOP

Assembler Syntax: [<label>] NOP

Operands: None

Operation: None

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	1	0	0	0	0	0	0	0

Description: No operation is performed.

Words: 1
Cycles: 1

Example: NOP

Note: NOP is useful as a "pad" or temporary instruction during program development.

Assembler Syntax:

Direct Addressing: [<label>] OR <dma>
Indirect Addressing: [<label>] OR {*|*+|*-}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: Zero. OR. high-order ACC bits: (dma). OR. low-order ACC bits→ACC

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 3.3						

Description The low-order bits of the accumulator are ORed with the contents of the specified data memory address concatenated with all zeroes ORed with the high-order bits of the accumulator. The result is stored in the accumulator. The OR operation follows the truth table below.

DATA MEMORY BIT	ACC BIT (BEFORE)	ACC BIT (AFTER)
0	0	0
0	1	1
1	0	1
1	1	1

Words: 1
Cycles: 1

Example: OR DAT88
or
OR * Where current auxiliary register contains the value 88.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 88	>F 0 0 0	>F 0 0 0
ACC	>0 0 1 0 0 0 0 2	>0 0 1 0 F 0 0 2

Note: This instruction is useful for comparing selected bits of a data word.

Assembler Syntax:

Direct Addressing:	[<label>]	OUT	<dma>,<PA>
Indirect Addressing:	[<label>]	OUT	{*|*+|*-},<PA>[,<ARP>]

Operands: $0 \leq dma \leq 127$
$0 \leq PA \leq 7$
ARP = 0 or 1

Operation: PA→ address lines PA2-PA0
(dma)→data bus D15-D0

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	0	1	PORT ADDRESS			0	DATA MEMORY ADDRESS						
Indirect:	0	1	0	0	1	PORT ADDRESS			1	SEE SECTION 3.3						

Description: The OUT instruction transfers data from data memory to an external peripheral. The first cycle of this instruction places the port address onto address lines A2/PA2-A0/PA0. During the same cycle, $\overline{WE}$ goes low and the data word is placed on the data bus D15-D0.

Words: 1
Cycles: 2

Example:

OUT	120,7	Output data word stored in memory location 120 to peripheral on port address 7.
OUT	*,5	Output data word referenced by current auxiliary register to peripheral on port address 5.

Notes: When the TMS32010 sends the port address onto the three LSBs of the address lines, the nine MSBs are set to zero.

The OUT instruction causes the $\overline{WE}$ line to go low during the first clock cycle of this instruction's execution. $\overline{MEN}$ remains high during the first cycle.

Assembler Syntax: [<label>] PAC

Operands: None

Operation: (P) → ACC

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	1	0	0	0	1	1	1	0

Description: The contents of the P register resulting from a multiply are loaded into the accumulator.

Words: 1
Cycles: 1

Example: PAC

	BEFORE INSTRUCTION	AFTER INSTRUCTION
P	144	144
ACC	23	144

Assembler Syntax: [<label>] POP

Operands: None

Operation: (TOS) → ACC

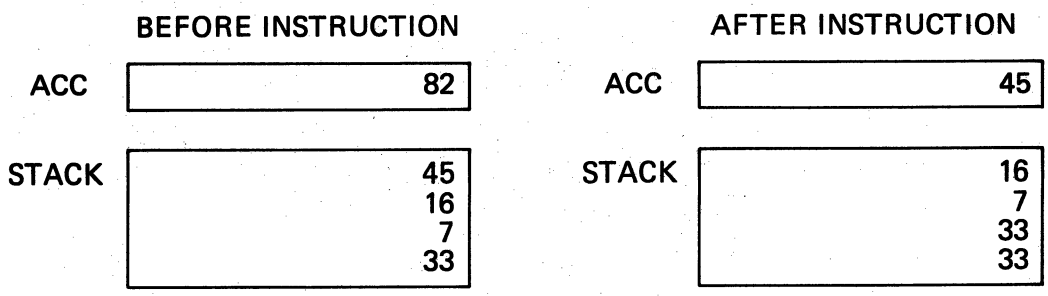
Encoding:

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

Description: The contents of the top of stack are loaded into the accumulator. The next element on the stack becomes the top of the stack.

Words: 1
Cycles: 2

Example: POP



Note: The 12 bits of the stack are put into the accumulator in bits 11 through 0, and bits 31 through 12 are zeroed. There is no provision to check stack underflow.

Push Accumulator onto Stack

PUSH

Assembler Syntax: [<label>] PUSH

Operands: None

Operation: (ACC) →TOS

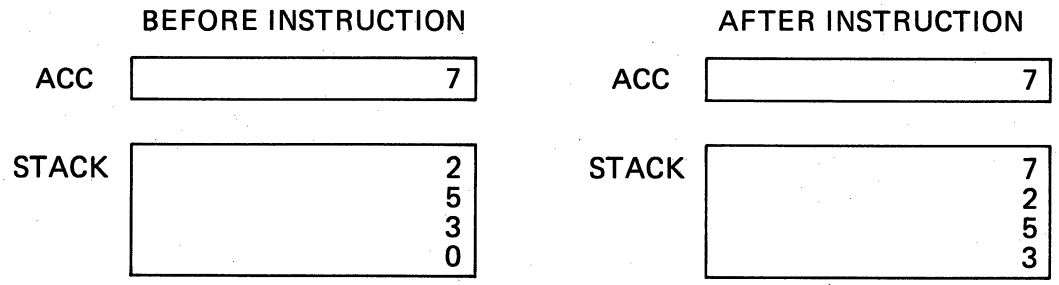
Encoding:

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

Description: The contents of the lower 12 bits (11-0) of the accumulator are pushed onto the top of the hardware stack.

Words: 1
Cycles: 2

Example: PUSH



Note: There is no provision for detecting a stack overflow. Therefore, if the stack is already full, the contents of the bottom stack element will be lost upon execution of PUSH.

RET — Return from Subroutine — RET

Assembler Syntax: [<label>] RET

Operands: None

Operation: (TOS) → PC

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	1	0	0	0	1	1	0	1

7F8D

Description: The top element is popped off of the stack and loaded into the program counter.

3

Words: 1
Cycles: 2

Example: RET

	BEFORE INSTRUCTION	AFTER INSTRUCTION
PC	96	37
STACK	37 45 75 75	45 75 75 75

Note: This instruction is used in conjunction with CALL and CALA for subroutines.

Assembler Syntax: [<label>] ROVM

Operand: None

Operation: 0→OVM

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	1	0	0	0	1	0	1	0

Description: This instruction will reset the TMS32010 from the overflow mode it was placed in by the SOVM instruction. The overflow mode will set the accumulator and the ALU to their highest positive/negative value when an overflow occurs.

Words: 1
Cycles: 1

Example: ROVM

Note: See SOVM.

Assembler Syntax:

Direct Addressing:	[<label>]	SACH	<dma>[,<shift>]
Indirect Addressing:	[<label>]	SACH	{*|*+|*-}[,<shift>[,<ARP>]]

Operands: $0 \le dma \le 127$
shift = 0, 1, or 4
ARP = 0 or 1

Operation: (ACC) x 2 − (16-shift) → dma

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	1	1	SHIFT			0	DATA MEMORY ADDRESS						
Indirect:	0	1	0	1	1	SHIFT			1	SEE SECTION 3.3						

Description: Store the upper half of the accumulator in data memory with shift. The shift can only be 0, 1, or 4.

Words: 1
Cycles: 1

Example: SACH DAT70,1
or
SACH *,1 If current auxiliary register contains the value 70.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
ACC	>0 4 2 0 8 0 0 1	>0 4 2 0 8 0 0 1
DATA MEMORY 70	6	>0 8 4 1

Notes: The SACH instruction copies the entire accumulator into a shifter. It then shifts this entire 32-bit number 0, 1, or 4 bits and copies the upper 16 bits of the shifted product into data memory. The accumulator itself remains unaffected.

For example, the following instruction sequence will store >8F35 in data memory location DAT1. Location DAT2 contains the number >A8F3. DAT3 contains >5000.

ZALH	DAT2	ACC = >A8F30000
ADDS	DAT3	ACC = >A8F35000
SACH	DAT1,4	DAT1 = >8F35
		ACC = >A8F35000

Assembler Syntax:

Direct Addressing:	[<label>]	SACL	<dma>[,<shift>]
Indirect Addressing:	[<label>]	SACL	{*|*+|*-}[,<shift>[,<ARP>]]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1
Shift = 0

Operation: (ACC bits 15 through 0) → dma

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 3.3						

Description: Store the low-order bits of the accumulator in data memory.

Words: 1
Cycles: 1

Example:
```
SACL    DAT71
or
SACL    *          If current auxiliary register contains the value 71.
```

	BEFORE INSTRUCTION	AFTER INSTRUCTION
ACC	>0 4 2 0 8 0 0 1	>0 4 2 0 8 0 0 1
DATA MEMORY 71	7	>8 0 0 1

Note: There is no shift associated with this instruction, although a shift code of zero MUST be specified if the ARP is to be changed.

Assembler Syntax:

Direct Addressing:	[<label>]	SAR	<AR>,<dma>
Indirect Addressing:	[<label>]	SAR	<AR>,{*|*+|*-}[,<ARP>]

Operands: $0 \leq dma \leq 127$
AR = 0 or 1
ARP = 0 or 1

Operation: (AR) → dma

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 3.3						

Description: The contents of the designated auxiliary register are stored in the specified data memory location.

Words: 1
Cycles: 1

Example: SAR AR0,DAT101

	BEFORE INSTRUCTION	AFTER INSTRUCTION
AR0	37	37
DATA MEMORY 101	18	37

also, LARP AR0
SAR AR0,*+

AR0	5	6
DATA MEMORY 5	0	6

WARNING

Special problems arise when SAR is used to store the current auxiliary register with indirect addressing if autoincrement/decrement is used.

(continued)

```
LARP    AR0
LARK    AR0,10
SAR     AR0,*+   or   SAR  ARO,*–
```

In this case, SAR AR0, *+ will cause the value 11 to be stored in location 10. SAR AR0, *– will cause the value 9 to be stored in location 10.

Note: For more information, see LAR.

Assembler Syntax: [<label>] SOVM

Operands: None

Operation: 1→OVM

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	1	0	0	0	1	0	1	1

Description: When placed in the overflow mode, the TMS32010 will set the accumulator and ALU to their highest positive/negative value if an overflow/underflow occurs. The highest positive value is >7FFFFFFF, and the lowest negative value is >80000000.

Words: 1
Cycles: 1

Example: SOVM

Assembler Syntax: [<label>] SPAC

Operands: None

Operation: (ACC) – (P) → ACC

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	1	0	0	1	0	0	0	0

Description: The contents of the P register are subtracted from the contents of the accumulator, and the result is stored in the accumulator.

Words: 1
Cycles: 1

Example: SPAC

	BEFORE INSTRUCTION	AFTER INSTRUCTION
P	36	36
ACC	60	24

Assembler Syntax:

Direct Addressing:	[<label>]	SST	<dma>
Indirect Addressing:	[<label>]	SST	{*|*+|*−}[,<ARP>]

Operands: $0 \leq dma \leq 15$
ARP = 0 or 1

Operation: status bits → specified data memory word on page 1

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 3.3						

Description: The status bits are saved into the specified data memory address on page 1.

Words: 1
Cycles: 1

Example:

```
SST     DAT1
or
SST     *,1          IF current auxiliary register contains the value 1.
```

Note: This instruction is used to load the TMS32010's status bits after interrupts and subroutine calls. These status bits include the Overflow Flag (OV) bit, Overflow Mode (OVM) bit, Interrupt Mask (INTM) bit, Auxiliary Register Pointer (ARP) bit, and the Data Memory Page Pointer (DP) bit. These bits are 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
OV	OVM	INTM	1	1	1	1	ARP	1	1	1	1	1	1	1	DP

Note: See LST.

Assembler Syntax:

Direct Addressing:	[<label>]	SUB	<dma>[,<shift>]
Indirect Addressing:	[<label>]	SUB	{*|*+|*-}[,<shift>[,<ARP>]]

Operands: $0 \leq shift \leq 15$
$0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (ACC) − [(dma) × 2^{shift}] → ACC

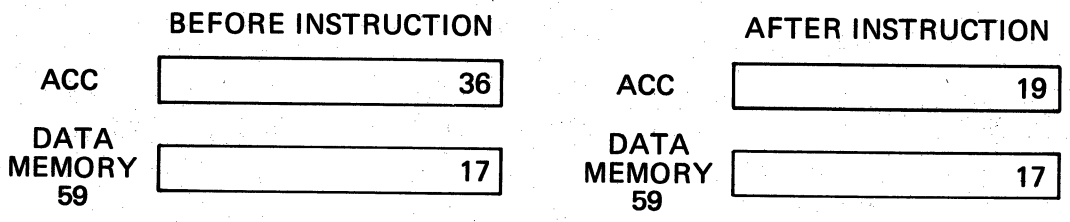
Encoding:

	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 3.3						

Description: Contents of data memory address are left-shifted and subtracted from the accumulator. During shifting, the low-order bits of data are zero-filled and the high-order bit is sign-extended. The result is stored in the accumulator.

Words: 1
Cycles: 1

Example:
```
SUB     DAT59
or
SUB     *
```
If current auxiliary register contains the value 59.



Assembler Syntax:

Direct Addressing:	[<label>]	SUBC	<dma>
Indirect Addressing:	[<label>]	SUBC	{*|*+|*-}[,<ARP>]

Operands: $0 \leq dma \leq 127$,
ARP = 0 or 1

Operation: (ACC) − [(dma) x 2^{15}] → adder output

If (high-order bits of adder output) ≥ 0
Then (adder output) * 2 + 1 → ACC
Else (ACC) × 2 → ACC

Encoding:

Direct:

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

Indirect:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	SEE SECTION 3.3						

Description: This instruction performs conditional subtraction which can be used for division in algorithms.

Words: 1
Cycles: 1

Note: The next instruction after SUBC cannot use the accumulator.

Assembler Syntax:

Direct Addressing:	[<label>]	SUBH	<dma>
Indirect Addressing:	[<label>]	SUBH	{*|*+|*-}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (ACC) − [(dma) × 2^{16}] → ACC

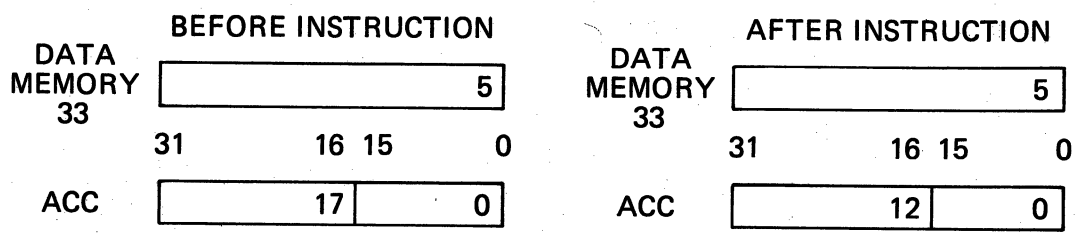
Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	0	1	0	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	0	0	1	0	1	SEE SECTION 3.3						

Description: Subtract the contents of specified data memory location from the upper half of the accumulator. The result is stored in the accumulator.

Words: 1
Cycles: 1

Example:
```
SUBH    DAT33
or
SUBH    *          If current auxiliary register contains the value 33.
```



Note: The SUBH instruction can be used for performing 32-bit arithmetic.

Subtract from Low Accumulator with Sign-Extension Suppressed

SUBS

Assembler Syntax:

Direct Addressing:	[<label>]	SUBS	<dma>
Indirect Addressing:	[<label>]	SUBS	{*|*+|*-}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (ACC) − (dma) → ACC

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	0	1	1	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	0	0	1	1	1	SEE SECTION 3.3						

Description: Subtract contents of a specified data memory location from accumulator with sign-extension suppressed. The data is treated as a 16-bit positive integer rather than a two's complement integer.

Words: 1
Cycles: 1

Example:

```
SUBS    DAT61
or
SUBS    *       If current auxiliary register contains the value 61.
```

	BEFORE INSTRUCTION	AFTER INSTRUCTION
ACC	>0 0 0 0 F 1 0 5	>0 0 0 0 0 1 0 2
DATA MEMORY 61	>F 0 0 3	>F 0 0 3

Assembler Syntax:

Direct Addressing:	[<label>]	TBLR	<dma>
Indirect Addressing:	[<label>]	TBLR	{*|*+|*−}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (PC) + 1 → TOS
(ACC) → PC → address lines A11 through A0
data bus D15 through D0 → dma
(TOS) → PC

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	1	1	1	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	0	1	1	1	1	SEE SECTION 3.3						

Description: This instruction transfers a word from anywhere in program memory (i.e., internal ROM, external ROM, external RAM) to the specified location in data memory. The three-cycle instruction is as follows:

Prefetch: $\overline{MEN}$ goes low and the TBLR instruction opcode is fetched. The previous instruction is executing.

Cycle 1: $\overline{MEN}$ goes low. The address of the next instruction is placed onto address bus, but data bus is not read. Program counter is pushed onto stack. Twelve LSBs of the accumulator contents are loaded into the program counter.

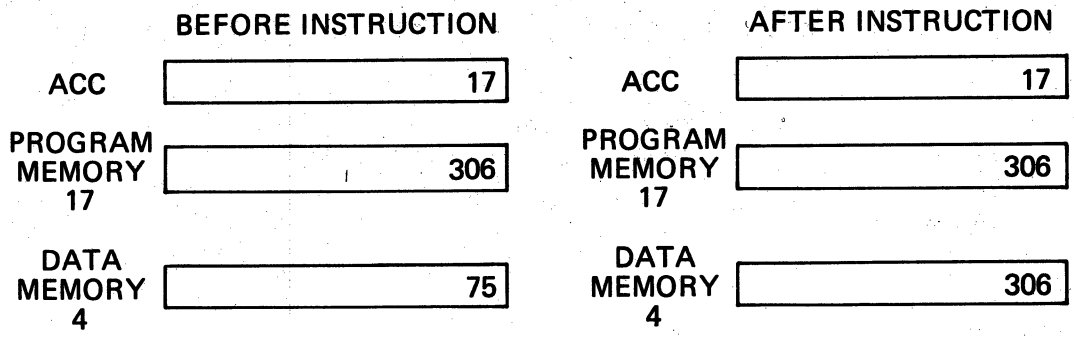
Cycle 2: $\overline{MEN}$ goes low. Contents of program counter are buffered to address lines. Address memory location is read and is copied into specified RAM location. The new program counter is popped from the stack.

Cycle 3: $\overline{MEN}$ goes low. Next instruction opcode is prefetched.

Words: 1
Cycles: 3

Example:
```
TBLR  DAT4
TBLR  *     If current auxiliary register contains the value 4.
```

(Continued)



Note: This instruction is useful for reading coefficients that have been stored in program ROM, or time-dependent data stored in RAM.

Assembler Syntax:

Direct Addressing:	[<label>]	TBLW	<dma>
Indirect Addressing:	[<label>]	TBLW	{*|*+|*-}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (PC) + 1 → TOS
(ACC) → PC → address lines A11 through A0
(dma) → data bus D15 through D0
(TOS) → PC

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 3.3						

Description: This instruction transfers a word from the specified location in data memory to a location in external program RAM. The three-cycle instruction is as follows:

Prefetch: $\overline{MEN}$ goes low and the TBLR instruction opcode is fetched. The previous instruction is executing.

Cycle 1: $\overline{MEN}$ goes low. The address of the next instruction is placed onto address bus, but data bus is not read. Program counter is pushed onto stack. Twelve LSBs of the accumulator contents are loaded into the program counter.

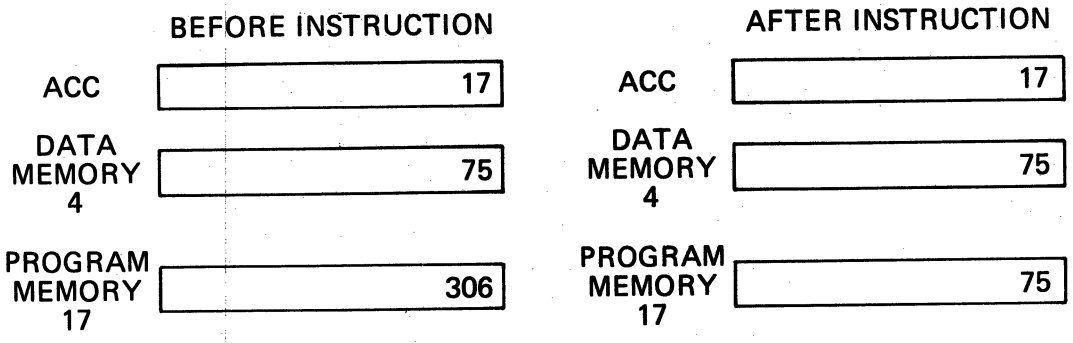
Cycle 2: $\overline{WE}$ goes low. Contents of program counter are buffered to address lines. Contents of specified data memory address are placed on the data bus. The new program counter is popped off of stack.

Cycle 3: $\overline{MEN}$ goes low. Next instruction opcode is prefetched.

Words: 1
Cycles: 3

Example: TBLW DAT4
TBLW * If current auxiliary register contains the value 4.

(Continued)



Note: The TBLW and OUT instructions use the same external signals and thus cannot be distinguished when writing to program memory addresses 0 through 7.

Assembler Syntax:

Direct Addressing: [<label>] XOR <dma>
Indirect Addressing: [<label>] XOR {*|*+|*-}[,<ARP>]

Operands: 0≤dma≤127
ARP=0 or 1

Operation: Zero. XOR. high-order ACC bits: (dma). XOR. low-order ACC bits→ACC

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 3.3						

Description: The low-order bits of the accumulator are exclusive-ORed with the specified data memory address and concatenated with the exclusive-OR of all zeroes and the high-order bits of the accumulator. The exclusive-OR operation follows the truth table below:

DATA MEMORY BIT	ACC BIT (BEFORE)	ACC BIT (AFTER)
0	0	0
0	1	1
1	0	1
1	1	0

Words: 1
Cycles: 1

Example: XOR DAT45
or
XOR * If current auxiliary register contains the value 45.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 45	>F F 0 0	>F F 0 0
ACC	>0 F F F 0 F F F	>0 F F F F 0 F F

Note: This instruction is useful for toggling or setting bits of a word for high-speed control. Also, the one's complement of a word can be found by exclusive-ORing it with all ones.

Assembler Syntax: [<label>] ZAC

Operands: None

Operation: 0 → ACC

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	1	0	0	0	1	0	0	1

Description: The accumulator is cleared (zeroed).

Words: 1
Cycles: 1

Example: ZAC

	BEFORE INSTRUCTION								AFTER INSTRUCTION							
ACC	A	F	F	F	F	F	F	F	ACC	0	0	0	0	0	0	0	0

Assembler Syntax:

Direct Addressing:	[<label>]	ZALH	<dma>
Indirect Addressing:	[<label>]	ZALH	{*|*+|*-}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (dma) $\times 2^{16} \rightarrow$ ACC

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	1	0	1	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	0	1	0	1	1	SEE SECTION 3.3						

Description: ZALH clears the accumulator and loads the contents of the specified data memory location into the upper half of the accumulator. The lower half of the accumulator remains clear.

Words: 1
Cycles: 1

Example: ZALH DAT29
or
ZALH * If current auxiliary register contains the value 29.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 29	>3 F 0 0	>3 F 0 0
ACC	>0 0 7 7 F F F F	>3 F 0 0 0 0 0 0

Note: ZALH can be used for implementing 32-bit arithmetic.

Assembler Syntax:

Direct Addressing:	[<label>]	ZALS	<dma>
Indirect Addressing:	[<label>]	ZALS	{*|*+|*-}[,<ARP>]

Operands: $0 \leq dma \leq 127$
ARP = 0 or 1

Operation: (dma) → ACC

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	1	1	0	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	0	1	1	0	1	SEE SECTION 3.3						

Description: Clear accumulator and load contents of specified data memory location into lower half of the accumulator. The data is treated as a 16-bit positive integer rather than a two's complement integer. Therefore, there is no sign-extension as with the LAC instruction.

Words: 1
Cycles: 1

Example:
```
ZALS   DAT22
  or
ZALS   *      If current auxiliary register contains the value 22.
```

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 22	>F 7 F F	>F 7 F F
ACC	>7 F F 0 0 0 3 3	>0 0 0 0 F 7 F F

Notes: The following routine reveals the difference between the ZALS and the LAC instruction. Data memory location 1 contains the number > FA37.

```
ZALS    DAT1    (ACC) = > 0000FA37
ZAC             Zero ACC
LAC     DAT1    (ACC) = > FFFFFA37
```

ZALS is useful for 32-bit arithmetic operations.

METHODOLOGY FOR APPLICATION DEVELOPMENT

4. METHODOLOGY FOR APPLICATION DEVELOPMENT

4.1 OUTLINE OF DEVELOPMENT PROCESS

A number of development tools are required for designing a system with a microprocessor. This section describes the facilities which are available for the TMS32010 and illustrates how to use them for developing an application. A typical application development flowchart is shown in Figure 4-1.

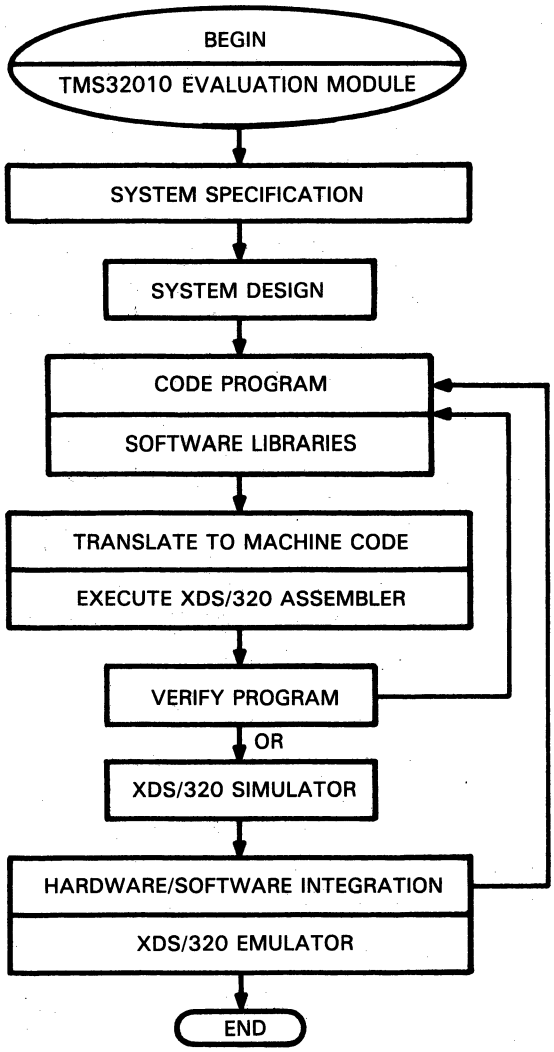


FIGURE 4-1 - FLOWCHART OF TYPICAL APPLICATION DEVELOPMENT

After defining the specifications of the system, the designer should draw a flowchart of the software and a block diagram of the hardware. The processor's performance is then evaluated to determine the feasibility of implementing the algorithm via the TMS32010 Evaluation Module. The full algorithm is coded using assembly language. The program is assembled and then verified using the XDS/320 Macro Assembler and Linker and, optionally, the XDS/320 Simulator. Several iterations of the program are usually required to correctly code the algorithm. The verified program is integrated into the hardware, and the prototype system is debugged by using the XDS/320 Emulator.

4.2 DESCRIPTION OF DEVELOPMENT FACILITIES

Five development facilities aid in the design and implementation of TMS32010 applications. Each of the following five development facilities provides a tool for one of the steps involved in developing an application:

- The TMS32010 Evaluation Module is used to appraise the performance of the processor. A software library capability is used to simplify and standardize code development.
- The XDS/320 Assembler and Linker translates an assembly language program into a loadable object module.
- The XDS/320 Simulator accepts downloaded object code and executes the program via a simulated TMS32010 in a debug mode, thus allowing software debug before attempting hardware debug.
- The XDS/320 Emulator integrates the processor into the hardware design by providing a means to debug both software and hardware together.

4.2.1 TMS32010 Evaluation Module

The TMS32010 Evaluation Module (EVM) is a single board which enables a user to determine inexpensively if the TMS32010 meets the speed and timing requirements of his application. The EVM is a stand-alone module which contains all the tools necessary to evaluate the TMS32010.

Communication to a host computer and to several peripherals is provided on the EVM. Dual EIA ports allow the EVM to be connected to a terminal and a host computer. The EVM can also be configured with a line printer on one port; the other port is connected to either a terminal or a host computer. As either the host computer or the terminal feeds the assembly language program to the EVM, the EVM assembles the code. A built-in cassette tape interface can also be used to save code on tape to be reloaded at a later time. An EPROM programmer is also provided for saving code. Alternatively, code can be executed directly by the EVM through its target connector.

The EVM can accept either source or object code from a host computer or terminal. A line-oriented text editor, an assembler which permits symbolic addressing of memory locations, and a reverse assembler that changes machine code back into assembly language instructions are provided for programming ease. The debug mode gives access to all of the TMS32010's registers and memory. Eight breakpoints on program addresses and the ability to single-step program execution have been incorporated for monitoring device operation.

4.2.2 XDS/320 Macro Assembler/Linker

The XDS/320 Macro Assembler translates TMS32010 assembly language 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. This allows software to be designed more efficiently and reliably.

The XDS/320 Macro Assembler supports macro calls and definitions along with conditional assembly. It provides the user with a comprehensive set of error diagnostics. The XDS/320 Macro Assembler produces a listing and an object file, and will optionally print a symbol table/cross-reference listing.

Assembler directives which affect program assembly are provided for the user. Some directives affect the location counter and make sections of the program relocatable. Constants for data and text are defined by using directives. Symbols defined in one assembly can be used in another assembly with the REF and DEF directives. These external symbols allow separate modules to be linked together.

The XDS/320 Linker permits a program to be designed and implemented in separate modules which will later be linked together to form the complete program. This allows the same modules (i.e., a filter module) to be used in many programs. The linker assigns values to relocatable code, creating an object file which can be executed by the simulator or emulator.

The linker resolves external definitions and references from different assemblies, and thereby links several modules together. More than one assembly may be linked together to create a module which may be linked again to the main program. An intermediate partial linkage does not require that all external references be resolved, but in the final linking process, there should be no unresolved references. Another function of the linker is to assign absolute values to relocatable code. The final output of the linker can then be loaded into either the simulator or the emulator.

A source code macro library can be maintained in a directory to be assembled with the main program. This allows commonly used routines to be accessed by more than one program and to be used to decrease program development time. The mnemonics are macro calls which expand into assembly code.

The macro library typically should contain user-defined macros and the macros defined in Section 7. These macros simplify the generation of an assembly language program. Examples include comparing a word in memory to a word in the accumulator, shifting right, and moving numbers between registers.

The XDS/320 Macro Assembler and Linker are currently available on several host computers, including the TI990(DX10) VAX(VMS) and IBM MVS and CMS operating systems. Currently in development is software to support the VAX(UNIX), DEC PDP11(RSX), IBM PC(DOS) and TI professional computer (DOS) operating system. Contact your local TI representative for availability or further details.

4.2.3 XDS/320 Simulator

The XDS/320 Simulator is a software program that simulates operation of the TMS32010 to allow program verification. The debug mode enables the user to monitor the state of the simulated TMS32010 while the program is executing.

The simulator program uses the TMS32010 object code, produced by the XDS/320 Macro Assembler/ Linker. Input and output files may be associated with the port addresses of the I/O instructions in order to simulate I/O devices which will be connected to the processor. The interrupt flag can be set periodically at a user-defined interval for simulating an interrupt signal. Before initiating program execution, breakpoints may be defined, and the trace mode set up.

During program execution, the internal registers and memory of the simulated TMS32010 are modified as each instruction is interpreted by the host computer. Execution is suspended when either 1) a breakpoint or error is encountered, 2) the step count goes to zero, or 3) a branch to 'self' is detected. Once program execution is suspended, the internal registers and both program and data memories can be inspected and/or modified. The trace memory can also be displayed. A record of the simulation session can be maintained in a journal file, so that it may be replayed to regain the same machine state during another simulation session.

The XDS/320 Simulator is currently available for the VAX(VMS).

4.2.4 XDS/320 Emulator

The XDS/320 Emulator is a self-contained system that has all the features necessary for real-time in-circuit emulation. This allows integration of the user hardware and software in the debug mode. Three EIA ports have been provided on the emulator to interface with a host system. The first EIA port provides a connection for a computer, the second port for a terminal, and the third port for a printer or a PROM programmer. Using a standard EIA port, the object file produced by the macro assembler/linker can be downloaded into the emulator, which can then be controlled through a terminal. In addition, source code can be downloaded to the emulator. A line-by-line assembler with forward and reverse referencing labels is provided on the XDS to assemble the source.

A pin-compatible target connector plugs into the TMS32010 socket to enable real-time emulation. Three clock options are available. First, a 20-MHz clock is available on the emulator. In addition, an external clock source can be used by attaching a crystal to the target connector, or by connecting a signal generator to the emulator.

The emulator operates in one of three memory modes: 1) software development mode, 2) microcomputer mode, or 3) microprocessor mode. In the software development mode, the entire 8K bytes of program memory reside within the emulator. In the microcomputer mode, 3K bytes reside within the emulator while 5K bytes reside on the target system. The microprocessor mode is used when all 8K bytes of program memory exist on the target system.

By setting breakpoints based on internal conditions or external events, execution of the user's program can be suspended and control given to the XDS monitor. While in the monitor, all registers and memory locations can be inspected and modified. Single-step execution is also available. A single read or write to an I/O port can be performed to test peripheral devices in the prototype system. Full trace capabilities at full speed and a reverse assembler that translates machine code back into assembly instructions are also included to increase debugging productivity.

4.3 APPLICATION DEVELOPMENT PROCESS EXAMPLE

The design and implementation of a TMS32010-based discrete-time filter is presented below to illustrate the development process. The filter design is derived from the system specification, using digital signal processing theory. A macro library is used to help code the program. The assembler and simulator verify that the program executes the filter properly. The processor is then integrated into the prototype system by using the emulator.

4.3.1 System Specification

Table 4-1 defines the specifications of the discrete-time filter.

TABLE 4-1 – FILTER SPECIFICATIONS

PARAMETER	VALUE	UNIT
Sample frequency (f_S)	10	kHz
Corner frequency (f_{CO})	2	kHz
Attenuation at $f = f_{CO}$	−2	dB
Attenuation at $f = 1.2\ f_{CO}$	−15	dB
Passband ripple	±1.5	dB

4.3.2 System Design

The equation for the above discrete-time filter was derived as follows:

$$y(n) = -.2302699\,x(n) + .1559177\,x(n-1) + .2211667\,x(n-2) + .1119031\,x(n-3) - .1124507\,x(n-4) - .1485743\,x(n-5) + .2046856\,x(n-6) + .7409326\,x(n-7) + 1.0\,x(n-8) + .7409326\,x(n-9) + .2046856\,x(n-10) - .1485743\,x(n-11) - .1124507\,x(n-12) + .1119031\,x(n-13) + .2211667\,x(n-14) + .1559177\,x(n-15) - .2302699\,x(n-16).$$

where x(n) is the current sample,
x(n – 1) is the sample from the previous period,
↓
x(n – 16) is the sample from the previous 16th period.

4.3.3 Code Development

The TMS32010 software development cycle is generally a three-step process for the purpose of translating the filter equation into TMS32010 assembly language. First, a flowchart of the program is drawn. Then, the example is coded in a high-level language, FORTRAN, to provide structure and to test if the algorithm is correct before implementing it in assembly language. Finally, the program is coded and tested in assembly language using some of the macro library routines.

4.3.3.1 *Discrete-Time Filter Flowchart*

Figure 4-2 is a flowchart for the software implementation of the discrete-time filter.

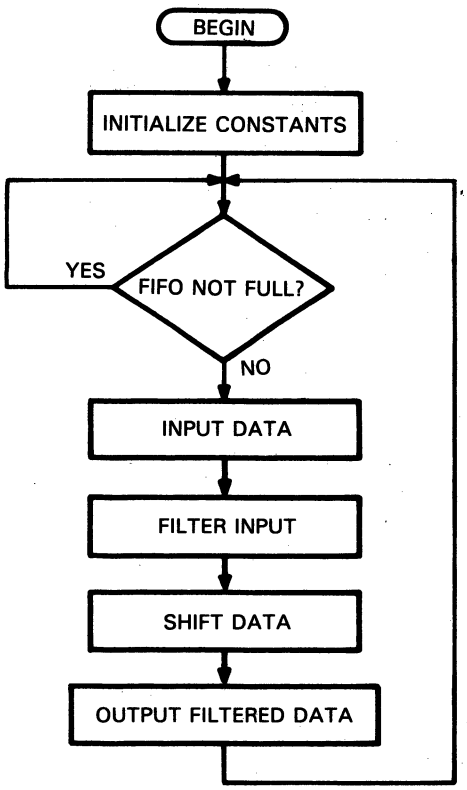


FIGURE 4-2 - FLOWCHART OF FILTER IMPLEMENTATION

4.3.3.2 *FORTRAN Program*

The following FORTRAN program implements the specified digital filter and provides 1000 outputs.

```
      PROGRAM FILTER
C
C y(n)=-.2302699 x(n) + .1559177 x(n-1) + .2211667 x(n-2) +.1119031 x(n-3)
C  - .1124507 x(n-4) - .1485743 x(n-5) + .2046856 x(n-6) + .7409326 x(n-7)
C  + 1.0 x(n-8) + .7409326 x(n-9) + .2046856 x(n-10) - .1485743 x(n-11)
C  - .1124507 x(n-12) + .1119031 x(n-13) + .2211667 x(n-14)
C  + .1559177 x(n-15) - .2302699 x(n-16).
C
      REAL*4 X(17),CX(17),Y
C
C     Initialize the constants for the filter equation
C
      DATA   CX /-.2302699,.1559177,.2211667,.1119031,-.1124507,
     1           -.1485743,.2046856,.7409326,1.0,.7409326,
     1           .2046856,-.1485743,-.1124507,.1119031,.2211667,
     1           .1559177,-.2302699/
C
      I = 0
  100       I = I + 1
C
C     Input sampled data
C
      READ (55,110) IX
  110       FORMAT (I6)
      X(1) = IX
C
C     Filter data
C
      Y = 0
      DO J = 1,17
         Y = Y + CX(J)*X(J)
      END DO
C
C     Shift data to new variables
C
      DO J = 16,1,-1
         X(J) = X(J-1)
      END DO
C
C     Output filtered data
C
      TYPE *,Y
C
      IF (I .LE. 1000) GO TO 100
  200       END
```

4.3.3.3 *Assembly Language Program Using Relocatable Code*

The same discrete-time filter can be implemented in TMS32010 assembly language using relocatable code. The FORTRAN program should not be directly translated into assembly language. Assembly language code can be made more efficient than the FORTRAN implementation by taking advantage of the processor's architecture. The assembly language implementation of the FORTRAN program is described in the following paragraphs.

Two library macros (PROG and MAIN) have been used in the example program to simplify the coding process and to standardize the program structure. One advantage of using macros for standardizing program structure is that different programmers can easily trade relocatable modules if they have used the same structure. The PROG macro begins the module with an IDT directive. This directive gives the module a name to be used later during link and also initializes some values in the assembler's symbol table. The macro MAIN labels the beginning of the main routine, initializes the constants ONE and MINUS, and defines the variables XR0 and XR1.

The coefficients in the equation are converted to integer arithmetic for this program. To maintain a maximum amount of accuracy, the coefficients should be factored by 2** — 15, which will create a Q15 number. After factoring the filter equation, it becomes:

$$y(n) = [\ -7545x(n) + 5109\ x(n-1) + 7247\ x(n-2) + 3667\ x(n-3) - 3685\ x(n-4) - 4868x(n-5) + 6707\ x(n-6) + 24279\ x(n-7) + 32767\ x(n-8) + 24279\ x(n-9) + 6706\ x(n-10) - 4868\ x(n-11) - 3685\ x(n-12) + 3667\ x(n-13) + 7247\ x(n-14) + 5109\ x(n-15) - 7545\ x(n-16)]*2^{**} - 15.$$

Contants are listed in program memory in a table so as to define the coefficients in data memory. Constants are then read into data memory using the TBLR instruction. The user loads a one in the T register to access the table. The MPYK instruction puts the address of the table into the P register. Then, the PAC instruction loads it into the accumulator. A loop is set up to move all of the constants into data memory.

The $\overline{BIO}$ pin is connected to the FIFO empty line. A BIOZ instruction is used to synchronize the external hardware with the program. As long as the FIFO buffer is empty, the processor polls the device until data is available.

The sampled data is read into data memory, and the filter equation is calculated. If the equation is coded in a loop, both of the auxiliary registers must be used as pointers. By starting one of the lists at location zero in data memory, the pointer for that list can also be used as the loop counter. The calculation time can be reduced by a factor of two if the equation is implemented using straight-line code. The user must decide whether program size or execution time is more important in his application.

The data is shifted in memory as the equation is computed, making a separate loop to do the shift operation unnecessary. A 0.5 is added to the result to round up the number before storing the result. The output is written to a D/A converter. Then the whole process is repeated.

The following assembly language program implements the digital filter:

```
*   The MLIB directive is used to reference a file containing  the
*   source code for the two macros, PROG and MAIN.
*
        MLIB    'MACRO.SRC'
*
        PROG    FLTR
*
*       REAL    4 X(17),CX(17),Y
*
        DSEG        BEGIN DATA SEGMENT
X1      BSS     16  16 WORDS NAME X1
X17     BSS     1   1 WORD NAME X17
CX1     BSS     16  16 WORDS NAME CX1
CX17    BSS     1   1 WORD NAME CX17
```

```
Y       BSS     1    1 WORD NAME Y
        DEND         END DATA SEGMENT
*
        B       FLTR
        RET
*
COEF    DATA    -7545,5109,7247,3667,-3685,-4868
        DATA    6707,24279,32767,24279,6707
        DATA    -4868,-3685,3667,7247,5109,-7545
*
        MAIN    FLTR

*****************************************************************
*       DATA   CX /-.2302699,.1559177,.2211667,.1119031,-.1124507,
*       1           -.1485743,.2046856,.7409326,1.0,.7409326,
*       1           .2046856,-.1485743,-.1124507,.1119031,.2211667,
*       1           .1559177,-.2302699/
*****************************************************************
*
*   ONE is a data memory location containing a 1. COEF is the address
*   where the filter coefficient table begins. The next four lines of
*   code put the value of COEF in the accumulator so that TBLR can be
*   used for reading in the coefficients.
*
        LT      ONE
        MPYK    COEF
        PAC
        LARK    AR0,16
        LARK    AR1,CX1
RCONST  LARP    1
        TBLR    *+,AR0
        ADD     ONE
        BANZ    RCONST
*
*   Test FIFO to see if it is empty. The next line of code branches on
*   itself till the BIO pin goes low.
*
WAIT    BIOZ    WAIT
*
*   Input sampled data
*
        IN      X1,PA0
*
*****************************************************************
*   DO J = 1,17
*      Y = Y + CX(J)*X(J)          Compute filter equation
*   END DO
*
*   DO J = 1,16
*      X(J) = X(J-1)               Shift variables
*   END DO
*****************************************************************
*
*   X17 is the data memory address of X(17).
*   CX17 is the data memory address of CX(17).
*
        LARK    AR0,X17
*
        LARK    AR1,CX17
        ZAC
        LT      *-,AR1
        MPY     *-,AR0
LOOP    LTD     *,AR1
        MPY     *-,AR0
```

```
        BANZ    LOOP
        APAC
*
*   Round up
*
        ADD     ONE,14
*
*   Output results
*
        SACH    Y,1
        OUT     Y,PA1
        B       WAIT
```

4.3.3.3.1 *Assembler Output*

The XDS/320 Macro Assembler requires a source file which contains the assembly language program. Two output files are created by the assembler. One output file is a listing file that prints the object code and the source statement for each instruction. The other output file contains the object code in standard 990 tagged format. The listing file for the filter program is shown below, although certain comment statements have been deleted. Object code followed by an apostrophe indicates that the code is relocatable (i.e., the B FLTR statement).


```
                                LISTING FILE

FLTR        320 FAMILY MACRO ASSEMBLER 2.0 83.010       9:20:28   2/21/83
                                                                  PAGE 0001

 0001            *   The MLIB directive is used to reference a file con-
 0002            *   taining source code for the two macros, PROG and MAIN.
 0003            *
 0004 0000               MLIB    'MACRO.SRC'
 0005            *
 0006                    PROG    FLTR
 0001                    IDT     'FLTR'
 0007            *
 0008            *       REAL    4 X(17),CX(17),Y
 0009            *
 0010 0000               DSEG        BEGIN DATA SEGMENT
 0011 0000       X1      BSS     16  16 WORDS NAME X1
 0012 0010       X17     BSS     1   1 WORD NAME X17
 0013 0011       CX1     BSS     16  16 WORDS NAME CX1
 0014 0021       CX17    BSS     1   1 WORD NAME CX17
 0015 0022       Y       BSS     1   1 WORD NAME Y
 0016 0023               DEND        END DATA SEGMENT
 0017            *
 0018 0000 F900          B       FLTR
      0001 0014'
 0019 0002 7F8D          RET
 0020            *
 0021 0003 E287  COEF    DATA    -7545,5109,7247,3667,-3685,-4868
      0004 13F5
      0005 1C4F
      0006 0E53
      0007 F19B
      0008 ECFC
 0022 0009 1A33          DATA    6707,24279,32767,24279,6707
      000A 5ED7
      000B 7FFF
      000C 5ED7
      000D 1A33
 0023 000E ECFC          DATA    -4868,-3685,3667,7247,5109,-7545
      000F F19B
```

```
     0010 0E53
     0011 1C4F
     0012 13F5
     0013 E287
0024              *
0025                        MAIN FLTR
0001 0014                   PSEG              PROG SEG
0002                        DEF  FLTR         ENTRY POINT
0003       0014' FLTR       EQU  $
0004 0014 7E01              LACK 1            MAKE CONSTANT ONE
0005 0015 5023"             SACL ONE,0        SAVE IT
0006 0016 7F89              ZAC               ZERO ACCUMULATOR
0007 0017 1023"             SUB  ONE,0        MAKE -1
0008 0018 5024"             SACL MINUS,0      SAVE IT
0009 0023                   DSEG
0010 0023         ONE       BSS  1            CONSTANT ONE
0011 0024         MINUS     BSS  1            CONSTANT -1
0012 0025         XR0       BSS  1            TEMP 0
0013 0026         XR1       BSS  1            TEMP 1
0014                        DEF  ONE,MINUS    ALLOW EXTERNAL USE
0015                        DEF  XR0,XR1      OF VARIABLE
0016 0027                   DEND              END OF DATA
0026              ************************************************************
0027              *       DATA   CX /-.2302699,.1559177,.2211667,.1119031,-.11
0028              *       1           -.1485743,.2046856,.7409326,1.0,.7409326
0029              *       1           .2046856,-.1485743,-.1124507,.1119031,.2
0030              *       1           .1559177,-.2302699/
0031              ************************************************************
0032              *
0033              * ONE is a data memory location containing a 1.  COEF is  the
0034              * address where the filter coefficient table begins. The next
0035              * four lines of code put the value of COEF in the accumulator
0036              * so that TBLR can be used for reading in the coefficients.
0037              *
0038
0039 0019 6A23"              LT       ONE
0040 001A 8003              MPYK     COEF
0041 001B 7F8E              PAC
0042 001C 7010              LARK     AR0,16
0043 001D 7111              LARK     AR1,CX1
0044 001E 6881    RCONST    LARP     1
0045 001F 67A0              TBLR     *+,AR0
0046 0020 0023"             ADD      ONE
0047 0021 F400              BANZ     RCONST
     0022 001E'
0048              *
0049              * Test FIFO to see if it is empty. The next line of code
0050              * branches on itself till the BIO pin goes low.
0051              *
0052 0023 F600    WAIT      BIOZ     WAIT
     0024 0023'
0053              *
0054              *   Input sampled data
0055              *
0056 0025 4000"             IN       X1,PA0
0057              *
0058              ************************************************************
0059              *   DO J = 1,17
0060              *      Y = Y + CX(J)*X(J)          Compute filter equation
0061              *   END DO
0062              *
0063              *   DO J = 1,16
0064              *      X(J) = X(J-1)               Shift variables
0065              *   END DO
0066              ************************************************************
```

```
0067                    *
0068                    * X17 is the data memory address of X(17).
0069                    * CX17 is the data memory address of CX(17).
0070                    *
0071 0026 7010                 LARK    AR0,X17
0072                    *
0073 0027 7121                 LARK    AR1,CX17
0074 0028 7F89                 ZAC
0075 0029 6A91                 LT      *-,AR1
0076 002A 6D90                 MPY     *-,AR0
0077 002B 6B81          LOOP   LTD     *,AR1
0078 002C 6D90                 MPY     *-,AR0
0079 002D F400                 BANZ    LOOP
     002E 002B'
0080 002F 7F8F                 APAC
0081                    *
0082                    *   Round up
0083                    *
0084 0030 0E23"                ADD     ONE,14
0085                    *
0086                    *   Output results
0087                    *
0088 0031 5922"                SACH    Y,1
0089 0032 4922"                OUT     Y,PA1
0090 0033 F900                 B       WAIT
     0034 0023'


THE FOLLOWING SYMBOLS ARE UNDEFINED
  *+
  *-
  $$LAB
  *
NO ERRORS, NO WARNINGS
```

Although the symbols above are undefined, this is a natural result of the macros used and should be ignored.

The following example is the tagged object code produced by the XDS/320 Assembler. The tags are used by the linker when it is producing a link module.

TAGGED OBJECT CODE

```
K0035FLTR    M0027$DATA 000050014FLTR  W0023ONE   00007F43AF            FLTR
W0025XR0   0000W0026XR1   0000W0024MINUS 0000A0000BF900C0014B7F8D7F1A9F FLTR
BE287B13F5B1C4FB0E53BF19BBECFCB1A33B5ED7B7FFFB5ED7B1A33BECFCBF19B7F036F FLTR
B0E53B1C4FB13F5BE287A0014B7E01#5023007FB7F89#1023007F#5024007F7F281F    FLTR
A0019#6A23007FB8003B7F8EB7010B7111B6881B67A0#0023007FBF400C001E7F250F   FLTR
BF600C0023#4000007FB7010B7121B7F89B6A91B6D90B6B81B6D90BF400C002B7F1D5F  FLTR
B7F8F#0E23007F#5922007F#4922007FBF900C00237F6E6F                        FLTR
:      FLTR       2/21/83   9:20:28    ASM320 2.0 83.010                FLTR
```

4.3.3.3.2 *Program Linkage*

The linker must be executed even if the program is contained in a single module. The control file required by the linker specifies the task name, defines the starting location for the data and program

segments, and indicates the object files to be linked. The control file which was used to link the example program is as follows:

```
FORMAT ASCII
TASK DEV
PROGRAM >0000
DATA >0000
INCLUDE S4USR.LVK111 .FLTR.OBJ
END
```

Two files are produced by the linker. The linked object file is an output file containing the load module. The link listing file is an output file containing a listing of the command control file, a map of the segments and modules which were linked, and a cross-reference listing of the externally defined variables. The link listing file and the linked object file are shown below. The object file can be loaded into the simulator or emulator for program debugging.

```
                                LINK LISTING FILE

DX/9900 LINKER   VERSION 2.0.0  82.312  2/21/83   9:29:30              PAGE   1
COMMAND LIST

FORMAT ASCII
TASK DEV
PROGRAM >0000
DATA >0000
INCLUDE  S4USR.LVK111.FLTR.OBJ
END
DX/9900 LINKER   VERSION 2.0.0  82.312  2/21/83   9:29:30              PAGE   2
LINK MAP

CONTROL FILE = S4USR.LVK111.FLTR.CF

LINKED OUTPUT FILE = S4USR.LVK111.FLTR.LINKOBJ

LIST FILE = S4USR.LVK111.FLTR.LINKLIS

OUTPUT FORMAT = ASCII

1 ---->OVERWRITTEN SEGMENTS IN MODULE DEV
DX/9900 LINKER   VERSION 2.0.0  82.312  2/21/83   9:29:30              PAGE   3

PHASE 0     DEV       MODULE  ORIGIN = 0000    LENGTH = 0000

MODULE    NO   ORIGIN    LENGTH      TYPE         DATE       TIME     CREATOR

FLTR       1    0000*     0035     INCLUDE       2/21/83    9:20:28  ASM320
$DATA      1    0000*     0027

                                D E F I N I T I O N S

 NAME    VALUE NO    NAME    VALUE NO    NAME    VALUE NO    NAME    VALUE NO

*FLTR    0014*  1   *MINUS   0024*  1   *ONE     0023*  1   *XR0     0025*  1
*XR1     0026*  1

LENGTH OF REGION FOR TASK            = 0000
```

```
NUMBER OF WARNINGS MESSAGES PRINTED   =    1

NUMBER OF RECORDS FOR MODULE DEV      =    6

TOTAL CARDS PRINTED                   =    6

**** LINKING COMPLETED     2/21/83   9:29:34
```

The following object file is an output produced by the linker:

LINKED OBJECT FILE

```
K0000DEV     90000BF900B0014B7F8DBE287B13F5B1C4FB0E53BF19BBECFC7F1C4F   DEV
B1A33B5ED7B7FFFB5ED7B1A33BECFCBF19BB0E53B1C4FB13F5BE28790014B7E017F0A0F DEV
B5023B7F89B1023B502490019B6A23B8003B7F8EB7010B7111B6881B67A0B00237F1B8F DEV
BF400B001EBF600B0023B4000B7010B7121B7F89B6A91B6D90B6B81B6D90BF4007F177F DEV
B002BB7F8FB0E23B5922B4922BF900B00237F80BF                             DEV
:     DEV         2/21/83   9:29:30    MPPLINK   82.312               DEV
```

4

4.3.3.4 *Assembly Language Program Using Absolute Code*

Through the use of the macros, PROG and MAIN, the above program is well structured and relocatable. During link time, the program and data memory locations for the coefficient CX (i.e., the value for the constant COEF), the data memory location of the variable X, and the program memory location of the MAIN program, FLTR, can be established.

In contrast to the relocatable code approach is one that uses absolute code. Although the use of absolute code makes it somewhat easier to write a single program, this program is not relocatable. The same program that was coded in relocatable code in Section 4.3.3.3 is shown below coded in absolute code.

SOURCE FILE

```
        IDT   'FLTR'
*
*   IDT is a directive which assigns a name to the module. The EQU
*   directive  assigns  values to constants.  The  constants below
*   will refer to locations  in  data  memory.  Unlike  the  above
*   program, these data memory locations are fixed  and  cannot be
*   changed at link time.  As a result, this module would  be very
*   difficult to use as part of another program.
*
X1      EQU   17
X17     EQU   33
CX17    EQU   16
Y       EQU   34
ONE     EQU   127
*
        AORG  10
*
*  The AORG directive establishes the location in program memory where
*  the code sequence will begin. In this case, the following section
*  of code will begin at program memory  location 10.  This  contrasts
*  with the above program (Section 4,3.3.3) which allows the block  of
*  memory the program will occupy to be established during link time.
*
```

```
        LARK  AR0,16
        LARK  AR1,0
*
RCONST  LARP  1
        TBLR  *+,AR0
        ADD   ONE
        BANZ  RCONST
*
WAIT    BIOZ  WAIT
*
        IN    X1,PA0
*
        LARK  AR0,X17
        LARK  AR1,CX17
        ZAC
        LT    *-,AR1
        MPY   *-,AR0
*
LOOP    LTD   *,AR1
        MPY   *-,AR0
        BANZ  LOOP
        APAC
*
        ADD   ONE,14
*
        SACH  Y,1
        OUT   Y,PA1
        B     WAIT
```

Below is the listing file for this program using absolute code.

LISTING FILE

```
FLTR        320 FAMILY MACRO ASSEMBLER  1.0          10:16: 5  12/22/82
                                                                PAGE 0001
 0001                        IDT   'FLTR'
 0002  *
 0003  * IDT is a directive which assigns a name to the module. The EQU
 0004  * directive assigns values to constants.  The constants  below *
 0005  * will  refer  to  locations  in data memory. Unlike the above *
 0006  * program, these data memory  locations  are fixed and cannot be
 0007  * changed at link time.  As a result, this  module would be very
 0008  * difficult to use as part of another program.
 0009  *
 0010      0011  X1      EQU   17
 0011      0021  X17     EQU   33
 0012      0010  CX17    EQU   16
 0013      0022  Y       EQU   34
 0014      007F  ONE     EQU   127
 0015            *
 0016 000A                AORG  10
 0017  *
 0018  * The AORG directive establishes the location in program memory
 0019  * where * the code sequence  will begin. In this case, the fol-
 0020  * lowing section of code will begin  at program memory location
 0021  * 10.  This  contrasts with the above program (Section 4.3.3.3)
 0022  * which allows the block of memory the program will  occupy  to
 0023  * be established during link time.
 0024  *
 0025 000A 7010           LARK  AR0,16
 0026 000B 7100           LARK  AR1,0
```

```
0027                    *
0028 000C 6881  RCONST  LARP  1
0029 000D 67A0          TBLR  *+,AR0
0030 000E 007F          ADD   ONE
0031 000F F400          BANZ  RCONST
     0010 000C
0032                    *
0033 0011 F600  WAIT    BIOZ  WAIT
     0012 0011
0034                    *
0035 0013 4011          IN    X1,PA0
0036                    *
0037 0014 7021          LARK  AR0,X17
0038 0015 7110          LARK  AR1,CX17
0039 0016 7F89          ZAC
0040 0017 6A91          LT    *-,AR1
0041 0018 6D90          MPY   *-,AR0
0042                    *
0043 0019 6B81  LOOP    LTD   *,AR1
0044 001A 6D90          MPY   *-,AR0
0045 001B F400          BANZ  LOOP
     001C 0019
0046 001D 7F8F          APAC
0047                    *
0048 001E 0E7F          ADD   ONE,14
0049                    *
0050 001F 5922          SACH  Y,1
0051 0020 4922          OUT   Y,PA1
0052 0021 F900          B     WAIT
     0022 0011
0053 0023
0054 0023
NO ERRORS, NO WARNINGS
```

5

PROCESSOR RESOURCE MANAGEMENT

5. PROCESSOR RESOURCE MANAGEMENT

5.1 FUNDAMENTAL OPERATIONS

An understanding of how to use the instructions to perform common tasks is necessary in order to make efficient use of the instruction set. The following sections discuss implementations of some fundamental operations using the TMS32010 instruction set.

5.1.1 Bit Manipulation

A specified bit of a word from data memory can either be set, cleared, or tested. Such bit manipulations are accomplished by using the built-in shifter and the logic instructions, AND, OR, and XOR. In the first example, operations on single bits are performed on the data word VALUE. In this and the following examples, data memory location ONE contains the value 1 and MINUS contains the value 1 (all bits set).

```
*
* Clear bit 5 of data memory location VALUE
*
        LAC     ONE,5           ACC = >00000020
        XOR     MINUS           Invert accumulator;  ACC = >0000FFDF
        AND     VALUE           Bit 5 of VALUE is zeroed
        SACL    VALUE
*
* Set bit 12 of VALUE
*
        LAC     ONE,12          ACC = >00001000
        OR      VALUE           Bit 12 of VALUE is set
        SACL    VALUE
*
* Test bit 3 of VALUE
*
        LAC     ONE,3           ACC = >00000008
        AND     VALUE           Test bit 3 of VALUE
        BZ      BIT3Z           Branch to BIT3Z if bit is clear
```

More than one bit can be set, cleared, or tested at one time if the necessary mask exists in data memory. In the next example, the six low-order bits in the word VALUE are cleared if MASK contains the value 127.

```
*
* Clear lower six bits of VALUE
*
        LAC     MASK            ACC = >0000003F
        XOR     MINUS           Invert accumulator; ACC = >0000FFC0
        AND     VALUE           Clear lower six bits
        SACL    VALUE
```

5.1.2 Data Shift

There are two types of shifts: logical and arithmetic. A logical shift is implemented by filling the empty bits to the left of the MSB with zeros, regardless of the value of the MSB. An arithmetic shift fills the empty bits to the left of the MSB with ones if the MSB is one, or with zeros if the MSB is zero. The second type of bit padding is referred to as sign extension.

The hardware shift which is built into the ADD, SUB, and LAC instructions performs an arithmetic left shift on a 16-bit word. This feature can also be used to peform right shifts. A right shift of n is implemented by peforming a left shift of 16-n and saving the upper word of the accumulator.

The first example performs an arithmetic right shift of seven on a 16-bit number in the accumulator.

```
SACL    TEMP            Move number to memory
LAC     TEMP,9          Shift left (16-7)
SACH    TEMP            Save high word in memory
LAC     TEMP            Return number back to accumulator
```

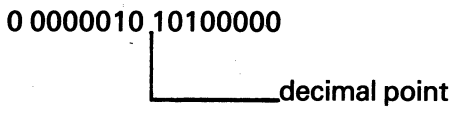
The second example performs a logical right shift of four on a 32-bit number stored in the accumulator. The 32-bit results of the shift are then stored in data memory. In this example, the accumulator initially contains the hex number >9D84C1B2. The variables, SHIFTH and SHIFTL, will receive the high word (>09D8) and low word (>4C1B) of the shifted results.

```
*
* Shift the lower word
*
        SACH    SHIFTH          SHIFTH = >9D84   Initial values
        SACL    SHIFTL          SHIFTL = >C1B2
        LAC     SHIFTL,12       ACC = >FC1B2000
        SACH    SHIFTL          SHIFTL = >FC1B
        LAC     MINUS,12        ACC = >FFFFF000
        XOR     MINUS           ACC = >FFFF0FFF
        AND     SHIFTL          ACC = >00000C1B
*
* Shift the upper word
*
        ADD     SHIFTH,12       ACC = >F9D84C1B
        SACL    SHIFTL          SHIFTL = >4C1B   Final low-order value
        SACH    SHIFTH          SHIFTH = >F9D8
        LAC     MINUS,12        ACC = >FFFFF000
        XOR     MINUS           ACC = >FFFF0FFF
        AND     SHIFTH          ACC = >000009D8
        SACL    SHIFTH          SHIFTH = >09D8   Final high-order value
```

An arithmetic right shift of four can be implemented using the same routine as shown above, except with the last four lines omitted.

5.1.3 Fixed-Point Arithmetic

Computation on the TMS32010 is based on a fixed-point two's complement representation of numbers. Each 16-bit number is evaluated with a sign bit, i integer bits, and 15-i fractional bits. Thus the number:



has a value of 2.625. This particular number is said to be represented in a Q8 format (8 fractional bits). Its range is between -128 (1000000000000000) and 127.996 (0111111111111111). The fractional accuracy of a Q8 number is about .004 (one part in 2**8 or 256).

Although particular situations (e.g., a combination of dynamic range and accuracy requirements) must use mixed notations, it is more common to work entirely with fractions represented in a Q15 format or integers in a Q0 format. This is especially true for signal processing algorithms where multiply-accumulate operations are dominant. The result of a fraction times a fraction remains a fraction, and the result of an integer times an integer remains an integer. No overflows are possible.

The difficulty comes during accumulations of the resulting products. In these situations, the programmer must understand the physical process which underlies the mathematics in order to take care of potential overflow conditions. The following sections discuss some of the techniques involved in using this kind of number representation.

5.1.3.1 *Multiplication*

There are a wide variety of situations which might be encountered when multiplying two numbers. Three of these scenarios are illustrated below:

CASE I -- FRACTION * FRACTION

Q15 * Q15 = Q30

```
  0100000000000000  =  0.5  in Q15 notation
* 0100000000000000  =  0.5  in Q15
-----------------------------------
00 0100000000000 0000000000000000  = 0.25  in Q30
   |
   └──────decimal point
```

Note: Two sign bits remain after the multiply.

5

Generally, the programmer will not want to maintain full precision. In fact, he will probably want to save a single-precision (16-bit) result. Unfortunately, the upper half of the result does not contain a full 15 bits of fractional precision since the multiply operation actually creates a second sign bit. In order to recover that precision, the product must be shifted left by one bit. The following code excerpt illustrates an implementation of this example:

```
LT      OP1      OP1 = >4000 (0.5 in Q15)
MPY     OP2      OP2 = >4000 (0.5 in Q15)
PAC
SACH    ANS,1    ANS = >2000 (0.25 in Q15)
```

The MPYK instruction in the TMS320 will allow the programmer the ability to multiply by a 13-bit signed constant. In fractional notation, this means he can multiply a Q15 number by a Q12 number. This case requires the programmer to shift the resulting number left by four bits to maintain full precision.

```
LT      OP1      OP1 = >4000 (0.5 in Q15)
MPYK    2048     OP2 = >0800 (0.5 in Q12)
PAC
SACH    ANS,4    ANS = >2000 (0.25 in Q15)
```

CASE II -- INTEGER * INTEGER

Q0 * Q0 = Q0

```
  0000000000010001  = 17   in Q0
* 1111111111111011  = 5    in Q0
-----------------------------------
11111111111111111111111110101011  =85   in Q0
                                  |_____ decimal point
```

Note: In this case, the extra sign bit does not come into play, and the desired product is entirely in the lower half of the product. The following program illustrates this example.

```
LT    OP1       OP1 = >0011 (17 in Q0)
MPY   OP2       OP2 = >0005 ( 5 in Q0)
PAC
SACL  ANS       ANS = >0055 (85 in Q0)
```

CASE III -- MIXED NOTATION

Q14 * Q14 = Q28

```
  0110000000000000  = 1.50 in Q14
* 0011000000000000  = 0.75 in Q14
-----------------------------------
0001,00100000000000000000000000  = 1.125 in Q28
  /
 /__ decimal point
```

The maximum magnitude of a Q14 number is just under two. Thus, the maximum magnitude of the product of two Q14 numbers is four. Two integer bits are required to allow for this possibility, leaving a maximum precision for the product of 13 bits. In general, the following rule applies:

> The product of a number with i integer bits and f fractional bits and a second number with j integer bits and g fractional bits will be a number with (i + j) integer bits and (f + g) fractional bits. The highest precision possible for a 16-bit representation of this number will have (i + j) integer bits and (15- i - j) fractional bits.

If, however, the programmer has a prior knowledge of the physical system which is being modelled, he may be able to increase the precision with which the number is modelled. For example, if he knows that the above product can be no more than 1.8, he could represent the product as a Q14 number rather than the theoretical worst case of Q13. The following program illustrates the above example:

```
LT      OP1       OP1 = >6000 (1.5 in Q14)
MPY     OP2       OP2 = >3000 (.75 in Q14)
PAC
SACH    ANS,1     ANS = >2400 (1.125 in Q13)
```

The techniques which have been illustrated above all truncate the result of the multiplication to the desired precision. The error which is generated as a result amounts to minus one full LSB. This is true whether the truncated number is positive or negative. It is possible to implement a simple rounding technique to reduce this potential error by a factor of two. This is illustrated by the following code sequence:

```
LT      OP1
MPY     OP2       OP1 * OP2
PAC
ADD     ONE,14    ROUND UP
SACH    ANS,1
```

The error generated in this example is plus one-half LSB whether ANS is positive or negative.

5.1.3.2 *Addition*

During the process of multiplication, the programmer is not concerned about overflows and needs only to adjust his decimal point following the operation. Addition is a much more complex process. First, both operands of an addition must be represented in the same Q-point notation. Second, the programmer must either allow enough head room in his result to accomodate bit growth or he must be prepared to handle oveflows. If the operands are only 16 bits long, the result may have to be represented as a double-precision number. The following example illustrates two approaches to adding 16-bit numbers:


```
                Maintaining 32-Bit Results:

        LAC     OP1       Q15
        ADD     OP2       Q15
        SACH    ANSHI     High-order 16 bits of result
        SACL    ANSLO     Low-order 16 bits of result


    Adjusted Decimal Point to Maintain 16-Bit Results:

        LAC     OP1,15    Q14 number in ACCH
        ADD     OP2,15    Q14 number in ACCH
        SACH    ANS       Q14
```

Double-precision operands present a more complex problem. In this case, actual arithmetic overflows or underflows might occur. The TMS32010 provides the programmer with the facility to check for the occurrence of these conditions using the BV instruction. A second technique is the use of saturation mode operations which will saturate the result of overflowing accumulations to the most positive or most negative number. Unfortunately, both techniques will result in a loss of precision. The best technique involves a thorough understanding of the underlying physical process and care in selecting number representations.

5.1.3.3 *Division*

Binary division is the inverse of multiplication. Multiplication consists of a series of shift and add operations, while division can be broken down into a series of subtracts and shifts. The following example illustrates this process:

Given an 8-bit accumulator, suppose the problem is to divide the number 10 by 3. The process consists of gradually shifting the divisor relative to the dividend, subtracting at each stage, and inserting bits into the quotient if the subraction was successful.

1. First line up the LSB of the divisor with the MSB of the dividend.

```
 00001010
-00011000
 11110010
```

2. Since the result is negative (the subtraction was unsuccessful), throw away the result, shift the dividend, and try again.

```
 00010100
-00011000
 11111000
```

3. The result is still negative. Throw away the result, shift, and try again.

```
 00101000
-00011000
 00010000
```

4. The answer is now positive. Shift the result and add one to set up the fourth and final subtraction.

```
 00100001
-00011000
 00001001
```

5. The answer is again positive. Shift the result and add one. The most significant four bits represent the remainder, while the least significant four bits represent the quotient.

```
00010011
 \   \
  \   \____ Quotient = 0011
   \_______ Remainder = 0001
```

The TMS32010 does not have an explicit divide instruction. However it is possible to implement an efficient flexible divide capability using the conditional subtract instruction, SUBC. The only restriction for the use of this instruction is that both operands be positive. It is also very important that the programmer understand the characteristics of his potential operands, such as whether the quotient can be represented as a fraction and the accuracy to which the quotient is to be computed. Each of these considerations can affect how the SUBC is used.

The examples below illustrate two different situations.

DIV1 — CASE 1 — NUMERATOR < DENOMINATOR — DIV1

TITLE: Division Routine I

NAME: DIV1

OBJECTIVE: To divide two binary two's complement numbers of any sign where the numerator is less than the denominator

ALGORITHM: ((((((A – B)*2) + 1) – B)*2) + 1) – B... = C

if, A – B> = 0,(((A – B)*2) + 1) – B > = 0 ...

where A = denominator, B = numerator, C = quotient

CALLING SEQUENCE: CALL DIV1

ENTRY CONDITIONS: Numerator < Denominator

EXIT CONDITIONS: Quotient stored in data memory location labelled QUOT

PROGRAM MEMORY REQUIRED: 22 words, excluding macros

DATA MEMORY REQUIRED: 4 words

STACK REQUIRED: None

EXECUTION TIME: 61-64 machine cycles

FLOWCHART: DIV1

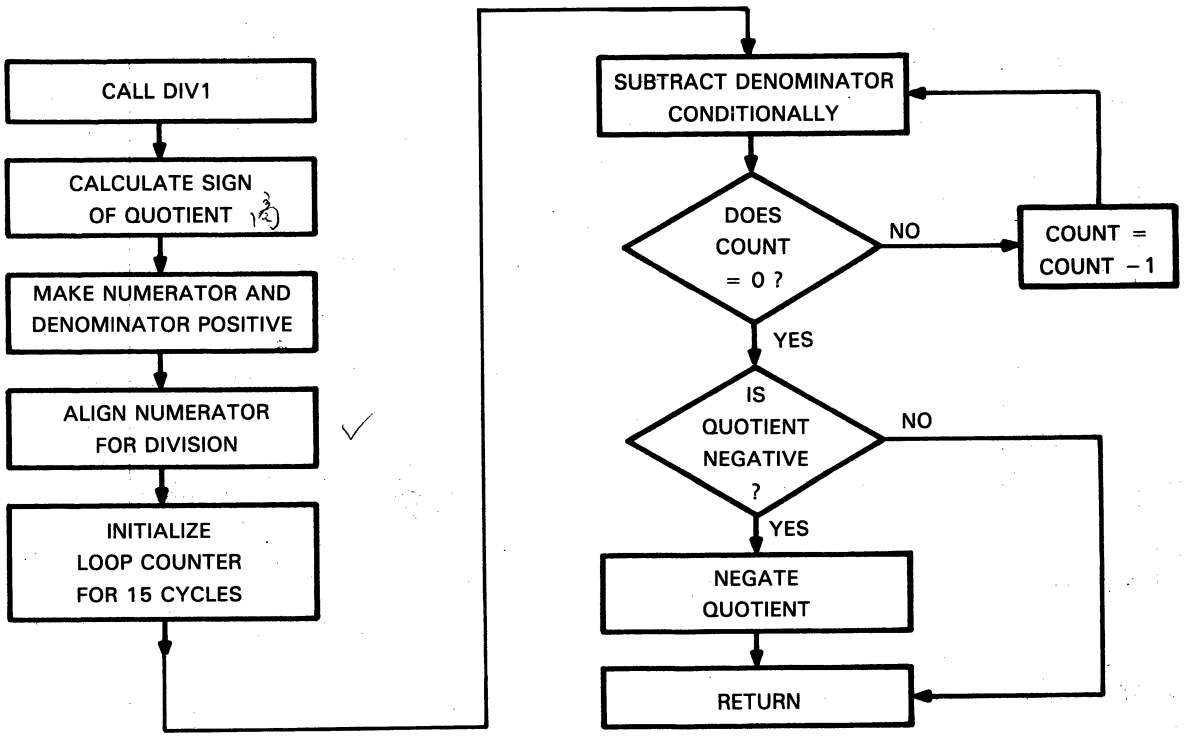


FIGURE 5-1 – DIVISION ROUTINE I FLOWCHART

SOURCE:

```
*
DIV1    LARP   0
        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                    Make numerator positive
        LARK   0,14
*
KPDVNG  SUBC   DENOM           15-cycle divide loop
        BANZ   KPDVNG
*
        SACL   QUOT
        LAC    TEMSGN
        BGEZ   DONE            Done if sign positive
*
        ZAC
        SUB    QUOT
        SACL   QUOT            Negate quotient if negative
*
DONE    RET
```


EXAMPLE:

CALL DIV1

	BEFORE INSTRUCTION	AFTER INSTRUCTION
NUMERA	21	21
DENOM	42	42
QUOT	0	.5

(0.1 0 0)

DIV2

CASE 2 — SPECIFY ACCURACY OF QUOTIENT

DIV2

TITLE: Division Routine II

NAME: DIV2

OBJECTIVE: To divide two binary two's complement numbers of any sign, specifying the fractional accuracy of the quotient

ALGORITHM: ((((((A – B)*2) + 1) – B)*2) + 1) – B... = C

if A – B> = 0,(((A – B)*2) + 1) – B > = 0,...

where A = numerator, B = denominator, C = quotient

CALLING SEQUENCE: CALL DIV2

ENTRY CONDITIONS: FRAC specifies accuracy of quotient

EXIT CONDITIONS: Quotient stored in data memory location labelled QUOT

PROGRAM MEMORY REQUIRED: 24 words, excluding macros

DATA MEMORY REQUIRED: 5 words

STACK REQUIRED: None

EXECUTION TIME: 67 – 70 + 3*FRAC clocks

FLOWCHART: DIV2

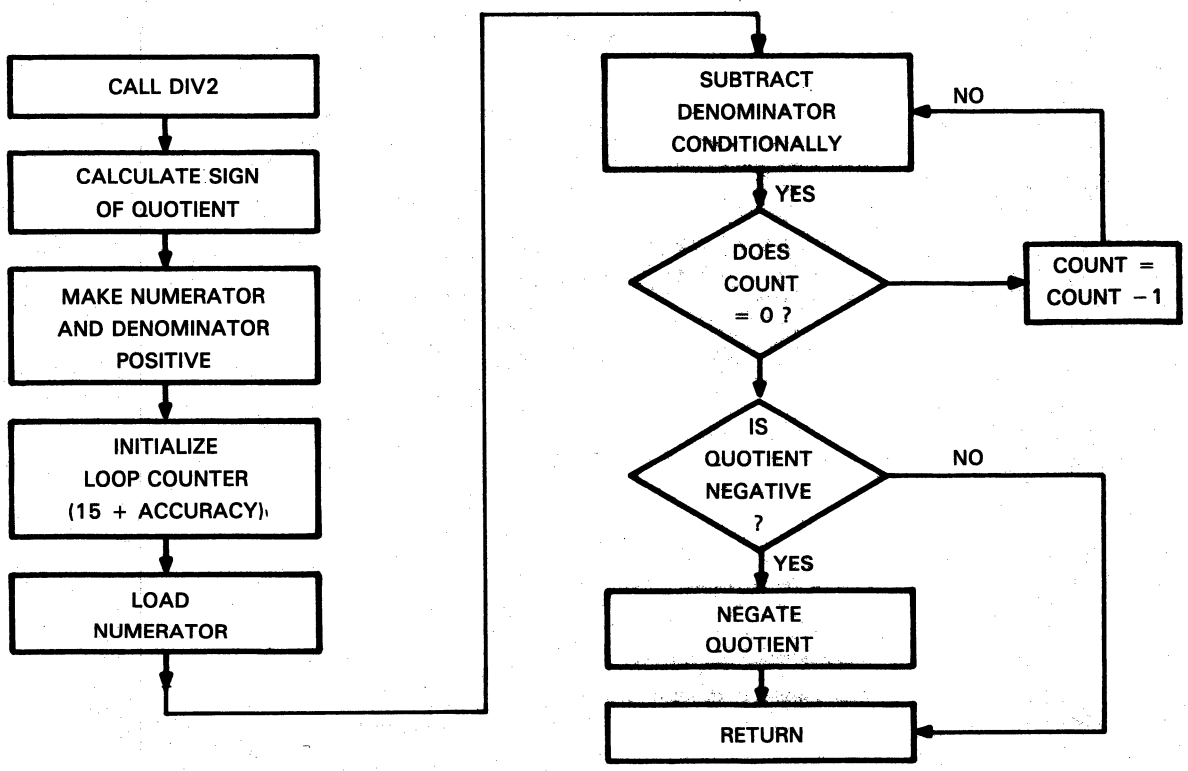


FIGURE 5-2 – DIVISION ROUTINE II FLOWCHART

SOURCE:

```
*
DIV2    LARP   0
        LT     NUMERA          Get sign of quotient
        MPY    DENOM
        PAC
        SACH   TEMSGN          Save sign of quotient
        LAC    DENOM
        ABS
        SACL   DENOM           Make denominator positive
        LACK      15
        ADD       FRAC
        SACL      FRAC         Compute loop count
        LAC    NUMERA          Align numerator
        ABS                    Make numerator positive
        LAR    0,FRAC
*
KPDVNG  SUBC   DENOM           16 + FRAC cycle divide loop
        BANZ   KPDVNG
*
        SACL   QUOT
        LAC    TEMSGN
        BGEZ   DONE            Done if sign positive
*
        ZAC
        SUB    QUOT
        SACL   QUOT            Negate quotient if negative
*
DONE    RET
```

EXAMPLE:

CALL DIV2

	BEFORE INSTRUCTION		AFTER INSTRUCTION
NUMERA	11	NUMERA	11
DENOM	8	DENOM	8
FRAC	3	FRAC	3
QUOT	17	QUOT	1.375

(1.0 1 1)

5.1.4 *Subroutines*

When a subroutine call is made using the CALL or CALA instruction, the PC + 1 (return address) is saved on the top of the stack. At the end of the subroutine, a RET instruction is executed which updates the PC with the value saved on the stack. The program will then resume execution at the instruction following the subroutine call.

There are two occasions in which a level of stack must be reserved for the machine's use. First, the TBLR and TBLW instructions use one level of stack. Second, when interrupts are enabled, the PC is saved on the stack during the interrupt routine. If a system is designed to use both interrupts and a TBLR or TBLW instruction, only two levels of stack are available for nesting subroutine calls.

NOTE

If the hardware emulator will be used for system development, the level of stack which is reserved for TBLR and TBLW will be used by the emulator to store a return address whenever the program execution is suspended by the emulator. Therefore, if neither the TBLR or TBLW instruction is used, one level of stack must still be reserved for use by the emulator.

Subroutine calls can be nested deeper than two levels if the return address is removed from the stack and saved in data memory. The POP instruction moves the top of stack (TOS) into the accumulator and pops the stack up one level. The return address can then be stored in data memory until the end of the subroutine when it is put back into the accumulator. The PUSH instruction will push the stack down one level and then move the accumulator onto the TOS. Therefore, when the RET instruction is executed, the PC is updated with the return address. This procedure will allow a second subroutine to be called inside the first routine without using another level of stack.

The POP and PUSH instructions can also be used to pass arguments to a subroutine. DATA directives following the subroutine call create a list of constants and/or variables to be passed to the subroutine. After the subroutine is called, the TOS points to the list of arguments following the CALL instruction. By moving the argument pointer from the TOS into the accumulator, the list of arguments can be read into data memory using the TBLR instruction. Between each TBLR instruction, the accumulator must be incremented by one to point to the next argument in the list. To create the return address, the argument pointer is incremented past the last element in the argument list. The PUSH instruction moves the return address onto the TOS, and the RET instruction updates the PC.

The following example illustrates a call which passes two arguments to a subroutine.

```
          .
          .
          .
          CALL    CBITS
          DATA    VALUE
          DATA    >0FFF
          .
          .
          .

*****************************************************
*                  Clear Bits                       *
* This subroutine clears the bits of a data word desig- *
* nated by a mask. The bits set to one in the mask  *
* indicate the bits in the data word to be cleared. All *
* other bits remain unchanged. Two arguments are passed *
* to this subroutine:                               *
```

```
*     1st argument = address of data word                    *
*     2nd argument = mask                                    *
*                                                            *
* Calling sequence:  CALL    CBITS                           *
*                    DATA    1st argument                    *
*                    DATA    2nd argument                    *
************************************************************

CBITS SAR     AR0,XR0       Save AR0 in temporary location

      POP                   Hold return address
      TBLR    XR1           1st argument = pointer to data
      LAR     AR0,XR1       Put 1st argument into AR0
      ADD     ONE
      TBLR    XR1           2nd argument = mask
      ADD     ONE
      PUSH                  Put return address on TOS

      LARP    0
      LAC     XR1           Load mask into accumulator
      XOR     MINUS         Invert mask
      AND     *             Clear bits
      SACL    *

      LAR     AR0,XR0       Restore AR0
      RET
```

5

5.1.5 *Computed GO TOs*

The CALA instruction executes a subroutine call based on the address contained in the accumulator. This instruction can be used to perform a computed GO TO. The address of the subroutine can be computed from a data value to determine which one of several routines will be executed. The return at the end of each of these routines will cause program execution to resume with the instruction following the CALA command. It should be noted that the CALA instruction will use a level of stack, because it is an indirect subroutine call and not just an indirect branch.

The example below illustrates how to compute a call to one of several routines. The subroutines are defined first, and then a table of branches to each subroutine is created. The main part of the program inputs a data value of 0, 1, or 2. The appropriate address in the table is calculated in the accumulator. An indirect subroutine call causes the proper branch in the table to be executed.

```
SUB1  IN      DAT1,PA0
      RET

SUB2  IN      DAT1,PA1
      RET

SUB3  IN      DAT1,PA2
      RET

TBL1  B       SUB1
      B       SUB2
      B       SUB3

      LT      ONE
      MPYK    TBL1            Get address of table
      PAC
      IN      VALUE,PA4       Input data from PA4
      LT      VALUE
```

```
        MPYK    2               Calculate offset
        APAC
        CALA                    Go to designated subroutine
        LAC     DAT1            Return here after subroutine
        .
        .
        .
```

5.2 ADDRESSING AND LOOP CONTROL WITH AUXILIARY REGISTERS

There are two auxiliary registers on the TMS32010. The auxiliary registers can be used either as loop counters or as pointers for indirect addressing.

5.2.1 Auxiliary Register Indirect Addressing

In the indirect addressing mode, the auxiliary register pointer (ARP) is used to determine which auxiliary register is selected. The LARP instruction sets the ARP equal to the value of the immediate operand. The value of the ARP can also be changed in the indirect addressing mode; the ARP is updated after the instruction has been executed.

The contents of the auxiliary register are interpreted as a data memory address when the indirect addressing mode is used. A sequential list of data can easily be accessed in the indirect mode by using the autoincrement or autodecrement feature of the auxiliary registers. If the auxiliary register contains a data memory address, the counter can be used to increment through the entire address space. The auxiliary register should not be used as a general purpose incrementer, because only the lower nine bits of the register actually count. A special instruction, MAR, allows the auxiliary register which is selected by the ARP to be incremented or decremented without implementing any other operation in parallel.

There are three instructions (LARK, LAR, SAR) which either load or store a value into an auxiliary register, independent of the value of the ARP. The first operand in each of these instructions determines which auxiliary register is to be either loaded or stored. This operand does not affect the value of the ARP for subsequent instructions.

The example below illustrates using an auxiliary register in the indirect addressing mode to input data into a block of memory.

```
        LARK    AR0,DATBLK      Initialize AR0 as a pointer to
                                DATBLK (an area of 8 words in
                                data memory)
        LARP    0               Select AR0
        LACK    8               Initialize accumulator as a counter

LOOP    IN      *+,PA0          Input data
        SUB     ONE             Decrement counter (ONE contains
                                value 1)
        BNZ     LOOP            Repeat until count=0
```

5.2.2 Loop Counter

An auxiliary register can also be used as a loop counter. The BANZ instruction will test and then decrement the auxiliary register selected by the ARP. Because the test for zero occurs before the auxiliary register is decremented, the value loaded into the auxiliary register must be one less than the number of times the loop should be executed. The maximum number of loops which can be counted is 512, because only nine bits of each auxiliary register are implemented as counters.

The example below inputs data and calculates the sum while the auxiliary register is used to count the number of loops. The accumulator will contain the result.

```
        LARK    AR0,3           Initialize AR0 as a counter
        LARP    0               Select AR0
        ZAC                     Clear accumulator

LOOP    IN      DATA1,PA2       Input data value
        ADD     DATA1           Add data to accumulator
        BANZ    LOOP            Repeat loop four times
```

5.2.3 Combination of Operational Modes

Both indirect addressing and loop counting can be performed at the same time to implement loops efficiently. If the data block is defined to start at location 0 in data memory, the same auxiliary which is counting the number of loops can also be the pointer for indirect addressing.

The example below illustrates using the same auxiliary register as both a counter and a pointer. Data locations 0 through 7 are loaded with input data.


```
        LARK    AR0,7           AR0 points to end of data block
LOOP    IN      *,PA0           Input data
        BANZ    LOOP            Repeat loop 8 times
```

The data block does not have to start at zero if one auxiliary register is used for counting and the other auxiliary register is used as a pointer. The following example illustrates how both auxiliary registers can be used at once.

```
        LARK    AR0,7           Initialize AR0 as a counter
        LARK    AR1,DATBLK      AR1 points to start of DATBLK,
                                data memory area
        ZAC
LOOP    LARP    1               Point to AR1
        ADD     *+,AR0          Calculate sum of data in block;
                                point to AR0
        BANZ    LOOP            Repeat loop 8 times
```

5.3 MULTIPLICATION AND CONVOLUTION

The hardware multiplier will peform a 16 X 16-bit multiply and produce a 32-bit result. This section will discuss the features of the multiplier and give examples which illustrate how to efficiently use the multiply instructions.

5.3.1 Pipelined Multiplications

A single multiply operation consists of three steps on the TMS32010. First, one of the operands is loaded into the T register from data memory using the LT instruction. The second step is performed by specifying the second operand using either the MPY or MPYK instruction. MPY obtains the second operand from data memory, and MPYK uses an immediate operand as the other operand to be multiplied. The third step moves the output from the (product) P register to the accumulator by using one of three instructions, PAC, APAC, or SPAC. The PAC instruction loads the accumulator

with the value from the P register; the APAC instruction adds the product register to the accumulator; and the SPAC instruction subtracts the P register from the accumulator. Since each of the steps is a one-clock cycle, a single multiply-accumulate operation takes 600 ns.

If several multiplies are to be performed consecutively, the first and third steps of the multiplication process can be done in parallel. This method reduces the time of a multiply-accumulate operation to 400 ns. Multiplication can be pipelined by using the LTA instruction. This instruction loads the T register with the first operand for the next multiplication and adds the P register to the accumulator for the current multiplication.

The example below performs a pipelined multiplication.

```
*************************************************************
* The equation to be calculated is:                         *
*      t = Aw + Bx + Cy + Dz                                *
*************************************************************

      ZAC
      LT      W
      MPY     A
      LTA     X               ACC = Aw
      MPY     B
      LTA     Y               ACC = Aw+Bx
      MPY     C
      LTA     Z               ACC = Aw+Bx+Cy
      MPY     D
      APAC                    ACC = Aw+Bx+Cy+Dz
      SACH    T1
      SACL    T2              Store results
```


5.3.2 Moving Data

When implementing a digital filter, the variables in the equation represent the inputs and outputs at discrete times. Typically this type of data structure is implemented as a shift register where the data at time t is shifted to the position previously occupied by the data at time t-1. If consecutive addresses in data memory correspond to consecutive time increments, then shifts can be accomplished simply by moving the data item at location d to that corresponding to d + 1. The DMOV command allows a data word to be written into the next higher memory location in a single cycle without affecting the accumulator. Therefore, if the variables are placed in consecutive locations, a DMOV command can be used to move each of the variables before the next calculation is peformed.

The data move operation is combined with the LTA instruction to create the LTD instruction. This instruction performs three operations in parallel. The operand of the instruction is loaded into the T register; the operand is also written into the next higher memory location; and the P register is added to the accumulator. When using the LTD instruction, the order of the multiply and accumulate operations becomes important because the data is being moved while the calculation is being performed. The oldest input variable must be multiplied by its constant and loaded into the accumulator first. Then the input, which is one time-unit delay less, is multiplied and accumulated. This process is repeated until the entire equation has been computed.

The following example illustrates the input variables being moved in memory as the results are calculated:

```
*********************************************************
* The following equation is used to implement a filter:  *
*    y(n)=[Ax(n-1)+Bx(n-2)+Cx(n-3)+Dx(n-4)] * 2**-16     *
*********************************************************

START IN      X1,PA0        Input sample
      ZAC
      LT      X4            x(n-4)
      MPY     D
      LTD     X3            ACC=Dx4; x(n-4)=x(n-3)
      MPY     C
      LTD     X2            ACC=Dx4+Cx3; x(n-3)=x(n-2)
      MPY     B
      LTD     X1            ACC=Dx4+Cx3+Bx2; x(n-2)=x(n-1)
      MPY     A
      APAC                  ACC=Dx4+Cx3+Bx2+Ax1
      SACH    Y
      OUT     Y,PA1         Output results
      B       START
```

5.3.3 Product Register

The product register stores the results of a multiplication until another multiplication is peformed. A user may want to use the multiplier during the interrupt routine, but the product register must be restored with the value it contained before the interrupt occurred. It is easy to save the product register in data memory, but it is very difficult to restore the product register with the value that was saved in memory. A hardware feature has been built into the interrupt logic to prevent an interrupt from occurring immediately after a multiply instruction (MPY or MPYK). If the contents of the product register are always transferred into the accumulator on the instruction following the multiply, the product register could be changed during the interrupt routine without having to be restored before returning from the interrupt. Therefore, a PAC, APAC, SPAC, LTA, or LTD should always follow a MPY or MPYK instruction. This rule should be followed whenever the multiplier is being used during the interrupt routine.

The value of the product register can be restored if the contents are saved in memory, but it is a very time-consuming process. If the magnitude of the value saved in memory is greater than fifteen bits, it must be factored into two smaller numbers in order to restore the product register.

5.4 MEMORY CONSIDERATIONS OF HARVARD ARCHITECTURE

The memory organization on the TMS32010 is referred to as a Harvard architecture. This means that the program memory is separate from the data memory. This type of architecture allows the next instruction fetch to occur while the current instruction is fetching data and executing the operation. While the concept of a Harvard architecture increases the speed of the machine, there are disadvantages in having the program memory totally separate from data memory. The instruction set, therefore, includes instructions which transfer a word between data memory and program memory. The following sections illustrate how to make efficient use of the ablility to exchange data between memories.

5.4.1 Moving Constants into Data Memory

Most signal processors have a separate memory space for storing constants. By allowing communication between data and program memory, the TMS32010 is able to incorporate a constant memory capability with its program memory. This method allows a more efficient use of memory space. The portion of memory not used for storing constants is available for use as program space.

There are five immediate instructions in the instruction set which provide an efficient way to execute operations using constants. Two immediate instructions, LARP and LDPK, modify the program context.

LARP changes the auxiliary register pointer, and LDPK changes the data page pointer. Three other immediate instructions, LACK, LARK, and MPYK, allow constants to be used in calculations. LACK and LARK both require an unsigned operand with a magnitude no greater than eight bits. The MPYK instruction allows a 13-bit signed number as an operand.

A 16-bit data value can be moved from program memory to data memory using the TBLR instruction. TBLR requires that the program memory address (the source) be in the accumulator, while the data memory address (the destination) is obtained from the operand of the instruction. The TBLR instruction is commonly used to look up values in a table in program memory. The address of the value in the table is computed in the accumulator before executing the instruction. TBLR then moves the value into data memory. TBLR is a three-cycle instruction and, therefore, takes longer than an immediate instruction. However, it has more flexibility since it operates on 16-bit constants.

The example below illustrates bringing the cosine value of a variable into data memory.

```
* First, a table containing the cosine values is created in
* program memory.

COSINE        DATA
              .
              .
              .
        .
        .
START   IN      X,PA0
        LACK    COSINE      Load table address
        ADD     X           Calculate program memory address
        TBLR    COSX        Move value into data memory
```

Note: If the address of COSINE is larger than 255, the address can be loaded into the accumulator by loading the T register with a one and then "multplying by the constant COSINE.

5.4.2 Data Memory Expansion

Often it is necessary to expand data storage capability by using external memory. If the storage requirements are small, additional memory can be added as a RAM extension of the program memory address space. This technique is very efficient in terms of additional hardware requirements, but it has two drawbacks. It requires that the combination of the memory required to store the program and accomodate data be limited to 4096 words. It also tends to limit system throughput, since access to data in program memory is relatively slow. The minimum memory access time using this technique is four clocks (800 ns), but six clocks (1200 ns) is a more likely average.

A system requiring larger memories or faster data access can be implemented by treating the expanded data memory as an I/O device. Since the TMS32010 lacks the capability to address a large I/O address space (it is limited to eight devices), this technique also requires the use of an external address register. This register can be implemented as a counter to allow efficient access to contiguous data buffers. See Section 6.1.3 on I/O design techniques for more details.

5.4.3 Program Memory Expansion

Using the MC/ $\overline{MP}$ pin on the TMS32010, the applications engineer can choose between two distinct techniques for structuring his program memory address space. (See Figure 5-3.) In the microcomputer mode, the internal masked ROM is active and consumes the low 1536 words of the address space. The remaining 2560 words can be implemented using external memory. If the microprocessor mode is selected, the entire 4096 word address space is assumed to exist external to the chip.

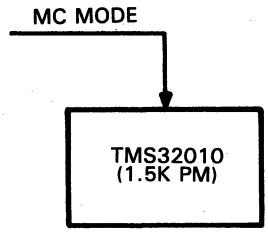


FIGURE 5-3A – USE OF INTERNAL PROGRAM MEMORY

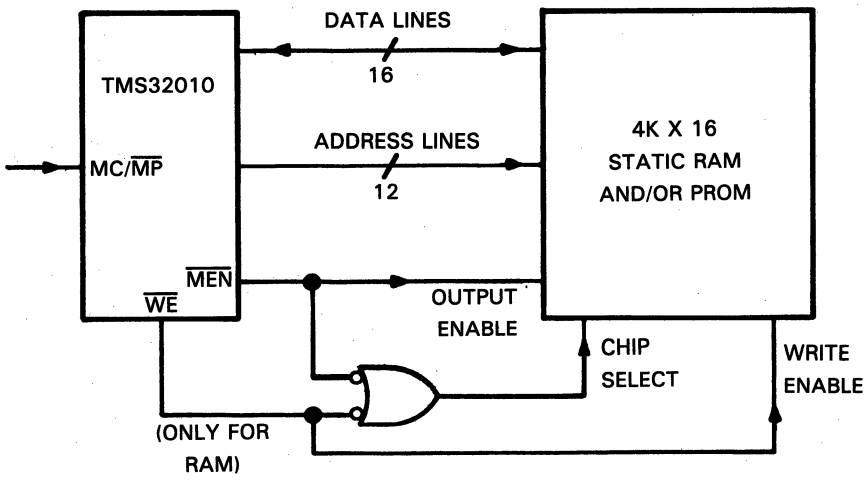


FIGURE 5-3B – USE OF EXTERNAL PROGRAM MEMORY

FIGURE 5-3 – TECHNIQUES FOR EXPANDING PROGRAM MEMORY

In the microcomputer mode, only the upper 2.5K words of external program memory are used. In the microprocessor mode, all 4K words of external memory are used. With some types of memory elements, additional chip-select logic may be necessary.

External program memory may utilize either RAM or ROM. In either case, system operation at the full 5-MHz clock rate requires that the memory exhibit an access time of less than 100 ns. If RAM is used, it may be loaded either via the TMS32010 itself using a boot ROM, or via a dual RAM port from an independent controller.

INPUT/OUTPUT DESIGN TECHNIQUES

6. INPUT/OUTPUT DESIGN TECHNIQUES

An interrupt-driven sampled data interface is the most common for signal processing applications, but other types of peripherals can also be used. This section illustrates several examples and discusses some of the hardware and software issues which should be considered when designing an I/O system for the TMS32010.

6.1 PERIPHERAL DEVICE TYPES

Using a three-bit port address, the TMS32010 is capable of accessing eight different input devices and eight different output devices. The port number is placed on the external address lines during the second cycle of the instruction. The address lines can be decoded to select one of several devices attached to the data bus or to activate a single control line. Three classes of peripherals are discussed below.

6.1.1 Registers

A register can be used for several different functions. The most simplistic interface uses a 16-bit dual port transceiver. Such a register allows two-way communication between the TMS32010 and another processor. Handshaking between the processors can be implemented by using interrupts on the TMS32010. In Figure 6-1, the acknowledge line from the other processor is connected to the BIO pin in order to synchronize the TMS32010.

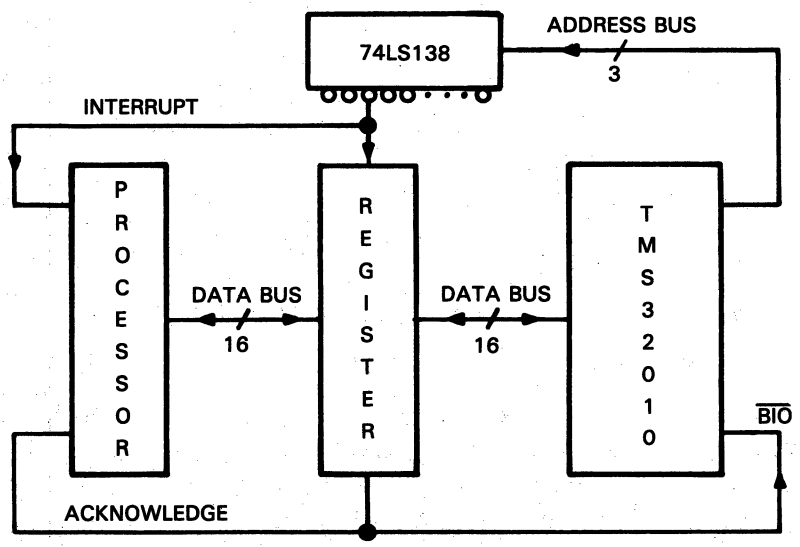


FIGURE 6-1 — COMMUNICATION BETWEEN PROCESSORS

In a more complicated configuration, a shift register can be used to convert a serial data stream into parallel data to be compatible with the I/O instructions. An analog device which can be interfaced to this processor is a codec. It is simply an A/D converter and D/A converter which is designed to operate in a telecommunications environment. This serial device produces eight-bit logarithmically-weighted digital data. Consequently, a codec interface must include a mechanism for serial to parallel conversion and a facility for code conversion. A shift register can provide the parallel input to the TMS32010. The code converter for A/D data can be implemented either in hardware using a 256 X 16-bit ROM or in software.

Another example of a register-based I/O system is a very simple A/D channel where the output of an A/D converter is buffered using a single parallel register. This requires that the A/D system be serviced before the next data sample overwrites the previous sample stored in the register. Unfortunately, a routine which only services a single data word for every interrupt can be very time consuming. The service overhead time can be reduced by multiword buffering (see Section 6.1.2 for discussion of FIFOs and interrupts).

6.1.2 FIFOs

The use of FIFOs instead of registers offers three definite advantages as follows:

1) Single address access to multiple data words,
2) Reduction of I/O overhead (since several words can be accessed for each interrupt),
3) Preservation of temporary information in data stream.

Figure 6-2 illustrates the use of a FIFO in a typical analog subsystem.

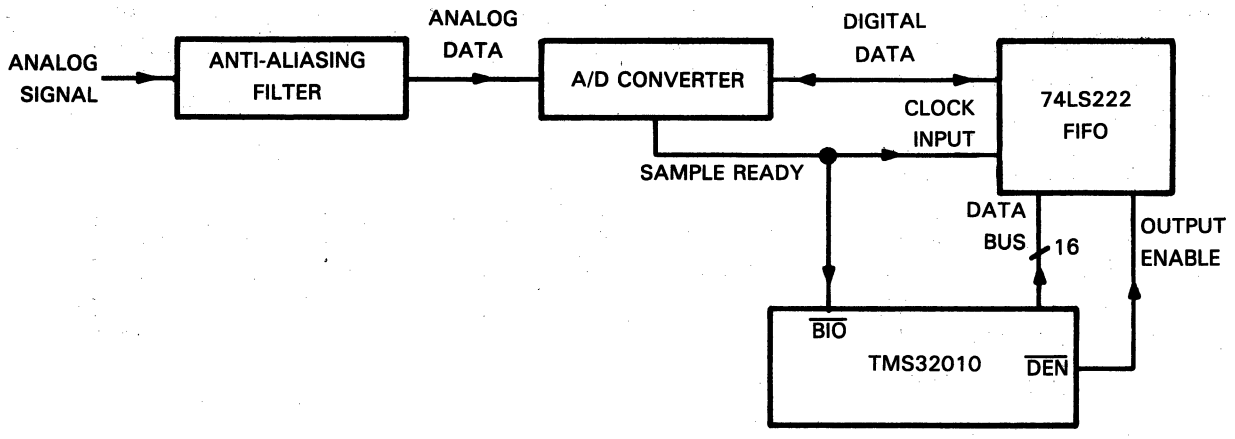


FIGURE 6-2 — TYPICAL ANALOG SYSTEM INTERFACE

6.1.3 Extended Memory Interface

The peripheral which requires the most hardware to implement is a large memory. Because the address lines only access locations 0-7 during an I/O operation an external address counter must be used to provide an address for the memory. It is also advisable to provide a buffer between the data bus of the TMS32010 and that of the memory itself. Although this buffer is probably not necessary for high-speed static memories, it is required for slower devices and large arrays where the drive capacity of the TMS32010 may be marginal.

Figure 6-3 gives an example of one way to extend data memory by using the IN and OUT instructions. The design consists of 16K words of static RAM, addressed by the lower 14 bits of a 16-bit counter. The location to address in this RAM is loaded into the counter by doing an OUT instruction to port 0. This loads the data bus into the counters. The appropriate data memory location is addressed by the lower 14 bits of the data. Bit 15 (MSB) of the data is loaded into the counters to determine whether to count up or down through data memory. Memory can then be read from or written to sequentially by doing an IN or OUT instruction to port 1. The MSB in the counters determines whether the memory address should be incremented (MSB = 0) or decremented (MSB = 1) after a read or write of data memory. Memory will continue to be addressed sequentially until new data is loaded into the counters.

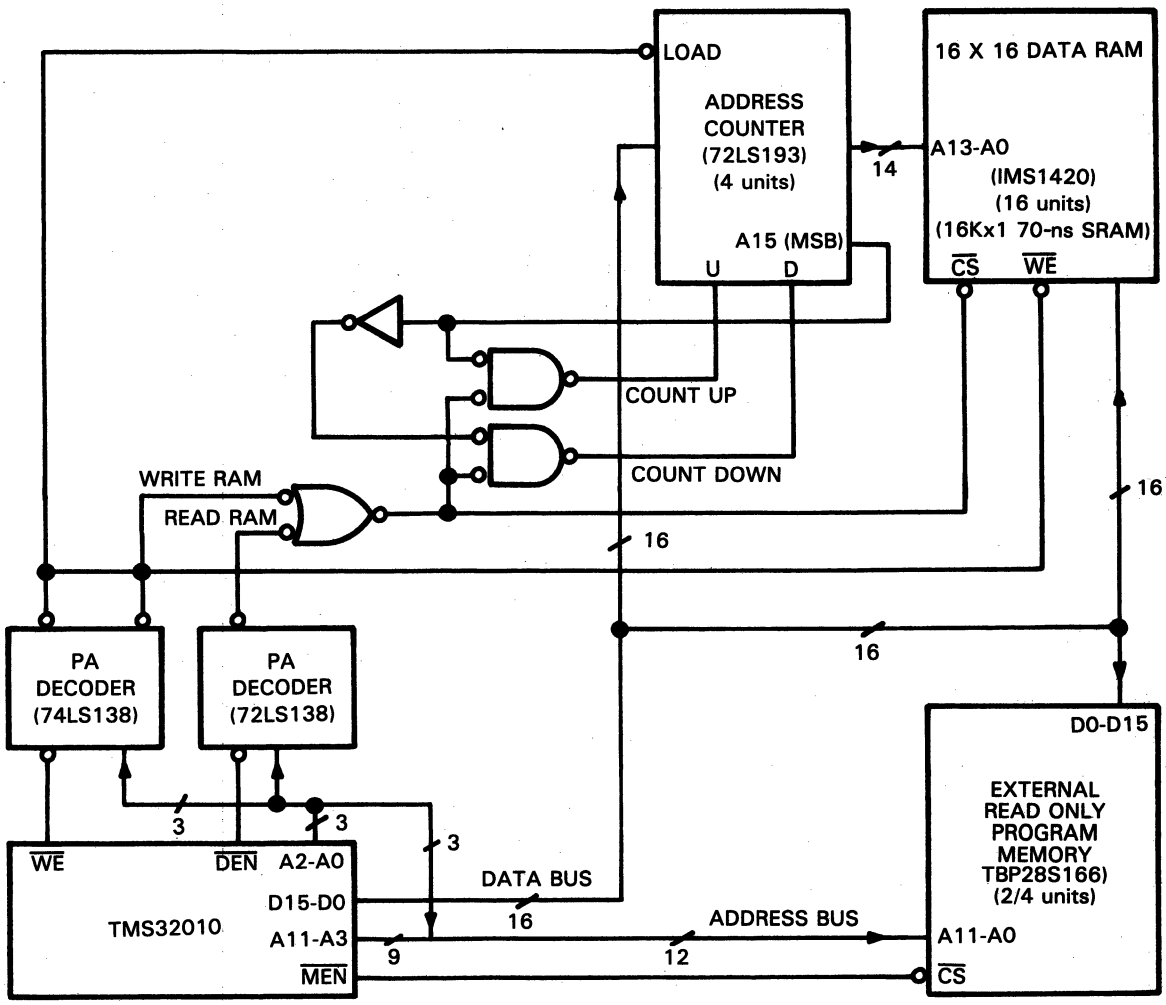


FIGURE 6-3 — TMS32010 EXTENDED MEMORY INTERFACE

Dynamic memories can also be used. However, those devices may impose software constraints on the system designer. For example, memory cycle times may not allow consecutive IN/OUT/IN instruction sequences. Memory refresh represents another problem. Since this processor has no capability to enter a "wait" state, memory refresh must be generated with external hardware.

6.2 INTERRUPTS

An interrupt routine allows the current process to be suspended while an I/O device is being serviced. The processor's execution may be suspended on a high-priority basis by using the $\overline{\text{INT}}$ pin. Otherwise, a lower priority interrupt can be serviced by using a software polling technique.

6.2.1 Software Methods

The BIOZ instruction can be used to poll (or test) the $\overline{\text{BIO}}$ pin to see if a device needs to be serviced. This method allows for a critical loop or set of instructions to be executed without a variation in execution time. Because the test for interrupts occurs at defined points in the program, context saves requirements are minimal.

The $\overline{\text{BIO}}$ pin can be used to monitor the status of a peripheral. If the FIFO full status line is connected to the $\overline{\text{BIO}}$ pin, the FIFO is serviced only when the FIFO is full. In the following example, the FIFO contains 16 data words. The $\overline{\text{BIO}}$ pin is tested after each time-critical function has been executed.

```
        BIOZ    SKIP
        CALL    SERVE
SKIP    .
        .
        .
```

The subroutine does not have to save the registers or the status, because a new procedure will be executed after the device is serviced.

```
SERVE LACK    AR0,15
      LACK    AR1,TABLE
LOOP  LARP    1
      IN      PA0,*+,AR0
      BANZ    LOOP
      RET
```

The FIFO must be serviced before another word is input or data may be lost. This fact determines the frequency at which the polling must take place.

6.2.2 Hardware Methods

The $\overline{\text{INT}}$ pin causes execution to be suspended at any point in the program except after a multiply instruction (see Section 4.1.3.3). The hardware interrupt can be masked at critical points in the program with the DINT instruction. If an interrupt occurs while the INTM (disabled interrupt mask) equals one, the interrupt will not be serviced until the interrupts are enabled again. If an interrupt is pending when an enable interrupt operation occurs, the interrupt is serviced after the execution of the instruction following the EINT command.

When an interrupt is serviced, the INTF (interrupt flag) is cleared, INTM is set to one, the current PC is pushed on the TOS, and the PC is set to 2. The user must save the context of the machine before servicing the peripheral. The context should be restored and the interrupts enabled prior to returning from the interrupt routine. The following paragraphs illustrate a technique for implementing an interrupt-driven analog input channel. It also shows the impact of multiple-level data buffering on system I/O overhead.

Generally, the class of analog systems which can be reasonably supported by the TMS32010 will have information bandwidths of less than 20 kHz. The desired sample rate can be generated by dividing the 5 MHz CLKOUT signal from the TMS32010. It is advisable to provide at least a one-level data buffer to insure the integrity of the data which is read by the processor. If an 8-kHz sample rate is used (for example), the system must then respond to an analog interrupt every 125 ms. The I/O overhead incurred by this arrangement can be computed by determining the number of clock times the TMS32010 will spend in the interrupt routine servicing each sample, and dividing by 625. For example, a typical interrupt routine might look like the following:

```
INT  SST   STATUS     Save status
     SACL  ACCL       Save accumulator low
     SACH  ACCH       Save accumulator high
     IN    SAMP,ADC   Read from ADC
     LAC   COUNT      Update sample counter
     ADD   ONE
     SACL  COUNT
     LACK  LIMIT      Check whether LIMIT clocks
     SUB   COUNT      received
     BGZ   OK
```

```
DONE  LACK  1          YES ===> Set flag
      SACL  FLAG
OK    ZALH  ACCH       Restore accumulator high
      ADDS  ACCL       Restore accumulator low
      LST   STATUS     Restore status
      EINT             Enable subsequent interrupts
      RET
```

The overhead required to service this system is 18/625 = 2.9 percent. This overhead burden can be reduced by using a FIFO to buffer the data. In this case, the TMS32010 need only be interrupted when the buffer has filled. If a 16-level FIFO is used in our example above, this interrupt will occur every 2 ms, and the overhead burden will be reduced to about 0.5 percent.

If two different kinds of devices are being serviced by the same interrupt routine, the $\overline{\text{BIO}}$ pin can be used to determine which device needs to be serviced.

MACRO LANGUAGE INSTRUCTIONS

7. MACRO LANGUAGE EXTENSIONS

The basic instruction set of the TMS32010 has been extended via the XDS/320 Macro Assembler to facilitate coding of commonly used assembly language constructs. In this section, a set of macros designed to ease assembly language coding is described. Some macros call routines from the set of utility routines described in Section 7.5.

7.1 CONVENTIONS USED IN MACRO DESCRIPTIONS

In the macro descriptions, the following conventions are used:

A	A previously defined† memory label
B	Another previously defined† label
A:A + 1	Like A, except refers to a double word
B:B + 1	Like B, except refers to a double word
TMP	A temporary location (previously defined)
AR	Auxiliary register 1 or auxiliary registor 0
@AR	Data RAM location pointed to by the selected auxiliary register
@AR: @AR + 1	Double word, starting at location pointed to by the selected auxiliary register
@AR — 1: @AR	Double word, starting at one before the location pointed to by the selected auxiliary register
AR1	Auxiliary register 1
@AR1	Data RAM location pointed to by AR1
AR0	Auxiliary register 0
@AR0	Data RAM location pointed to by AR0
AC	Accumulator
AC low	Low-order 16 bits of the accumulator
AC high	High-order 16 bits of the accumulator
@AC	Data RAM location pointed to by the accumulator
P	P register
T	T register
ARP	Auxiliary register pointer

*	Indirect operand
* +	Indirect reference and increment
* –	Indirect reference and decrement
[f]	Field f optional (i.e., may be replaced by a null operand)
C	Constant. (It may be written as C{n< C< m} to indicate a range limit between n and m. C1 and C2 will be used as constants when two are required in a description.

† Some macros generate different code sequences for constant operands and memory operands. Memory operands can be confused with constants unless the memory labels (operand names) have been defined to the assembler prior to their use in a macro call. This limitation corresponds to the requirement in some higher-level languages like PASCAL that variables be declared prior to their use in expressions.

7.2 MACRO SET SUMMARY

Table 7-1 lists alphabetically all the macros described in Section 7-3.

TABLE 7-1 – MACRO INDEX

TABLE 7-1 – MACRO INDEX (CONTINUED)

Table 7-2 summarizes all the legal parameters of the macros described in Section 7-3.

TABLE 7-2 – MACRO SET SUMMARY

MACRO INSTRUCTION	OPERAND NUMBER	OPT	OPERAND SIZE†	OPERAND TYPES‡							CONSTANT RANGE	
				C	S	*	*+	*–	AC	AR	LOWEST	HIGHEST
ACTAR	1									X		
	2	X	1		X						temporary	
ADAR	1									X		
	2		1	X	X						–32768	32767
	3	X	1		X						temporary	
ADDX	1		2		X	X	X	X				
ARTAC	1									X		
	2	X	1		X						temporary	
BIC	1		1		X	X	X	X				
	2		1		X	X						
BIS	1		1		X	X	X	X				
	2		1		X	X						
BIT	1		1		X	X	X	X				
	2		1		X	X	X	X				
CMP	1		1		X	X	X	X				
	2		1		X	X	X	X				
CMPX	1		2		X	X	X	X				
	2		2		X	X	X	X				
DEC	1	X	1		X	X			X			
	2	X								X		
DECX	1	X	2		X	X	X	X	X			
INC	1	X	1		X	X			X			
	2	X								X		
INCX	1	X	2		X	X	X	X	X			
LACARY	##		1						X			
	1	X		X							0	15
LASH	1		1		X							
	2		1		X							
	3			X							0	15
LASX	1		2		X							
	2		2		X							
	3			X							0	15
LAXARY	##		2									
LCAC	1		1	X	X						–32768	32767
	2	X		X							0	15
LCACAR	##		1						X			
	1	X		X							0	15
	2	X	1		X						temporary	
LCAR	1									X		
	2		1	X	X						–32768	32767
LCAX	1		2	X							–2**31	2**31-1
LCAXAR	##		2						X			
	1	X	2		X						temporary	
LCP	1		1	X	X						–4096	4095
LCPAC	1		1	X	X						–4096	4095
LDAX	1		2		X	X	X	X				
LTK	1		1	X	X						–32768	32767
MAX	1		1		X							
	2		1		X							
MAXX	1		2		X							
	2		2		X							
MIN	1		1		X							
	2		1		X							
MINX	1		2		X							
	2		2		X							
MOV	1		1		X	X	X	X	X			
	2		1		X	X	X	X	X			
MOVCON	1		?	X								
	2		?		X	X			X			
MOVDAT	1		?		X	X			X			
program →	2		?		X	X		X				
data	3	X		X							–32768	32767

TABLE 7-2 – MACRO SET SUMMARY (Concluded)

MACRO INSTRUCTION	OPERAND NUMBER	OPT	OPERAND SIZE†	OPERAND TYPES‡							CONSTANT RANGE	
				C	S	*	*+	*–	AC	AR	LOWEST	HIGHEST
MOVE	1		?		X	X			X			
data →	2		?		X	X			X			
data	3	X		X							–32768	32767
MOVROM	1		?		X	X			X			
data →	2		?		X	X			X			
program	3	X		X							–32768	32767
MOVX	1		2		X	X	X	X	X			
	2		2		X	X	X	X	X			
NEG	1		1		X	X						
NEGX	1		2		X	X	X	X				
NOT	1	X	1		X	X	X	X	X			
RASH	1		1		X							
	2		1		X							
	3			X							0	15
RASX	1		2		X							
	2		2		X							
	3			X							0	15
REPCON	1			X							–32768	32767
	2		?		X							
	3			X							–32768	32767
RIPPLE	1		?		X							
	2			X							–32768	32767
	3	X									dummy argument	
RLSH	1		1		X							
	2		1		X							
	3			X							0	15
RLSX	1		2		X							
	2		2		X							
	3			X							0	15
SACX	1		2		X	X	X	X				
SAT	1		1		X							
	2		1	X	X						–32768	32767
	3		1	X	X						–32768	32767
SBAR	1									X		
	2		1	X	X						–32768	32767
	3	X	1		X						temporary	
SBIC	1			X							0	15
	2		1		X	X						
SBIS	1			X							0	15
	2		1		X	X						
SBIS	1			X							0	15
	2		1		X	X						
SBIT	1			X							0	15
	2		1		X	X	X	X				
STOX	1		1		X							
	2		2		X							
SUBX	1		2		X	X	X	X				
TST	1		1		X	X	X	X				
TSTX	1		2		X	X	X	X				
XTOS	1		2		X							
	2		1		X							

NOTES:

† Blank in size field means that operand is not a data (program) location, but is a field in an instruction (i.e., has no word size).

‡
- C — Constant
- S — Symbolic address
- *,*+,*– — Indirect through the selected address register (ARP)
- AC — Operand is the AC (usually shown in the instruction as null or blank operand: MOV,A)
- AR — An address register (AR0 or AR1)

ϕ 32-bit constant expressed as a two-word constant list: (C1,C2)

? Variable length operand (length given by argument 3)

Implied operand in accumulator

7.3 MACRO DESCRIPTIONS

Each macro instruction is named, followed by a summary table. A flowchart for clarifying the macro source then follows and specific examples of all legal forms.

The macros described in this section use a number of assembler symbols for internal purposes during macro expansion. Most of these internal symbols and any operands the user supplies to the macros are entered into the assembler symbol table as undefined (unless they are user-defined already) and will be printed at the end of the assembler printed output as undefined. This is not an error. Only undefined symbol errors flagged under assembly language statements in the program listing are actual fatal errors. Only these errors will be tallied in the assembly error count. Undefined symbols listed after the program are for information only.

TITLE: Move Accumulator to Auxiliary Register

NAME: ACTAR

OBJECTIVE: Pass data word to named auxiliary register from accumulator

ALGORITHM: (ACC) → temp (XR0)
(temp) → AR

CALLING SEQUENCE: ACTAR AR [,TEMP]

ENTRY CONDITIONS: AR = 0,1; 0 ⩽ TEMP ⩽ 127

EXIT CONDITIONS: Accumulator stored in auxiliary register;
ARP now points to auxiliary register specified

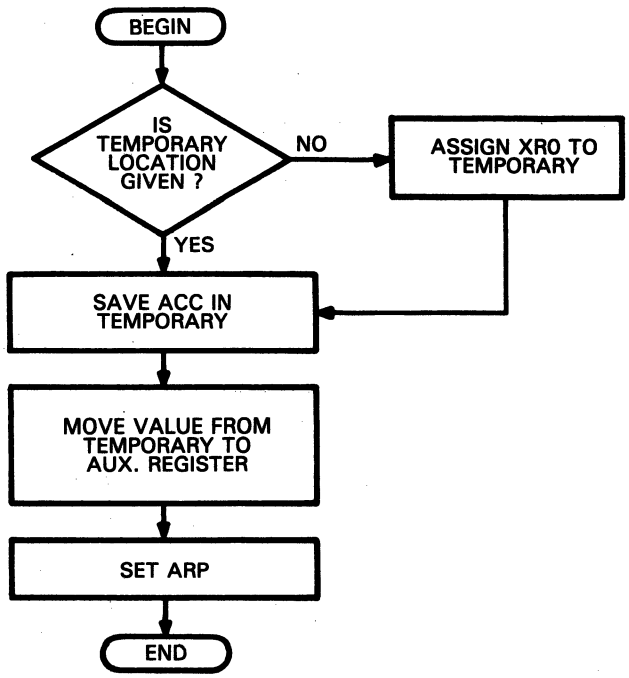
PROGRAM MEMORY REQUIRED: 3 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 3 cycles

FLOWCHART: ACTAR



SOURCE:

```
*MOVE AC TO AR
*
ACTAR  $MACRO  A,T
       $IF  T.L=0          ASSIGN XR0 AS TEMP
       $ASG 'XR0' TO T.S
       $ENDIF
       SACL :T:,0          STORE AC TO :T:
       LAR  :A:,:T:        RE-LOAD :A:
       LARP :A:            LOAD AR POINTER
       $END
```

EXAMPLE 1:

```
0013                  ACTAR  AR0
0001 0009 5004"        SACL XR0,0        STORE AC TO XR0
0002 000A 3804"        LAR  AR0,XR0      RE-LOAD AR0
0003 000B 6880         LARP AR0          LOAD AR POINTER
```

EXAMPLE 2:

```
0015                  ACTAR  0,C
0001 000C 5000"        SACL C,0          STORE AC TO C
0002 000D 3800"        LAR  0,C          RE-LOAD 0
0003 000E 6880         LARP 0            LOAD AR POINTER
```

TITLE: Add Variable to Auxiliary Register

NAME: ADAR

OBJECTIVE: Add data word to named auxiliary register

ALGORITHM: (AR) + (dma) → ACC
(ACC) → AR

CALLING SEQUENCE: ADAR AR, B [,TEMP]

ENTRY CONDITIONS: AR = 0,1; 0 ⩽ B ⩽ 127; 0 ⩽ TEMP ⩽ 127

EXIT CONDITIONS: Sum of memory location and auxiliary register is stored in named auxiliary register

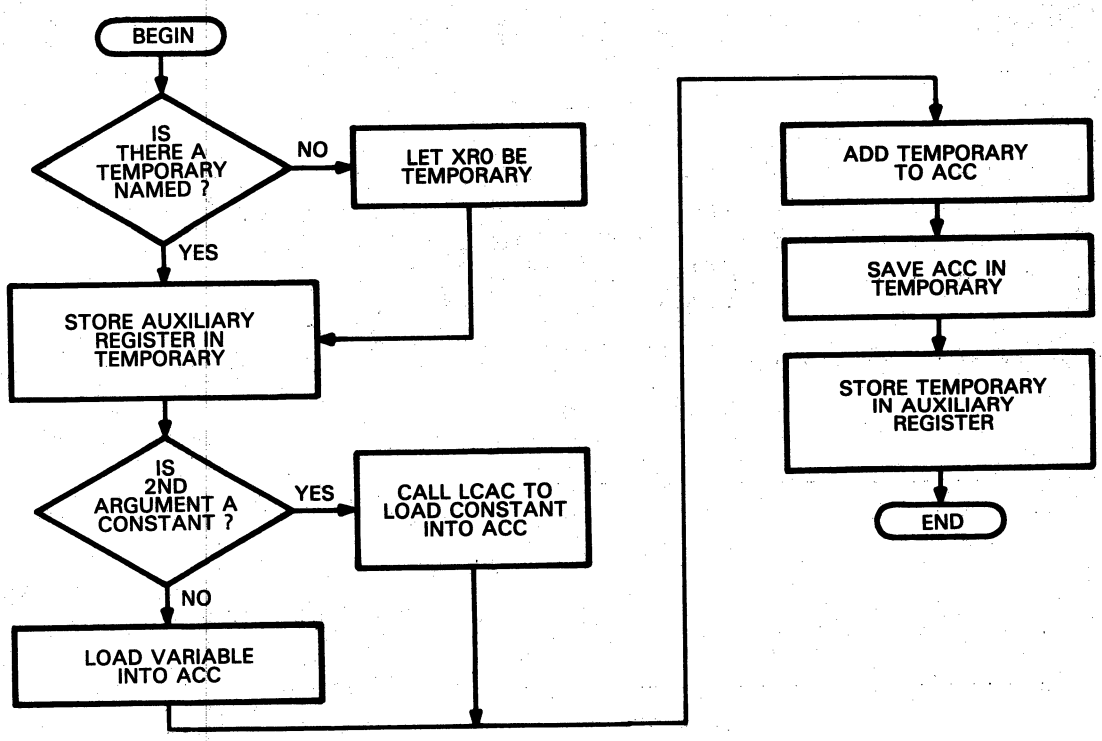
PROGRAM MEMORY REQUIRED: 5 – 7 words (plus LDAC$ routine)

DATA MEMORY REQUIRED: 2 words

STACK REQUIRED: 0 – 2 levels

EXECUTION TIME: 5 – 17 cycles

FLOWCHART: ADAR



SOURCE:

```
*ADD TO AR
*
ADAR  $MACRO  A,B,T
      $IF  T.L=0        USE XR1 AS TEMP
      $ASG 'XR1' TO T.S
      $ENDIF
      SAR   :A:,:T:     STORE :A:
      $IF  B.SA&$UNDF
      LCAC :B:          LOAD CONST :B: INTO AC
      $ELSE
      LAC :B:,0         LOAD VAR :B: INTO AC
      $ENDIF
      ADD   :T:,0       ADD TEMP :T: TO AC
      SACL :T:,0        STORE :T:
      LAR   :A:,:T:     LOAD BACK INTO :A:
      $END
```

EXAMPLE 1:

```
0007                  ADAR   A,3
0001 0006 3103"        SAR  A,XR1        STORE A
0002                   LCAC 3            LOAD CONSTANT 3 INTO AC
0001      0003  V$1 EQU 3
0002 0007 7E03          LACK V$1         LOAD AC WITH V$1
0003 0008 0003"        ADD  XR1,0        ADD TEMP XR1 TO AC
0004 0009 5003"        SACL XR1,0        STORE XR1
0005 000A 3903"        LAR  A,XR1        LOAD BACK INTO A
```

EXAMPLE 2:

```
0009                  ADAR   AR0,C,B
0001 000B 3008         SAR  AR0,B        STORE AR0
0002 000C 2004"        LAC C,0           LOAD VARIABLE C INTO AC
0003 000D 0008         ADD  B,0          ADD TEMP B TO AC
0004 000E 5008         SACL B,0          STORE B
0005 000F 3808         LAR  AR0,B        LOAD BACK INTO AR0
```

EXAMPLE 3:

```
0011                  ADAR   0,D
0001 0010 3003"        SAR  0,XR1        STORE 0
0002 0011 2005"        LAC D,0           LOAD VARIABLE D INTO AC
0003 0012 0003"        ADD  XR1,0        ADD TEMP XR1 TO AC
0004 0013 5003"        SACL XR1,0        STORE XR1
0005 0014 3803"        LAR  0,XR1        LOAD BACK INTO 0
```

TITLE: Double-Word Add

NAME: ADDX

OBJECTIVE: Add double word to accumulator

ALGORITHM:

ADDX *	– causes→	(ACC) + (@AR:@AR + 1) → ACC
ADDX * –	– causes→	(ACC) + (@AR – 1:@AR) → ACC (AR) – 2 → AR
ADDX * +	– causes→	(ACC) + (@AR:@AR + 1) → ACC (AR) + 2 → AR
ADDX A	– causes→	(ACC) + (A:A + 1) → ACC

CALLING SEQUENCE: ADDX {A,*,* – ,* + }

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127$

EXIT CONDITIONS: Accumulator contains updated value after addition; auxiliary register is updated if necessary

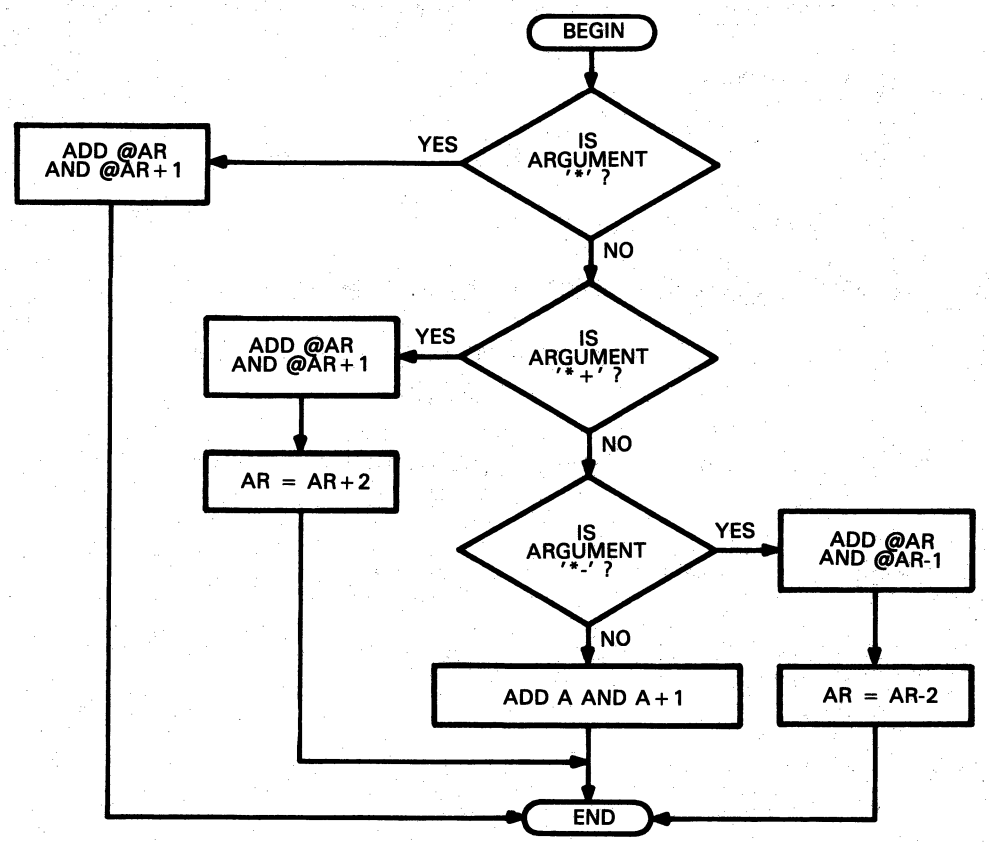
PROGRAM MEMORY REQUIRED: 2 words

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 2 cycles

FLOWCHART: ADDX



SOURCE:

```
*ADD DOUBLE PRECISION
*
ADDX    $MACRO  A          ADD DOUBLE PRECISION
        $VAR ST,SP,SM
        $ASG '*+' TO SP.S
        $ASG '*-' TO SM.S
        $ASG '*' TO ST.S
        $IF A.SV=ST.SV
        ADDH *+            ADD HIGH
        ADDS *-            ADD LOW '*'
        $ELSE
        $IF A.SV=SP.SV
        ADDH *+            ADD HIGH
        ADDS *+            ADD LOW '*+'
        $ELSE
        $IF A.SV=SM.SV
        ADDS *-            ADD LOW
        ADDH *-            ADD HIGH '*-'
        $ELSE
        ADDH :A:           ADD :A: HIGH
        ADDS :A:+1         ADD :A: LOW
        $ENDIF
        $ENDIF
        $ENDIF
        $END
```

EXAMPLE 1:

```
0011                    ADDX A
0001 0006 6007           ADDH A          ADD A HIGH
0002 0007 6108           ADDS A+1        ADD A LOW
```

EXAMPLE 2:

```
0013                    ADDX *
0001 0008 60A8           ADDH *+         ADD HIGH
0002 0009 6198           ADDS *-         ADD LOW '*'
```

EXAMPLE 3:

```
0015                    ADDX *-
0001 000A 6198           ADDS *-         ADD LOW
0002 000B 6098           ADDH *-         ADD HIGH '*-'
```

EXAMPLE 4:

```
0017                    ADDX *+
0001 000C 60A8           ADDH *+         ADD HIGH
0002 000D 61A8           ADDS *+         ADD LOW '*+'
```

TITLE: Move Auxiliary Register to Accumulator

NAME: ARTAC

OBJECTIVE: Load data from auxiliary register into accumulator

ALGORITHM: (AR) → temp
(temp) → ACC

CALLING SEQUENCE: ARTAC AR [,TEMP]

ENTRY CONDITIONS: AR = 0,1; 0 ⩽ TEMP ⩽ 127

EXIT CONDITIONS: Accumulator contains same value as auxiliary register

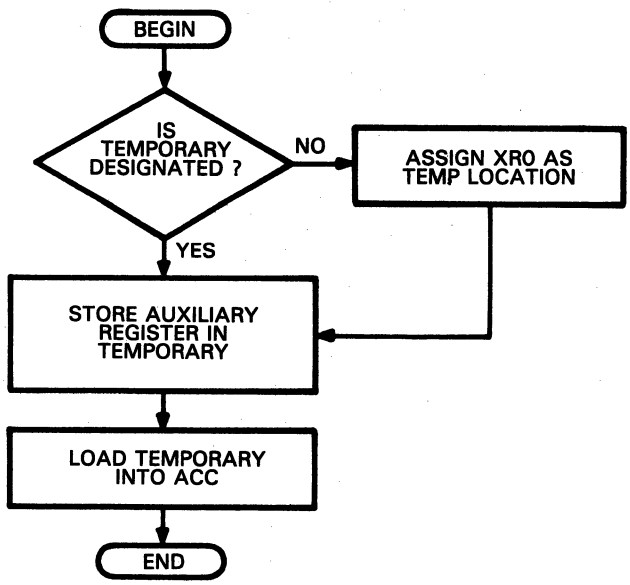
PROGRAM MEMORY REQUIRED: 2 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 2 cycles

FLOWCHART: ARTAC



SOURCE:

```
*COPY AR TO AC
*
ARTAC  $MACRO  A,T
       $IF  T.L=0          USE XR0 AS TEMP
       $ASG 'XR0' TO T.S
       $ENDIF
       SAR   :A:,:T:       SAVE :A:
       LAC   :T:,0         LOAD INTO AC
       $END
```

EXAMPLE 1:

```
0013                    ARTAC  AR0
0001 0008 3004"          SAR  AR0,XR0       SAVE AR0
0002 0009 2004"          LAC  XR0,0         LOAD INTO AC
```

EXAMPLE 2:

```
0014              ***
0015                    ARTAC  0,C
0001 000A 3000"          SAR  0,C           SAVE 0
0002 000B 2000"          LAC  C,0           LOAD INTO AC
```

TITLE: Clear Bits in Data Word

NAME: BIC

OBJECTIVE: Clear bits in data word specified by one bit in mask

ALGORITHM: (data) .AND. .NOT. (mask) → data

CALLING SEQUENCE: BIC mask,data

ENTRY CONDITIONS: 0 ⩽ mask ⩽ 127; 0 ⩽ data ⩽ 127

EXIT CONDITIONS: Data word contains initial value with specified bits cleared

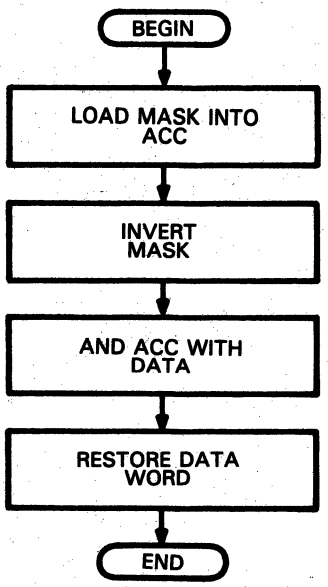
PROGRAM MEMORY REQUIRED: 4 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 4 cycles

FLOWCHART: BIC



SOURCE:

```
*BIT CLEAR - CLEAR BITS IN B WHERE A HAS ZEROS
*
BIC     $MACRO  A,B         BIT CLEAR
        LAC   :A:,0         LOAD :A:
```

```
        XOR  MINUS        INVERT MASK
        AND  :B:          AND :B:
        SACL :B:,0        SAVE RESULT IN :B:
        $END
```

EXAMPLE 1:

```
0014                  BIC   B,A
0001 000A 2008         LAC  B,0          LOAD B
0002 000B 7803"        XOR  MINUS        INVERT MASK
0003 000C 7901         AND  A            AND A
0004 000D 5001         SACL A,0          SAVE RESULT IN A
```

EXAMPLE 2:

```
0016                  BIC   D,C
0001 000E 2001"        LAC  D,0          LOAD D
0002 000F 7803"        XOR  MINUS        INVERT MASK
0003 0010 7900"        AND  C            AND C
0004 0011 5000"        SACL C,0          SAVE RESULT IN C
```

EXAMPLE 3:

```
0018                  BIC   D,A
0001 0012 2001"        LAC  D,0          LOAD D
0002 0013 7803"        XOR  MINUS        INVERT MASK
0003 0014 7901         AND  A            AND A
0004 0015 5001         SACL A,0          SAVE RESULT IN A
```

TITLE: Set Bits in Data Word

NAME: BIS

OBJECTIVE: Set bits in data word specified by one bit in mask

ALGORITHM: (data) .OR. (mask) → data

CALLING SEQUENCE: BIS mask,data

ENTRY CONDITIONS: 0 ≤ mask ≤ 127; 0 ≤ data ≤ 127

EXIT CONDITIONS: Data word contains initial value with specified bits set

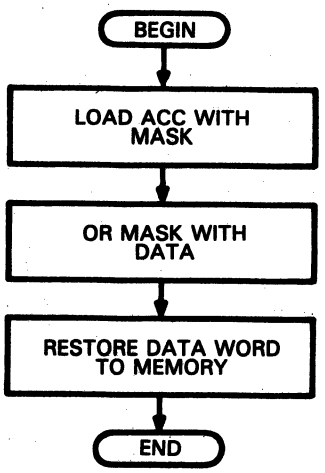
PROGRAM MEMORY REQUIRED: 3 words

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 3 cycles

FLOWCHART: BIS



SOURCE:

```
*SET BITS IN B CORRESPONDING TO ONES IN A
*
BIS     $MACRO   A,B       BIT SET
        LAC   :A:,0        LOAD :A:
        OR    :B:          OR WITH :B:
        SACL  :B:,0        SAVE BACK TO :A:
        $END
```

EXAMPLE 1:

```
0014                    BIS   B,A
0001 000A 2008           LAC  B,0          LOAD B
0002 000B 7A01           OR   A            OR WITH A
0003 000C 5001           SACL A,0          SAVE BACK TO B
```

EXAMPLE 2:

```
0016                    BIS   D,C
0001 000D 2001"          LAC  D,0          LOAD D
0002 000E 7A00"          OR   C            OR WITH C
0003 000F 5000"          SACL C,0          SAVE BACK TO D
```

TITLE: Test Bits in Data Word

NAME: BIT

OBJECTIVE: Test bits in data word specified by one bit in mask

ALGORITHM: (data) .AND. (mask) → ACC

CALLING SEQUENCE: BIT mask,data

ENTRY CONDITIONS: $0 \leq$ mask ≤ 127; $0 \leq$ data ≤ 127

EXIT CONDITIONS: ACC contains zero if no bits of mask are set in data word: any bits masked that are set in data word will be set in ACC

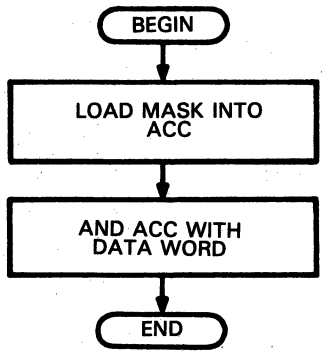
PROGRAM MEMORY REQUIRED: 2 words

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 2 cycles

FLOWCHART: BIT



SOURCE:

```
*BIT TEST - BITS IN B TESTED BY MASK IN A
*
BIT     $MACRO  A,B         BIT TEST
        LAC   :A:,0         LOAD :A:, MASK
        AND   :B:           AND WITH :B:
        $END
```

EXAMPLE:

```
0014                    BIT   B,A
0001 000A 2008           LAC  B,0          LOAD B, MASK
0002 000B 7901           AND  A            AND WITH A
```

TITLE: Compare Two Words

NAME: CMP

OBJECTIVE: Load word into accumulator; then subtract the other word, allowing comparison

ALGORITHM: CMPX A,B – causes→ (A) – (B) → ACC

CALLING SEQUENCE: CMP {A,*,* –,* +},{B,*,* –,* +}

ENTRY CONDITIONS: 0 ⩽ A ⩽ 127; 0 ⩽ B ⩽ 127

EXIT CONDITIONS: Accumulator contains value of second word subtracted from the first word; auxiliary register is updated if necessary

PROGRAM MEMORY REQUIRED: 2 words

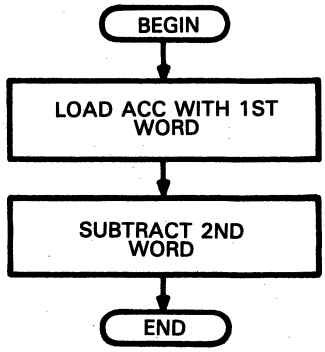
DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 2 cycles

7

FLOWCHART: CMP



SOURCE:

```
*COMPARE A TO B
*
CMP     $MACRO   A,B       COMPARE
        LAC   :A:,0        LOAD :A:
        SUB   :B:,0        SUBTRACT :B:
        $END
```

EXAMPLE 1:

```
0007                    CMP    A,B
0001 0006 2001           LAC   A,0          LOAD A
0002 0007 1008           SUB   B,0          SUBTRACT B
```

EXAMPLE 2:

```
0009                    CMP    *,B
0001 0008 2088           LAC   *,0          LOAD *
0002 0009 1008           SUB   B,0          SUBTRACT B
```

EXAMPLE 3:

```
0011                    CMP    C,*+
0001 000A 2004"          LAC   C,0          LOAD C
0002 000B 10A8           SUB   *+,0         SUBTRACT *+
```

EXAMPLE 4:

```
0013                    CMP    *,*
0001 000C 2088           LAC   *,0          LOAD *
0002 000D 1088           SUB   *,0          SUBTRACT *
```

TITLE: Compare Two Double Words

NAME: CMPX

OBJECTIVE: Load double word into accumulator; then subtract the other double word, allowing comparison

ALGORITHM: CMPX A,B – causes→ (A:A + 1) – (B:B + 1) → ACC

CALLING SEQUENCE: CMPX {A,*,* – ,* + },{B,*,* – ,* + }

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127; 0 \leqslant B \leqslant 127$

EXIT CONDITIONS: Accumulator contains value of second double word subtracted from the first double word; auxiliary register is updated if necessary.

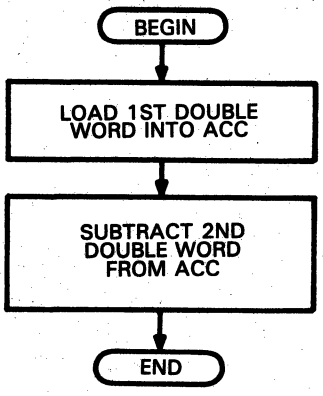
PROGRAM MEMORY REQUIRED: 4 words

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 4 cycles

FLOWCHART: CMPX



SOURCE:

```
*COMPARE A TO B, DOUBLE
*
CMPX    $MACRO  A,B         COMPARE DOUBLE
        LDAX :A:            LOAD DOUBLE :A:
        SUBX :B:            SUBTRACT DOUBLE :B:
        $END
```

EXAMPLE 1:

```
0011                    CMPX A,B
0001                     LDAX A          LOAD DOUBLE A
0001 0006 6507            ZALH A        LOAD HIGH A
0002 0007 6108            ADDS A+1      LOAD LOW A
0002                     SUBX B          SUBTRACT DOUBLE B
0001 0008 6209            SUBH B        SUBTRACT HIGH
0002 0009 630A            SUBS B+1      SUBTRACT LOW
```

EXAMPLE 2:

```
0013                    CMPX C,*
0001                     LDAX C          LOAD DOUBLE C
0001 000A 6500"           ZALH C        LOAD HIGH C
0002 000B 6101"           ADDS C+1      LOAD LOW C
0002                     SUBX *          SUBTRACT DOUBLE *
0001 000C 62A8            SUBH *+       SUBTRACT HIGH
0002 000D 6398            SUBS *-       SUBTRACT LOW
```

EXAMPLE 3:

```
0015                    CMPX *-,D
0001                     LDAX *-         LOAD DOUBLE *-
0001 000E 6698            ZALS *-       LOAD LOW
0002 000F 6098            ADDH *-       LOAD HIGH '*-'
0002                     SUBX D          SUBTRACT DOUBLE D
0001 0010 6202"           SUBH D        SUBTRACT HIGH
0002 0011 6303"           SUBS D+1      SUBTRACT LOW
```

EXAMPLE 4:

```
0017                    CMPX *+,*+
0001                     LDAX *+         LOAD DOUBLE *+
0001 0012 65A8            ZALH *+       LOAD HIGH
0002 0013 61A8            ADDS *+       LOAD LOW '*+'
0002                     SUBX *+         SUBTRACT DOUBLE *+
0001 0014 62A8            SUBH *+       SUBTRACT HIGH
0002 0015 63A8            SUBS *+       SUBTRACT LOW
```

EXAMPLE 5:

```
0019                    CMPX *-,*-
0001                     LDAX *-         LOAD DOUBLE *-
0001 0016 6698            ZALS *-       LOAD LOW
0002 0017 6098            ADDH *-       LOAD HIGH '*-'
0002                     SUBX *-         SUBTRACT DOUBLE *-
0001 0018 6398            SUBS *-       SUBTRACT LOW
0002 0019 6298            SUBH *-       SUBTRACT HIGH
```

TITLE: Decrement Word

NAME: DEC

OBJECTIVE: Decrement word or accumulator

ALGORITHM: DEC – causes→ (ACC) – 1 → ACC

DEC A – causes→ (A) – 1 → (A)

DEC ,AR – causes→ (AR) – 1 → AR

CALLING SEQUENCE: DEC [A][,AR]

ENTRY CONDITIONS: $0 \leq A \leq 127$; AR = 0,1

EXIT CONDITIONS: Specified word or auxiliary register is decremented; auxiliary register pointer will point to specified auxiliary register

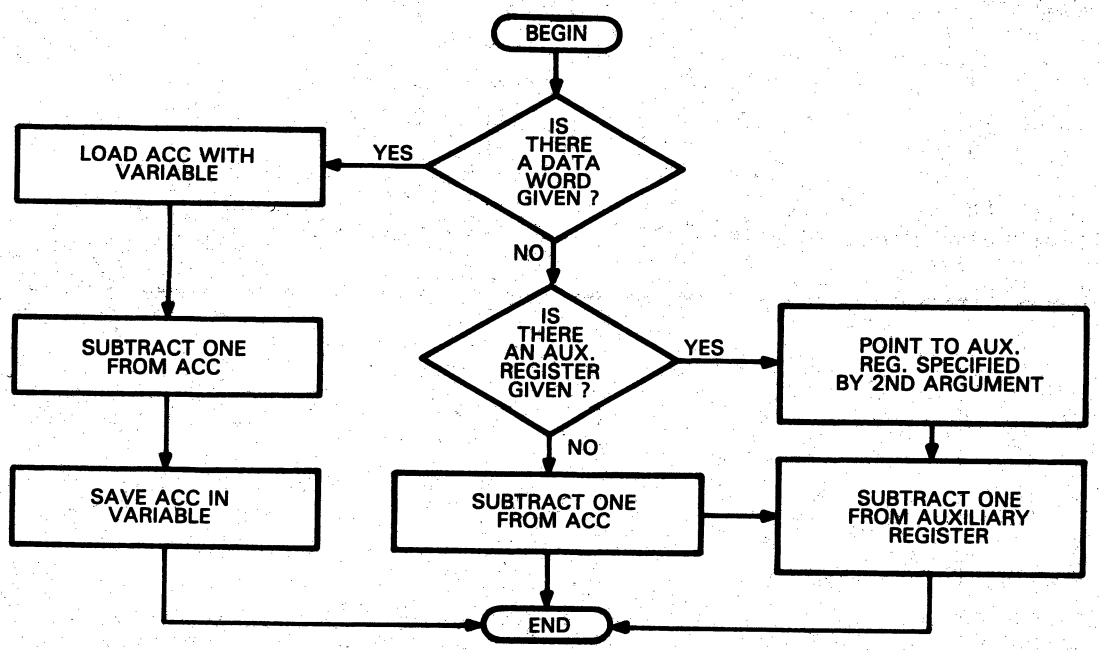
PROGRAM MEMORY REQUIRED: 1 – 3 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 1 – 3 cycles

FLOWCHART: DEC



SOURCE:

```
*DECREMENT THE ACCUMULATOR, AN AUXILIARY
*REGISTER, OR MEMORY
*
DEC     $MACRO A,B          DECREMENT
        $IF  A.L=0
        $IF  B.L=0
        SUB  ONE,0          DECREMENT AC
        $ELSE
        LARP :B:            LOAD ARP WITH :B:
        MAR  *-             DECREMENT
        $ENDIF
        $ELSE
        LAC  :A:,0          LOAD :A:
        SUB  ONE,0          DECREMENT
        SACL :A:,0          SAVE :A:
        $ENDIF
        $END
```

EXAMPLE 1:

```
0007                    DEC    A
0001 0006 2001           LAC  A,0          LOAD A
0002 0007 1000"          SUB  ONE,0        DECREMENT
0003 0008 5001           SACL A,0          SAVE A
```

EXAMPLE 2:

```
0009                    DEC    ,A
0001 0009 6881           LARP A            LOAD ARP WITH A
0002 000A 6898           MAR  *-           DECREMENT
```

EXAMPLE 3:

```
0011                    DEC
0001 000B 1000"          SUB  ONE,0        DECREMENT THE ACCUMULATOR
```

EXAMPLE 4:

```
0015                    DEC    ,AR0
0001 000F 6880           LARP AR0          LOAD ARP WITH AR0
0002 0010 6898           MAR  *-           DECREMENT
```

TITLE: Double-Word Decrement

NAME: DECX

OBJECTIVE: Decrement double word or accumulator

ALGORITHM:

DECX *	– causes→	(@AR:@AR + 1) – 1 → @AR:@AR + 1
DECX * –	– causes→	(@AR – 1:@AR) – 1 → @AR – 1:@AR (AR) – 2 → AR
DECX * +	– causes→	(@AR:AR:@AR + 1) – 1 → @AR:@AR + 1 (AR) + 2 → AR
DECX A	– causes→	(A:A + 1) – 1 → A:A + 1
DECX	– causes→	(ACC) – 1 → ACC

CALLING SEQUENCE: DECX [A,*,* – ,* +]

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127$

EXIT CONDITIONS: Specified double word is decremented; auxiliary register is updated as necessary

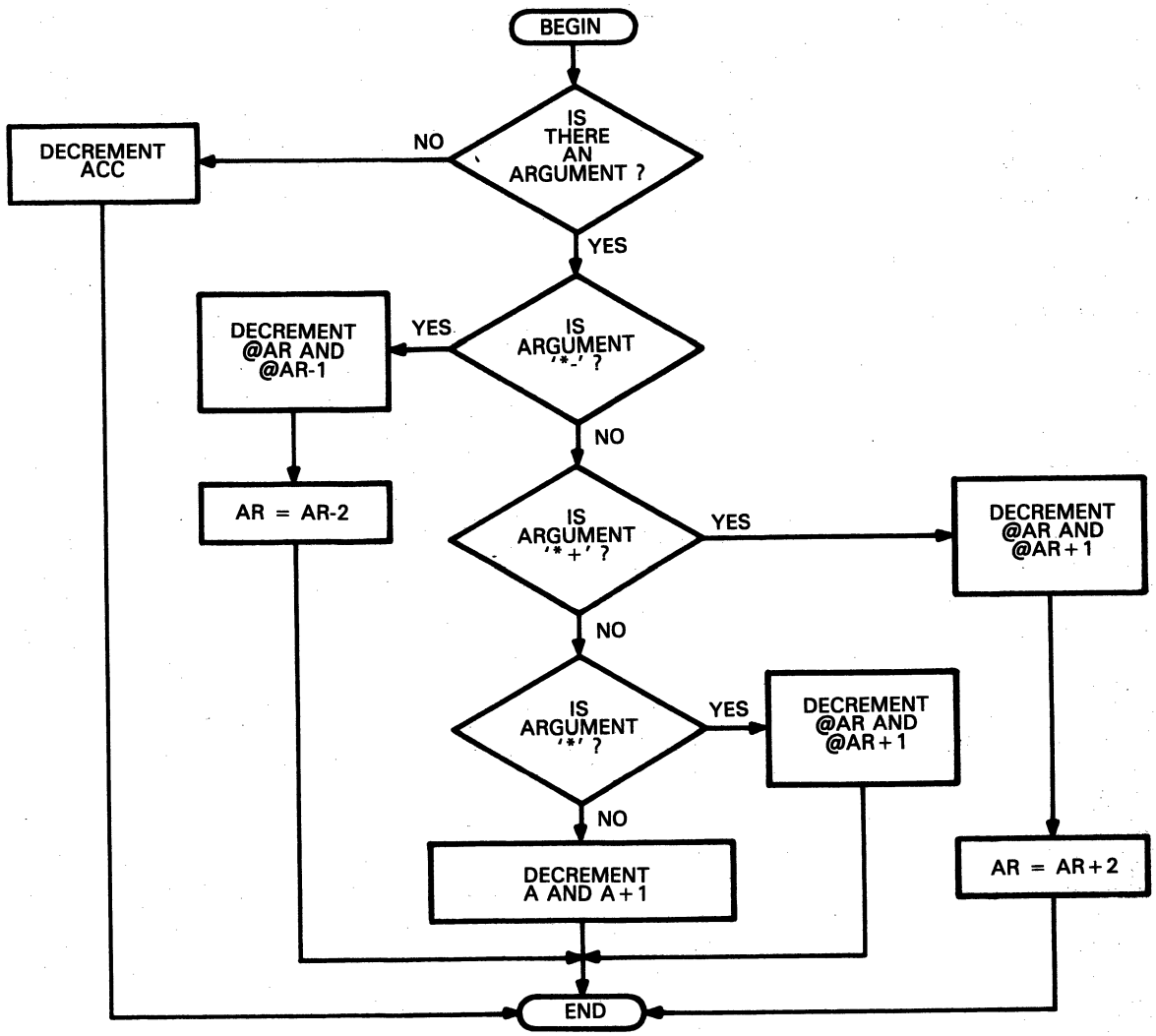
PROGRAM MEMORY REQUIRED: 1 – 5 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 1 – 5 cycles

FLOWCHART: DECX



SOURCE:

```
*DECREMENT DOUBLE
*
DECX    $MACRO A            DECREMENT DOUBLE
        $VAR ST,SP,SM
        $ASG '*+' TO SP.S
        $ASG '*-' TO SM.S
        $ASG '*' TO ST.S
        $IF  A.L=0
        SUB  ONE,0          DECREMENT AC
        $ELSE
        $IF A.SV=SM.SV
        ZALS *-
        ADDH *+             LOAD '*-'
        SUB  ONE,0          DECREMENT
        SACX *-             SAVE '*-'
        $ELSE
        $IF A.SV=SP.SV
        LDAX *              LOAD '*'
        SUB  ONE,0          DECREMENT
        SACX *+             SAVE '*+'
```

```
        $ELSE
        $IF A.SV=ST.SV
        LDAX *              LOAD '*'
        SUB  ONE,0          DECREMENT
        SACX *              SAVE '*'
        $ELSE
        LDAX :A:            LOAD :A:
        SUB  ONE,0          DECREMENT
        SACX :A:            SAVE :A:
        $ENDIF
        $END
```

EXAMPLE 1:

```
0011                    DECX A
0001                     LDAX A             LOAD A
0001 0006 6507           ZALH A             LOAD HIGH A
0002 0007 6108           ADDS A+1           LOAD LOW A
0002 0008 1004"         SUB  ONE,0         DECREMENT
0003                    SACX A             SAVE A
0001 0009 5807           SACH A,0           STORE HIGH
0002 000A 5008           SACL A+1,0         STORE LOW
```

EXAMPLE 2:

```
0013                    DECX *
0001                     LDAX *             LOAD '*'
0001 000B 65A8           ZALH *+            LOAD HIGH
0002 000C 6198           ADDS *-            LOAD LOW '*'
0002 000D 1004"         SUB  ONE,0         DECREMENT
0003                    SACX *             SAVE '*'
0001 000E 58A8           SACH *+,0          STORE HIGH
0002 000F 5098           SACL *-,0          STORE LOW
```

EXAMPLE 3:

```
0015                    DECX *-
0001 0010 6698           ZALS *-
0002 0011 60A8           ADDH *+            LOAD '*-'
0003 0012 1004"         SUB  ONE,0         DECREMENT
0004                    SACX *-            SAVE '*-'
0001 0013 5098           SACL *-,0          STORE LOW
0002 0014 5898           SACH *-,0          STORE HIGH
```

EXAMPLE 4:

```
0017                    DECX *+
0001                     LDAX *             LOAD '*'
0001 0015 65A8           ZALH *+            LOAD HIGH
0002 0016 6198           ADDS *-            LOAD LOW '*'
0002 0017 1004"         SUB  ONE,0         DECREMENT
0003                    SACX *+            SAVE '*+'
0001 0018 58A8           SACH *+,0          STORE HIGH
0002 0019 50A8           SACL *+,0          STORE LOW
```

EXAMPLE 5:

```
0019                    DECX
0001 001A 1004"          SUB  ONE,0         DECREMENT AC
```

TITLE: Increment Word

NAME: INC

OBJECTIVE: Increment word or accumulator

ALGORITHM: INC – causes→ (ACC) + 1 → ACC

INC A – causes→ (A) + 1 → (A)

INC ,AR – causes→ (AR) + 1 → AR

CALLING SEQUENCE: INC [A][,AR]

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127$; AR = 0,1

EXIT CONDITIONS: Specified word or auxiliary register is incremented; auxiliary register pointer specifies the named auxiliary register

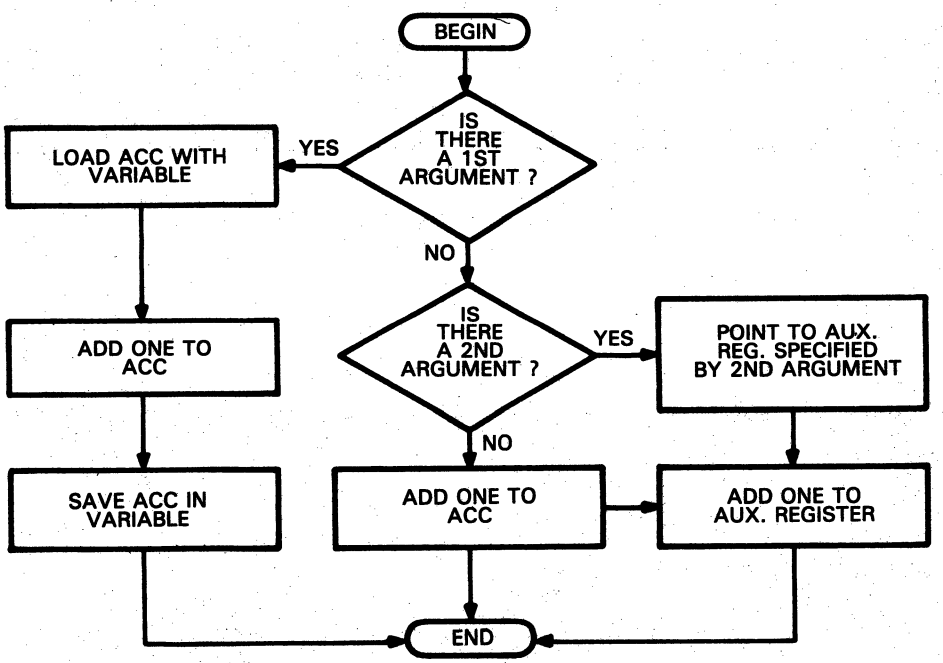
PROGRAM MEMORY REQUIRED: 1 – 3 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 1 – 3 cycle

FLOWCHART: INC



SOURCE:

```
*INCREMENT AC, AR, OR MEM
*
INC     $MACRO  A,B         INCREMENT
        $IF  A.L=0
        $IF  B.L=0
        ADD  ONE,0          INCREMENT AC
        $ELSE
        LARP :B:            LOAD ARP WITH :B:
        MAR  *+             INCREMENT
        $ENDIF
        $ELSE
        LAC  :A:,0          LOAD :A:
        ADD  ONE,0          INCREMENT
        SACL :A:,0          SAVE :A:
        $ENDIF
        $END
```

EXAMPLE 1:

```
 0007                   INC    A
 0001 0006 2001          LAC  A,0          LOAD A
 0002 0007 0000"         ADD  ONE,0        INCREMENT
 0003 0008 5001          SACL A,0          SAVE A
```

EXAMPLE 2:

```
 0009                   INC    ,AR1
 0001 0009 6881          LARP AR1          LOAD ARP WITH AR1
 0002 000A 68A8          MAR  *+           INCREMENT
```

EXAMPLE 3:

```
 0011                   INC
 0001 000B 0000"         ADD  ONE,0        INCREMENT
```

EXAMPLE 4:

```
 0015                   INC    ,AR0
 0001 000F 6880          LARP AR0          LOAD ARP WITH AR0
 0002 0010 68A8          MAR  *+           INCREMENT
```

TITLE: Double-Word Increment

NAME: INCX

OBJECTIVE: Increment double word or accumulator

ALGORITHM:

INCX *	– causes→	(@AR:@AR + 1) + 1 → @AR:@AR + 1
INCX * –	– causes→	(@AR – 1:@AR) + 1 → @AR – 1: @A (AR) – 2 → AR
INCX * +	– causes→	(@AR:@ AR + 1) + 1 → @AR:@AR + 1 (AR) + 2 → AR
INCX A	– causes→	(A:A + 1) + 1 → A:A + 1
INCX	– causes→	(ACC) + 1 → ACC

CALLING SEQUENCE: INCX [A,*,* – ,* +]

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127$

EXIT CONDITIONS: Specified double word is incremented; auxiliary register is updated as necessary

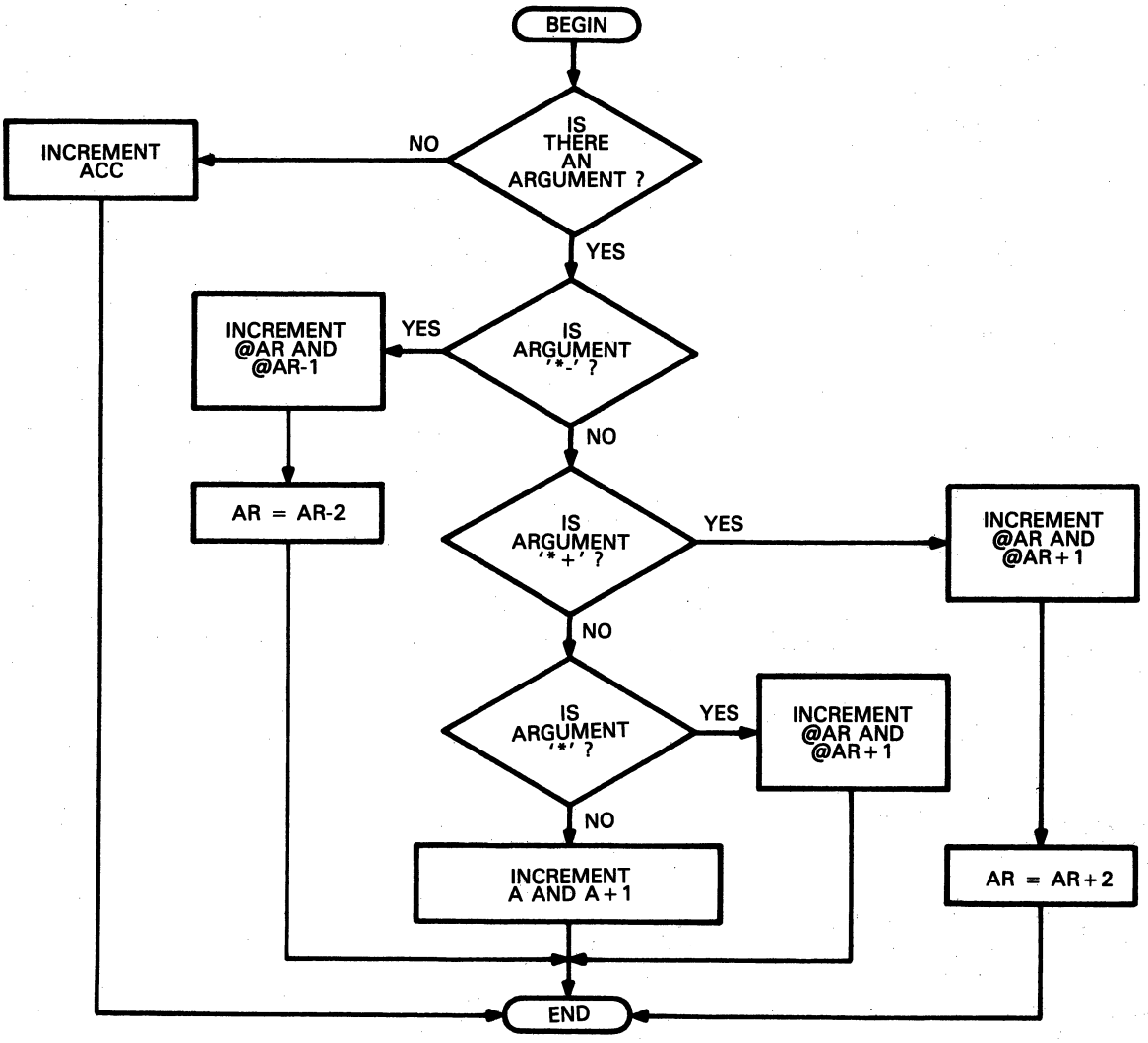
PROGRAM MEMORY REQUIRED: 1 – 5 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 1 – 5 cycles

FLOWCHART: INCX



SOURCE:

```
*INCREMENT DOUBLE
*
INCX    $MACRO  A           INCREMENT DOUBLE
        $VAR ST,SP,SM
        $ASG '*+' TO SP.S
        $ASG '*-' TO SM.S
        $ASG '*' TO ST.S
        $IF  A.L=0
        ADD  ONE,0          INCREMENT AC
        $ELSE
        $IF A.SV=SM.SV
        ZALS *-
        ADDH *+             LOAD '*-'
        ADD  ONE,0          INCREMENT
        SACX *-             SAVE '*-'
        $ELSE
        $IF A.SV=SP.SV
        LDAX *              LOAD '*'
        ADD  ONE,0          INCREMENT
        SACX *+             SAVE '*+'
```

```
        $ELSE
        $IF A.SV=ST.SV
        LDAX *              LOAD '*'
        ADD  ONE,0          INCREMENT
        SACX *              SAVE '*'
        $ELSE
        LDAX :A:            LOAD :A:
        ADD  ONE,0          INCREMENT
        SACX :A:            SAVE :A:
        $ENDIF
        $END
```

EXAMPLE 1:

```
0011                    INCX A
0001                     LDAX A             LOAD A
0001 0006 6507           ZALH A             LOAD HIGH A
0002 0007 6108           ADDS A+1           LOAD LOW A
0002 0008 0004"         ADD  ONE,0         INCREMENT
0003                     SACX A             SAVE A
0001 0009 5807           SACH A,0           STORE HIGH
0002 000A 5008           SACL A+1,0         STORE LOW
```

EXAMPLE 2:

```
0013                    INCX *
0001                     LDAX *             LOAD '*'
0001 000B 65A8           ZALH *+            LOAD HIGH
0002 000C 6198           ADDS *-            LOAD LOW '*'
0002 000D 0004"         ADD  ONE,0         INCREMENT
0003                     SACX *             SAVE '*'
0001 000E 58A8           SACH *+,0          STORE HIGH
0002 000F 5098           SACL *-,0          STORE LOW
```

EXAMPLE 3:

```
0015                    INCX *-
0001 0010 6698           ZALS *-
0002 0011 60A8           ADDH *+            LOAD '*-'
0003 0012 0004"         ADD  ONE,0         INCREMENT
0004                     SACX *-            SAVE '*-'
0001 0013 5098           SACL *-,0          STORE LOW
0002 0014 5898           SACH *-,0          STORE HIGH
```

EXAMPLE 4:

```
0017                    INCX *+
0001                     LDAX *             LOAD '*'
0001 0015 65A8           ZALH *+            LOAD HIGH
0002 0016 6198           ADDS *-            LOAD LOW '*'
0002 0017 0004"         ADD  ONE,0         INCREMENT
0003                     SACX *+            SAVE '*+'
0001 0018 58A8           SACH *+,0          STORE HIGH
0002 0019 50A8           SACL *+,0          STORE LOW
```

EXAMPLE 5:

```
0019                    INCX
0001 001A 0004"          ADD  ONE,0         INCREMENT AC
```

TITLE: Load Accumulator from Address in Accumulator

NAME: LACARY

OBJECTIVE: Load accumulator from array in data RAM; the address of the data RAM location is in the accumulator; the data will be left-shifted in the accumulator

ALGORITHM: (ACC) → AR1
(@AR1) * 2^{shift} → ACC

CALLING SEQUENCE: LACARY [shift]

ENTRY CONDITIONS: $0 \leqslant shift < 16$; $0 \leqslant (ACC) \leqslant 143$

EXIT CONDITIONS: Data RAM location pointed to by accumulator is stored in the accumulator; AR1 is overwritten

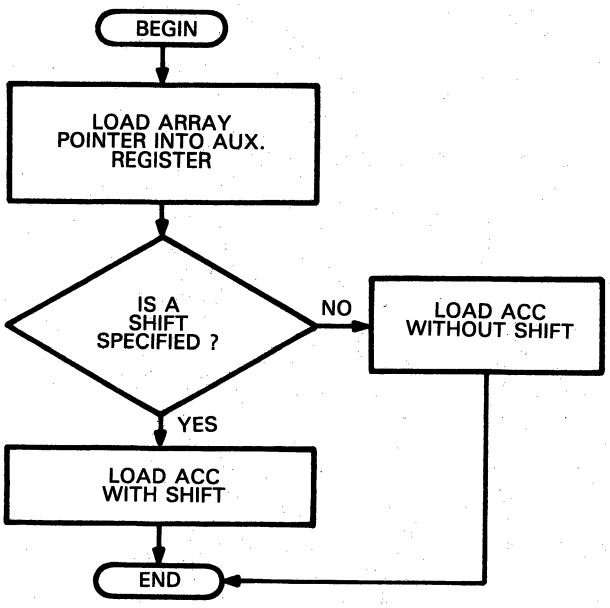
PROGRAM MEMORY REQUIRED: 4 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 4 cycles

FLOWCHART: LACARY



SOURCE:

```
*LOAD AC FROM ADDRESS IN AC
*
LACARY $MACRO  A
       ACTAR AR1          AC TO AR1
       $IF  A.L=0
       LAC  *,0           LOAD
       $ELSE
       LAC  *,:A:         LOAD AND SHIFT
       $ENDIF
       $END
```

EXAMPLE 1:

```
 0011                  LACARY 8
 0001                   ACTAR AR1          AC TO AR1
 0001 0006 5006"         SACL XR0,0        STORE AC TO XR0
 0002 0007 3906"         LAR  AR1,XR0      RE-LOAD AR1
 0003 0008 6881          LARP AR1          LOAD AR POINTER
 0002 0009 2888         LAC  *,8           LOAD AND SHIFT
```

EXAMPLE 2:

```
 0013                  LACARY
 0001                   ACTAR AR1          AC TO AR1
 0001 000A 5006"         SACL XR0,0        STORE AC TO XR0
 0002 000B 3906"         LAR  AR1,XR0      RE-LOAD AR1
 0003 000C 6881          LARP AR1          LOAD AR POINTER
 0002 000D 2088         LAC  *,0           LOAD
```

7

TITLE: Arithmetic Left Shift

NAME: LASH

OBJECTIVE: Move word from one data location to another with an arithmetic left shift

ALGORITHM: $(A) * 2^{shift} \rightarrow B$

CALLING SEQUENCE: LASH A,B,shift

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127; 0 \leqslant B \leqslant 127; 0 \leqslant shift < 16$

EXIT CONDITIONS: B contains the shifted value of A

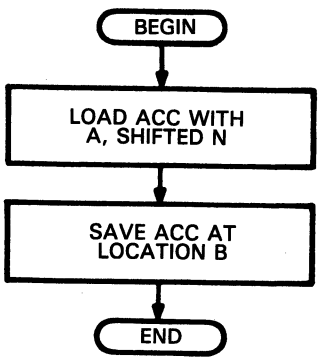
PROGRAM MEMORY REQUIRED: 2 words

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 2 cycles

FLOWCHART: LASH



SOURCE:

```
*MOVE A TO B (SINGLE-VAR) WITH N (CONST) BIT
*LEFT ARITHMETIC SHIFT
*
LASH   $MACRO A,B,N    MOVE WITH LEFT ARITH. SHIFT
       LAC   :A:,:N:   LOAD :A: LEFT SHIFT
       SACL  :B:,0     STORE TO :B:
       $END
```

EXAMPLE:

```
0013                    LASH A,B,5
0001 0008 2507           LAC  A,5        LOAD A LEFT SHIFT
0002 0009 5008           SACL B,0        STORE TO B
```

TITLE: Double-Word Arithmetic Left Shift

NAME: LASX

OBJECTIVE: Move double word from one data location to another in data memory with left shift

ALGORITHM: $(A:A + 1) * 2^{shift} \rightarrow B:B + 1$

CALLING SEQUENCE: LASX A,B,shift

ENTRY CONDITIONS: $0 \leqslant A \leqslant 126$; $0 \leqslant B \leqslant 126$; $0 \leqslant shift < 16$

EXIT CONDITIONS: B:B + 1 contains shifted value of A:A + 1

PROGRAM MEMORY REQUIRED: 8 words

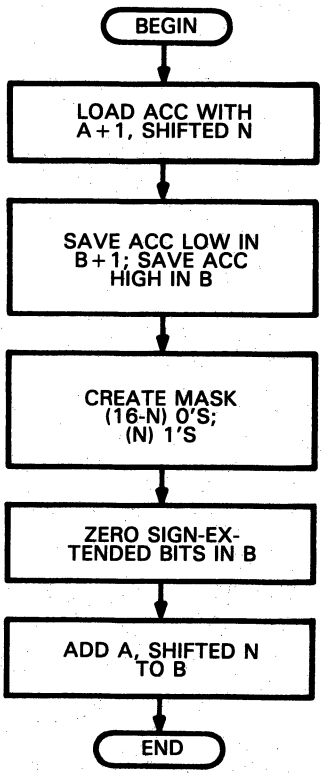
DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 8 cycles

7

FLOWCHART: LASX



SOURCE:

```
*MOVE A TO B (DOUBLE VAR) WITH N (CONST) BIT
*LEFT ARITHMETIC SHIFT
*
LASX  $MACRO  A,B,N     MOVE DOUBLE WITH ARITH. SHIFT
      LAC   :A:+1,:N:   LOAD LOW, SHIFT LEFT
      SACL  :B:+1,0     SAVE IN LOW
      SACH  :B:,0       SAVE HIGH OVERFLOW
      LAC   MINUS,:N:   GET MASK
      NOT
      AND   :B:         TAKE SIGNIFICANT BITS
      ADD   :A:,:N:     ADD IN SHIFT HIGH PART
      SACL  :B:,0       SAVE HIGH
      $END
```

EXAMPLE:

```
 0011                   LASX A,B,3
 0001 0006 2308          LAC  A+1,3        LOAD LOW, SHIFT LEFT
 0002 0007 500A          SACL B+1,0        SAVE IN LOW
 0003 0008 5809          SACH B,0          SAVE HIGH OVERFLOW
 0004 0009 2305"         LAC  MINUS,3      GET MASK
 0005                    NOT
 0001 000A 7805"          XOR  MINUS       INVERT
 0006 000B 7909          AND  B            TAKE SIGNIFICANT BITS
 0007 000C 0307          ADD  A,3          ADD IN SHIFT HIGH PART
 0008 000D 5009          SACL B,0          SAVE HIGH
```

7

TITLE: Load Double Word into Accumulator from Address in Accumulator

NAME: LAXARY

OBJECTIVE: Load accumulator from double-word array in data RAM; the address of the first RAM location is in the accumulator

ALGORITHM: (ACC) → AR1
(@AR1) → ACC high
(@AR1 + 1) → ACC low

CALLING SEQUENCE: LAXARY

ENTRY CONDITIONS: $0 \leqslant (ACC) \leqslant 143$

EXIT CONDITIONS: Double word pointed to by accumulator is stored in the accumulator; AR1 is overwritten

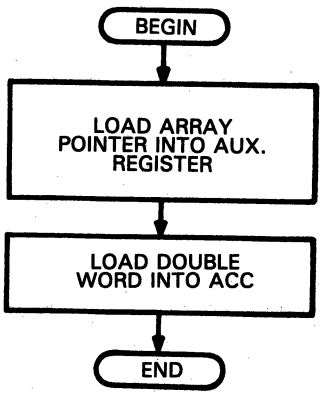
PROGRAM MEMORY REQUIRED: 5 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 5 cycles

FLOWCHART: LAXARY



SOURCE:

```
*LOAD DOUBLE AC FROM ADDRESS IN AC
*
LAXARY $MACRO
       ACTAR AR1          AC TO AR1
       LDAX *+            LOAD DOUBLE
       $END
```

LAXARY

EXAMPLE:

```
0011                    LAXARY
0001                     ACTAR AR1          AC TO AR1
0001 0006 5006"           SACL XR0,0        STORE AC TO XR0
0002 0007 3906"           LAR  AR1,XR0      RE-LOAD AR1
0003 0008 6881            LARP AR1          LOAD AR POINTER
0002                     LDAX *+            LOAD DOUBLE
0001 0009 65A8            ZALH *+           LOAD HIGH
0002 000A 61A8            ADDS *+           LOAD LOW '*+'
```

TITLE: Load Constant into Accumulator

NAME: LCAC

OBJECTIVE: Move constant value into accumulator with possible left shift

ALGORITHM: Constant → ACC
if shift → (ACC) →
temp * 2^{shift} → ACC

CALLING SEQUENCE: LCAC constant,shift,temp

ENTRY CONDITIONS: $-32768 \leqslant constant \leqslant 32767$; $0 \leqslant shift < 16$;
$0 \leqslant temp \leqslant 127$

EXIT CONDITIONS: Accumulator contains value of the constant

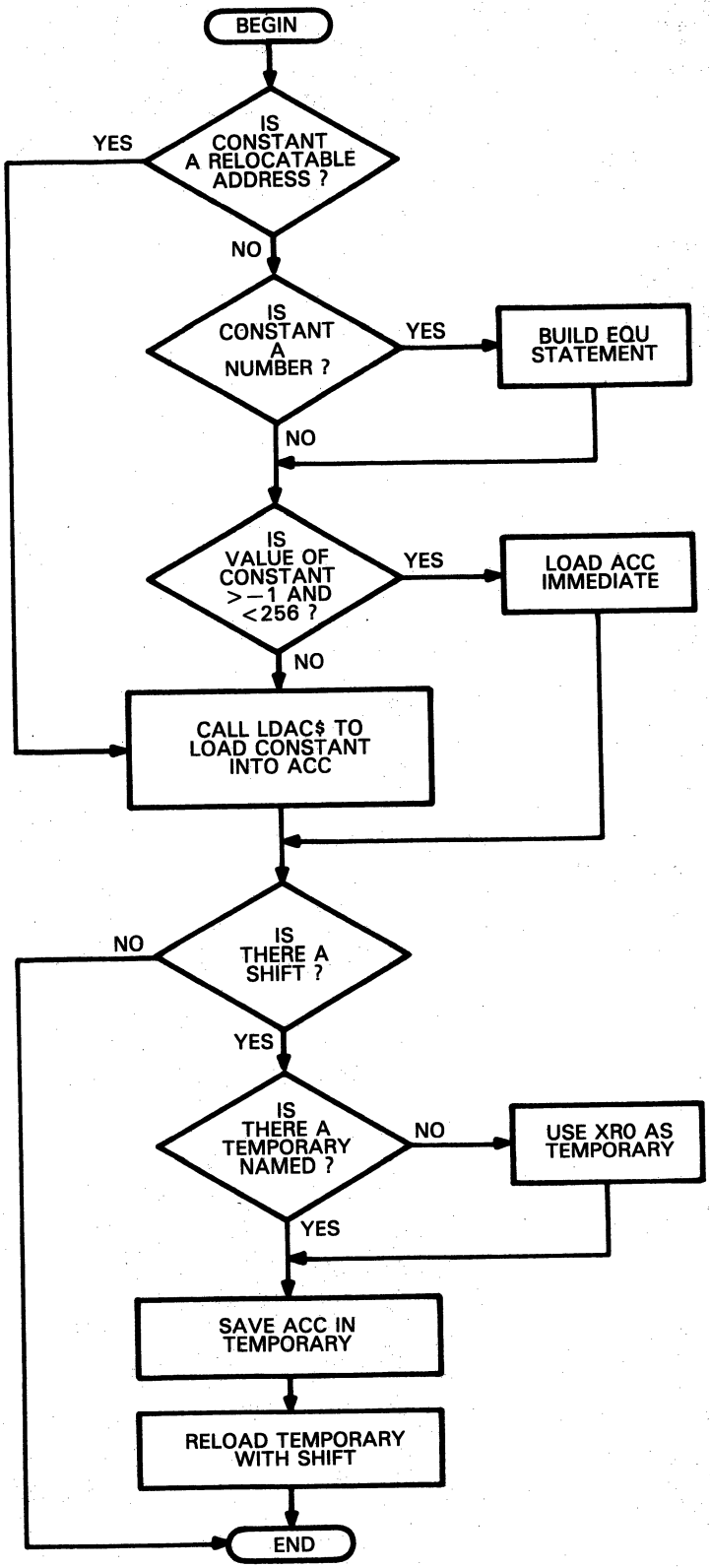
PROGRAM MEMORY REQUIRED: 1 – 5 words + LDAC$ routine

DATA MEMORY REQUIRED: 0 – 2 words

STACK REQUIRED: 2 levels with LDAC$

EXECUTION TIME: 1 – 15 cycles

FLOWCHART: LCAC



SOURCE:

```
*
*LOAD CONSTANT TO AC
*       LCAC A              LOAD CONSTANT A
*       LCAC A,B            LOAD CONSTANT A, SHIFTED B, USE TEMP XR0
*       LCAC A,B,T          LOAD CONSTANT A, SHIFTED B, USE TEMP T
*
LCAC    $MACRO  A,B,T
        $IF   A.SA&$REL
        CALL LDAC$          LOAD AC WITH:
        REF   LDAC$
        DATA :A:                  :A:
        $ELSE
        $IF   A.SA&$UNDF
        $VAR L,Q
        $ASG '$$LAB' TO L.S
        $ASG L.SV+1 TO L.SV
V$:L.SV: EQU :A:
        $ASG 'V$' TO Q.S
        $ASG :Q.S::L.SV: TO A.S
        $ENDIF
        $IF   (A.SV<256)&(A.SV>-1)
        LACK :A:            LOAD AC WITH :A:
        $ELSE
        CALL LDAC$          LOAD AC WITH:
        REF   LDAC$
        DATA :A:                  :A:
        $ENDIF
        $ENDIF
        $IF   B.L#=0
        $IF   (B.V>0)
        $IF   T.L=0         XR0 AS TEMP
        $ASG 'XR0' TO T.S
        $ENDIF
        SACL :T:,0          STORE UNSHIFTED CONSTANT
        LAC   :T:,:B:       LOAD SHIFTED
        $ENDIF
        $ENDIF
        $END
```

EXAMPLE 1:

```
0012                    LCAC 1,5
0001       0001  V$2 EQU 1
0002 0007 7E01          LACK V$2          LOAD AC WITH V$2
0003 0008 5003"         SACL XR0,0        STORE UNSHIFTED CONSTANT
0004 0009 2503"         LAC  XR0,5        LOAD SHIFTED
```

EXAMPLE 2:

```
0014                    LCAC 128,0
0001       0080  V$3 EQU 128
0002 000A 7E80          LACK V$3          LOAD AC WITH V$3
```

EXAMPLE 3:

```
0018                    LCAC -1000,5
0001       FC18  V$5 EQU -1000
0002 000E F800          CALL LDAC$        LOAD AC WITH:
     000F 0000
```

```
0003                    REF  LDAC$
0004 0010 FC18          DATA V$5            V$5
0005 0011 5003"         SACL XR0,0      STORE UNSHIFTED CONSTANT
0006 0012 2503"         LAC  XR0,5      LOAD SHIFTED
```

EXAMPLE 4:

```
0022                   LCAC A,6,B
0001 0016 7E07          LACK A          LOAD AC WITH A
0002 0017 5008          SACL B,0        STORE UNSHIFTED CONSTANT
0003 0018 2608          LAC  B,6        LOAD SHIFTED
```

TITLE: Load Constant to Accumulator from Program Address in Accumulator

NAME: LCACAR

OBJECTIVE: Load accumulator from array in program RAM; the address of the program ROM location is in the accumulator; the data will be left-shifted in the accumulator

ALGORITHM: (@ACC) → temp
(temp) * 2^{shift} → ACC

CALLING SEQUENCE: LCACAR [C][,TEMP]

ENTRY CONDITIONS: $0 \leqslant shift < 16$; $0 \leqslant TEMP \leqslant 127$; $0 \leqslant (ACC) \leqslant 4095$

EXIT CONDITIONS: Program ROM location pointed to by accumulator is stored in the accumulator

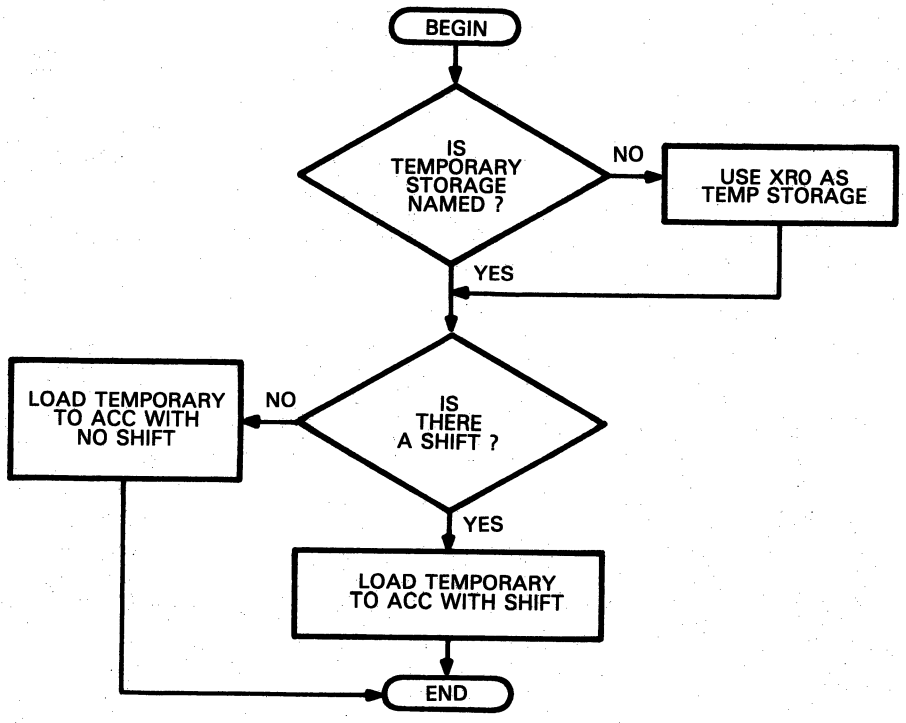
PROGRAM MEMORY REQUIRED: 2 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: 1 level

EXECUTION TIME: 4 cycles

FLOWCHART: LCACAR



SOURCE:

```
*LOAD CONSTANT ADDRESS BY AC IN AC
*     (IN ROM)
*
LCACAR $MACRO  A,T
       $IF  T.L=0       ASSIGN TEMP
       $ASG 'XR0' TO T.S
       $ENDIF
       TBLR :T:         READ FROM ROM TO :T:
       $IF  A.L=0
       LAC  :T:,0       LOAD :T: UNSHIFTED
       $ELSE
       LAC  :T:,:A:     LOAD :T: SHIFTED
       $ENDIF
       $END
```

EXAMPLE 1:

```
0011                    LCACAR 8
0001 0006 6706"          TBLR XR0          READ FROM ROM TO XR0
0002 0007 2806"          LAC  XR0,8        LOAD XR0 SHIFTED
```

EXAMPLE 2:

```
0013                    LCACAR 4,A
0001 0008 6707           TBLR A            READ FROM ROM TO A
0002 0009 2407           LAC  A,4          LOAD A SHIFTED
```

EXAMPLE 3:

```
0015                    LCACAR
0001 000A 6706"          TBLR XR0          READ FROM ROM TO XR0
0002 000B 2006"          LAC  XR0,0        LOAD XR0 UNSHIFTED
```

EXAMPLE 4:

```
0017                    LCACAR ,C
0001 000C 6700"          TBLR C            READ FROM ROM TO C
0002 000D 2000"          LAC  C,0          LOAD C UNSHIFTED
```

TITLE: Load Constant into Auxiliary Register

NAME: LCAR

OBJECTIVE: Move constant value into auxiliary register

ALGORITHM: Constant → AR

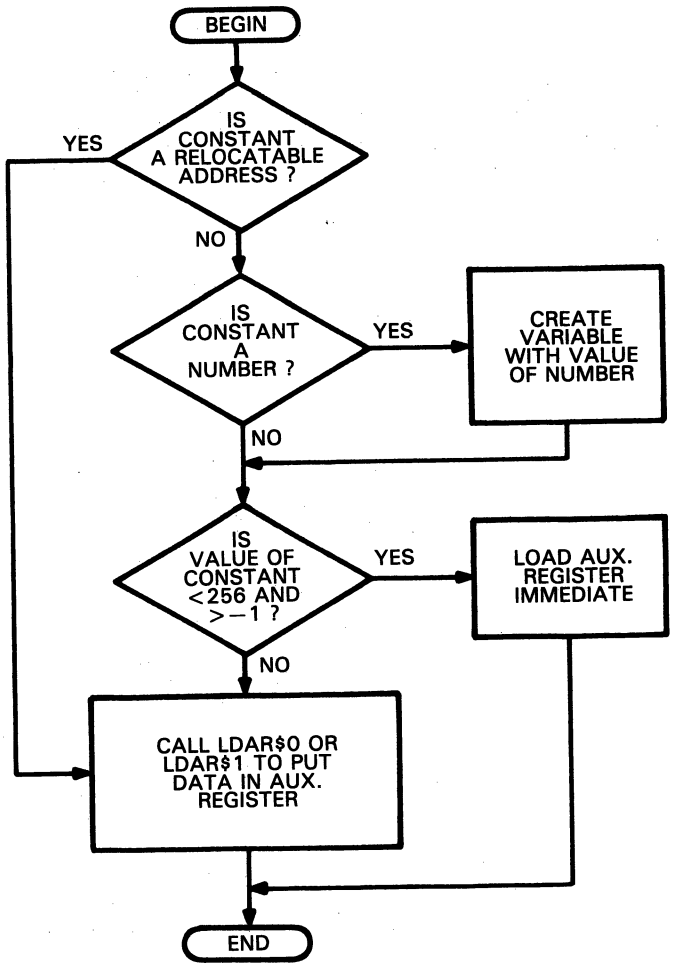
CALLING SEQUENCE: LCAR AR,constant

ENTRY CONDITIONS: $-32768 \leqslant$ constant $\leqslant 32767$; AR = 0,1

EXIT CONDITIONS: Auxiliary register contains value of the constant

PROGRAM MEMORY REQUIRED:	1 – 3 words (+ LDAR$0 and LDAR$1 routines)	**DATA MEMORY REQUIRED:**	0 – 2 words
STACK REQUIRED:	2 levels with LDAR$	**EXECUTION TIME:**	1 – 13 cycles

FLOWCHART: LCAR



SOURCE:

```
*LOAD CONSTANT (TO AR0/1)
*       LCAR AR0/1,CONSTANT
*
LCAR    $MACRO  A,B
        $IF    B.SA&$REL
        CALL LDAR$:A.V:    LOAD :A: WITH:
        REF  LDAR$:A.V:
        DATA :B:                :B:
        $ELSE
        $IF  B.SA&$UNDF
        $VAR L,Q
        $ASG '$$LAB' TO L.S
        $ASG L.SV+1 TO L.SV
V$:L.SV: EQU :B:
        $ASG 'V$' TO Q.S
        $ASG :Q.S::L.SV: TO B.S
        $ENDIF
        $IF  (B.SV<256)&(B.SV>-1)
        LARK :A:,:B:       LOAD :A: WITH :B:
        $ELSE
        CALL LDAR$:A.V:    LOAD :A: WITH:
        REF  LDAR$:A.V:
        DATA :B:                :B:
```

```
        $ENDIF
        $ENDIF
        $END
```

EXAMPLE 1:

```
0010                    LCAR 0,A
0001 0006 7007           LARK 0,A          LOAD 0 WITH A
```

EXAMPLE 2:

```
0012                    LCAR 1,C
0001 0007 F800           CALL LDAR$1       LOAD 1 WITH:
     0008 0000
0002                     REF  LDAR$1
0003 0009 0000"          DATA C            C
```

EXAMPLE 3:

```
0014                   LCAR AR1,-1000
0001       FC18  V$1 EQU -1000
0002 000A F800           CALL LDAR$1       LOAD AR1 WITH:
     000B 0000
0003                     REF  LDAR$1
0004 000C FC18           DATA V$1          V$1
```

EXAMPLE 4:

```
0016                   LCAR AR0,3333
0001       0D05  V$2 EQU 3333
0002 000D F800           CALL LDAR$0       LOAD AR0 WITH:
     000E 0000
0003                     REF  LDAR$0
0004 000F 0D05           DATA V$2          V$2
```

TITLE: Load Double-Word Constant into Accumulator

NAME: LCAX

OBJECTIVE: Move double-word constant value into accumulator

ALGORITHM: Constant → ACC

CALLING SEQUENCE: LCAX (upper,lower)

ENTRY CONDITIONS: $-32768 \leqslant upper \leqslant 32767$; $-32768 \leqslant lower \leqslant 32767$

EXIT CONDITIONS: Accumulator contains value of the constant

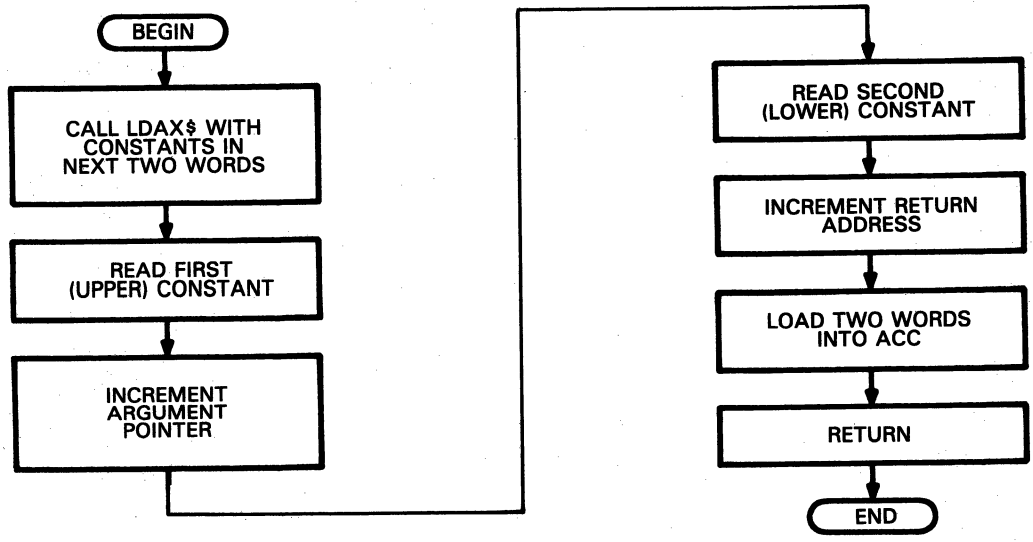
PROGRAM MEMORY REQUIRED: 2 words + LDAX$ routine

DATA MEMORY REQUIRED: 3 words

STACK REQUIRED: 2 levels

EXECUTION TIME: 18 cycles

FLOWCHART: LCAX



SOURCE:

```
*LOAD DOUBLE CONSTANT (TO AC)
*      LCAX (HIGH VALUE,LOW VALUE)
*
LCAX   $MACRO  A
       CALL LDAX$          LOAD DOUBLE
       REF  LDAX$
       DATA :A:            DATA LIST
       $END
```

EXAMPLE 1:

```
0010                    LCAX (128,3)
0001 0006 F800           CALL LDAX$          LOAD DOUBLE
     0007 0000
0002                     REF  LDAX$
0003 0008 0080           DATA 128,3          DATA LIST
     0009 0003
```

EXAMPLE 2:

```
0012                    LCAX (-1000,5)
0001 000A F800           CALL LDAX$          LOAD DOUBLE
     000B 0000
0002                     REF  LDAX$
0003 000C FC18           DATA -1000,5        DATA LIST
     000D 0005
```

EXAMPLE 3:

```
0014                    LCAX (A,B)
0001 000E F800           CALL LDAX$          LOAD DOUBLE
     000F 0000
0002                     REF  LDAX$
0003 0010 0007           DATA A,B            DATA LIST
0011 0009
```

TITLE: Load Double-Word Constant to Accumulator from Program Memory

NAME: LCAXAR

OBJECTIVE: Load accumulator from double-word array in program RAM; the address of the first program ROM location is in the accumulator

ALGORITHM: (@ACC) → temp
(@ACC + 1) → temp + 1
(temp:temp + 1) → ACC

CALLING SEQUENCE: LCAXAR [TEMP]

ENTRY CONDITIONS: $0 \leqslant TEMP \leqslant 127$; $0 \leqslant (ACC) \leqslant 4095$

EXIT CONDITIONS: Program ROM double-word location pointed to by accumulator is stored in the accumulator

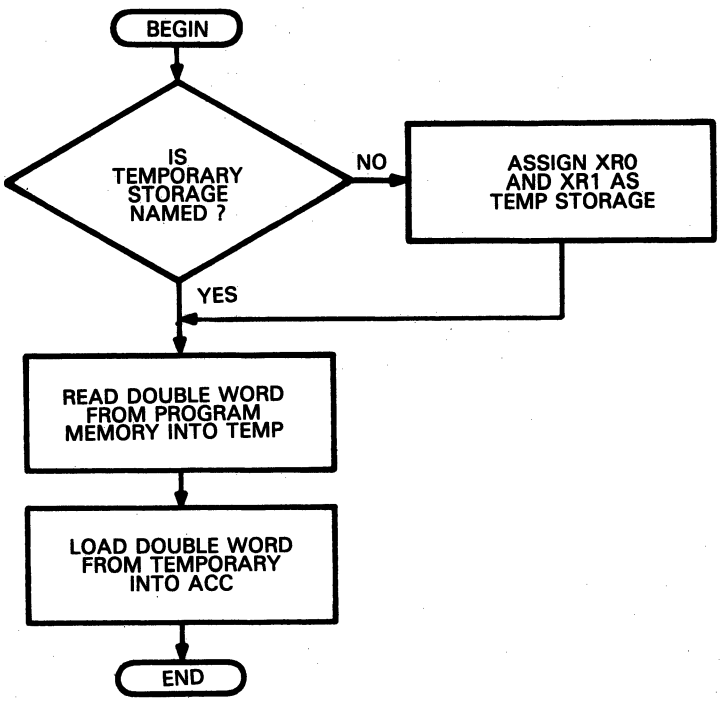
PROGRAM MEMORY REQUIRED: 5 words

DATA MEMORY REQUIRED: 2 words

STACK REQUIRED: 1 level

EXECUTION TIME: 9 cycles

FLOWCHART: LCAXAR



SOURCE:

```
*LOAD FROM ROW AT ADDRESS IN ACCUMULATOR,
*DOUBLE CONSTANT TO ACCUMULATOR
*
LCAXAR $MACRO  T
       $IF  T.L=0      ASSIGN TEMP
       $ASG 'XR0' TO T.S
       $ENDIF
       TBLR :T:        READ HIGH PART OF :T:
       ADD  ONE,0      INCREMENT AC
       TBLR :T:+1      READ LOW PART OF :T:
       LDAX :T:        LOAD TO AC
       $END
```

EXAMPLE 1:

```
0011                    LCAXAR
0001 0006 6706"          TBLR XR0          READ HIGH PART OF XR0
0002 0007 0004"          ADD  ONE,0        INCREMENT AC
0003 0008 6707"          TBLR XR0+1        READ LOW PART OF XR0
0004                     LDAX XR0          LOAD TO AC
0001 0009 6506"           ZALH XR0         LOAD HIGH XR0
0002 000A 6107"           ADDS XR0+1       LOAD LOW XR0
```

EXAMPLE 2:

```
0013                    LCAXAR C
0001 000B 6700"          TBLR C            READ HIGH PART OF C
0002 000C 0004"          ADD  ONE,0        INCREMENT AC
0003 000D 6701"          TBLR C+1          READ LOW PART OF C
0004                     LDAX C            LOAD TO AC
0001 000E 6500"           ZALH C           LOAD HIGH C
0002 000F 6101"           ADDS C+1         LOAD LOW C
```

TITLE: Load Constant into P Register

NAME: LCP

OBJECTIVE: Move constant value into P register

ALGORITHM: 1 * constant → P

CALLING SEQUENCE: LCP constant

ENTRY CONDITIONS: $-4096 \leqslant$ constant $\leqslant 4095$

EXIT CONDITIONS: P register contains value of the constant;
T register contains value 1

PROGRAM MEMORY REQUIRED: 2 words

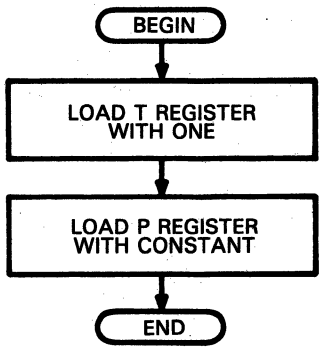
DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 2 cycles

7

FLOWCHART: LCP



SOURCE:

```
*LCP   LOAD A CONSTANT TO THE P REGISTER
*
LCP    $MACRO  A
       LT   ONE          LOAD A ONE
       MPYK :A:          MAKE CONSTANT
       $END
```

EXAMPLE 1:

```
0013                    LCP  A
0001 0008 6A01"          LT   ONE          LOAD A ONE
0002 0009 8007           MPYK A            MAKE CONSTANT
```

EXAMPLE 2:

```
0015                    LCP -4096
0001 000A 6A01"          LT   ONE          LOAD A ONE
0002 000B 9000           MPYK -4096        MAKE CONSTANT
```

EXAMPLE 3:

```
0017                    LCP 4095
0001 000C 6A01"          LT   ONE          LOAD A ONE
0002 000D 8FFF           MPYK 4095         MAKE CONSTANT
```

EXAMPLE 4:

```
0019                    LCP  -4000
0001 000E 6A01"          LT   ONE          LOAD A ONE
0002 000F 9060           MPYK -4000        MAKE CONSTANT
```

TITLE: Load Constant into P Register and Accumulator

NAME: LCPAC

OBJECTIVE: Move constant value into P register and accumulator

ALGORITHM: 1 * constant → P
(P) → ACC

CALLING SEQUENCE: LCPAC constant

ENTRY CONDITIONS: $-4096 \leqslant$ constant $\leqslant 4095$

EXIT CONDITIONS: P register and accumulator contain value of the constant;
T register contains the value 1

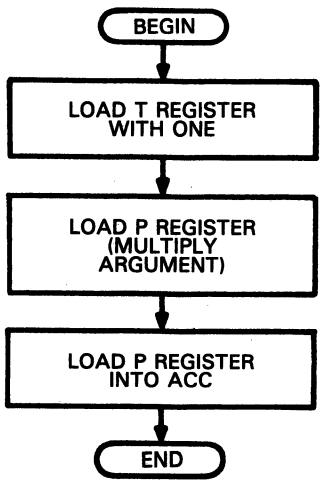
PROGRAM MEMORY REQUIRED: 3 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 3 cycles

FLOWCHART: LCPAC



SOURCE:

```
*LCPAC   LOAD A CONST TO P AND AC REGISTERS
*
LCPAC  $MACRO  A
```

```
        LT   ONE          LOAD A ONE
        MPYK :A:          MAKE CONSTANT
        PAC               TO THE AC
        $END
```

EXAMPLE 1:

```
0013                    LCPAC  A
0001 0009 6A01"          LT    ONE          LOAD A ONE
0002 000A 8007           MPYK  A            MAKE CONSTANT
0003 000B 7F8E           PAC                TO THE AC
```

EXAMPLE 2:

```
0015                    LCPAC -4096
0001 000C 6A01"          LT    ONE          LOAD A ONE
0002 000D 9000           MPYK -4096         MAKE CONSTANT
0003 000E 7F8E           PAC                TO THE AC
```

EXAMPLE 3:

```
0017                    LCPAC 4095
0001 000F 6A01"          LT    ONE          LOAD A ONE
0002 0010 8FFF           MPYK 4095          MAKE CONSTANT
0003 0011 7F8E           PAC                TO THE AC
```

EXAMPLE 4:

```
0019                    LCPAC  -4000
0001 0012 6A01"          LT    ONE          LOAD A ONE
0002 0013 9060           MPYK -4000         MAKE CONSTANT
0003 0014 7F8E           PAC                TO THE AC
```

TITLE: Load Double Word

NAME: LDAX

OBJECTIVE: Load double word into accumulator

ALGORITHM:

LDAX *	– causes→	(@AR:@AR + 1) → ACC
LDAX * –	– causes→	(@AR – 1: @ AR) → ACC (AR) – 2 → AR
LDAX * +	– causes→	(@AR:@ AR + 1) → ACC (AR) + 2 → AR
LDAX A	– causes→	(A:A + 1) → ACC

CALLING SEQUENCE: LDAX {A,*,* – ,* + }

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127$

EXIT CONDITIONS: Accumulator contains value of double word; auxiliary register is updated if necessary

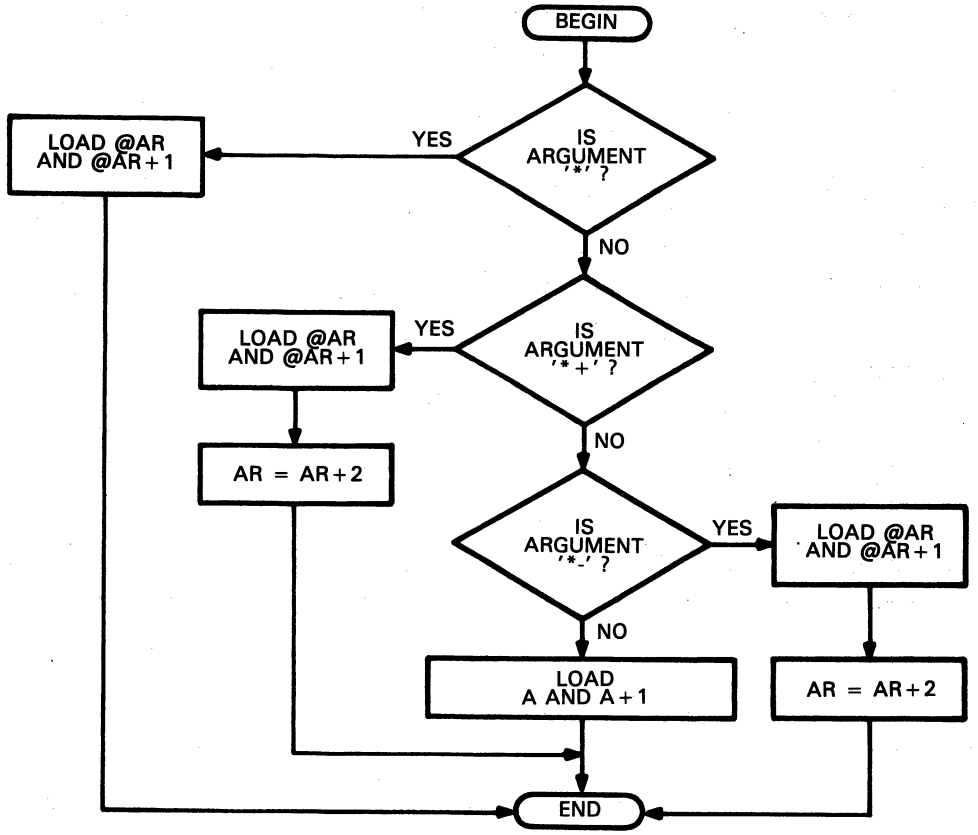
PROGRAM MEMORY REQUIRED: 2 words

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 2 cycles

FLOWCHART: LDAX



SOURCE:

```
*LOAD DOUBLE PRECISION
*
LDAX    $MACRO  A          LOAD DOUBLE
        $VAR ST,SP,SM
        $ASG '*' TO ST.S
        $ASG '*+' TO SP.S
        $ASG '*-' TO SM.S
        $IF A.SV=ST.SV
        ZALH *+            LOAD HIGH
        ADDS *-            LOAD LOW '*'
        $ELSE
        $IF A.SV=SP.SV
        ZALH *+            LOAD HIGH
        ADDS *+            LOAD LOW '*+'
        $ELSE
        $IF A.SV=SM.SV
        ZALS *-            LOAD LOW
        ADDH *-            LOAD HIGH '*-'
        $ELSE
        ZALH :A:           LOAD HIGH :A:
        ADDS :A:+1         LOAD LOW :A:
        $ENDIF
        $ENDIF
        $ENDIF
        $END
```

EXAMPLE 1:

```
0011                LDAX A
0001 0006 6507       ZALH A          LOAD HIGH A
0002 0007 6108       ADDS A+1        LOAD LOW A
```

EXAMPLE 2:

```
0013                LDAX *
0001 0008 65A8       ZALH *+         LOAD HIGH
0002 0009 6198       ADDS *-         LOAD LOW '*'
```

EXAMPLE 3:

```
0015                LDAX *-
0001 000A 6698       ZALS *-         LOAD LOW
0002 000B 6098       ADDH *-         LOAD HIGH '*-'
```

EXAMPLE 4:

```
0017                LDAX *+
0001 000C 65A8       ZALH *+         LOAD HIGH
0002 000D 61A8       ADDS *+         LOAD LOW '*+'
```

TITLE: Load Constant into T Register

NAME: LTK

OBJECTIVE: Move constant value into T register

ALGORITHM: Constant → T

CALLING SEQUENCE: LTK constant

ENTRY CONDITIONS: $-32768 \leqslant$ constant $\leqslant 32767$

EXIT CONDITIONS: T register contains value of the constant

PROGRAM MEMORY REQUIRED: 3 words (+ LTK$ routine)

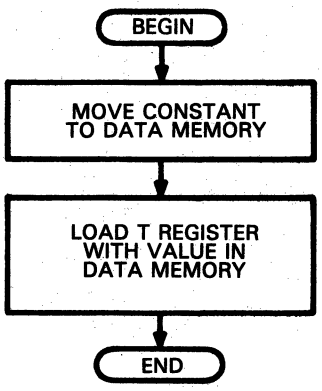
DATA MEMORY REQUIRED: 2 words

STACK REQUIRED: 2 levels

EXECUTION TIME: 13 cycles

7

FLOWCHART: LTK



SOURCE:

```
*LOAD CONSTANT TO T
*
LTK     $MACRO  A
        CALL LTK$           LOAD :A: TO T
        REF  LTK$
        DATA :A:
        $END
```

EXAMPLE 1:

```
0012                    LTK  A
0001 0009 F800           CALL LTK$          LOAD A TO T
     000A 0000
0002                     REF  LTK$
0003 000B 0007           DATA A
```

EXAMPLE 2:

```
0014                    LTK >7FFF
0001 000C F800           CALL LTK$          LOAD >7FFF TO T
     000D 0000
0002                     REF  LTK$
0003 000E 7FFF           DATA >7FFF
```

EXAMPLE 3:

```
0016                    LTK >8000
0001 000F F800           CALL LTK$          LOAD >8000 TO T
     0010 0000
0002                     REF  LTK$
0003 0011 8000           DATA >8000
```

TITLE: Select Maximum of Two Words

NAME: MAX

OBJECTIVE: Load maximum of two words into accumulator

ALGORITHM: If (A) > (B) then (A) → ACC
else (B) → ACC

CALLING SEQUENCE: MAX A,B

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127; 0 \leqslant B \leqslant 127$

EXIT CONDITIONS: Accumulator contains maximum value of two words

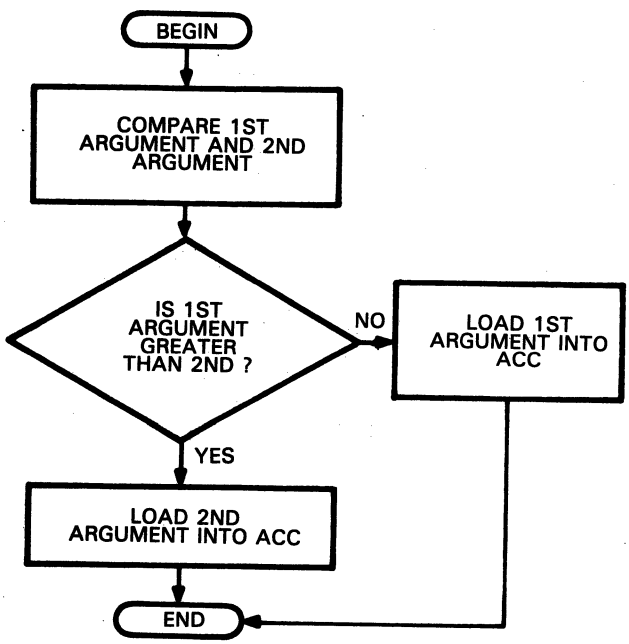
PROGRAM MEMORY REQUIRED: 8 words

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 5 – 7 cycles

FLOWCHART: MAX



SOURCE:

```
*SELECT MAXIMUM OF SINGLE A OR B
*A AND B ARE VARIABLES
*
MAX     $MACRO  A,B
        LAC   :A:,0          LOAD :A:
        SUB   :B:,0          COMPARE :B:
        $VAR L,L1,L2
        $ASG '$$LAB' TO L.S
        $ASG L.SV+2 TO L.SV     UNIQUE LABEL
        $ASG L.SV-1 TO L1.V
        $ASG L.SV   TO L2.V
        BGZ   L$:L1.V:       BRANCH IS :A:>:B:
        LAC   :B:,0          LOAD :B:
        B     L$:L2.V:       TO CONTINUE
L$:L1.V: LAC :A:,0            LOAD :A:
L$:L2.V: EQU $               CONTINUE
        $END
```

EXAMPLE:

```
 0011                    MAX A,B
 0001 0006 2007           LAC  A,0          LOAD A
 0002 0007 1008           SUB  B,0          COMPARE B
 0003 0008 FC00           BGZ  L$1          BRANCH IS A>B
      0009 000D'
 0004 000A 2008           LAC  B,0          LOAD B
 0005 000B F900           B    L$2          TO CONTINUE
      000C 000E'
 0006 000D 2007  L$1      LAC A,0           LOAD A
 0007      000E' L$2 EQU $                  CONTINUE
```

TITLE: Select Maximum of Two Double Words

NAME: MAXX

OBJECTIVE: Load maximum of two double words into accumulator

ALGORITHM: If (A:A + 1) > (B:B + 1) then (A:A + 1) → ACC
else (B:B + 1) → ACC

CALLING SEQUENCE: MAXX A,B

ENTRY CONDITIONS: 0 < = A < ,PI6,171 126; 0 < = B < = 126

EXIT CONDITIONS: Accumulator contains maximum value of two double words; saturation mode is reset

PROGRAM MEMORY REQUIRED: 14 words

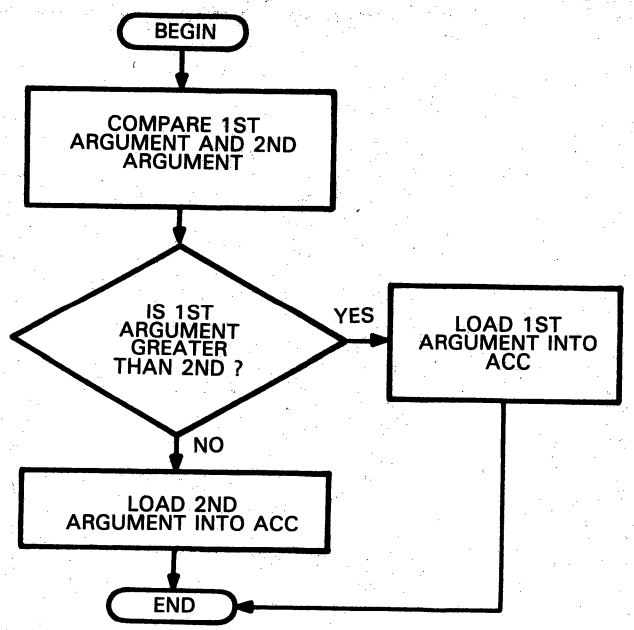
DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 10 – 12 cycles

7

FLOWCHART: MAXX



SOURCE:

```
*SELECT MAX OF DOUBLE A OR B (VARIABLES)
*
MAXX    $MACRO  A,B
        SOVM                SET OVERFLOW MODE
        LDAX :A:            LOAD :A:
        SUBX :B:            COMPARE TO :B:
        $VAR L,L1,L2
        $ASG '$$LAB' TO L.S
        $ASG L.SV+2 TO L.SV   UNIQUE LABEL
        $ASG L.SV-1 TO L1.V
        $ASG L.SV   TO L2.V
        BGZ  L$:L1.V:       BRANCH IF :A:>:B:
        LDAX :B:            LOAD :B:
        B    L$:L2.V:       TO CONTINUE
L$:L1.V: LDAX :A:           LOAD :A:
L$:L2.V: ROVM               CONTINUE
        $END
```

EXAMPLE:

```
 0013                    MAXX C,D
 0001 0013 7F8B           SOVM             SET OVERFLOW MODE
 0002                     LDAX C           LOAD C
 0001 0014 6500"           ZALH C          LOAD HIGH C
 0002 0015 6101"           ADDS C+1        LOAD LOW C
 0003                     SUBX D           COMPARE TO D
 0001 0016 6202"           SUBH D          SUBTRACT HIGH
 0002 0017 6303"           SUBS D+1        SUBTRACT LOW
 0004 0018 FC00           BGZ  L$3         BRANCH IF C>D
      0019 001E'
 0005                     LDAX D           LOAD D
 0001 001A 6502"           ZALH D          LOAD HIGH D
 0002 001B 6103"           ADDS D+1        LOAD LOW D
 0006 001C F900           B    L$4         TO CONTINUE
      001D 0020'
 0007              L$3    LDAX C           LOAD C
 0001 001E 6500"           ZALH C          LOAD HIGH C
 0002 001F 6101"           ADDS C+1        LOAD LOW C
 0008 0020 7F8A  L$4      ROVM             CONTINUE
```

TITLE: Select Minimum of Two Words

NAME: MIN

OBJECTIVE: Load minimum of two words into accumulator

ALGORITHM: If (A) < (B) then (A) → ACC
else (B) → ACC

CALLING SEQUENCE: MIN A,B

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127; 0 \leqslant B \leqslant 127$

EXIT CONDITIONS: Accumulator contains minimum value of two words

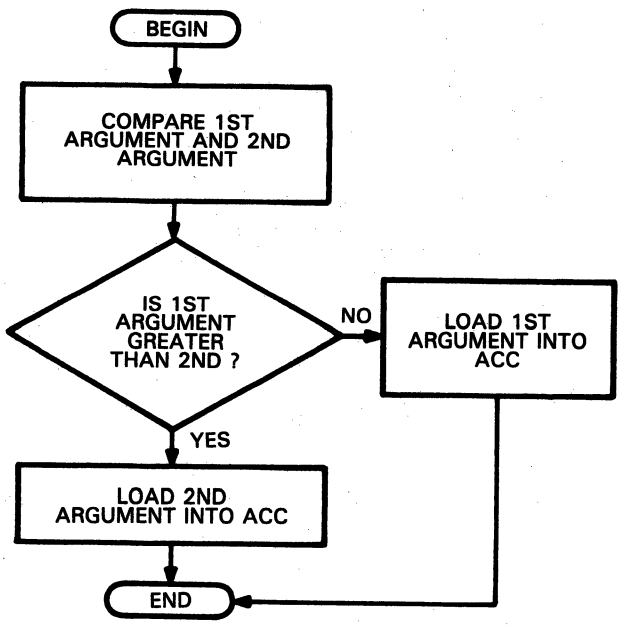
PROGRAM MEMORY REQUIRED: 8 words

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 5 – 7 cycle

FLOWCHART: MIN



SOURCE:

```
*SELECT MINUMUM OF SINGLE A OR B (VARIABLES)
*
MIN     $MACRO   A,B
        LAC   :A:,0          LOAD :A:
        SUB   :B:,0          COMPARE TO :B:
        $VAR L,L1,L2
        $ASG '$$LAB' TO L.S
        $ASG L.SV+2 TO L.SV
        $ASG L.SV-1 TO L1.V
        $ASG L.SV   TO L2.V
        BLZ   L$:L1.V:       BRANCH IF :A:<:B:
        LAC   :B:,0          LOAD :B:
        B     L$:L2.V:       TO CONTINUE
L$:L1.V: LAC :A:,0           LOAD :A:
L$:L2.V: EQU $               CONTINUE
        $END
```

EXAMPLE:

```
 0011                    MIN A,B
 0001 0006 2007           LAC  A,0          LOAD A
 0002 0007 1008           SUB  B,0          COMPARE TO B
 0003 0008 FA00           BLZ  L$1          BRANCH IF A<B
      0009 000D'
 0004 000A 2008           LAC  B,0          LOAD B
 0005 000B F900           B    L$2          TO CONTINUE
      000C 000E'
 0006 000D 2007  L$1      LAC A,0           LOAD A
 0007      000E' L$2 EQU $                  CONTINUE
```

7

TITLE: Select Minimum of Two Double Words

NAME: MINX

OBJECTIVE: Load minimum of two double words into accumulator

ALGORITHM: If (A:A + 1) < (B:B + 1) then (A:A + 1) → ACC
else (B:B + 1) → ACC

CALLING SEQUENCE: MINX A,B

ENTRY CONDITIONS: $0 \leqslant A \leqslant 126; 0 \leqslant B \leqslant 126$

EXIT CONDITIONS: Accumulator contains minimum value of two double words; saturation mode is reset

PROGRAM MEMORY REQUIRED: 14 words

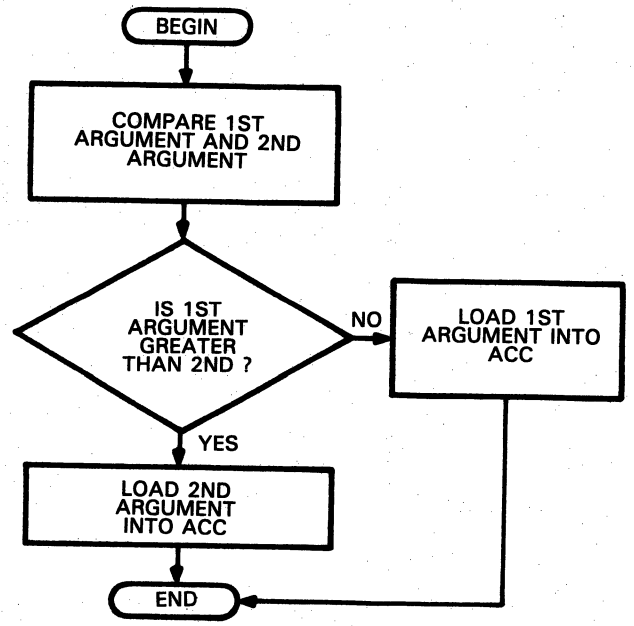
DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 10 – 12 cycles

7

FLOWCHART: MINX



MINX

SOURCE:

```
*SELECT MINIMUM OF DOUBLE A OR B (VARIABLES)
*
MINX    $MACRO  A,B
        SOVM                SET OVERFLOW MODE
        LDAX :A:            LOAD :A:
        SUBX :B:            COMPARE TO :B:
        $VAR L,L1,L2
        $ASG '$$LAB' TO L.S
        $ASG L.SV+2 TO L.SV
        $ASG L.SV-1 TO L1.V
        $ASG L.SV   TO L2.V
        BLZ  L$:L1.V:       BRANCH IF :A:<:B:
        LDAX :B:            LOAD :B:
        B    L$:L2.V:       TO CONTINUE
L$:L1.V: LDAX :A:           LOAD :A:
L$:L2.V: ROVM               CONTINUE
        $END
```

EXAMPLE:

```
0011                    MINX A,B
0001 0005 7F8B           SOVM           SET OVERFLOW MODE
0002                     LDAX A         LOAD A
0001 0006 6507            ZALH A        LOAD HIGH A
0002 0007 6108            ADDS A+1      LOAD LOW A
0003                     SUBX B         COMPARE TO B
0001 0008 6209            SUBH B        SUBTRACT HIGH
0002 0009 630A            SUBS B+1      SUBTRACT LOW
0004 000A FA00           BLZ  L$1       BRANCH IF A<B
     000B 0010'
0005                     LDAX B         LOAD B
0001 000C 6509            ZALH B        LOAD HIGH B
0002 000D 610A            ADDS B+1      LOAD LOW B
0006 000E F900           B    L$2       TO CONTINUE
     000F 0012'
0007              L$1    LDAX A         LOAD A
0001 0010 6507            ZALH A        LOAD HIGH A
0002 0011 6108            ADDS A+1      LOAD LOW A
0008 0012 7F8A   L$2     ROVM           CONTINUE
```

TITLE: Move Word in Data Memory

NAME: MOV

OBJECTIVE: Copy word from one location to another in data memory

ALGORITHM: (A) → B or
(@ACC) → B

CALLING SEQUENCE: MOV [A],B

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127; 0 \leqslant B \leqslant 127$

EXIT CONDITIONS: Word at B contains value of word located at A;
AR0 may be overwritten; accumulator is overwritten

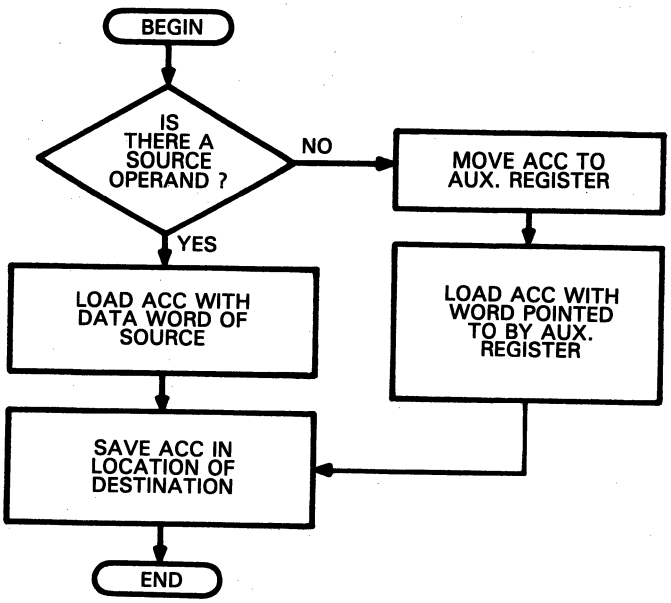
PROGRAM MEMORY REQUIRED: 2 – 5 words

DATA MEMORY REQUIRED: 0 – 1 words

STACK REQUIRED: None

EXECUTION TIME: 2 – 5 cycles

FLOWCHART: MOV



SOURCE:

```
*MOVE ONE WORD (A TO B)
*
MOV     $MACRO  A,B
        $IF  A.L=0          IF A IS AC
        SACL XR0,0          SAVE AC
        LAR  AR0,XR0        LOAD TO AR0
        LARP AR0            SELECT AR0
        LAC  *,0            LOAD *
        $ELSE
        LAC  :A:,0          LOAD :A:
        $ENDIF
        SACL :B:,0          STORE :B:
        $END
```

EXAMPLE 1:

```
0012                    MOV    A,B
0001 0006 2001           LAC  A,0          LOAD A
0002 0007 5008           SACL B,0          STORE B
```

EXAMPLE 2:

```
0014                    MOV    *,B
0001 0008 2088           LAC  *,0          LOAD *
0002 0009 5008           SACL B,0          STORE B
```

EXAMPLE 3:

```
0016                    MOV    C,*+
0001 000A 2000"          LAC  C,0          LOAD C
0002 000B 50A8           SACL *+,0         STORE *+
```

EXAMPLE 4:

```
0018                    MOV    ,D
0001 000C 5004"          SACL XR0,0        SAVE AC
0002 000D 3804"          LAR  AR0,XR0      LOAD TO AR0
0003 000E 6880           LARP AR0          SELECT AR0
0004 000F 2088           LAC  *,0          LOAD *
0005 0010 5001"          SACL D,0          STORE D
```

EXAMPLE 5:

```
0020                    MOV    *-,B
0001 0011 2098           LAC  *-,0         LOAD *-
0002 0012 5008           SACL B,0          STORE B
```

EXAMPLE 6:

```
0022                    MOV    *+,A
0001 0013 20A8           LAC  *+,0         LOAD *+
0002 0014 5001           SACL A,0          STORE A
```

EXAMPLE 7:

```
0024                    MOV    D,*-
0001 0015 2001"          LAC  D,0          LOAD D
0002 0016 5098           SACL *-,0         STORE *-
```

TITLE: Move Constants to Data Memory

NAME: MOVCON

OBJECTIVE: Move list of constants to data memory

ALGORITHM: For each constant in list,
C → A[i] (data memory location)

CALLING SEQUENCE: MOVCON C [,A|,*] or
MOVCON (C1,C2,...Cn) [,A|,*]

ENTRY CONDITIONS: $0 \leqslant A \leqslant 143$; $-32768 \leqslant C \leqslant 32767$

EXIT CONDITIONS: Data memory addresses starting at specified locations are filled with constants; AR0 and AR1 may be overwritten

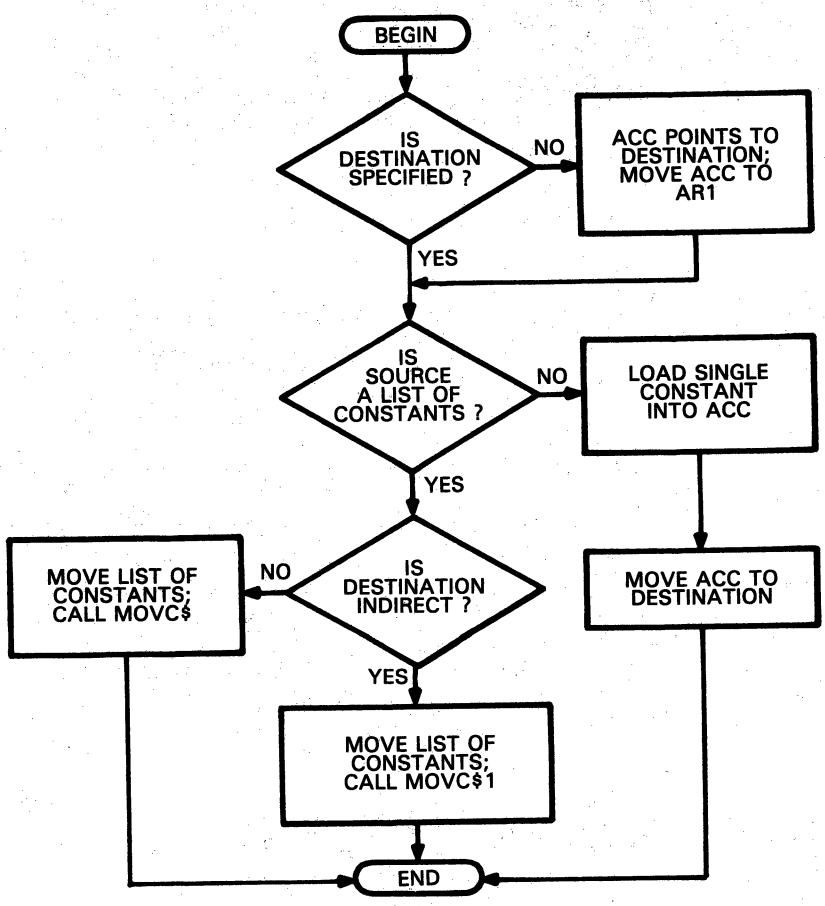
PROGRAM MEMORY REQUIRED: 8 words (+ MOVC$ routines)

DATA MEMORY REQUIRED: 3 words

STACK REQUIRED: 2 levels

EXECUTION TIME: (max) 9 + (7 x of C's) cycles

FLOWCHART: MOVCON



SOURCE:

```
MOVCON $MACRO  A,B
       $VAR ST
       $ASG '*' TO ST.S
       $IF  B.L=0
       ACTAR AR1
       $ASG '*' TO B.S
       $ENDIF
       $IF  A.A&$POPL      A IS LIST OF CONST
       $IF  B.SV=ST.SV
       CALL MOVC$1         MOVE CONSTANTS
       REF  MOVC$1
       $ELSE
       CALL MOVC$          MOVE CONSTANTS
       REF  MOVC$
       DATA :B:            TO :B:
       $ENDIF
       DATA :A.V:          LENGTH OF LIST
       DATA :A:            CONSTANT LIST
       $ELSE
       LCAC :A:
       SACL :B:,0          STORE CONSTANT
       $ENDIF
       $END
```

EXAMPLE 1:

```
0012                    MOVCON 1,B
0001                     LCAC 1
0001       0001  V$1 EQU 1
0002 0006 7E01           LACK V$1          LOAD AC WITH V$1
0002 0007 5008           SACL B,0          STORE CONSTANT
```

EXAMPLE 2:

```
0014                    MOVCON 3,*
0001                     LCAC 3
0001       0003  V$2 EQU 3
0002 0008 7E03           LACK V$2          LOAD AC WITH V$2
0002 0009 5088           SACL *,0          STORE CONSTANT
```

EXAMPLE 3:

```
0016                   MOVCON 6,
0001                    ACTAR AR1
0001 000A 5004"          SACL XR0,0        STORE AC TO XR0
0002 000B 3904"          LAR  AR1,XR0      RE-LOAD AR1
0003 000C 6881           LARP AR1          LOAD AR POINTER
0002                    LCAC 6
0001       0006  V$3 EQU 6
0002 000D 7E06           LACK V$3          LOAD AC WITH V$3
0003 000E 5088           SACL *,0          STORE CONSTANT
```

EXAMPLE 4:

```
0018                   MOVCON (32,15,2,13),B
0001 000F F800          CALL MOVC$         MOVE CONSTANTS
     0010 0000
0002                    REF  MOVC$
0003 0011 0008          DATA B             TO B
0004 0012 0004          DATA 4             LENGTH OF LIST
0005 0013 0020          DATA 32,15,2,13    CONSTANT LIST
     0014 000F
     0015 0002
     0016 000D
```

EXAMPLE 5:

```
0020                   MOVCON (22,1,56),*
0001 0017 F800          CALL MOVC$1        MOVE CONSTANTS
     0018 0000
0002                    REF  MOVC$1
0003 0019 0003          DATA 3             LENGTH OF LIST
0004 001A 0016          DATA 22,1,56       CONSTANT LIST
     001B 0001
     001C 0038
```

EXAMPLE 6:

```
0022                   MOVCON (33,34,35),
0001                    ACTAR AR1
0001 001D 5004"          SACL XR0,0        STORE AC TO XR0
0002 001E 3904"          LAR  AR1,XR0      RE-LOAD AR1
0003 001F 6881           LARP AR1          LOAD AR POINTER
0002 0020 F800          CALL MOVC$1        MOVE CONSTANTS
     0021 0000
0003                    REF  MOVC$1
0004 0022 0003          DATA 3             LENGTH OF LIST
```

MOVCON

```
0005 0023 0021          DATA 33,34,35      CONSTANT LIST
     0024 0022
     0025 0023
```

TITLE: Move Words to Data Memory

NAME: MOVDAT

OBJECTIVE: Copy data from program memory to data memory

ALGORITHM: For number of elements in array,

```
MOVDAT    A,B,C  – causes→ (A) → @B
MOVDAT    A,*,C  – causes→ (A) → @AR1
MOVDAT    A, ,C  – causes→ (A) → @ACC

MOVDAT    *,B,C  – causes→ (@AR0) → @B
MOVDAT    *,*,C  – causes→ (@AR0) → @AR1
MOVDAT    *, ,C  – causes→ (@AR0) → @ACC

MOVDAT     ,B,C  – causes→ (@ACC) → @B
MOVDAT     ,*,C  – causes→ (@ACC) → @AR1
```

CALLING SEQUENCE: MOVDAT [A|*],[B|*][,C]

ENTRY CONDITIONS: $0 \leqslant B + C \leqslant 143$; $0 \leqslant A < 4095$

7

EXIT CONDITIONS: Elements of B contain data from program memory starting at A; AR0 and AR1 may be overwritten

PROGRAM MEMORY REQUIRED: 12 words (+ routines)

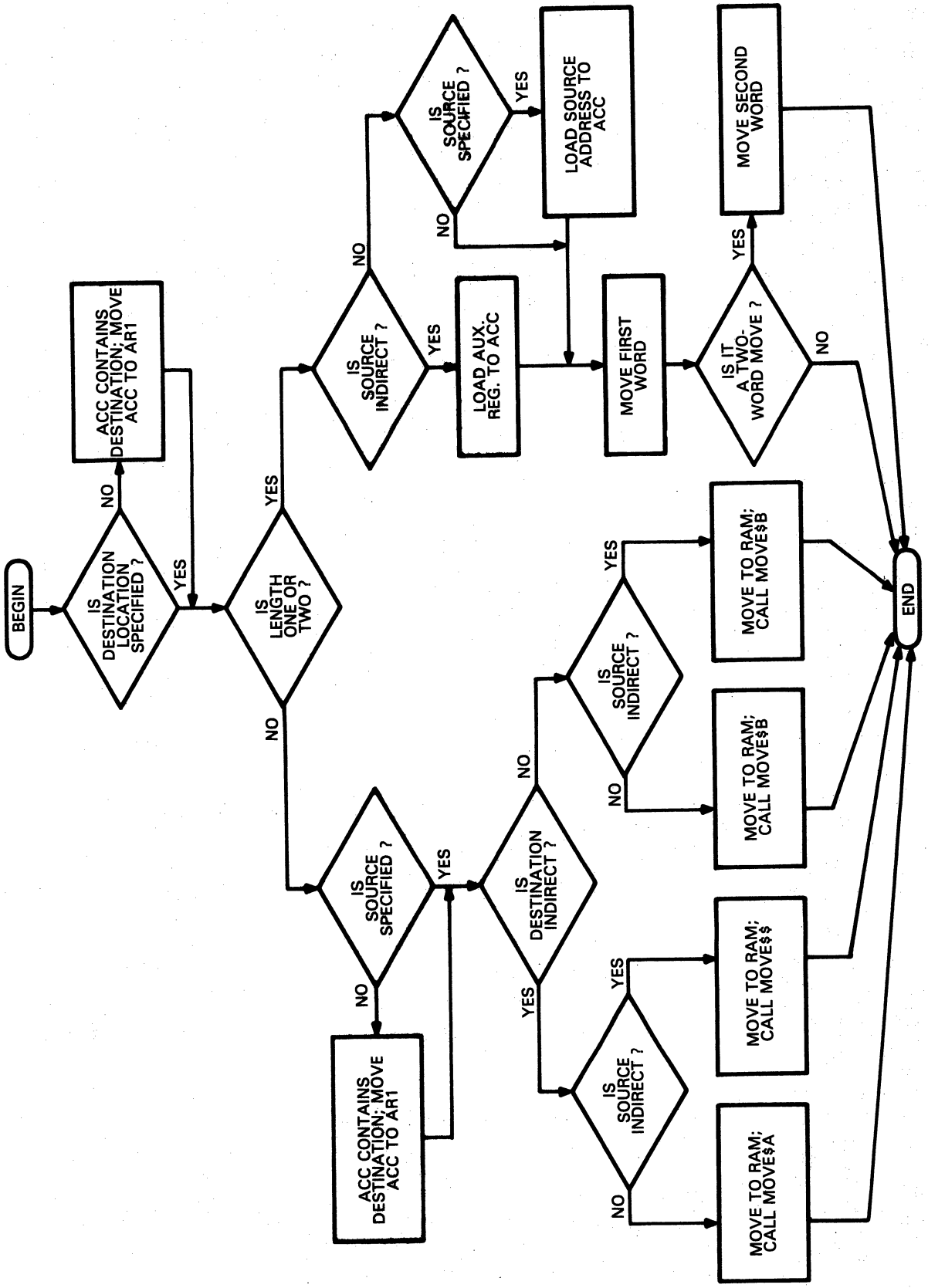
DATA MEMORY REQUIRED: 3 words

STACK REQUIRED: 2 levels

EXECUTION TIME: (max) 31 + (7x length) cycles

FLOWCHART: MOVDAT

MOVDAT



SOURCE:

```
*MOVE L(CONST) WORDS FROM A(ROM ITEM)
*TO B(RAM VAR)
*ROM ITEM IS:
*
MOVDAT $MACRO  A,B,L
       $VAR ST
       $ASG '*' TO ST.S
       $IF  B.L=0
       ACTAR AR1
       $ASG '*' TO B.S
       $ENDIF
       $IF  L.V<3          ONE OR TWO WORDS
       $IF  A.SV=ST.SV     A = *
       ARTAC AR0
       $ELSE
       $IF  A.L#=0         A = PROGRAM ADDRESS
       LCAC :A:
       $ENDIF
       $ENDIF
       $IF  B.SV=ST.SV
       LARP 1
       TBLR *+             READ FIRST WORD
       $ELSE
       TBLR :B:
       $ENDIF
       $IF  L.V=2          TWO WORDS
       ADD  ONE,0          INCREMENT POINTER
       $IF  B.SV=ST.SV
       TBLR *+             READ NEXT WORD
       $ELSE
       TBLR :B:+1
       $ENDIF
       $ENDIF
       $ENDIF
       $IF  L.V>2
       $IF  A.L=0
       ACTAR AR0
       $ASG '*' TO A.S
       $ENDIF
       $IF  B.SV=ST.SV
       $IF  A.SV#=ST.SV
       CALL MOVC$A         MOVE
       REF  MOVC$A
       DATA :A:            FROM :A:
       $ELSE
       CALL MOVC$$         MOVE
       REF  MOVC$$
       $ENDIF
       $ELSE
       $IF  A.SV#=ST.SV
       CALL MOVA$B         MOVE
       REF  MOVA$B
       DATA :A:            FROM :A:
       $ELSE
       CALL MOVC$B         MOVE
       REF  MOVC$B
       $ENDIF
       DATA :B:            TO :B:
       $ENDIF
       DATA :L:            FOR :L: WORDS
       $ENDIF
       $END
```

EXAMPLE 1:

```
0012                    MOVDAT A,B
0001                     LCAC A
0001 0006 7E01           LACK A          LOAD AC WITH A
0002 0007 6708           TBLR B
```

EXAMPLE 2:

```
0014                    MOVDAT *,B,2
0001                     ARTAC AR0
0001 0008 3004"          SAR  AR0,XR0    SAVE AR0
0002 0009 2004"          LAC  XR0,0      LOAD INTO AC
0002 000A 6708           TBLR B
0003 000B 0002"          ADD  ONE,0      INCREMENT POINTER
0004 000C 6709           TBLR B+1
```

EXAMPLE 3:

```
0016                    MOVDAT *,*,2
0001                     ARTAC AR0
0001 000D 3004"          SAR  AR0,XR0    SAVE AR0
0002 000E 2004"          LAC  XR0,0      LOAD INTO AC
0002 000F 6881           LARP 1
0003 0010 67A8           TBLR *+         READ FIRST WORD
0004 0011 0002"          ADD  ONE,0      INCREMENT POINTER
0005 0012 67A8           TBLR *+         READ NEXT WORD
```

EXAMPLE 4:

```
0018                    MOVDAT C,*,B
0001 0013 F800           CALL MOVC$A     MOVE
     0014 0000
0002                     REF  MOVC$A
0003 0015 0000"          DATA C          FROM C
0004 0016 0008           DATA B          FOR B WORDS
```

EXAMPLE 5:

```
0020                    MOVDAT *,,5
0001                     ACTAR AR1
0001 0017 5004"          SACL XR0,0      STORE AC TO XR0
0002 0018 3904"          LAR  AR1,XR0    RE-LOAD AR1
0003 0019 6881           LARP AR1        LOAD AR POINTER
0002 001A F800           CALL MOVC$$     MOVE
     001B 0000
0003                     REF  MOVC$$
0004 001C 0005           DATA 5          FOR 5 WORDS
```

EXAMPLE 6:

```
0022                    MOVDAT ,B
0001 001D 6708           TBLR B
```

EXAMPLE 7:

```
0024                    MOVDAT ,*,5
0001                     ACTAR AR0
0001 001E 5004"          SACL XR0,0      STORE AC TO XR0
0002 001F 3804"          LAR  AR0,XR0    RE-LOAD AR0
0003 0020 6880           LARP AR0        LOAD AR POINTER
0002 0021 F800           CALL MOVC$$     MOVE
     0022 0000
```

```
0003                    REF  MOVC$$
0004 0023 0005          DATA 5              FOR 5 WORDS
```

EXAMPLE 8:

```
0026                   MOVDAT D,*
0001                    LCAC D
0001 0024 F800           CALL LDAC$         LOAD AC WITH:
     0025 0000
0002                     REF  LDAC$
0003 0026 0001"          DATA D             D
0002 0027 6881          LARP 1
0003 0028 67A8          TBLR *+             READ FIRST WORD
```

EXAMPLE 9:

```
0028                   MOVDAT D,,3
0001                    ACTAR AR1
0001 0029 5004"          SACL XR0,0         STORE AC TO XR0
0002 002A 3904"          LAR  AR1,XR0       RE-LOAD AR1
0003 002B 6881           LARP AR1           LOAD AR POINTER
0002 002C F800          CALL MOVC$A         MOVE
     002D 0000
0003                    REF  MOVC$A
0004 002E 0001"         DATA D              FROM D
0005 002F 0003          DATA 3              FOR 3 WORDS
```

EXAMPLE 10:

```
0030                   MOVDAT *,*
0001                    ARTAC AR0
0001 0030 3004"          SAR  AR0,XR0       SAVE AR0
0002 0031 2004"          LAC  XR0,0         LOAD INTO AC
0002 0032 6881          LARP 1
0003 0033 67A8          TBLR *+             READ FIRST WORD
```

EXAMPLE 11:

```
0032                   MOVDAT *,*,9
0001 0034 F800          CALL MOVC$$        MOVE
     0035 0000
0002                    REF  MOVC$$
0003 0036 0009          DATA 9              FOR 9 WORDS
```

TITLE: Move Data Array

NAME: MOVE

OBJECTIVE: Copy data from one array to another in data memory.

ALGORITHM: For number of elements in array,
(A[i]) → B[i]

CALLING SEQUENCE: MOVE A,B,length

ENTRY CONDITIONS: $0 \leqslant A + length \leqslant 143; 0 \leqslant B + length \leqslant 143$

EXIT CONDITIONS: Elements of B contain corresponding elements of A;
AR0 or AR1 may be overwritten

PROGRAM MEMORY REQUIRED: 5 – 7 words (+ MOV$ routines)

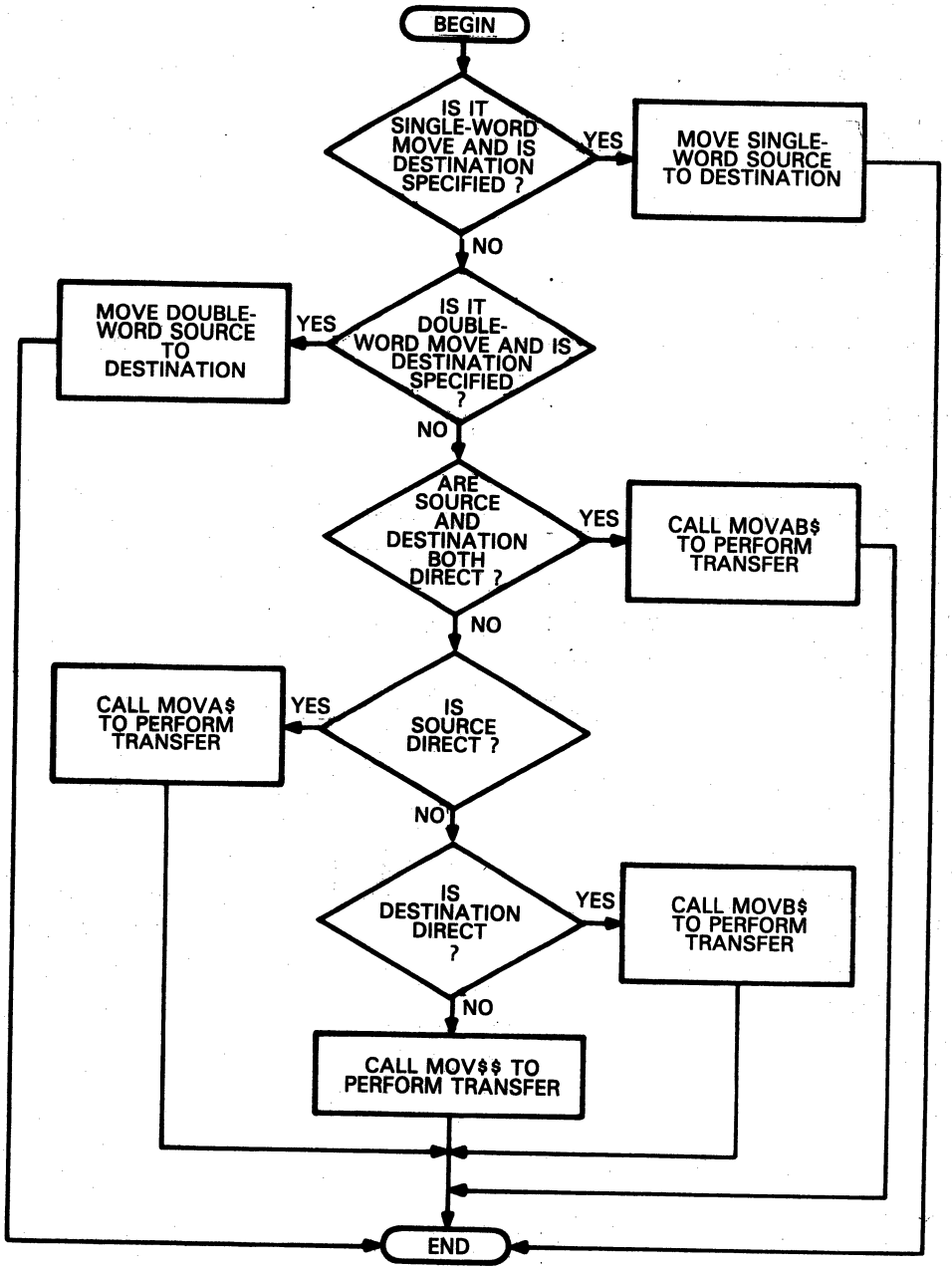
DATA MEMORY REQUIRED: 1 – 3 words

STACK REQUIRED: 2 levels

EXECUTION TIME: (max) 29 + (7 x length) cycles

7

FLOWCHART: MOVE



SOURCE:

```
*MOVE L(CONST) WORDS FROM A(RAM VAR)
*TO B(RAM VAR)
*
MOVE    $MACRO   A,B,L
        $IF  (L.V<2)&(B.L#=0)
        MOV  :A:,:B:       MOVE SINGLE
        $ENDIF
        $IF  (L.V=2)&(B.L#=0)
        MOVX :A:,:B:       MOVE DOUBLE
        $ENDIF
```

```
$IF  (L.V>2)++(B.L=0)
$VAR ST
$ASG '*' TO ST.S
$IF  (A.L#=0)&(B.L#=0)
$IF  (A.SV#=ST.SV)&(B.SV#=ST.SV)
CALL MOVAB$          MOVE
REF  MOVAB$
DATA :A:             FROM :A:
DATA :B:             TO :B:
DATA :L.V:           FOR :L.V: WORDS
$ENDIF
$ENDIF
$IF  (A.SV#=ST.SV)&(A.L#=0)
$IF  (B.L=0)++(B.SV=ST.SV)
$IF  B.L=0
ACTAR AR1            AC TO AR1
$ENDIF
CALL MOVA$           MOVE
REF  MOVA$
DATA :A:             FROM :A:
DATA :L.V:           FOR :L.V: WORDS
$ENDIF
$ENDIF
$IF  (B.SV#=ST.SV)&(B.L#=0)
$IF  (A.L=0)++(A.SV=ST.SV)
$IF  A.L=0
ACTAR AR0            MOVE AC TO AR0
$ENDIF
CALL MOVB$           MOVE
REF  MOVB$
DATA :B:             TO :B:
DATA :L.V:           FOR :L.V: WORDS
$ENDIF
$ENDIF
$IF  (A.L=0)++(A.SV=ST.SV)
$IF  (B.L=0)++(B.SV=ST.SV)
$IF  A.L=0
ACTAR AR0            AC TO AR0
$ENDIF
$IF  B.L=0
ACTAR AR1            AC TO AR1
$ENDIF
CALL MOV$$           MOVE
REF  MOV$$
DATA :L.V:           FOR :L.V: WORDS
$ENDIF
$ENDIF
$ENDIF
$END
```

EXAMPLE 1:

```
0012                 MOVE   A,B
0001                  MOV  A,B          MOVE SINGLE
0001 0006 2001         LAC  A,0         LOAD A
0002 0007 5008         SACL B,0         STORE B
```

EXAMPLE 2:

```
0014                 MOVE   *,B,2
0001                  MOVX *,B         MOVE DOUBLE
0001                   LDAX *          LOAD DOUBLE *
0001 0008 65A8           ZALH *+       LOAD HIGH
0002 0009 6198           ADDS *-       LOAD LOW '*'
```

```
0002                    SACX B          STORE DOUBLE *
0001 000A 5808          SACH B,0        STORE HIGH
0002 000B 5009          SACL B+1,0      STORE LOW
```

EXAMPLE 3:

```
0016                  MOVE   C,*,B
0001 000C F800         CALL MOVA$       MOVE
     000D 0000
0002                   REF  MOVA$
0003 000E 0000"        DATA C           FROM C
0004 000F 0008         DATA 8           FOR 8 WORDS
```

EXAMPLE 4:

```
0018                  MOVE   *,,5
0001                   ACTAR AR1        AC TO AR1
0001 0010 5004"         SACL XR0,0      STORE AC TO XR0
0002 0011 3904"         LAR  AR1,XR0    RE-LOAD AR1
0003 0012 6881          LARP AR1        LOAD AR POINTER
0002 0013 F800         CALL MOV$$       MOVE
     0014 0000
0003                   REF  MOV$$
0004 0015 0005         DATA 5           FOR 5 WORDS
```

EXAMPLE 5:

```
0020                  MOVE    ,B
0001                   MOV   ,B         MOVE SINGLE
0001 0016 5004"         SACL XR0,0      SAVE AC
0002 0017 3804"         LAR  AR0,XR0    LOAD TO AR0
0003 0018 6880          LARP AR0        SELECT AR0
0004 0019 2088          LAC  *,0        LOAD *
0005 001A 5008          SACL B,0        STORE B
```

EXAMPLE 6:

```
0022                  MOVE    ,*,5
0001                   ACTAR AR0        AC TO AR0
0001 001B 5004"         SACL XR0,0      STORE AC TO XR0
0002 001C 3804"         LAR  AR0,XR0    RE-LOAD AR0
0003 001D 6880          LARP AR0        LOAD AR POINTER
0002 001E F800         CALL MOV$$       MOVE
     001F 0000
0003                   REF  MOV$$
0004 0020 0005         DATA 5           FOR 5 WORDS
```

EXAMPLE 7:

```
0024                  MOVE   D,*
0001                   MOV  D,*         MOVE SINGLE
0001 0021 2001"         LAC  D,0        LOAD D
0002 0022 5088          SACL *,0        STORE *
```

EXAMPLE 8:

```
0026                  MOVE   D,,3
0001                   ACTAR AR1        AC TO AR1
0001 0023 5004"         SACL XR0,0      STORE AC TO XR0
0002 0024 3904"         LAR  AR1,XR0    RE-LOAD AR1
0003 0025 6881          LARP AR1        LOAD AR POINTER
0002 0026 F800         CALL MOVA$       MOVE
```

```
     0027 0000
0003                 REF  MOVA$
0004 0028 0001"      DATA D              FROM D
0005 0029 0003       DATA 3              FOR 3 WORDS
```

TITLE: Move Words to Program Memory

NAME: MOVROM

OBJECTIVE: Copy data from data memory to program memory

ALGORITHM: For number of elements in array,

```
MOVROM    A,B,C – causes→ (A) → @B
MOVROM    A,*,C – causes→ (A) → @AR1
MOVROM    A, ,C – causes→ (A) → @ACC

MOVROM    *,B,C – causes→ (@AR0) → @B
MOVROM    *,*,C – causes→ (@AR0) → @AR1
MOVROM    *, ,C – causes→ (@AR0) → @ACC

MOVROM     ,B,C – causes→ (@ACC) → @B
MOVROM     ,*,C – causes→ (@ACC) → @AR1
```

CALLING SEQUENCE: MOVROM [A,*],[B,*][,length]

ENTRY CONDITIONS: $0 \leq A + length \leq 143$; $0 \leq B \leq 4095$

7

EXIT CONDITIONS: Program memory starting at B contains data elements starting at A; AR0 and AR1 may be overwritten

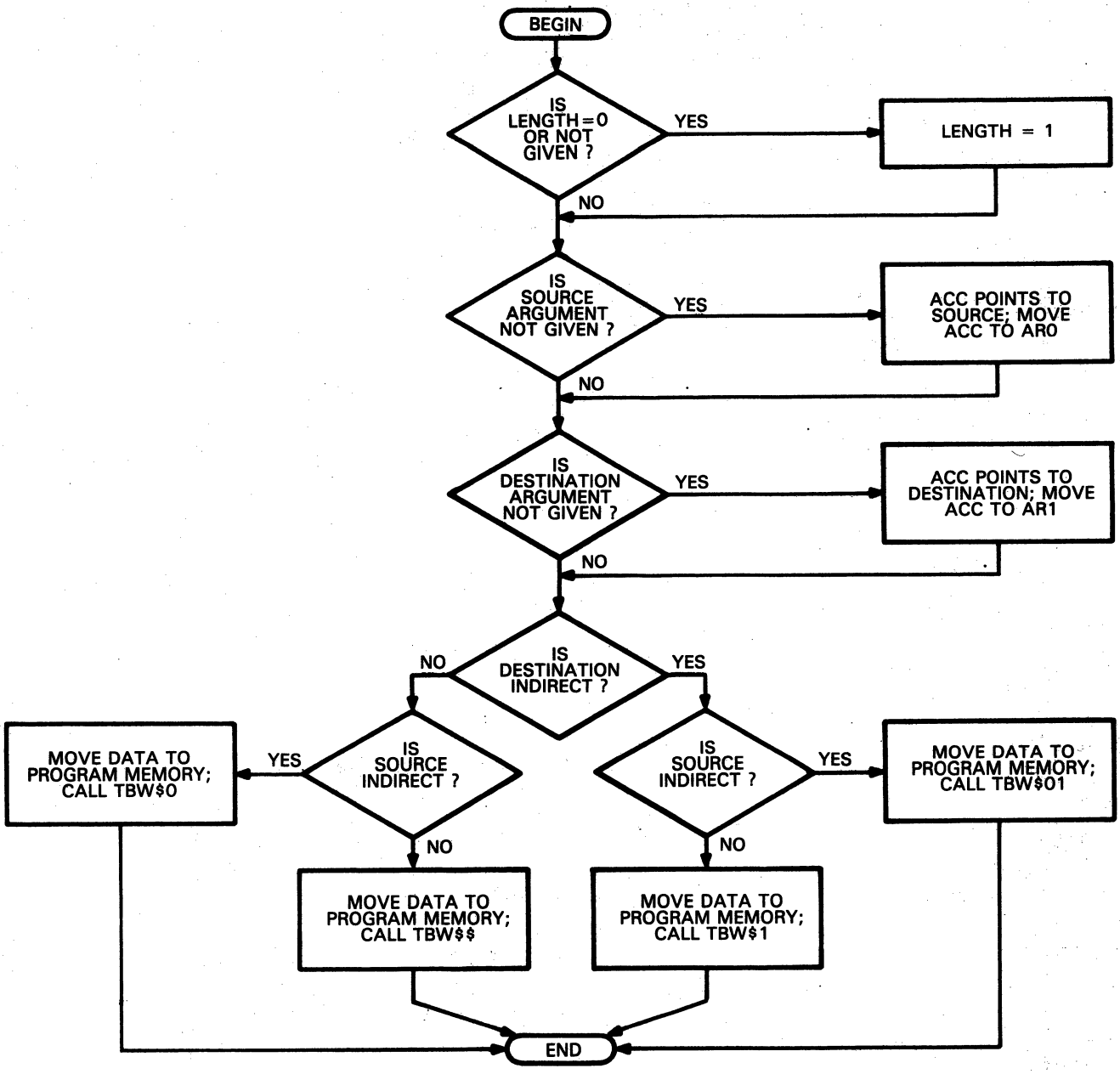
PROGRAM MEMORY REQUIRED: 8 words (+ TBW$ routines)

DATA MEMORY REQUIRED: 3 words

STACK REQUIRED: 2 levels

EXECUTION TIME: (max) 31 + (7 x length) cycles

FLOWCHART: MOVROM



SOURCE:

```
*MOVE L(CONST) WORDS FROM A(RAM VAR)
*TO B(ROM VAR)
*
MOVROM $MACRO  A,B,L
       $VAR ST
       $ASG '*' TO ST.S
       $IF  L.V=0          DEFAULT 0 TO 1
       $ASG 1 TO L.V
       $ENDIF
       $IF  A.L=0
       ACTAR AR0           AC TO AR0
       $ENDIF
       $IF  B.L=0
```

```
        ACTAR AR1           AC TO AR1
        $ENDIF
        $IF   (B.SV=ST.SV)++(B.L=0)
        $IF   (A.SV=ST.SV)++(A.L=0)
        CALL TBW$01         MOVE RAM->ROM
        REF   TBW$01
        DATA :L.V:          FOR :L.V: WORDS
        $ELSE
        CALL TBW$1          MOVE RAM->ROM
        REF   TBW$1
        DATA :A:            FROM :A:
        DATA :L.V:          FOR :L.V: WORDS
        $ENDIF
        $ELSE
        $IF   (A.SV=ST.SV)++(A.L=0)
        CALL TBW$0          MOVE RAM->ROM
        REF   TBW$0
        DATA :B:            TO :B:
        DATA :L.V:          FOR :L.V: WORDS
        $ELSE
        CALL TBW$$          MOVE RAM->ROM
        REF   TBW$$
        DATA :A:            FROM :A:
        DATA :B:            TO :B:
        DATA :L.V:          FOR :L.V: WORDS
        $ENDIF
        $ENDIF
        $END
```

EXAMPLE 1:

```
0012                    MOVROM A,B
0001 0006 F800           CALL TBW$$         MOVE RAM->ROM
     0007 0000
0002                     REF  TBW$$
0003 0008 0001           DATA A             FROM A
0004 0009 0008           DATA B             TO B
0005 000A 0001           DATA 1             FOR 1 WORDS
```

EXAMPLE 2:

```
0014                    MOVROM *,B,2
0001 000B F800           CALL TBW$0         MOVE RAM->ROM
     000C 0000
0002                     REF  TBW$0
0003 000D 0008           DATA B             TO B
0004 000E 0002           DATA 2             FOR 2 WORDS
```

EXAMPLE 3:

```
0016                    MOVROM C,*,B
0001 000F F800           CALL TBW$1         MOVE RAM->ROM
     0010 0000
0002                     REF  TBW$1
0003 0011 0000"          DATA C             FROM C
0004 0012 0008           DATA 8             FOR 8 WORDS
```

EXAMPLE 4:

```
0018                    MOVROM *,,5
0001                     ACTAR AR1          AC TO AR1
0001 0013 5004"           SACL XR0,0        STORE AC TO XR0
0002 0014 3904"           LAR  AR1,XR0      RE-LOAD AR1
```

```
0003 0015 6881          LARP AR1          LOAD AR POINTER
0002 0016 F800          CALL TBW$01       MOVE RAM->ROM
     0017 0000
0003                    REF  TBW$01
0004 0018 0005          DATA 5            FOR 5 WORDS
```

EXAMPLE 5:

```
0020                    MOVROM ,B
0001                     ACTAR AR0        AC TO AR0
0001 0019 5004"           SACL XR0,0      STORE AC TO XR0
0002 001A 3804"           LAR  AR0,XR0    RE-LOAD AR0
0003 001B 6880            LARP AR0        LOAD AR POINTER
0002 001C F800           CALL TBW$0       MOVE RAM->ROM
     001D 0000
0003                     REF  TBW$0
0004 001E 0008           DATA B           TO B
0005 001F 0001           DATA 1           FOR 1 WORDS
```

EXAMPLE 6:

```
0022                    MOVROM ,*,5
0001                     ACTAR AR0        AC TO AR0
0001 0020 5004"           SACL XR0,0      STORE AC TO XR0
0002 0021 3804"           LAR  AR0,XR0    RE-LOAD AR0
0003 0022 6880            LARP AR0        LOAD AR POINTER
0002 0023 F800           CALL TBW$01      MOVE RAM->ROM
     0024 0000
0003                     REF  TBW$01
0004 0025 0005           DATA 5           FOR 5 WORDS
```

EXAMPLE 7:

```
0024                    MOVROM D,*
0001 0026 F800           CALL TBW$1       MOVE RAM->ROM
     0027 0000
0002                     REF  TBW$1
0003 0028 0001"          DATA D           FROM D
0004 0029 0001           DATA 1           FOR 1 WORDS
```

EXAMPLE 8:

```
0026                    MOVROM D,,3
0001                     ACTAR AR1        AC TO AR1
0001 002A 5004"           SACL XR0,0      STORE AC TO XR0
0002 002B 3904"           LAR  AR1,XR0    RE-LOAD AR1
0003 002C 6881            LARP AR1        LOAD AR POINTER
0002 002D F800           CALL TBW$1       MOVE RAM->ROM
     002E 0000
0003                     REF  TBW$1
0004 002F 0001"          DATA D           FROM D
0005 0030 0003           DATA 3           FOR 3 WORDS
```

EXAMPLE 9:

```
0028                    MOVROM *,*
0001 0031 F800           CALL TBW$01      MOVE RAM->ROM
     0032 0000
0002                     REF  TBW$01
0003 0033 0001           DATA 1           FOR 1 WORDS
```

EXAMPLE 10:

```
0030                    MOVROM *,*,1
0001 0034 F800           CALL TBW$01        MOVE RAM->ROM
     0035 0000
0002                     REF  TBW$01
0003 0036 0001           DATA 1             FOR 1 WORDS
```

TITLE: Move Double Word

NAME: MOVX

OBJECTIVE: Copy double word from one location to another in data memory

ALGORITHM: (A:A + 1) → B:B + 1 or
(@ACC:@ACC + 1) → B:B + B

CALLING SEQUENCE: MOVX [A],B

ENTRY CONDITIONS: $0 \leqslant A \leqslant 126; 0 \leqslant B \leqslant 126$

EXIT CONDITIONS: Double word at B contains value of double word located at A; AR0 may be overwritten

PROGRAM MEMORY REQUIRED: 4 – 8 words

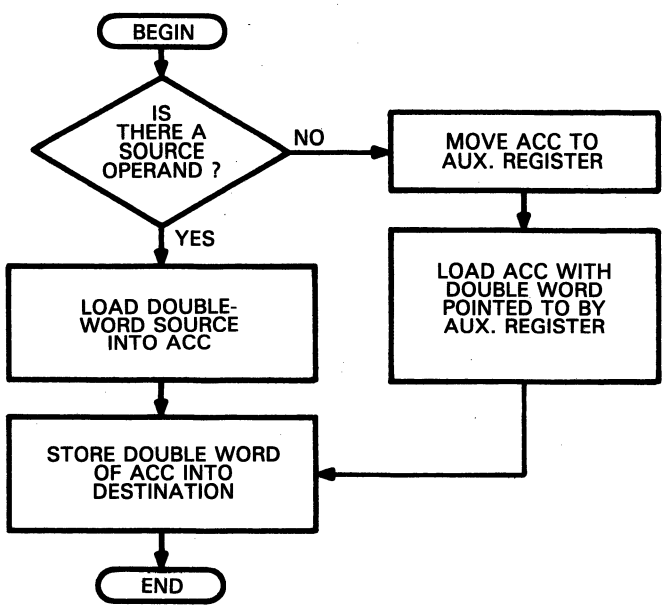
DATA MEMORY REQUIRED: 0 – 2 words

STACK REQUIRED: None

EXECUTION TIME: 4 – 8 cycles

7

FLOWCHART: MOVX



SOURCE:

```
*MOVE DOUBLE FROM A TO B
*
MOVX    $MACRO   A,B        MOVE DOUBLE
        $IF  A.L=0          A IN AC
        SACH XR0,0
        SACL XR1,0          SAVE AC TO XR0
        LAR  AR0,XR0        TO AR0
        LARP AR0            SELECT AR0
        LDAX *              LOAD *
        $ELSE
        LDAX :A:            LOAD DOUBLE :A:
        $ENDIF
        SACX :B:            STORE DOUBLE :A:
        $END
```

EXAMPLE 1:

```
0011                    MOVX   A,B
0001                     LDAX A             LOAD DOUBLE A
0001 0006 6501           ZALH A             LOAD HIGH A
0002 0007 6102           ADDS A+1           LOAD LOW A
0002                    SACX B              STORE DOUBLE A
0001 0008 5808           SACH B,0           STORE HIGH
0002 0009 5009           SACL B+1,0         STORE LOW
```

EXAMPLE 2:

```
0013                    MOVX   *,B
0001                     LDAX *             LOAD DOUBLE *
0001 000A 65A8           ZALH *+            LOAD HIGH
0002 000B 6198           ADDS *-            LOAD LOW '*'
0002                    SACX B              STORE DOUBLE *
0001 000C 5808           SACH B,0           STORE HIGH
0002 000D 5009           SACL B+1,0         STORE LOW
```

EXAMPLE 3:

```
0015                    MOVX   C,*+
0001                     LDAX C             LOAD DOUBLE C
0001 000E 6500"          ZALH C             LOAD HIGH C
0002 000F 6101"          ADDS C+1           LOAD LOW C
0002                    SACX *+             STORE DOUBLE C
0001 0010 58A8           SACH *+,0          STORE HIGH
0002 0011 50A8           SACL *+,0          STORE LOW
```

EXAMPLE 4:

```
0017                    MOVX    ,D
0001 0012 5806"          SACH XR0,0
0002 0013 5007"          SACL XR1,0         SAVE AC TO XR0
0003 0014 3806"          LAR  AR0,XR0       TO AR0
0004 0015 6880           LARP AR0           SELECT AR0
0005                     LDAX *             LOAD *
0001 0016 65A8           ZALH *+            LOAD HIGH
0002 0017 6198           ADDS *-            LOAD LOW '*'
0006                    SACX D              STORE DOUBLE
0001 0018 5802"          SACH D,0           STORE HIGH
0002 0019 5003"          SACL D+1,0         STORE LOW
```

EXAMPLE 5:

```
0019                    MOVX   *-,B
0001                     LDAX *-            LOAD DOUBLE *-
0001 001A 6698           ZALS *-            LOAD LOW
0002 001B 6098           ADDH *-            LOAD HIGH '*-'
0002                     SACX B             STORE DOUBLE *-
0001 001C 5808           SACH B,0           STORE HIGH
0002 001D 5009           SACL B+1,0         STORE LOW
```

EXAMPLE 6:

```
0021                    MOVX   *+,A
0001                     LDAX *+            LOAD DOUBLE *+
0001 001E 65A8           ZALH *+            LOAD HIGH
0002 001F 61A8           ADDS *+            LOAD LOW '*+'
0002                     SACX A             STORE DOUBLE *+
0001 0020 5801           SACH A,0           STORE HIGH
0002 0021 5002           SACL A+1,0         STORE LOW
```

EXAMPLE 7:

```
0023                    MOVX   D,*-
0001                     LDAX D             LOAD DOUBLE D
0001 0022 6502"          ZALH D             LOAD HIGH D
0002 0023 6103"          ADDS D+1           LOAD LOW D
0002                     SACX *-            STORE DOUBLE D
0001 0024 5098           SACL *-,0          STORE LOW
0002 0025 5898           SACH *-,0          STORE HIGH
```

TITLE: Arithmetic Negation

NAME: NEG

OBJECTIVE: Find negative value of argument

ALGORITHM: $-(A) \rightarrow A$

CALLING SEQUENCE: NEG A

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127$

EXIT CONDITIONS: Data word A contains the negative of its previous value

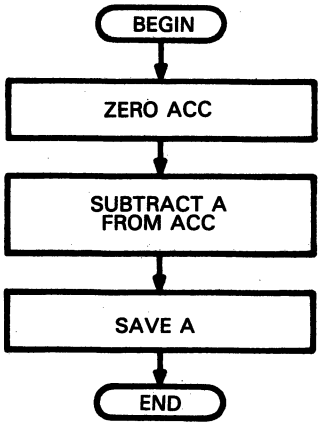
PROGRAM MEMORY REQUIRED: 3 words

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 3 cycles

FLOWCHART: NEG



SOURCE:

```
*NEGATE VAR A
*
NEG     $MACRO  A          NEGATE
        ZAC                ZERO AC
        SUB   :A:,0        SUBTRACT :A:
        SACL  :A:,0        RESTORE
        $END
```

EXAMPLE:

```
0015                    NEG D
0001 000C 7F89           ZAC              ZERO AC
0002 000D 1001"          SUB  D,0         SUBTRACT D
0003 000E 5001"          SACL D,0         RESTORE
```

TITLE: Double-Word Arithmetic Negation

NAME: NEGX

OBJECTIVE: Find negative value of double-word argument

ALGORITHM:

NEGX *	– causes→	– (@AR:@AR + 1) →@AR + 1
NEGX * –	– causes→	– (@AR – 1:@AR) → @AR – 1:@AR (AR) – 2 → AR
NEGX * +	– causes→	– (@AR:@AR + 1) → @AR:@AR + 1 (AR) + 2 → AR
NEGX A	– causes→	– (A:A + 1) → A:A + 1

CALLING SEQUENCE: NEGX {A,*,* – ,* + }

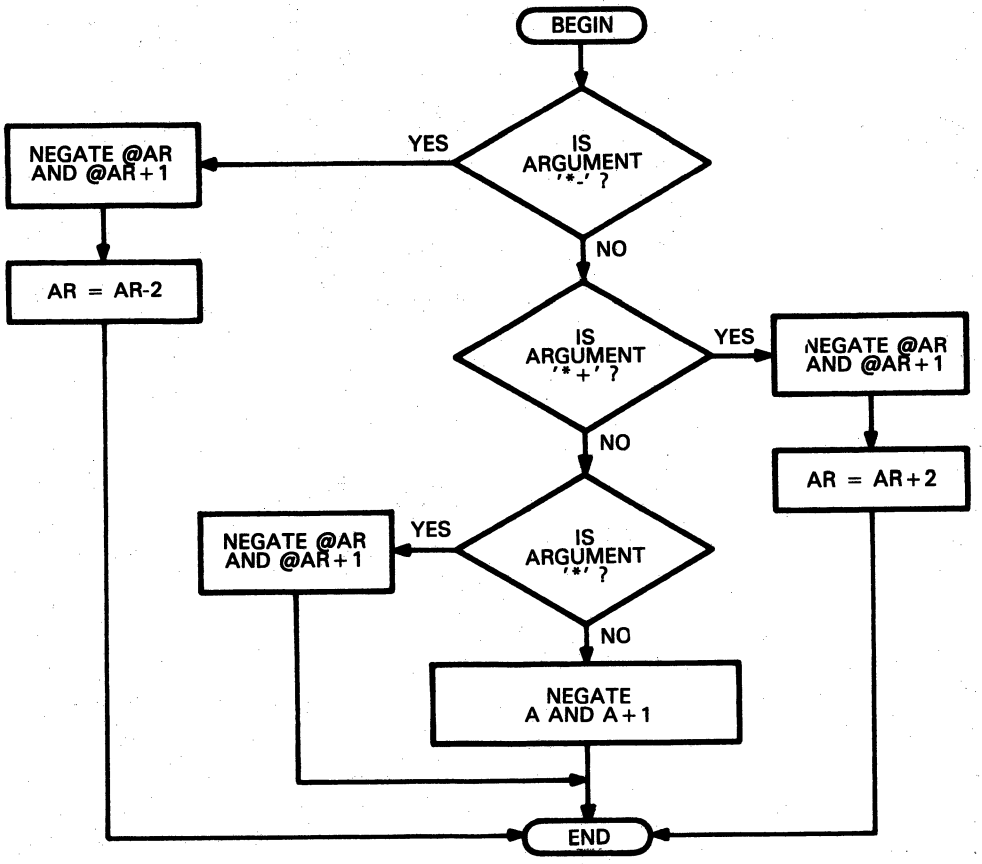
ENTRY CONDITIONS: 0 ⩽ A ⩽ 127

EXIT CONDITIONS: Specified data words contain negative of previous value; auxiliary register is updated as necessary

7

PROGRAM MEMORY REQUIRED:	5 words	**DATA MEMORY REQUIRED:**	None
STACK REQUIRED:	None	**EXECUTION TIME:**	5 cycles

FLOWCHART: NEGX



SOURCE:

```
*NEGATE DOUBLE WORD
*
NEGX    $MACRO A            NEGATE DOUBLE
        $VAR ST,SP,SM
        $ASG '*+' TO SP.S
        $ASG '*-' TO SM.S
        $ASG '*' TO ST.S
        ZAC
        $IF A.SV=SM.SV
        SUBS *-
        SUBH *+             SUBTRACT  '*-'
        SACX *-             SAVE '*-'
        $ELSE
        $IF A.SV=SP.SV
        SUBX *              SUBTRACT  '*'
        SACX *+             SAVE '*+'
        $ELSE
        $IF A.SV=ST.SV
        SUBX *              SUBTRACT  '*'
        SACX *              SAVE '*'
        $ELSE
        SUBX :A:            SUBTRACT  :A:
        SACX :A:            SAVE :A:
        $ENDIF
        $END
```

EXAMPLE 1:

```
0011                    NEGX A
0001 0006 7F89           ZAC
0002                     SUBX A             SUBTRACT  A
0001 0007 6207            SUBH A            SUBTRACT HIGH
0002 0008 6308            SUBS A+1          SUBTRACT LOW
0003                     SACX A             SAVE A
0001 0009 5807            SACH A,0          STORE HIGH
0002 000A 5008            SACL A+1,0        STORE LOW
```

EXAMPLE 2:

```
0013                    NEGX *
0001 000B 7F89           ZAC
0002                     SUBX *             SUBTRACT  '*'
0001 000C 62A8            SUBH *+           SUBTRACT HIGH
0002 000D 6398            SUBS *-           SUBTRACT LOW
0003                     SACX *             SAVE '*'
0001 000E 58A8            SACH *+,0         STORE HIGH
0002 000F 5098            SACL *-,0         STORE LOW
```

EXAMPLE 3:

```
0015                    NEGX *-
0001 0010 7F89           ZAC
0002 0011 6398           SUBS *-
0003 0012 62A8           SUBH *+           SUBTRACT  '*-'
0004                     SACX *-           SAVE '*-'
0001 0013 5098            SACL *-,0        STORE LOW
0002 0014 5898            SACH *-,0        STORE HIGH
```

EXAMPLE 4:

```
0017                    NEGX *+
0001 0015 7F89           ZAC
0002                     SUBX *             SUBTRACT  '*'
0001 0016 62A8            SUBH *+           SUBTRACT HIGH
0002 0017 6398            SUBS *-           SUBTRACT LOW
0003                     SACX *+            SAVE '*+'
0001 0018 58A8            SACH *+,0         STORE HIGH
0002 0019 50A8            SACL *+,0         STORE LOW
```

TITLE: Boolean Not

NAME: NOT

OBJECTIVE: Calculate one's complement of accumulator or data word

ALGORITHM: (A) .XOR. –1 → A

CALLING SEQUENCE: NOT [A]

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127$

EXIT CONDITIONS: A (accumulator) contains one's complement of previous value

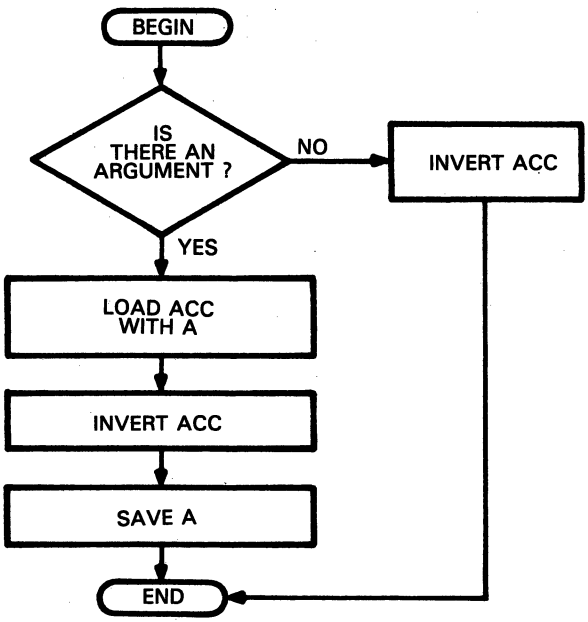
PROGRAM MEMORY REQUIRED: 3 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 1 – 3 cycles

FLOWCHART: NOT



SOURCE:

```
*NOT AC OR WORD A
*
NOT     $MACRO  A          INVERT
        $IF  A.L#=0
        LAC  :A:,0         LOAD AC
        XOR  MINUS         INVERT
        SACL :A:,0         RESTORE
        $ELSE
        XOR  MINUS         INVERT
        $ENDIF
        $END
```

EXAMPLE 1:

```
0011                    NOT
0001 0006 7803"          XOR  MINUS        INVERT
```

EXAMPLE 2:

```
0017                    NOT C
0001 000D 2000"          LAC  C,0          LOAD AC
0002 000E 7803"          XOR  MINUS        INVERT
0003 000F 5000"          SACL C,0          RESTORE
```

TITLE: Arithmetic Right Shift

NAME: RASH

OBJECTIVE: Move shifted data from one location to another in data memory

ALGORITHM: (A) * 2 – shift → B

CALLING SEQUENCE: RASH A,B,shift

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127$; $0 \leqslant B \leqslant 127$; $0 \leqslant shift < 16$

EXIT CONDITIONS: B contains shifted value of A

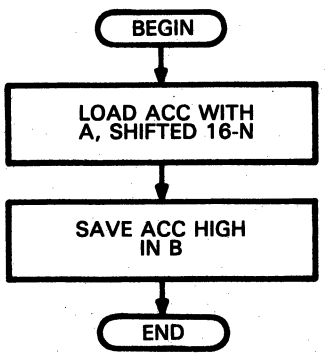
PROGRAM MEMORY REQUIRED: 2 words

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 2 cycles

FLOWCHART: RASH



SOURCE:

```
*MOVE A TO B (SINGLE-VAR) WITH N (CONST) BIT
*RIGHT ARITHMETIC SHIFT
*
RASH  $MACRO A,B,N      MOVE WITH RIGHT ARITH. SHIFT
      LAC  :A:,16-:N:   LOAD :A: RIGHT SHIFT
      SACH :B:,0        STORE HIGH TO :B:
      $END
```

EXAMPLE:

```
0011                    RASH A,B,3
0001 0006 2D07           LAC  A,16-3        LOAD A RIGHT SHIFT
0002 0007 5808           SACH B,0           STORE HIGH TO B
```

TITLE: Double-Word Arithmetic Right Shift

NAME: RASX

OBJECTIVE: Move shifted double word from one location to another in data memory

ALGORITHM: $(A:A+1) * 2^{shift} \rightarrow B:B+1$

CALLING SEQUENCE: RASX A,B,shift

ENTRY CONDITIONS: $0 \leqslant A \leqslant 126$; $0 \leqslant B \leqslant 126$; $0 \leqslant shift < 16$

EXIT CONDITIONS: Double word at B contains shifted value of double word at A

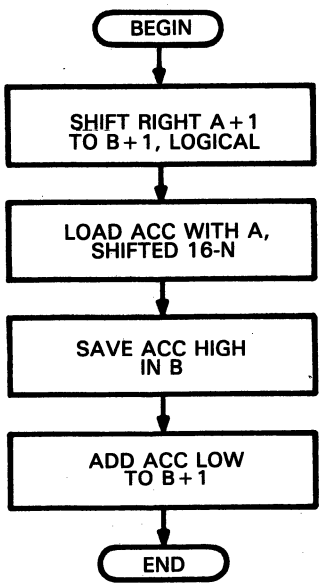
PROGRAM MEMORY REQUIRED: 10 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 10 cycles

FLOWCHART: RASX



SOURCE:

```
*MOVE A TO B (DOUBLE VAR) WITH N (CONST) BIT
*RIGHT ARITHMETIC SHIFT
*
RASX  $MACRO  A,B,N     MOVE DOUBLE WITH ARITH. SHIFT
```

```
        RLSH :A:+1,:B:+1,:N:
        LAC  :A:,16-:N:    LOAD HIGH, RIGHT SHIFT
        SACH :B:,0         SAVE IN :B: HIGH
        OR   :B:+1         COMBINE WITH :B: LOW
        SACL :B:+1,0       SAVE BACK
        $END
```

EXAMPLE:

```
0011                    RASX A,B,3
0001                     RLSH A+1,B+1,3
0001 0006 2D08            LAC  A+1,16-3     LOAD, RIGHT SHIFT
0002 0007 580A            SACH B+1,0        SAVE HIGH PART
0003 0008 2D03"           LAC  MINUS,16-3   GET MASK
0004                      NOT
0001 0009 7803"            XOR  MINUS       INVERT
0005 000A 790A            AND  B+1          APPLY MASK
0006 000B 500A            SACL B+1,0        STORE BACK TO B+1
0002 000C 2D07           LAC  A,16-3        LOAD HIGH, RIGHT SHIFT
0003 000D 5809           SACH B,0           SAVE IN B HIGH
0004 000E 7A0A           OR   B+1           COMBINE WITH B LOW
0005 000F 500A           SACL B+1,0         SAVE BACK
```

TITLE: Move One-Word Constant into Array

NAME: REPCON

OBJECTIVE: Initialize an array in data memory with a constant

ALGORITHM: Constant → ACC
For number of elements in array,
(ACC) → data memory

CALLING SEQUENCE: REPCON constant,array,length

ENTRY CONDITIONS: $-32768 \leq$ constant ≤ 32767; $0 \leq$ array + length ≤ 143

EXIT CONDITIONS: Array contains constant in each location

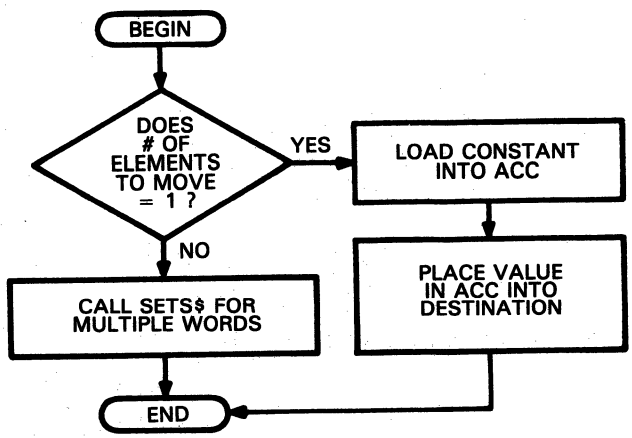
PROGRAM MEMORY REQUIRED: 2 – 4 words (+ SETS$ and LAC$ routines)

STACK REQUIRED: 2 levels

DATA MEMORY REQUIRED: 0 – 3 words

EXECUTION TIME: (max) 27 + (4 x length) cycles

FLOWCHART: REPCON



SOURCE:

```
*REPLICATE CONSTANTS
*A IS A CONSTANT
*B IS A MEM LOCATION
*L IS LENGTH TO REPLICATE
*
REPCON $MACRO  A,B,L
       $IF  L.V<2
       LCAC :A:          LOAD CONSTANT
       SACL :B:,0        SET IT
       $ELSE
       CALL SETS$        CALL FOR SET MEMORY
       REF  SETS$
       DATA :A:          CONSTANT
       DATA :L:          LENGTH
       DATA :B:          DESTINATION
       $ENDIF
       $END
```

EXAMPLE 1:

```
0014                    REPCON -252,A,10
0001 000B F800           CALL SETS$        CALL FOR SET MEMORY
     000C 0000
0002                     REF  SETS$
0003 000D FF04           DATA -252         CONSTANT
0004 000E 000A           DATA 10           LENGTH
0005 000F 0001           DATA A            DESTINATION
```

EXAMPLE 2:

```
0016                    REPCON 2,B,1
0001                     LCAC 2            LOAD CONSTANT
0001       0002  V$1 EQU 2
0002 0010 7E02            LACK V$1         LOAD AC WITH V$1
0002 0011 5008            SACL B,0         SET IT
```

TITLE: Ripple Data Array One Position

NAME: RIPPLE

OBJECTIVE: Move each element of array in data memory to next higher location

ALGORITHM: (array element N – 1) → array element N
(array element N – 2) → array element N – 1
:
:
(array element 2) → array element 3
(array element 1) → array element 2

CALLING SEQUENCE: RIPPLE array [,length[,inline]]

ENTRY CONDITIONS: 0 ⩽ array + length ⩽ 143; inline = any string

EXIT CONDITIONS: All array elements N contain value of previous location N – 1; AR0 and AR1 may be overwritten

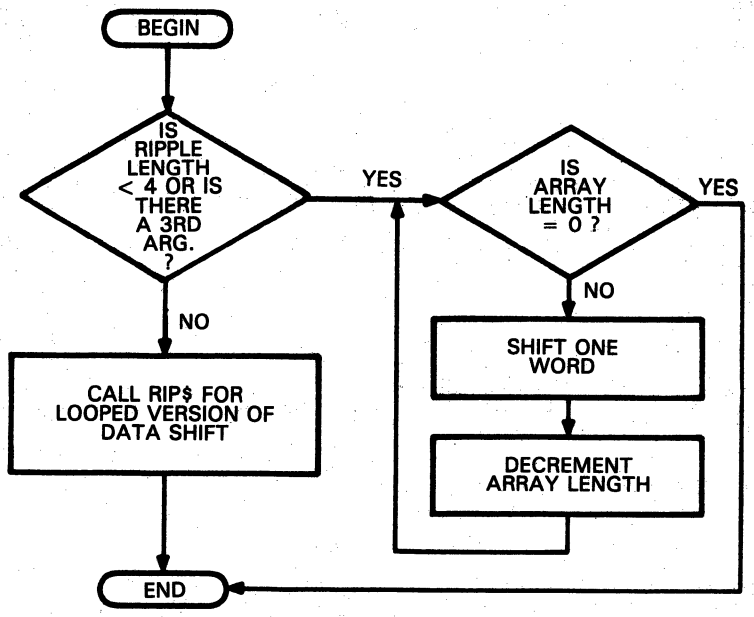
PROGRAM MEMORY REQUIRED: Inline – length words; looped – 4 + RIP$ function (23 words)

DATA MEMORY REQUIRED: 3 words

STACK REQUIRED: 2 levels (looped)

EXECUTION TIME: Inline – length cycles; looped – 30 + (4 * length)

FLOWCHART: RIPPLE



SOURCE 1:

```
RIPPLE $MACRO  A,L,C
       $IF  (L.V<4)++(C.L#=0)
       INRIP :A:,:L:
       $ELSE
       CALL RIP$        CALL FOR RIPPLE LOOP
       REF  RIP$
       DATA :L:         FOR :L:-1 WORDS
       DATA :A:         FROM :A:+:L:-1
       $ENDIF
       $END
```

SOURCE 2:

```
*RIPPLE DOWN ARRAY
*A IS ARRAY LOCATION
*L IS LENGTH OF ARRAY
*
INRIP $MACRO  A,L
       $IF  L.V>16
       INRIP :A:+16,:L:-16
       $ENDIF
       $IF  L.V>15
       DMOV :A:+15
       $ENDIF
       $IF  L.V>14
       DMOV :A:+14
       $ENDIF
       $IF  L.V>13
       DMOV :A:+13
       $ENDIF
       $IF  L.V>12
```

```
        DMOV :A:+12
        $ENDIF
        $IF  L.V>11
        DMOV :A:+11
        $ENDIF
        $IF  L.V>10
        DMOV :A:+10
        $ENDIF
        $IF  L.V>9
        DMOV :A:+9
        $ENDIF
        $IF  L.V>8
        DMOV :A:+8
        $ENDIF
        $IF  L.V>7
        DMOV :A:+7
        $ENDIF
        $IF  L.V>6
        DMOV :A:+6
        $ENDIF
        $IF  L.V>5
        DMOV :A:+5
        $ENDIF
        $IF  L.V>4
        DMOV :A:+4
        $ENDIF
        $IF  L.V>3
        DMOV :A:+3
        $ENDIF
        $IF  L.V>2
        DMOV :A:+2
        $ENDIF
        $IF  L.V>1
        DMOV :A:+1
        $ENDIF
        $IF  L.V>0
        DMOV :A:
        $ENDIF
        $END
```

7

EXAMPLE 1:

```
0007                    RIPPLE A,3
0001                     INRIP A,3
0001 0006 6909            DMOV A+2
0002 0007 6908            DMOV A+1
0003 0008 6907            DMOV A
```

EXAMPLE 2:

```
0009                    RIPPLE A,4
0001 0009 F800           CALL RIP$          CALL FOR RIPPLE LOOP
     000A 0000
0002                     REF  RIP$
0003 000B 0004           DATA 4             FOR 4-1 WORDS
0004 000C 0007           DATA A             FROM A+4-1
```

EXAMPLE 3:

```
0011                    RIPPLE A,5,L
0001                     INRIP A,5
0001 000D 690B            DMOV A+4
0002 000E 690A            DMOV A+3
```

```
0003 000F 6909          DMOV A+2
0004 0010 6908          DMOV A+1
0005 0011 6907          DMOV A
```

TITLE: Right Logical Shift

NAME: RLSH

OBJECTIVE: Move right-shifted data from one location to another in data memory

ALGORITHM: $[(A) * 2^{-shift}]$ .and. $[2^{16-shift}-1] \rightarrow B$

CALLING SEQUENCE: RLSH A,B,shift

ENTRY CONDITIONS: $0 \leq A \leq 127$; $0 \leq B \leq 127$; $0 \leq shift < 16$

EXIT CONDITIONS: B contains shifted value of A

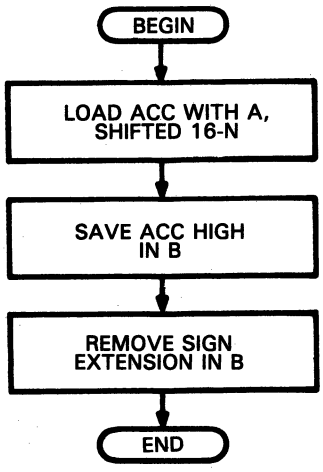
PROGRAM MEMORY REQUIRED: 6 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 6 cycles

FLOWCHART: RLSH



SOURCE:

```
*MOVE A TO B (SINGLE VAR) WITH N (CONST) BIT
*RIGHT LOGICAL SHIFT
*
RLSH  $MACRO A,B,N       MOVE WITH RIGHT LOGICAL SHIFT
      LAC   :A:,16-:N:   LOAD, RIGHT SHIFT
      SACH  :B:,0        SAVE HIGH PART
```

```
        LAC  MINUS,16-:N: GET MASK
        NOT
        AND  :B:          APPLY MASK
        SACL :B:,0        STORE BACK TO :B:
        $END
```

EXAMPLE:

```
0011                    RLSH A,B,3
0001 0006 2D07           LAC  A,16-3        LOAD, RIGHT SHIFT
0002 0007 5808           SACH B,0           SAVE HIGH PART
0003 0008 2D03"          LAC  MINUS,16-3    GET MASK
0004                     NOT
0001 0009 7803"           XOR  MINUS        INVERT
0005 000A 7908           AND  B             APPLY MASK
0006 000B 5008           SACL B,0           STORE BACK TO B
```

TITLE: Double-Word Logical Right Shift

NAME: RLSX

OBJECTIVE: Move right-shifted double word from one location to another in data memory

ALGORITHM: $[(A:A + 1) * 2^{-shift}].and.[2^{16-shift} - 1] \rightarrow B:B + 1$

CALLING SEQUENCE: RLSX A,B,shift

ENTRY CONDITIONS: $0 \leqslant A \leqslant 126$; $0 \leqslant B \leqslant 126$; $0 \leqslant shift < 16$

EXIT CONDITIONS: Double word at B contains shifted value of double word at A

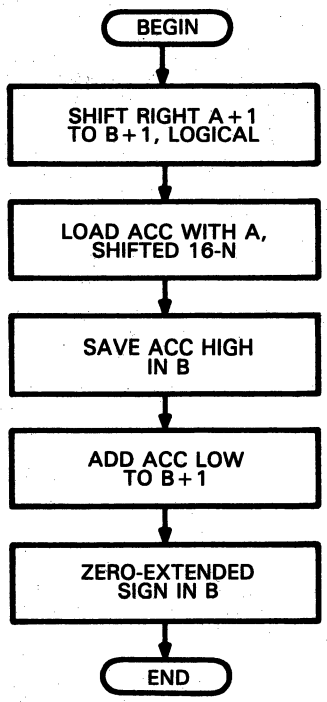
PROGRAM MEMORY REQUIRED: 14 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 14 cycles

FLOWCHART: RLSX



SOURCE:

```
*MOVE A TO B (DOUBLE VAR) WITH N(CONST) BIT
*RIGHT LOGICAL SHIFT
*
RLSX  $MACRO  A,B,N      MOVE DOUBLE WITH LOGICAL SHIFT
      RLSH :A:+1,:B:+1,:N: SHIFT RIGHT LOWER
      LAC  :A:,16-:N:    GET UPPER (RIGHT SHIFT)
      SACH :B:,0         SAVE IN :B: HIGH
      OR   :B:+1         COMBINE LOW PARTS
      SACL :B:+1,0       SAVE IN :B: LOW
      LAC  MINUS,16-:N:  GET MASK
      NOT
      AND  :B:           MASK HIGH :B:
      SACL :B:,0         SAVE BACK IN :B:
      $END
```

EXAMPLE:

```
 0011                  RLSX A,B,3
 0001                   RLSH A+1,B+1,3    SHIFT RIGHT LOWER
 0001 0006 2D08          LAC  A+1,16-3    LOAD, RIGHT SHIFT
 0002 0007 580A          SACH B+1,0       SAVE HIGH PART
 0003 0008 2D05"         LAC  MINUS,16-3  GET MASK
 0004                    NOT
 0001 0009 7805"          XOR  MINUS      INVERT
 0005 000A 790A          AND  B+1         APPLY MASK
 0006 000B 500A          SACL B+1,0       STORE BACK TO B+1
 0002 000C 2D07         LAC  A,16-3       GET UPPER (RIGHT SHIFT)
 0003 000D 5809         SACH B,0          SAVE IN B HIGH
 0004 000E 7A0A         OR   B+1          COMBINE LOW PARTS
 0005 000F 500A         SACL B+1,0        SAVE IN B LOW
 0006 0010 2D05"        LAC  MINUS,16-3   GET MASK
 0007                   NOT
 0001 0011 7805"         XOR  MINUS       INVERT
 0008 0012 7909         AND  B            MASK HIGH B
 0009 0013 5009         SACL B,0          SAVE BACK IN B
```

TITLE: Store Double Word

NAME: SACX

OBJECTIVE: Store double word from accumulator

ALGORITHM:

SACX *	– causes→	(ACC) → @AR:@AR + 1
SACX * –	– causes→	(ACC) → @AR-1:@AR (AR) – 2 → AR
SACX * +	– causes→	(ACC) → @AR:@AR + 1 (AR) + 2 → AR
SACX A	– causes→	(ACC) → A:A + 1

CALLING SEQUENCE: SACX {A,*,* – ,* + }

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127$

EXIT CONDITIONS: Specified double word contains value from accumulator; auxiliary register is updated if necessary

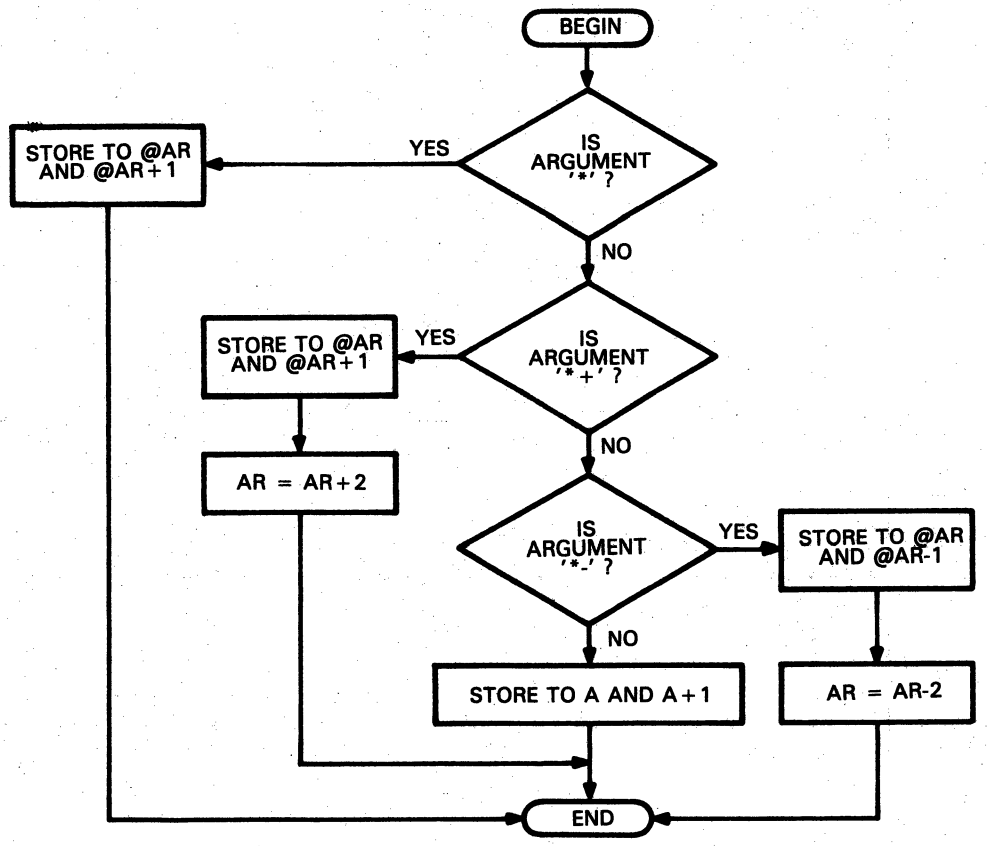
PROGRAM MEMORY REQUIRED: 2 words

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 2 cycles

FLOWCHART: SACX



SOURCE:

```
*STORE DOUBLE
*
SACX    $MACRO  A           STORE DOUBLE
        $VAR ST,SP,SM
        $ASG '*' TO ST.S
        $ASG '*-' TO SM.S
        $ASG '*+' TO SP,S
        $IF A.SV=ST.SV
        SACH *+,0           STORE HIGH
        SACL *-,0           STORE LOW
        $ELSE
        $IF A.SV=SP.SV
        SACH *+,0           STORE HIGH
        SACL *+,0           STORE LOW
        $ELSE
        $IF A.SV=SM.SV
        SACL *-,0           STORE LOW
        SACH *-,0           STORE HIGH
        $ELSE
        SACH :A:,0          STORE HIGH
        SACL :A:+1,0        STORE LOW
        $ENDIF
        $ENDIF
        $ENDIF
        $END
```

EXAMPLE 1:

```
0011                    SACX A
0001 0006 5807           SACH A,0          STORE HIGH
0002 0007 5008           SACL A+1,0        STORE LOW
```

EXAMPLE 2:

```
0013                    SACX *
0001 0008 58A8           SACH *+,0         STORE HIGH
0002 0009 5098           SACL *-,0         STORE LOW
```

EXAMPLE 3:

```
0015                    SACX *-
0001 000A 5098           SACL *-,0         STORE LOW
0002 000B 5898           SACH *-,0         STORE HIGH
```

EXAMPLE 4:

```
0017                    SACX *+
0001 000C 58A8           SACH *+,0         STORE HIGH
0002 000D 50A8           SACL *+,0         STORE LOW
```

TITLE: Saturate Data Word between Upper and Lower Bounds

NAME: SAT

OBJECTIVE: Insure that a data word falls within boundary conditions

ALGORITHM:
If (A) > upper, then upper → A
Else if (A) < lower, then lower → A

CALLING SEQUENCE: SAT data,lower,upper

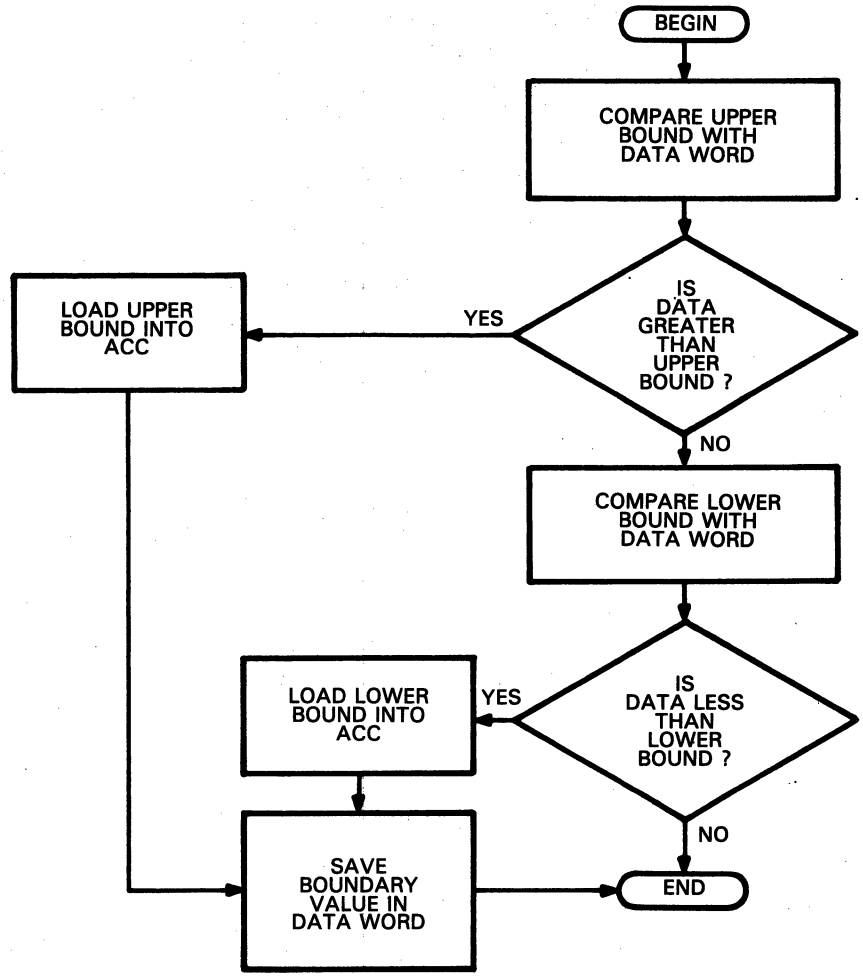
ENTRY CONDITIONS: $0 \leqslant data \leqslant 127$; $-32768 \leqslant lower \leqslant upper \leqslant 32767$

EXIT CONDITIONS: Data word contains value within bounds; staturation mode is reset

PROGRAM MEMORY REQUIRED:	16 – 24 words (+ LDAC$ routine)	**DATA MEMORY REQUIRED:**	2 words
STACK REQUIRED:	2 levels	**EXECUTION TIME:**	10 – 48 cycles

7

FLOWCHART: SAT



SOURCE:

```
*SATURATE VALUE IN A BETWEEN VALUES B AND C
*A IS A VARIABLE
*B AND C ARE VARIABLES OR CONSTANTS
*
SAT     $MACRO  A,B,C
        $VAR L,L1,L2,L3
        $ASG '$$LAB' TO L.S
        $ASG L.SV+3 TO L.SV     GET A LABEL
        $ASG L.SV-2 TO L1.V
        $ASG L.SV-1 TO L2.V
        $ASG L.SV   TO L3.V
        SOVM            SET OVERFLOW MODE
        $IF  C.SA&$UNDF
        LCAC :C:        LOAD UPPER BOUND :C:
        $ELSE
        LAC  :C:,0      LOAD UPPER BOUND :C:
        $ENDIF
        SUB  :A:,0      COMPARE TO :A:
        BGEZ L$:L1.V:   BRANCH IF :A:<=:C:
        $IF  C.SA&$UNDF
        LCAC :C:        RELOAD :C: AS VALUE
        $ELSE
```

```
        LAC   :C:,0         RELOAD :C: AS VALUE
        $ENDIF
        B     L$:L2.V:      BRANCH TO CONTINUE
L$:L1.V: EQU $              CHECK LOWER
        $IF   B.SA&$UNDF
        LCAC  :B:           LOAD LOWER BOUND :B:
        $ELSE
        LAC   :B:,0         LOAD LOWER BOUND :B:
        $ENDIF
        SUB   :A:,0         COMPARE TO :A:
        BLEZ  L$:L3.V:      BRANCH IF :A:>:B:
        $IF   B.SA&$UNDF
        LCAC  :B:           RELOAD :B: AS VALUE
        $ELSE
        LAC   :B:,0         RELOAD :B: AS VALUE
        $ENDIF
L$:L2.V: SACL :A:,0         RESTORE :A:
L$:L3.V: ROVM               CONTINUE
        $END
```

EXAMPLE 1:

```
 0011                    SAT  A,25,50
 0001 0005 7F8B           SOVM             SET OVERFLOW MODE
 0002                     LCAC 50          LOAD UPPER BOUND 50
 0001      0032  V$4 EQU 50
 0002 0006 7E32            LACK V$4        LOAD AC WITH V$4
 0003 0007 1007           SUB  A,0         COMPARE TO A
 0004 0008 FD00           BGEZ L$1         BRANCH IF A<=50
      0009 000D'
 0005      0032           LCAC 50          RELOAD 50 AS VALUE
 0001      0032  V$5 EQU 50
 0002 000A 7E32            LACK V$5        LOAD AC WITH V$5
 0006 000B F900           B    L$2         BRANCH TO CONTINUE
      000C 0012'
 0007      000D' L$1 EQU $                 CHECK LOWER
 0008      000D'          LCAC 25          LOAD LOWER BOUND 25
 0001      0019  V$6 EQU 25
 0002 000D 7E19            LACK V$6        LOAD AC WITH V$6
 0009 000E 1007           SUB  A,0         COMPARE TO A
 0010 000F FB00           BLEZ L$3         BRANCH IF A>25
      0010 0013'
 0011      0019           LCAC 25          RELOAD 25 AS VALUE
 0001      0019  V$7 EQU 25
 0002 0011 7E19            LACK V$7        LOAD AC WITH V$7
 0012 0012 5007  L$2      SACL A,0         RESTORE A
 0013 0013 7F8A  L$3      ROVM             CONTINUE
```

EXAMPLE 2:

```
 0013                    SAT  A,C,D
 0001 0014 7F8B           SOVM             SET OVERFLOW MODE
 0002 0015 2002"          LAC  D,0         LOAD UPPER BOUND D
 0003 0016 1007           SUB  A,0         COMPARE TO A
 0004 0017 FD00           BGEZ L$8         BRANCH IF A<=D
      0018 001C'
 0005 0019 2002"          LAC  D,0         RELOAD D AS VALUE
 0006 001A F900           B    L$9         BRANCH TO CONTINUE
      001B 0021'
 0007      001C' L$8 EQU $                 CHECK LOWER
 0008 001C 2000"          LAC  C,0         LOAD LOWER BOUND C
 0009 001D 1007           SUB  A,0         COMPARE TO A
```

```
0010 001E FB00          BLEZ L$10       BRANCH IF A>C
     001F 0022'
0011 0020 2000"         LAC  C,0        RELOAD C AS VALUE
0012 0021 5007  L$9     SACL A,0        RESTORE A
0013 0022 7F8A  L$10    ROVM            CONTINUE
```

TITLE: Subtract Variable from Auxiliary Register

NAME: SBAR

OBJECTIVE: Subtract data word from named auxiliary register

ALGORITHM: (ACAR) – (dma) → ACC
(ACC) → AR

CALLING SEQUENCE: SBAR AR, B [,TEMP]

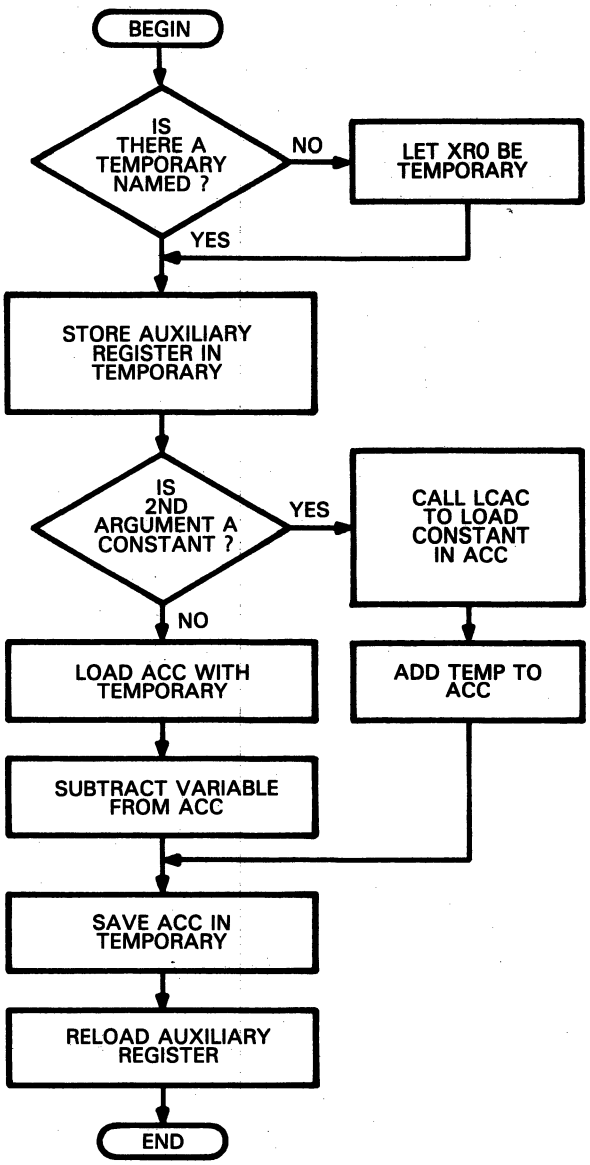
ENTRY CONDITIONS: AR = 0,1; 0 ⩽ B ⩽ 127; 0 ⩽ TEMP ⩽ 127

EXIT CONDITIONS: Difference between memory location and auxiliary regi ster is stored in named auxiliary register

PROGRAM MEMORY REQUIRED:	5 – 7 words (plus LDAC$ routine)	**DATA MEMORY REQUIRED:**	2 words
STACK REQUIRED:	0 – 2 levels	**EXECUTION TIME:**	5 – 17 cycles

7

FLOWCHART: SBAR



SOURCE:

```
*SUB FROM AR
*A IS AR1 OR AR0
*B IS CONST OR VAR
*
SBAR    $MACRO  A,B,T
        $IF  T.L=0          ASSIGN TEMP
        $ASG 'XR1' TO T.S
        $ENDIF
        SAR  :A:,:T:        SAVE :A:
        $IF  B.SA&$UNDF
        $ASG -B.V TO B.V
        LCAC :B.V:          LOAD -:B: VALUE
        ADD  :T:,0          ADD :T: VALUE
        $ELSE
        LAC :T:,0           LOAD :T:
        SUB :B:,0           SUB :B: VALUE
```

```
        $ENDIF
        SACL :T:,0          RESTORE
        LAR  :A:,:T:        RELOAD :A:
        $END
```

EXAMPLE 1:

```
0007                    SBAR   AR1,3
0001 0006 3103"          SAR  AR1,XR1       SAVE AR1
0002                     LCAC -3            LOAD -3 VALUE
0001      FFFD  V$1 EQU -3
0002 0007 F800           CALL LDAC$         LOAD AC WITH:
     0008 0000
0003                     REF  LDAC$
0004 0009 FFFD           DATA V$1           V$1
0003 000A 0003"         ADD  XR1,0          ADD XR1 VALUE
0004 000B 5003"         SACL XR1,0          RESTORE
0005 000C 3903"         LAR  AR1,XR1        RELOAD AR1
```

EXAMPLE 2:

```
0009                    SBAR   AR0,C,B
0001 000D 3008           SAR  AR0,B         SAVE AR0
0002 000E 2008           LAC B,0            LOAD B
0003 000F 1004"          SUB C,0            SUB C VALUE
0004 0010 5008           SACL B,0           RESTORE
0005 0011 3808           LAR  AR0,B         RELOAD AR0
```

EXAMPLE 3:

```
0011                    SBAR   0,D
0001 0012 3003"          SAR  0,XR1         SAVE 0
0002 0013 2003"          LAC XR1,0          LOAD XR1
0003 0014 1005"          SUB D,0            SUB D VALUE
0004 0015 5003"          SACL XR1,0         RESTORE
0005 0016 3803"          LAR  0,XR1         RELOAD 0
```

TITLE: Clear Single Bit in Data Word

NAME: SBIC

OBJECTIVE: Clear bit in data word specified by bit position argument

ALGORITHM: (A) .AND. .NOT. 2^{bit} → (A)

CALLING SEQUENCE: SBIC bit,A

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127; 0 \leqslant bit \leqslant 15$

EXIT CONDITIONS: A contains initial value with specified bit cleared

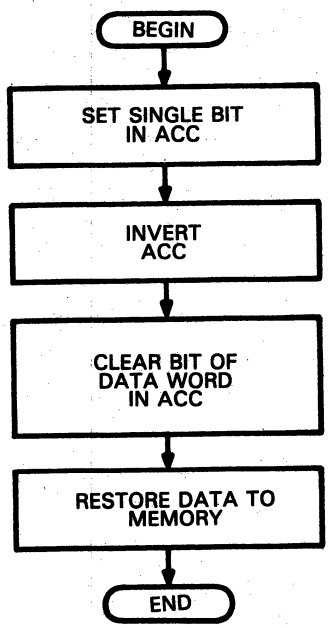
PROGRAM MEMORY REQUIRED: 4 words

DATA MEMORY REQUIRED: 2 words

STACK REQUIRED: None

EXECUTION TIME: 4 cycles

FLOWCHART: SBIC



SOURCE:

```
*BIC A SELECTED BIT
*A IS BIT NUMBER
*B IS VAR
*
SBIC    $MACRO  A,B        SINGLE BIT CLEAR
        LAC  ONE,:A:       GET SELECT BIT
        XOR  MINUS         INVERT MASK
        AND  :B:           AND :B:
        SACL :B:,0         STORE TO :B:
        $END
```

EXAMPLE 1:

```
0012                    SBIC   B,C
0001 000A 2802"          LAC  ONE,B        GET SELECT BIT
0002 000B 7803"          XOR  MINUS        INVERT MASK
0003 000C 7900"          AND  C            AND C
0004 000D 5000"          SACL C,0          STORE TO C
```

EXAMPLE 2:

```
0014                    SBIC   3,D
0001 000E 2302"          LAC  ONE,3        GET SELECT BIT
0002 000F 7803"          XOR  MINUS        INVERT MASK
0003 0010 7901"          AND  D            AND D
0004 0011 5001"          SACL D,0          STORE TO D
```

EXAMPLE 3:

```
0016                    SBIC   12,B
0001 0012 2C02"          LAC  ONE,12       GET SELECT BIT
0002 0013 7803"          XOR  MINUS        INVERT MASK
0003 0014 7908           AND  B            AND B
0004 0015 5008           SACL B,0          STORE TO B
```

TITLE: Set Single Bit in Data Word

NAME: SBIS

OBJECTIVE: Set bit in data word specified by bit position argument

ALGORITHM: (data) .OR. 2^{bit} → data

CALLING SEQUENCE: SBIS bit,A

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127; 0 \leqslant bit \leqslant 15$

EXIT CONDITIONS: A contains initial value with specified bit set

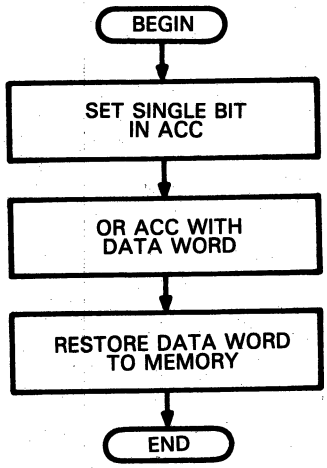
PROGRAM MEMORY REQUIRED: 3 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 3 cycles

FLOWCHART: SBIS



SOURCE:

```
*SET SELECTED BIT
*A IS BIT NUMBER
*B IS VAR
*
SBIS    $MACRO  A,B         SINGLE BIT SET
        LAC  ONE,:A:        GET SELECT BIT
        OR    :B:           SET TO :B:
        SACL :B:,0          RESTORE
        $END
```

EXAMPLE 1:

```
0012                    SBIS  B,C
0001 0009 2802"          LAC  ONE,B       GET SELECT BIT
0002 000A 7A00"          OR   C           SET TO C
0003 000B 5000"          SACL C,0         RESTORE
```

EXAMPLE 2:

```
0014                    SBIS  3,D
0001 000C 2302"          LAC  ONE,3       GET SELECT BIT
0002 000D 7A01"          OR   D           SET TO D
0003 000E 5001"          SACL D,0         RESTORE
```

EXAMPLE 3:

```
0016                    SBIS  12,B
0001 000F 2C02"          LAC  ONE,12      GET SELECT BIT
0002 0010 7A08           OR   B           SET TO B
0003 0011 5008           SACL B,0         RESTORE
```

TITLE: Test Single Bit in Data Word

NAME: SBIT

OBJECTIVE: Test bit in data word specified by bit position argument

ALGORITHM: data .AND. 2^{bit} → ACC

CALLING SEQUENCE: SBIT bit,A

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127; 0 \leqslant bit \leqslant 15$

EXIT CONDITIONS: ACC contains zero if specified bit is cleared, non-zero else

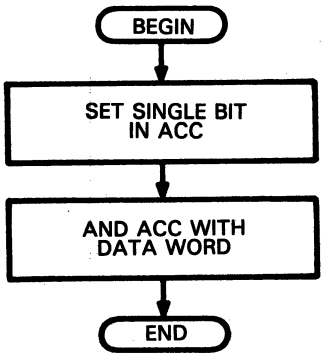
PROGRAM MEMORY REQUIRED: 2 words

DATA MEMORY REQUIRED: 1 word

STACK REQUIRED: None

EXECUTION TIME: 2 cycles

FLOWCHART: SBIT



SOURCE:

```
*TEST SELECTED BIT
*A IS BIT NUMBER
*B IS VAR TO TEST
*
SBIT    $MACRO  A,B         SINGLE BIT TEST
        LAC   ONE,:A:       GET BIT :A:
        AND   :B:           TEST FOR IT
        $END
```

EXAMPLE:

```
0014                    SBIT  3,D
0001 000A 2302"          LAC  ONE,3        GET BIT 3
0002 000B 7901"          AND  D            TEST FOR IT
```

TITLE: Convert Single Word to Double Word

NAME: STOX

OBJECTIVE: Convert single word to a double word and save

ALGORITHM: (A) → B:B + 1

CALLING SEQUENCE: STOX single,double

ENTRY CONDITIONS: 0 ⩽ single ⩽ 127 ; 0 ⩽ double ⩽ 127

EXIT CONDITIONS: Double word contains value of single word

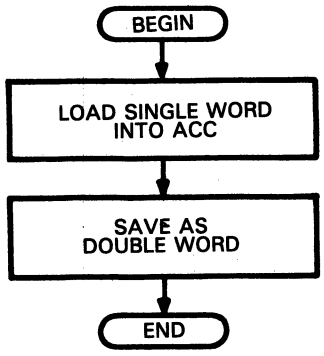
PROGRAM MEMORY REQUIRED: 3 words

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 3 cycles

FLOWCHART: STOX



SOURCE:

```
*SINGLE TO DOUBLE (A TO B)
*
STOX    $MACRO  A,B
        LAC   :A:,0         LOAD SINGLE
        SACX  :B:           STORE DOUBLE
        $END
```

EXAMPLE:

```
0011                    STOX A,D
0001 0006 2007           LAC  A,0          LOAD SINGLE
0002                     SACX D            STORE DOUBLE
0001 0007 5802"           SACH D,0         STORE HIGH
0002 0008 5003"           SACL D+1,0       STORE LOW
```

TITLE: Double-Word Subtract

NAME: SUBX

OBJECTIVE: Subtract double word from accumulator

ALGORITHM:

SUBX *	– causes→	(ACC) – (@AR:@AR + 1) → ACC
SUBX * –	– causes→	(ACC) – (@AR-1:@AR) → ACC (AR) – 2 → AR
SUBX * +	– causes→	(ACC) – (@AR:@AR + 1) → ACC (AR) + 2 → AR
SUBX A	– causes→	(ACC) – (A:A + 1) → ACC

CALLING SEQUENCE: SUBX {A,*,* – ,* + }

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127$

EXIT CONDITIONS: Accumulator contains updated value after subtraction; auxiliary register is updated if necessary

PROGRAM MEMORY REQUIRED: 2 words

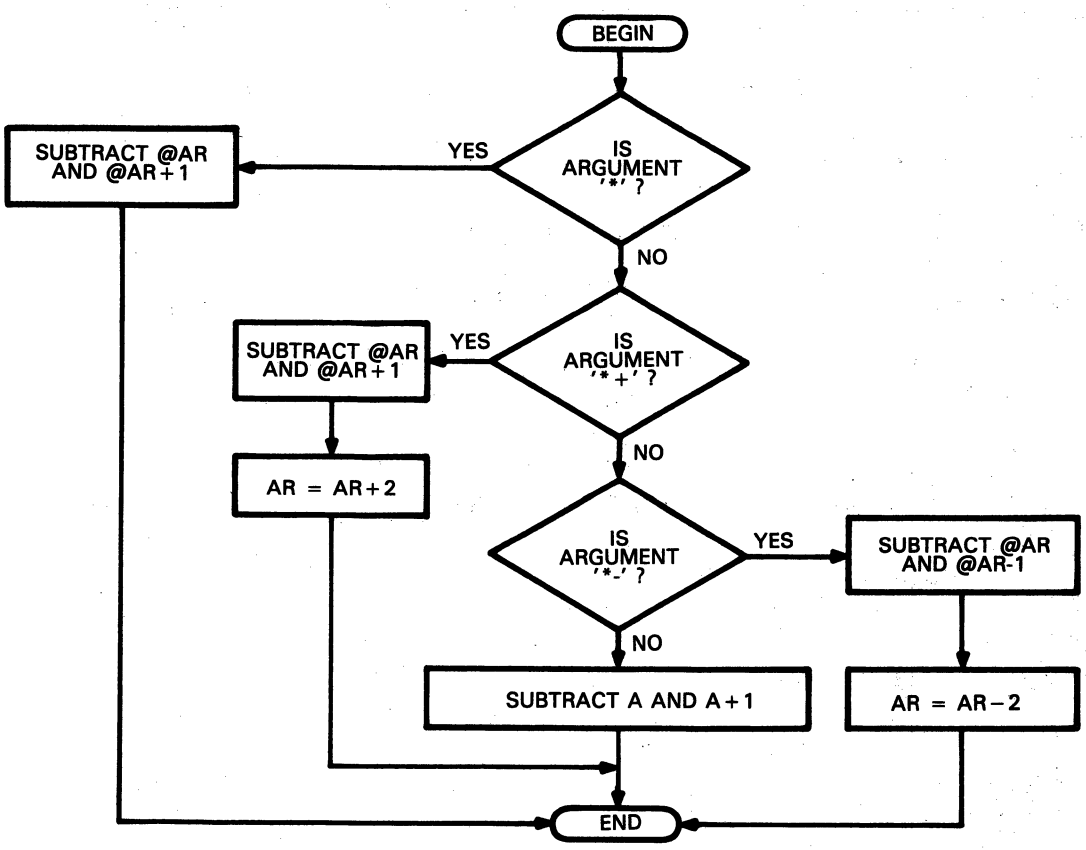
DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 2 cycles

7

FLOWCHART: SUBX



7

SOURCE:

```
*SUBTRACT DOUBLE
*
SUBX    $MACRO  A           SUBTRACT DOUBLE
        $VAR ST,SM,SP
        $ASG '*' TO ST.S
        $ASG '*+' TO SP.S
        $ASG '*-' TO SM.S
        $IF A.SV=ST.SV
        SUBH *+             SUBTRACT HIGH
        SUBS *-             SUBTRACT LOW
        $ELSE
        $IF A.SV=SP.SV
        SUBH *+             SUBTRACT HIGH
        SUBS *+             SUBTRACT LOW
        $ELSE
        $IF A.SV=SM.SV
        SUBS *-             SUBTRACT LOW
        SUBH *-             SUBTRACT HIGH
        $ELSE
        SUBH :A:            SUBTRACT HIGH
        SUBS :A:+1          SUBTRACT LOW
        $ENDIF
        $ENDIF
        $ENDIF
        $END
```

EXAMPLE 1:

```
0011                    SUBX A
0001 0006 6207           SUBH A          SUBTRACT HIGH
0002 0007 6308           SUBS A+1        SUBTRACT LOW
```

EXAMPLE 2:

```
0013                    SUBX *
0001 0008 62A8           SUBH *+         SUBTRACT HIGH
0002 0009 6398           SUBS *-         SUBTRACT LOW
```

EXAMPLE 3:

```
0015                    SUBX *-
0001 000A 6398           SUBS *-         SUBTRACT LOW
0002 000B 6298           SUBH *-         SUBTRACT HIGH
```

EXAMPLE 4:

```
0017                    SUBX *+
0001 000C 62A8           SUBH *+         SUBTRACT HIGH
0002 000D 63A8           SUBS *+         SUBTRACT LOW
```

EXAMPLE 5:

```
0019                    SUBX 3
0001 000E 6203           SUBH 3          SUBTRACT HIGH
0002 000F 6304           SUBS 3+1        SUBTRACT LOW
```

TITLE: Test Word

NAME: TST

OBJECTIVE: Load word into accumulator, allowing comparison with zero

ALGORITHM: (A) → ACC

CALLING SEQUENCE: TST {A,*,*–,*+}

ENTRY CONDITIONS: 0 ⩽ A ⩽ 127

EXIT CONDITIONS: Accumulator contains value of word

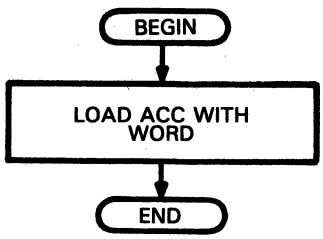
PROGRAMM MEMORY REQUIRED: 1 word

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 1 cycle

FLOWCHART: TST



SOURCE:

```
*TEST SINGLE VAR
*
TST     $MACRO  A          COMPARE TO ZERO
        LAC   :A:,0       LOAD IT
        $END
```

EXAMPLE 1:

```
0007                    TST    A
0001 0006 2001           LAC  A,0          LOAD IT
```

EXAMPLE 2:

```
0009                    TST    *
0001 0007 2088           LAC  *,0          LOAD IT
```

EXAMPLE 3:

```
0011                    TST    C
0001 0008 2004"          LAC  C,0          LOAD IT
```

EXAMPLE 4:

```
0013                    TST    *+
0001 0009 20A8           LAC  *+,0         LOAD IT
```

TITLE: Test Double Word

NAME: TSTX

OBJECTIVE: Load double word into accumulator, allowing comparison with zero

ALGORITHM:

TSTX *	– causes→	(@AR:@AR + 1) → ACC
TSTX * –	– causes→	(@AR – 1:@AR) → ACC (AR) – 2 → AR
TSTX * +	– causes→	(@AR:@ AR + 1) → ACC (AR) + 2 → AR
TSTX A	– causes←	(A:A + 1) → ACC

CALLING SEQUENCE: TSTX {A,*,* – ,* + }

ENTRY CONDITIONS: $0 \leqslant A \leqslant 127$

EXIT CONDITIONS: Accumulator contains value of double word; auxiliary register is updated if necessary

7

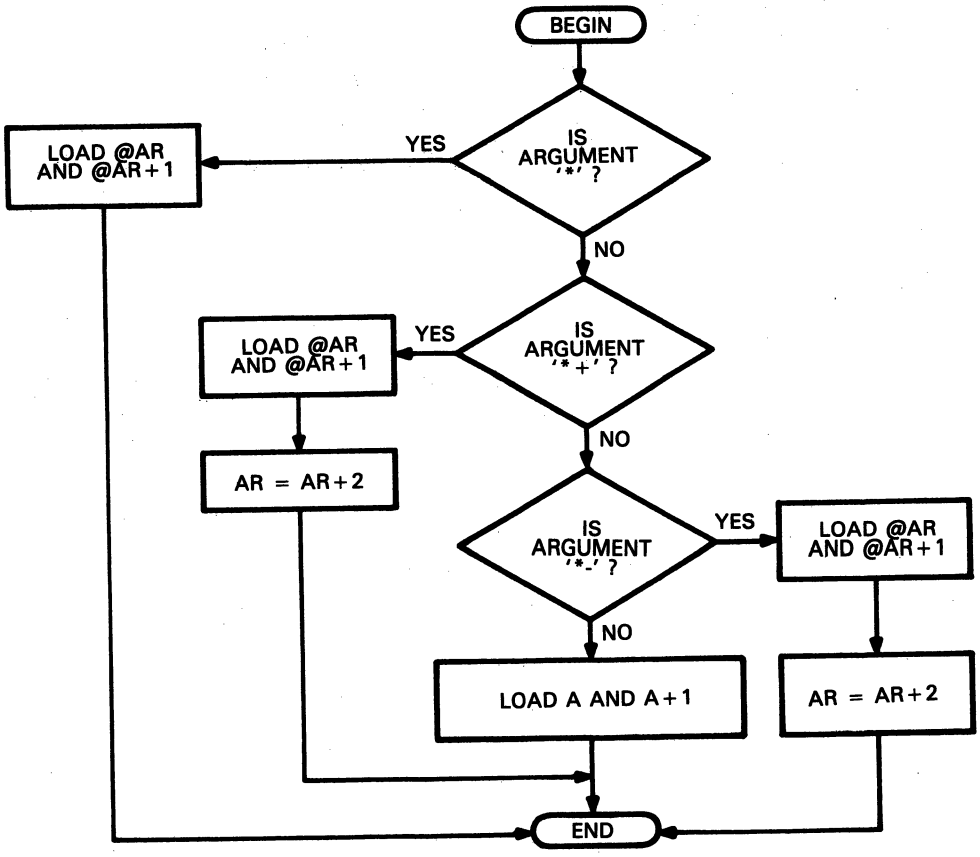
PROGRAM MEMORY REQUIRED: 2 words

DATA MEMORY REQUIRED: None

STACK REQUIRED: None

EXECUTION TIME: 2 cycles

FLOWCHART: TSTX



SOURCE:

```
*TEST DOUBLE VAR
*
TSTX    $MACRO  A       COMPARE TO ZERO DOUBLE
        LDAX :A:        LOAD IT DOUBLE
        $END
```

EXAMPLE 1:

```
 0011                   TSTX A
 0001                    LDAX A          LOAD IT DOUBLE
 0001 0006 6507           ZALH A         LOAD HIGH A
 0002 0007 6108           ADDS A+1       LOAD LOW A
```

EXAMPLE 2:

```
 0013                   TSTX *
 0001                    LDAX *          LOAD IT DOUBLE
 0001 0008 65A8           ZALH *+        LOAD HIGH
 0002 0009 6198           ADDS *-        LOAD LOW '*'
```

EXAMPLE 3:

```
 0015                   TSTX *-
 0001                    LDAX *-         LOAD IT DOUBLE
 0001 000A 6698           ZALS *-        LOAD LOW
 0002 000B 6098           ADDH *-        LOAD HIGH '*-'
```

EXAMPLE 4:

```
0017                    TSTX *+
0001                     LDAX *+          LOAD IT DOUBLE
0001 000C 65A8            ZALH *+        LOAD HIGH
0002 000D 61A8            ADDS *+        LOAD LOW '*+'
```

TITLE: Convert Double Word To Single Word

NAME: XTOS

OBJECTIVE: Convert double word to a single word and save

ALGORITHM:

	If (A:A + 1) >	32767	then	32767	→ B
Else	if (A:A + 1) <	−32768	then	−32768	→ B
			Else	(A + 1)	→ B

CALLING SEQUENCE: XTOS double,single

ENTRY CONDITIONS: 0 ⩽ single ⩽ 127 ; 0 ⩽ double ⩽ 127

EXIT CONDITIONS: Single word contains value of double word or saturation value

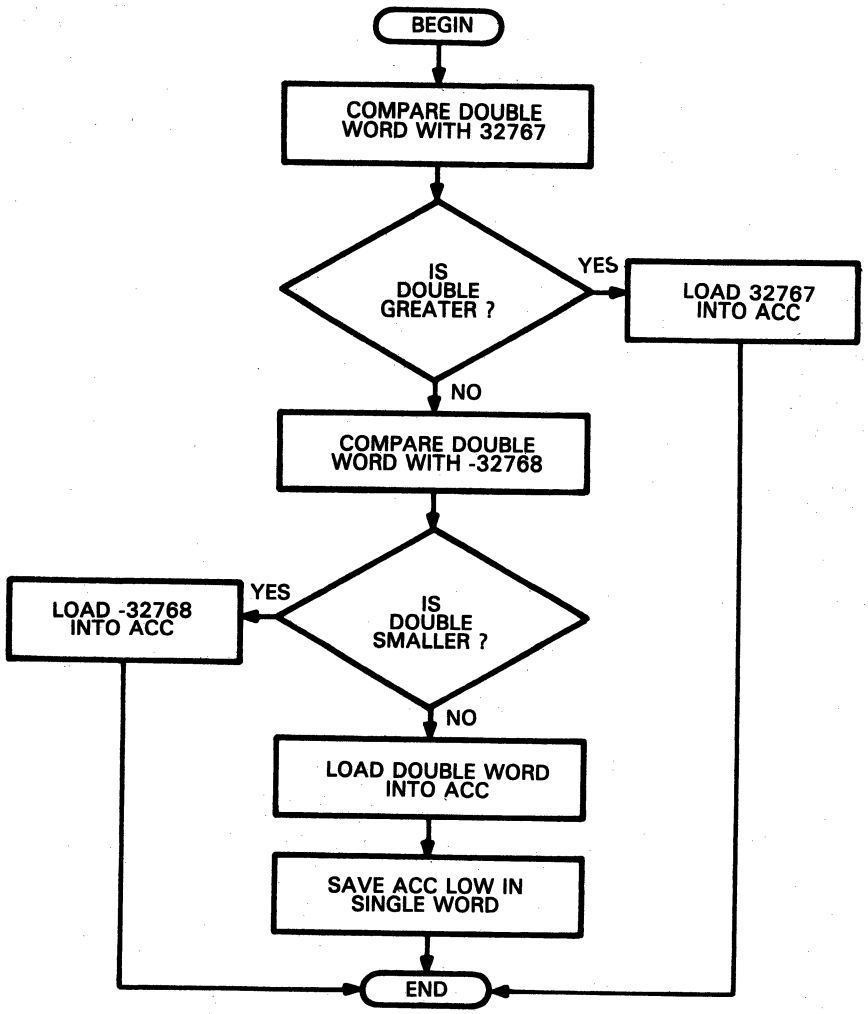
PROGRAM MEMORY REQUIRED: 27 words (+ LDAC$ routine)

DATA MEMORY REQUIRED: 2 words

STACK REQUIRED: 2 levels

EXECUTION TIME: 33 – 50 cycles

FLOWCHART: XTOS



SOURCE:

```
*DOUBLE TO SINGLE (A TO B)
*
XTOS    $MACRO  A,B
        $VAR L,L1,L2,L3
        $ASG '$$LAB' TO L.S
        $ASG L.SV+3 TO L.SV     GET LABEL
        $ASG L.SV-2 TO L1.V
        $ASG L.SV-1 TO L2.V
        $ASG L.SV   TO L3.V
        LCAC 32767          GET BIGGEST SINGLE
        SUBX :A:            COMPARE :A:
        BGEZ L$:L1.V:       IF :A: >= 32767 THEN
        LCAC 32767          SATURATE AT 32767
        B    L$:L3.V:       JUMP TO DONE
L$:L1.V: LCAC -32768        GET MOST NEG SINGLE
        SUBX :A:            COMPARE :A:
        BLEZ L$:L2.V:       IF :A: <= -32768 THEN
        LCAC -32768         SATURATE AT -32768
        B    L$:L3.V:       JUMP TO DONE
L$:L2.V: LDAX :A:           LOAD :A:
L$:L3.V: SACL :B:,0         RESTORE TO :B:
        $END
```

EXAMPLE:

```
0013                    XTOS C,B
0001                    LCAC 32727          GET BIGGEST SINGLE
0001       7FD7  V$11 EQU 32727
0002 0021 F800           CALL LDAC$         LOAD AC WITH:
     0022 0000
0003                     REF  LDAC$
0004 0023 7FD7           DATA V$11          V$11
0002 0024               SUBX C              COMPARE C
0001 0024 6200"          SUBH C             SUBTRACT HIGH
0002 0025 6301"          SUBS C+1           SUBTRACT LOW
0003 0026 FD00          BGEZ L$8            IF C >= 32767 THEN
     0027 002D'
0004 0028               LCAC 32727          SATURATE AT 32767
0001       7FD7  V$12 EQU 32727
0002 0028 F800           CALL LDAC$         LOAD AC WITH:
     0029 0000
0003                     REF  LDAC$
0004 002A 7FD7           DATA V$12          V$12
0005 002B F900          B    L$10           JUMP TO DONE
     002C 003B'
0006 002D         L$8    LCAC -32768        GET MOST NEGATIVE SINGLE
0001       8000  V$13 EQU -32768
0002 002D F800           CALL LDAC$         LOAD AC WITH:
     002E 0000
0003                     REF  LDAC$
0004 002F 8000           DATA V$13          V$13
0007 0030               SUBX C              COMPARE C
0001 0030 6200"          SUBH C             SUBTRACT HIGH
0002 0031 6301"          SUBS C+1           SUBTRACT LOW
0008 0032 FB00          BLEZ L$9            IF C <= -32768 THEN
     0033 0039'
0009 0034               LCAC -32768         SATURATE AT -32768
0001       8000  V$14 EQU -32768
0002 0034 F800           CALL LDAC$         LOAD AC WITH:
     0035 0000
0003                     REF  LDAC$
0004 0036 8000           DATA V$14          V$14
0010 0037 F900          B    L$10           JUMP TO DONE
     0038 003B'
0011 0039         L$9   LDAX C              LOAD C
0001 0039 6500"          ZALH C             LOAD HIGH C
0002 003A 6101"          ADDS C+1           LOAD LOW C
0012 003B 5009  L$10    SACL B,0            RESTORE TO B
```

7.4 STRUCTURED PROGRAMMING MACROS

The program structure macros, PROG AND MAIN, need to be used with most of the other macros described in Section 7.3 in order to set up internal symbols and utility variables used by those macros.

PROG — Begin Program – Macro — PROG

PROG – Begin Program

The program directive does two things. First, it defines the module IDT name (the name of the module printed on the link editor memory map listing). More importantly, it initializes several internal symbols used in many of the macros from Section 7.3. Syntax is as follows:

PROG < name>

Where < name> is a string of up to six characters. This name is used to generate:

IDT '< name> '

To end the module, use the assembly language END statement:

END

SOURCE:

```
*
* Prog Routine Initializes Internal Variables, and
*     Outputs IDT Statement
*
PROG    $MACRO              A
        $VAR Q
        $ASG '''' TO Q.S
        IDT  :Q::A::Q:
*
* Initialize unique label counter
*
        $ASG '$$LAB' TO Q.S
        $ASG 0 TO Q.SV
*
* Assign unique values to indirect symbols
*
        $ASG '*' TO Q.S
        $ASG >F0F0 TO Q.SV
        $ASG '*+' TO Q.S
        $ASG >F0F1 TO Q.SV
        $ASG '*-' TO Q.S
        $ASG >F0F2 TO Q.SV
        $END
```

MAIN— Begin Main Procedure

MAIN < name>

The MAIN directive begins the main procedure. < name> is the label (created by the macro) of the first instruction of the main routine (up to six characters). MAIN allocates the variables ONE, MINUS, XR0, and XR1 in data RAM (in the DSEG), and initializes ONE to 1, and MINUS to – 1.

SOURCE:

```
*
*  Main Procedure Definition Macro
*
*  A is Main Program Name (<6 CHAR)
*
MAIN    $MACRO              A
        PSEG                PROG SEG
        DEF   :A:           ENTRY POINT
:A:     EQU   $
*
*  Initialize Variables
*
        LACK 1              MAKE CONSTANT ONE
        SACL ONE,0          SAVE IT
        ZAC                 ZERO ACCUMULATOR
        SUB  ONE,0          MAKE -1
        SACL MINUS,0        SAVE IT
*
*  Data Segment
*
        DSEG
ONE     BSS  1              CONSTANT ONE
MINUS   BSS  1              CONSTANT -1
XR0     BSS  1              TEMP 0
XR1     BSS  1              TEMP 1
        DEF  ONE,MINUS      ALLOW EXTERNAL USE
        DEF  XR0,XR1        OF VARIABLES
        DEND                END OF DATA
        $END
```

EXAMPLES OF PROG AND MAIN USAGE:

```
      MLIB 'MACROS'          Declare directory of macros,
*                            including PROG and MAIN
      PROG  MACTST           Set up symbol table variables
*
*
```

```
      DSEG                    User's program variables
VAR1  BSS 1
VAR2  BSS 1
*     .
*     .
*     .
      DEND
*
*     |
*     |
*  Interrupt Routine (user defined)
*     |
*     |
*
      MAIN  START             Start of main routine
*
*     |
*     |
*  Main Program - Instructions and Macros
*     |
*     |
      END
```

LISTING:

```
0001 0000                MLIB 'MACROS'      Declare directory of macros,
0002               *                        including PROG and MAIN
0003                     PROG  MACTST       Set up symbol table variables
0001                      IDT  'MACTST'
0004               *
0005               *
0006 0000                DSEG               User's program variables
0007 0000          VAR1  BSS 1
0008 0001          VAR2  BSS 1
0009               *     .
0010               *     .
0011               *     .
0012 0002                DEND
0013               *
0014               *     |
0015               *     |
0016               *  Interrupt Routine (user defined)
0017               *     |
0018               *     |
0019               *
0020                     MAIN  START        Start of main routine
0001 0000                 PSEG              PROG SEG
0002                      DEF   START         ENTRY POINT
0003       0000' START     EQU  $
0004 0000 7E01           LACK 1             MAKE CONSTANT ONE
0005 0001 5002"          SACL ONE,0         SAVE IT
0006 0002 7F89           ZAC                ZERO ACCUMULATOR
0007 0003 1002"          SUB  ONE,0         MAKE -1
0008 0004 5003"          SACL MINUS,0       SAVE IT
0009 0002                DSEG
0010 0002          ONE   BSS  1             CONSTANT ONE
0011 0003          MINUS BSS  1             CONSTANT -1
0012 0004          XR0   BSS  1             TEMP 0
0013 0005          XR1   BSS  1             TEMP 1
0014                     DEF  ONE,MINUS     ALLOW EXTERNAL USE
0015                     DEF  XR0,XR1       OF VARIABLES
0016 0006                DEND               END OF DATA
0021               *
```

```
0022            *
0023            *     |
0024            *     |
0025            *  Main Program - Instructions and Macros
0026            *     |
0027            *     |
0028                   END
```

7.5 UTILITY SUBROUTINES

The subroutines in this section are called by many of the macros described in Section 7.3. Subroutines are used to save program space. Instead of inserting the code into each macro, the code occurs as a separate subroutine. Since the code is not expanded with each macro call, program space is saved. These routines should be assembled separately from the calling program and linked with the main program.

SOURCE FILE OF UTILITY SUBROUTINES:

```
      IDT 'SUBR'
*
*  SUBROUTINES USED AS UTILITIES IN VARIOUS MACRO LANGUAGE EXTENSIONS
*  AND SIGNAL PROCESSING LANGUAGE MACROS.
*
      REF ONE,MINUS
      REF XR0,XR1
*
*  LDAC$ - Load the accumulator with value found in program memory
*          at location pointed to by address on the top of the stack.
*
      DEF LDAC$
LDAC$ POP
      TBLR XR0
      ADD  ONE
      PUSH
      LAC  XR0
      RET
*
*
*  RIP$ - SUBROUTINE USED FOR LOOPED VERSION OF RIPPLE MACRO
*
      DEF RIP$
RIP$  POP
      TBLR XR0            1st argument = length
      LAR  AR0,XR0        R0 = count
      LARP AR0
      MAR  *-             Decrement count
      SAR  AR0,XR0        Store L-1 in XR0
      ADD  ONE            Increment argument pointer
      TBLR XR1            2nd argument = address
      LAR  AR1,XR1        Save address in R1
      SACL XR1            Save argument pointer
      LAC  XR0            ACC = L-1
      SAR  AR1,XR0        Get address from R1
      ADD  XR0            ACC = address + L-1
      SACL XR0            Save address
      LAR  AR1,XR0        R1 = address pointer
RIP$L LARP AR1
      DMOV *-,AR0         Shift data
      BANZ RIP$L
      LAC  XR1            Restore argument pointer
      ADD  ONE            Decrement argument pointer
```

```
      PUSH                Put return address on top of stack
      RET
*
* LDAX$ - Load accumulator with double word
*
      DEF LDAX$
LDAX$ POP                 Get address of constants
      TBLR XR1            Read upper half
      ADD ONE
      TBLR XR0            Read lower half
      ADD ONE
      PUSH
      ZALH XR1            Load upper half
      ADDS XR0            Load lower half
      RET
*
* LDAR$0 - Load Auxiliary Register 0 with word from program memory
*
       DEF LDAR$0
LDAR$0 POP                Get address of word
       TBLR XR0           Read word into data memory
       LAR  AR0,XR0       Load into AR0
       ADD  ONE
       PUSH               Restore return address
       RET
*
* LDAR$1 - Load Auxiliary Register 1 with word from program memory
*
       DEF LDAR$1
LDAR$1 POP                Get address of word
       TBLR XR0           Read word into data memory
       LAR  AR1,XR0       Load into AR1
       ADD  ONE
       PUSH               Restore return address
       RET
*
* LTK$  - Load T Register with word from program memory
*
      DEF LTK$
LTK$  POP                 Get address of word
      TBLR XR0            Read word into data memory
      LT   XR0            Load word into T register
      ADD  ONE
      PUSH                Restore return address
      RET
*
* Instructions for MOVE macro.  There are four different entry
* positions, but all of them use code starting at MOV$M to do
* actual data transfer.
*
*
* MOVAB$  - MOVE A,B
*
MOVAB$ POP
       TBLR XR0           Read A into AR0
       LAR  AR0,XR0
       ADD  ONE
MOVB$$ TBLR XR0           Read B into AR1
       LAR  AR1,XR0
       ADD  ONE
       B    MOV$M         Move data
*
* MOVA$  - MOVE A,*
*
```

```
MOVA$  POP
       TBLR XR0          Move A into AR0
       LAR  AR0,XR0
       ADD  ONE
       B    MOV$M
*
* MOVB$  - MOVE *,B
*
MOVB$  POP
       B    MOVB$$       Move B into AR1
*
* MOV$$  - MOVE *,*
*
MOV$$  POP
MOV$M  TBLR XR0          Read number of elements to move
       SACL XR1          Save return address
       LARP 0
MOV$L  LAC  *+,0,AR1     Move @AR0 to ACC
       SACL *+,0,AR0     Move ACC  to @AR1
       LAC  XR0
       SUB  ONE          Decrement loop counter
       SACL XR0
       BNZ  MOV$L        Loop back for another move
       LAC  XR1
       ADD  ONE
       PUSH              Restore return address
       RET
       DEF  MOVAB$,MOVA$,MOVB$,MOV$$
*
*   SETS$ - Move constant into L positions of data memory
*
SETS$ POP
      TBLR XR0           Get 1st argument - constant
      ADD  ONE
      TBLR XR1           Get 2nd argument - count
      LAR  AR0,XR1       Use AR0 as counter
      LARP 0
      MAR  *-
      ADD  ONE
      TBLR XR1           Get 3rd argument - destination
      LAR  AR1,XR1       Use AR1 as pointer
      SACL XR1           Save return address
      LAC  XR0           Load constant into accumulator
SET$L LARP 1
      SACL *+,0,AR0      Move constant to data memory
      BANZ SET$L         Repeat L times
      LAC  XR1
      ADD  ONE
      PUSH               Restore return address
      RET
      DEF  SETS$
*
*  MOVC$ AND MOVC$1 - Move list of constants to data memory
*
MOVC$  POP               Get argument pointer
       TBLR XR0          1st argument = destination
       LAR  AR1,XR0      Use AR1 as pointer
       ADD  ONE          Increment argument pointer
       B    MOVC$M
MOVC$1 POP
MOVC$M TBLR XR0          Read length of data
       LAR  AR0,XR0      AR0 is loop counter
       LARP 0
       MAR  *-           Decrement counter
```

```
        ADD  ONE            Increment argument pointer
MOVC$L  LARP 1
        TBLR *+,AR0         Read constant
        ADD  ONE
        BANZ MOVC$L         Loop for length of data
        PUSH                Restore return address
        RET
        DEF MOVC$,MOVC$1
*
* Routines for MOVDAT macro
*
* MOVA$B -  MOVDAT A,B,L
*
  MOVA$B POP
         TBLR XR0           1st Argument is source
         LAR  AR0,XR0
         ADD  ONE           Increment pointer
  MOVCB$ TBLR XR0           Next argument is destination
         LAR  AR1,XR0
         ADD  ONE           Increment pointer
         B    MOV$$M
  *
  * MOVC$A -  MOVDAT A,*,L or MOVDAT A,,L
  *
  MOVC$A POP
         TBLR XR0           Read source argument
         LAR  AR0,XR0
         ADD  ONE           Increment pointer
         B    MOV$$M
  *
  * MOVC$B -  MOVDAT *,B,L or MOVDAT ,B,L
  *
  MOVC$B POP
         B    MOVCB$        Get destination argument
  *
  * MOVC$$ - MOVDAT ,*,L or MOVDAT *,,L or MOVDAT *,*,L
  *
  MOVC$$ POP
  MOV$$M SAR  AR0,XR0       Save source location
         TBLR XR1           Read length
         LAR  AR0,XR1
         LARP 0
         MAR  *-            Decrement count
         SACL XR1           Save return address
         LAC  XR0           Load start address
  MOV$$L LARP 1
         TBLR *+,AR0        Move to data memory
         ADD  ONE           Update source pointer
         BANZ MOV$$L        Loop on array length
         LAC  XR1
         ADD  ONE
         PUSH               Restore return address
         RET
         DEF MOVA$B,MOVC$A,MOVC$B,MOVC$$
  *
  *  MOVROM routines
  *
  *  TBW$$ - MOVROM A,B,L
  *
  TBW$$  POP
         TBLR XR0           Read source address
         LAR  AR0,XR0
         ADD  ONE           Update pointer
  TBW0$  TBLR XR0           Read destination address
```

```
        LAR   AR1,XR0
        ADD   ONE             Update pointer
        B     TBW$M
*
*  TBW$1 - MOVROM A,*,L or MOVROM A,,L
*
TBW$1   POP
        TBLR  XR0             Read source address
        LAR   AR0,XR0
        ADD   ONE             Update pointer
        B     TBW$M
*
*  TBW$0 -  MOVROM *,B,L or MOVROM ,B,L
*
TBW$0   POP
        B     TBW0$           Read destination address
*
*  TBW$$ -  MOVROM *,*,L or MOVROM *,,L or MOVROM ,*,L
*
TBW$01  POP
TBW$M   SAR   AR1,XR0         Save destination address
        TBLR  XR1             Read length of move
        LAR   AR1,XR1
        LARP  1
        MAR   *-              Decrement counter
        SACL  XR1             Save return address
        LAC   XR0             Load destination address
TBW$L   LARP  0
        TBLW  *+,AR1          Move data
        ADD   ONE             Increment pointer
        BANZ  TBW$L           Loop on length
        LAC   XR1
        ADD   ONE
        PUSH                  Restore return address
        RET
        DEF   TBW$$,TBW$1,TBW$0,TBW$01
        END

* End of subroutines
```

DIGITAL SIGNAL PROCESSING

8. DIGITAL SIGNAL PROCESSING

All of the digital signal processing information presented in this Section 8 has been provided to Texas Instruments by Ronald W. Schafer, Russell M. Mersereau, and Thomas P. Barnwell, III, of Atlanta Signal Processors, Inc., and of Georgia Institute of Technology, School of Electrical Engineering.

The purpose of this section is to review the fundamentals of digital signal processing in order to highlight some of the important features of the digital approach and to illustrate how DSP techniques can be applied. The important issues in sampling analog signals will be presented, followed by a discussion of the basic theory of discrete signals and systems. A description of the basic algorithms that are widely used in applications of DSP techniques is also provided, along with some examples of how DSP can be used in the areas of speech and audio processing and in communications. Referral to references listed in Section 8.7 is indicated by brackets surrounding a reference number.

8.1 A-TO-D AND D-TO-A CONVERSION

In most applications, signals originate in analog form, i.e., as continuously varying patterns or waveforms. Thus, the first step in applying DSP techniques to a signal is to convert from continuous to discrete form, thereby obtaining a representation of the signal in terms of a sequence or array of numbers. In practice, this is called analog-to-digital (A-to-D) conversion.

Once the signal has been represented in discrete form, it can be processed or transformed into another sequence or set of numbers by a numerical computation procedure (see Figure 8-1). There is also the possibility of converting from the discrete representation back to analog form using a digital-to-analog (D-to-A) converter. This last stage is often not necessary, especially when the purpose of digital processing is to automatically extract information from the signal. The study of digital signal processing is concerned with both the A-to-D and D-to-A conversion processes as well as with the analysis and design of numerical processing algorithms. Although it is important to fully understand both aspects, they can be treated somewhat independently.

FIGURE 8-1 — BLOCK DIAGRAM OF DIGITAL SIGNAL PROCESSING

A-to-D conversion is conveniently analyzed by representing it as in Figure 8-2. First, it involves a sampling operation wherein a sequence x[n] is obtained by periodically sampling an analog signal. The samples are:

$$x[n] = x_a(nT), \quad -\infty < n < +\infty \tag{1}$$

where T is the sampling period, n is an integer, and 1/T is the sampling frequency or sampling rate with units of samples/s. (The sampling rate is often stated in units of frequency, i.e., Hz or kHz.) In most practical settings, these samples must be represented using binary numbers with finite precision. This involves quantizing the sample values. Thus, the sequence of quantized samples is:

$$\hat{x}[n] = Q[x[n]] \tag{2}$$

where Q[ ] is a nonlinear transformation, such as rounding or truncating to the nearest allowed amplitude level.

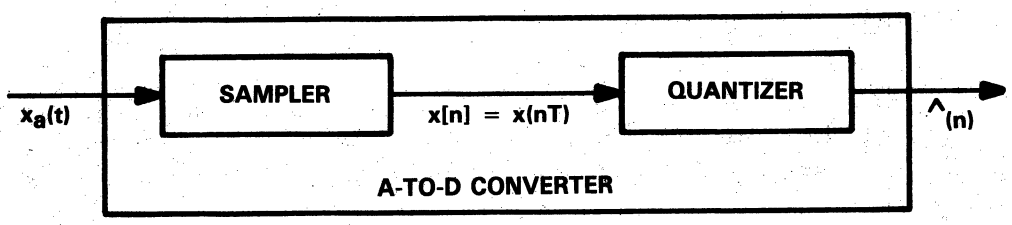


FIGURE 8-2 — ANALOG-TO-DIGITAL CONVERSION PROCESS

8.1.1 Sample Analysis

The important considerations in the sampling operation can be illustrated by a sinusoidal signal:

$$x_a(t) = \cos(\omega_0 t) \tag{3}$$

The resulting sequence of samples is:

$$x[n] = \cos(\omega_0 nT) \tag{4}$$

With this signal, it is simple to illustrate that there is a fundamentally unique problem in the sampling process, i.e., a given sequence of samples can be obtained by sampling an infinite number of analog signals. For example, consider the signal:

$$x_r(t) = \cos((\omega_0 + 2\pi r/T)t) \tag{5}$$

where r is any positive or negative integer. If the sampling period is T, the sampled sequence is:

$$x_r[n] = \cos((\omega_0 + 2\pi r/T)nT) = \cos(\omega_0 nT + 2\pi rn) \tag{6}$$

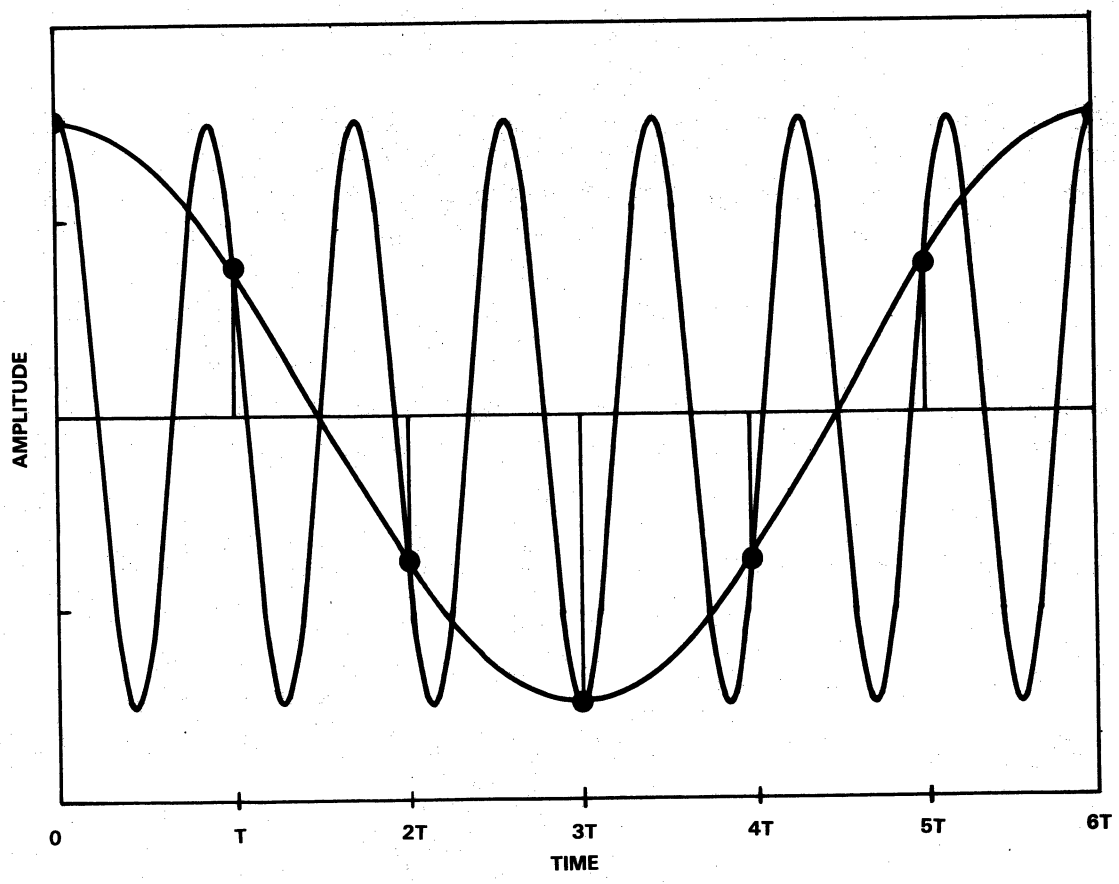
Using a familiar trigonometric identity, xr[n] can be expressed as:

$$x_r[n] = \cos(\omega_0 nT) \cdot \cos(2\pi rn) - \sin(\omega_0 nT) \cdot \sin(2\pi rn) \tag{7}$$

and since both n and r are integers:

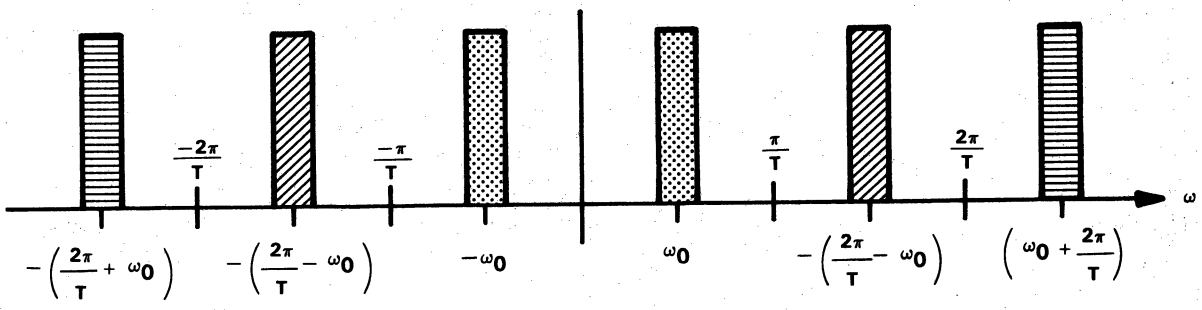
$$x_r[n] = \cos(\omega_0 nT) = x_0[n] \tag{8}$$

Thus, the sequences $x_r[n]$ are all identical to $x_0[n]$, or in other words, the frequencies $(\omega_0 + 2\pi r/T)$ are indistinguishable from the frequency ω_0 after sampling. This is illustrated in Figure 8-3, where two cosine waves are shown passing through the same sample points. The descriptive term for this confused identity is 'aliasing.' The frequency domain representations of the cosine and its aliases are shown in Figure 8-4. The positive and negative frequency components of the cosine wave at $+ - \omega_0$ are shown together with frequency components at $+ - (\omega_0 + 2\pi/T)$ and at $+ - (\omega_0 - 2\pi/T)$ which produce the identical set of samples when the sampling rate is 1/T.



NOTE: The two cosine waves have the same samples when the sampling period is T.

FIGURE 8-3 — TWO COSINE WAVES SAMPLED WITH PERIOD T



NOTE: The positive and negative frequency components of three cosine waves that have the same samples.

FIGURE 8-4 — FREQUENCY COMPONENTS OF THREE COSINE WAVES

The ambiguity of this situation can be removed by imposing a constraint on the size of ω_0 relative to the sampling frequency $\omega_S = 2\pi/T$ (in radians/s). If $\omega_0 < \pi/T$, then all of the frequencies $\omega_r = (\omega_0 + 2\pi r/T)$ will be larger in magnitude than ω_0. Thus, there is no ambiguity if it is determined in advance that $\omega_S > 2\omega_0$, i.e., SAMPLING MUST OCCUR AT A RATE THAT IS GREATER THAN TWICE THE HIGHEST FREQUENCY IN THE SIGNAL. This is true in general for any signal whose Fourier transform is bandlimited, as explained in the following paragraphs.

If the above condition is met, it is possible to recover xa(t) from x[n] by continuously interpolating between the samples, using an interpolation formula of the form:

$$\bar{x}_a(t) = \sum_{n=-\infty}^{\infty} x[n] \cdot P_a(t-nT) \qquad (9)$$

If $P_a(t)$ is a square pulse of duration T, the resulting interpolated waveform (reconstructed signal) has a staircase appearance, as in Figure 8-5. This is a good model for the output of most practical D-to-A converters. A better approximation to the original analog signal can be obtained by smoothing the sharp pulses with a lowpass filter. [1-4] If the effective pulse shape in (9) is:

$$P_a(t) = \frac{\sin \frac{\pi}{T} t}{\frac{\pi}{T} t} \qquad (10)$$

then the original signal $x_a(t)$ can be recovered from the samples x[n] if the Fourier transform of $x_a(t)$ is bandlimited (i.e., identically zero above some frequency which is less than π/T).

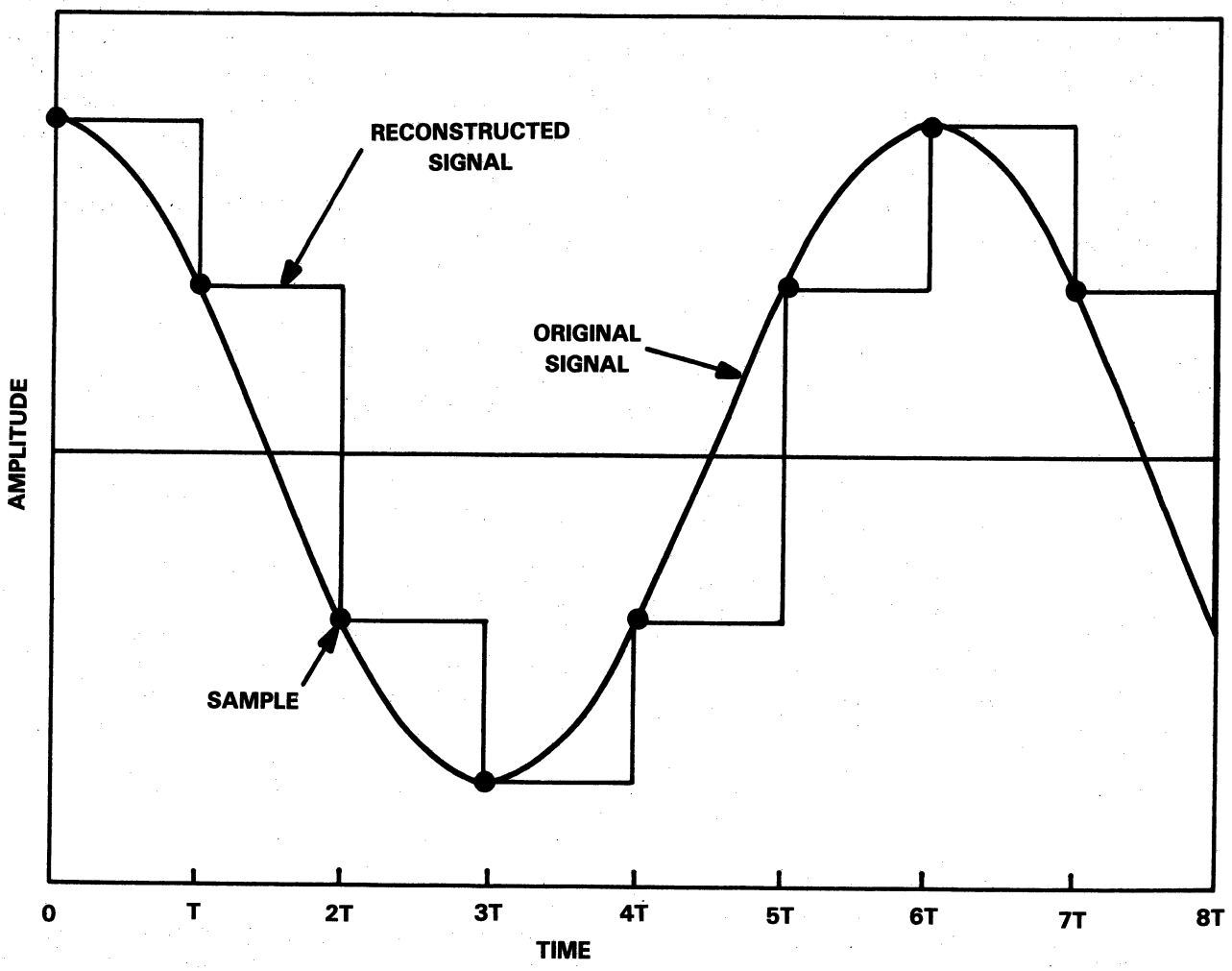


FIGURE 8-5 — D-TO-A CONVERSION USING A ZERO-ORDER HOLD

8.1.2 Sample Quantization

The other aspect of A-to-D conversion is concerned with the quantization of the samples. Figure 8-6 shows an eight-level quantizer which illustrates the important aspects of the quantization operation. Each quantization level is represented by a binary number (three bits in this case). Although the assignment of binary codes to the quantization levels is arbitrary, it is obviously advantageous to assign binary symbols in a scheme which permits convenient implementation of arithmetic operations on the samples (e.g., two's complement, as in Figure 8-6).

Once the number of quantization levels has been fixed (usually between 2^8 and 2^{16} for most signal processing applications), the binary numerical representation of the samples is related to the amplitude of the analog signal by the quantization stepsize Δ. The choice of Δ depends upon the peak-to-peak amplitude range of the signal. If the B-bit code is used, then Δ should be chosen so that:

$$\Delta \cdot 2^B = \text{Peak-to-peak signal amplitude} \qquad (11)$$

With this constraint, the maximum error in a sample value would be $+ -\Delta/2$, so that in general, the average quantization error will be proportional to Δ. This points up a fundamental dilemma in quantization, i.e., for a fixed stepsize, the relative error becomes large as the sample amplitude decreases. Thus, if signal amplitude varies widely (i.e., the signal has a wide dynamic range), then it may be necessary to use a large number of quantization levels to keep the relative quantization error within acceptable limits. Alternative approaches, often used in speech processing, are the use of either a nonuniform set of quantization levels or the adaptation of the stepsize to the amplitude of the input signal. [2]

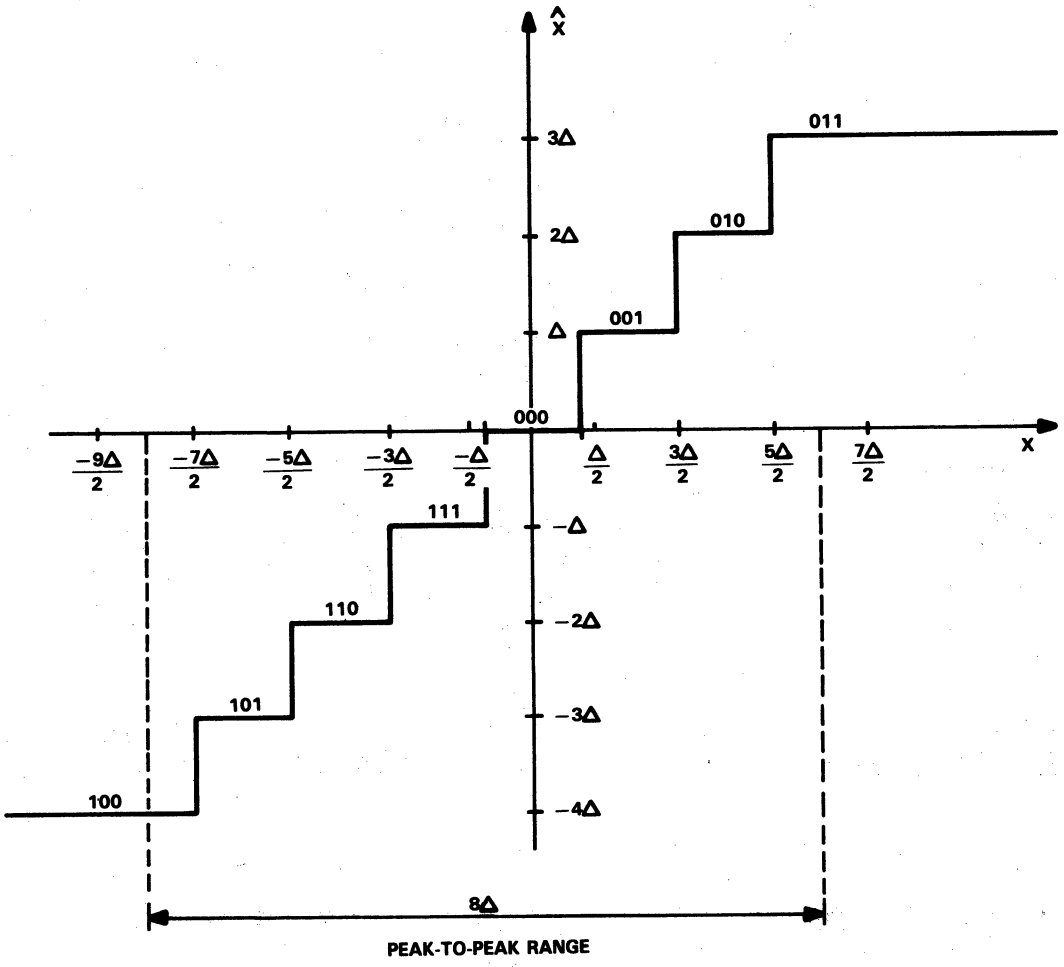


FIGURE 8-6 — AN EIGHT LEVEL (THREE-BIT) QUANTIZER

In the uniform stepsize non-adaptive case, it is often useful to represent the quantized signal as:

$$\hat{x}[n] = x[n] + e[n] \tag{12}$$

where e[n] is, by definition, the quantization error. This model for A-to-D conversion is depicted in Figure 8-7. As seen above:

$$-\Delta/2 \leqslant e[n] < +\Delta/2 \tag{13}$$

As a result, the root mean squared value of e[n] is proportional to Δ, which in turn is inversely proportional to 2^B where B is the number of bits in the binary coded samples. Thus, the signal-to-quantization noise ratio defined as:

$$SNR = 10 \cdot \log_{10}\left(\frac{\text{signal power}}{\text{noise power}}\right) \tag{14}$$

increases by 6 dB for each doubling of the number of quantization levels (i.e., for each additional bit in the word length).

Another important point is that from the viewpoint of statistical measurements, the sequence of noise samples appears to be uniformly distributed in amplitude and uncorrelated from sample to sample whenever the number of quantization levels (bits) is large. Thus, the model of the A-to-D conversion operation in Figure 8-7 consists of an ideal sampler whose output samples are corrupted by an additive white noise whose power increases exponentially as the number of bits/sample decreases.

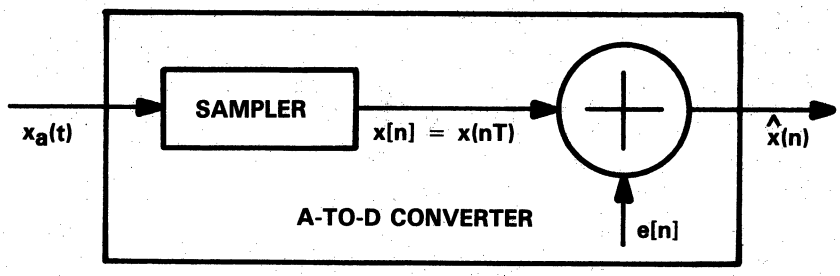


FIGURE 8-7 — QUANTIZATION AS ADDITIVE NOISE

8

8.2 BASIC THEORY OF DISCRETE SIGNALS AND SYSTEMS

Since signals are represented in discrete form as sequences of samples, a discrete system or digital signal processor is simply a computational algorithm for transforming an input sequence of samples into an output sequence.

8.2.1 Linear Systems

As in analog systems, a linear system is one which obeys the principle of superposition, and a time-invariant (or in general, shift-invariant) system is one for which the input-to-output transformation algorithm does not change with time. Linear time-invariant systems are exceedingly important because they are relatively easy to design and because they can be used to perform a wide variety of signal processing functions.

As a direct consequence of linearity and time invariance, the output sequence for any linear time-invariant system is obtained from the input sequence by the repeated evaluation of the convolution sum relation:

$$y[n] = \sum_{k=-\infty}^{\infty} h[k] \cdot x[n-k] \qquad -\infty < n < \infty \tag{15}$$

where h[n] is the response of the system to the unit sample (or impulse) sequence:

$$\delta[n] = \begin{cases} 1 & n = 0 \\ 0 & n \neq 0 \end{cases} \tag{16}$$

The convolution sum equation is very similar in form to the convolution integral that describes the operation of a continuous-time linear time-invariant system. In contrast to the analog system, however, the convolution sum equation (15) serves not only as a theoretical description of discrete linear time-invariant systems in general, but it can be used to implement certain types of linear systems.

8.2.2 Fourier Transform Representations

As in the analog case, Fourier analysis is a valuable tool in the theory and design of discrete signals and systems. The discrete-time Fourier transform representation is defined by the equations:

$$X(e^{j\omega T}) = \sum_{n=-\infty}^{\infty} x[n] \cdot e^{-j\omega nT} \tag{17A}$$

$$x[n] = \frac{T}{2\pi} \int_{-\pi}^{\pi} X(e^{j\omega T}) e^{j\omega nT} d\omega \tag{17B}$$

The first equation (17A) is a direct Fourier transform of the sequence x[n], and the second equation (17B) is the inverse Fourier transform. A notable property of $X(e^{j\omega T})$ is that it is always a periodic function of ω with period $2\pi/T$.

In the analog case, the Laplace transform is often more useful and convenient than the Fourier transform, because it can be used to represent a wider class of signals and because algebraic expressions involving the Laplace transform are less cumbersome than those involving Fourier transforms. For these same reasons, the z-transform is often preferred to the Fourier transform for discrete sequences. The z-transform representation is defined by:

$$X(z) = \sum_{n=-\infty}^{\infty} x[n] z^{-n} \tag{18A}$$

$$x[n] = \frac{1}{2\pi j} \oint_C X(z) z^{n-1} dz \tag{18B}$$

where C is a closed contour lying in the region of convergence of the power series in (18A).

Comparison of the Fourier transform (17A) and the z-transform (18A) shows that:

$$X(e^{j\omega T}) = X(z)\Big|_{z = e^{j\omega T}} \tag{18C}$$

i.e., the Fourier transform, when it exists, is just the z-transform evaluated on a circle of radius one in the complex z-plane.

One of the most important reasons for the use of frequency domain representations is the result that if y[n] is the output of a linear time-invariant system, then its z-transform (and thus its Fourier transform) satisfies the equation:

$$Y(z) = H(z) \cdot X(z) \tag{19}$$

where H(z) and X(z) are the z-transforms of the unit sample response of the system and the input to the system, respectively. Many of the design techniques which are available are based upon approximating a desired transfer function H(z).

Another advantage of the Fourier transform representation is that it provides a very convenient means of showing the relationship between a sequence of samples and the original analog signal from which the samples were obtained. Specifically, if $x[n] = x_a(nT)$, then:

$$X(e^{j\omega T}) = \frac{1}{T} \sum_{k=-\infty}^{\infty} X_a(\omega + 2\pi k/T) \tag{20}$$

where $X_a(\omega)$ is the Fourier transform of the analog signal $x_a(t)$. [1]

From this relationship between the Fourier transform of the sequence x[n] and the Fourier transform of the analog signal, it is clear that what is true for the cosine wave is also true in general. That is, there is a possibility that the images of the analog Fourier transform may overlap and since they are added together, it would be impossible to unscramble the effects of this aliasing distortion. Figure 8-8 illustrates the implications of (20) for two sampling rates. Figure 8-8A shows a bandlimited analog Fourier transform where $X_a(\omega) = 0$ for $|\omega| > \omega N$. The frequency ωN is often called the Nyquist frequency. Figure 8-8B shows the Fourier transform of a sequence of samples where the sampling frequency $\omega S = 2\pi/T$ is such that $\omega S > 2\omega N$. Figure 8-8c shows the case when $\omega S > 2\omega N$. No aliasing distortion occurs if $X_a(\omega)$ is bandlimited and if the sampling frequency is greater than twice the Nyquist frequency. Thus, it is essential that analog signals be bandlimited to the proper frequency before sampling. Even if the signal is 'naturally' bandlimited, it is well to remember that since additive noise may have a much broader spectrum than the signal, analog lowpass filtering is almost always necessary prior to sampling. Since it is generally desirable to minimize the sampling rate so as to minimize the computational intensity of the processor, sharp cutoff analog filters may be required. In situations where the expense of such filters is prohibitive, but sufficient numerical processing capability is available, it is possible to use low-order analog filters and sample at a higher sampling rate to avoid aliasing. Then, the resulting sequence of samples can be filtered digitally and the sampling rate reduced appropriately by decimating (throwing away samples) the digitally filtered sequence. [2] Such techniques are also useful in implementing low-noise A-to-D conversion systems, using delta modulation or other simple digitizing systems. [5]

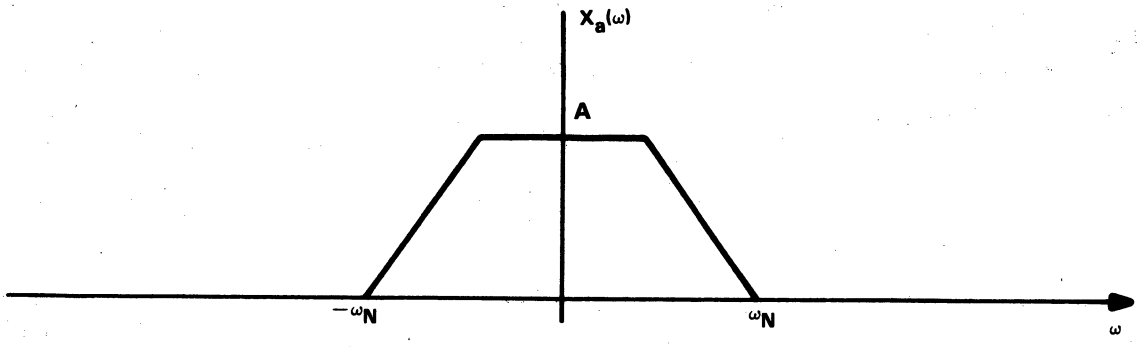


FIGURE 8-8A — FOURIER TRANSFORM OF ANALOG SIGNAL

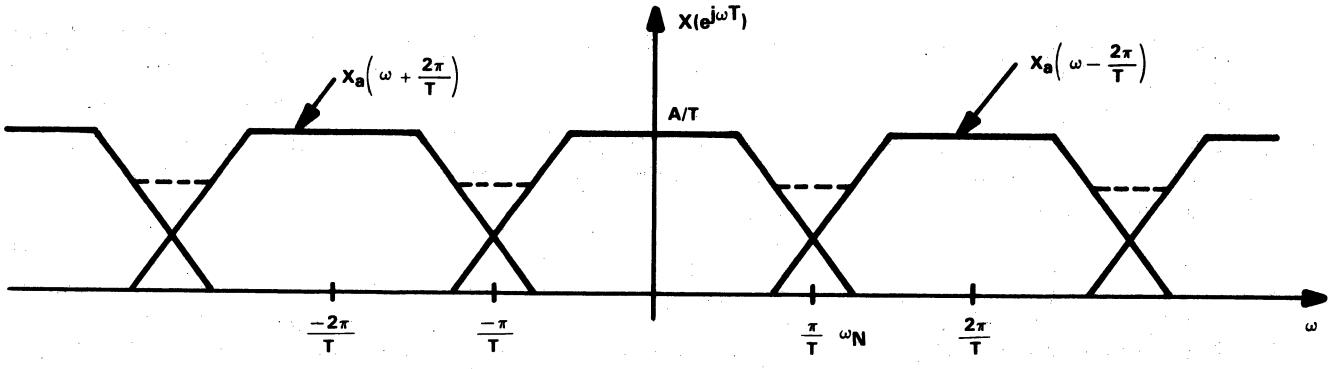


FIGURE 8-8B — FOURIER TRANSFORM OF SAMPLES FOR $2\pi/T > 2\omega_N$

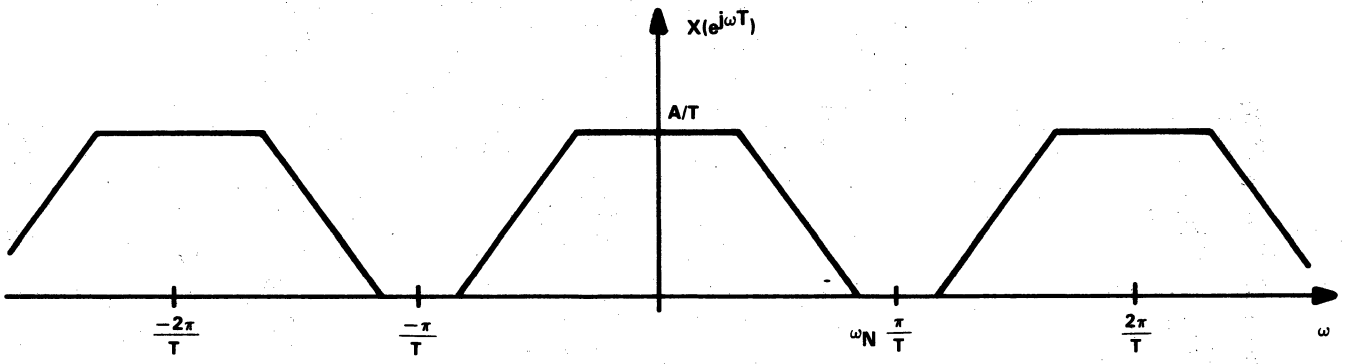


FIGURE 8-8C — FOURIER TRANSFORM OF SAMPLES FOR $2\pi/T > 2\omega_N$

FIGURE 8-8 — FOURIER TRANSFORM SAMPLING

8.3 DESIGN AND IMPLEMENTATION OF DIGITAL FILTERS

Linear filtering is one of the most important digital signal processing operations. As in the analog system, digital filters can be used for separating signals from noise, for compensating for previous linear distortions, for separating signal components from an additive combination of signals, and in modeling of many classes of signals. Some of the important techniques for implementation and design of digital filters are presented in the following paragraphs.

8.3.1 Digital Filter Structures

There are two classes of linear shift-invariant systems. The first class contains all such systems for which the unit sample response is of finite length, e.g., $h[n] = 0$ for $n > 0$ and for $n > M$. Such systems are called finite duration impulse response (FIR) systems. For such systems, it is clear from the convolution sum equation (15) that:

$$y[n] = \sum_{k=0}^{M} h[k] \cdot x[n-k] \tag{21}$$

so that the computation of each value of the output sequence requires $M + 1$ multiplications and M additions, i.e., the accumulation of $M + 1$ products. Thus, the convolution sum expression can be used to implement FIR systems.

Systems which have infinite duration impulse responses are called IIR systems. In general, it is not feasible to use the convolution sum expression to compute the output of such systems. However, an interesting and useful class of IIR systems does exist. These are systems whose input and output satisfy a linear constant coefficient difference equation of the form:

$$y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k] \tag{22}$$

For such systems, this equation can be used recursively to compute the output from the input sequence and N previously computed output samples. When all the a_k's are zero, (22) reduces to (21) so that (22) turns out to be a general description of all computationally feasible (i.e., realizable) linear time-invariant systems.

By finding the z-transform of both sides of (22), the transfer function of this class of systems is easily found to be:

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{k=1}^{N} a_k z^{-k}} \tag{23}$$

Since $b_k x[n-k]$ has z-transform $b_k z^{-k} X(z)$, there is a direct correspondence between terms in the numerator and denominator of H(z) in (23) and terms in the difference equation (22).

Block diagrams may be used to depict the computational procedure for implementing a digital filter. Figure 8-9 depicts two systems whose input and output satisfy the difference equation (22) and thus have the same transfer function (23). The operation of addition and multiplication are represented in standard block diagram notation while the delays are represented by systems with transfer functins z^{-1}. ($M = N = 4$ is used for convenience only.) Figure 8-9A shows the direct representation of the difference equation (22). This is sometimes called the Direct Form I structure for a system with transfer function (23). If $N = 0$ (i.e., all the a_k's are zero), then the system is a FIR system. Thus, the left half of Figure 8-9A is illustrative of the general Direct Form implementation of a FIR system. Also note that in general the left half implements the numerator (or zeros) of H(z) while the right half implements the denominator (or poles) of the transfer function.

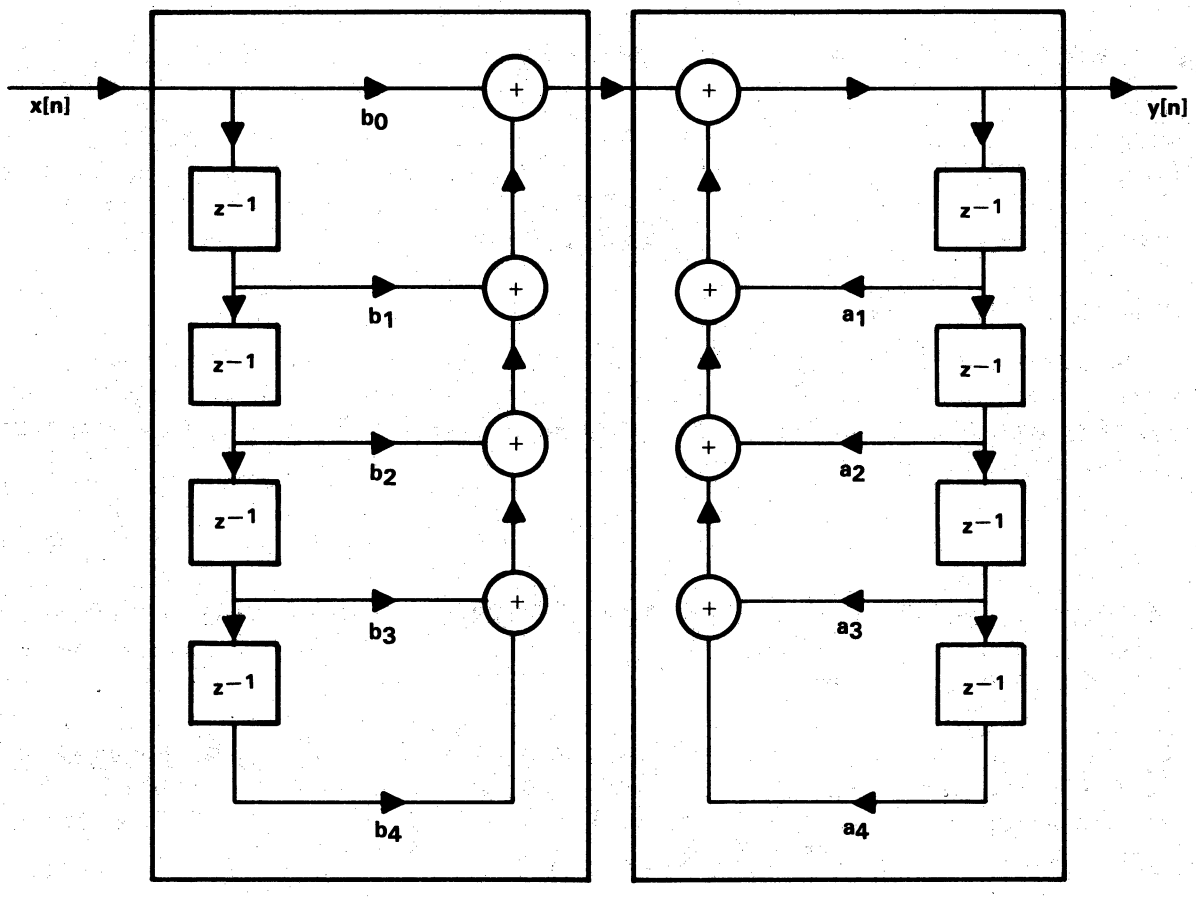


FIGURE 8-9A — DIRECT FORM I

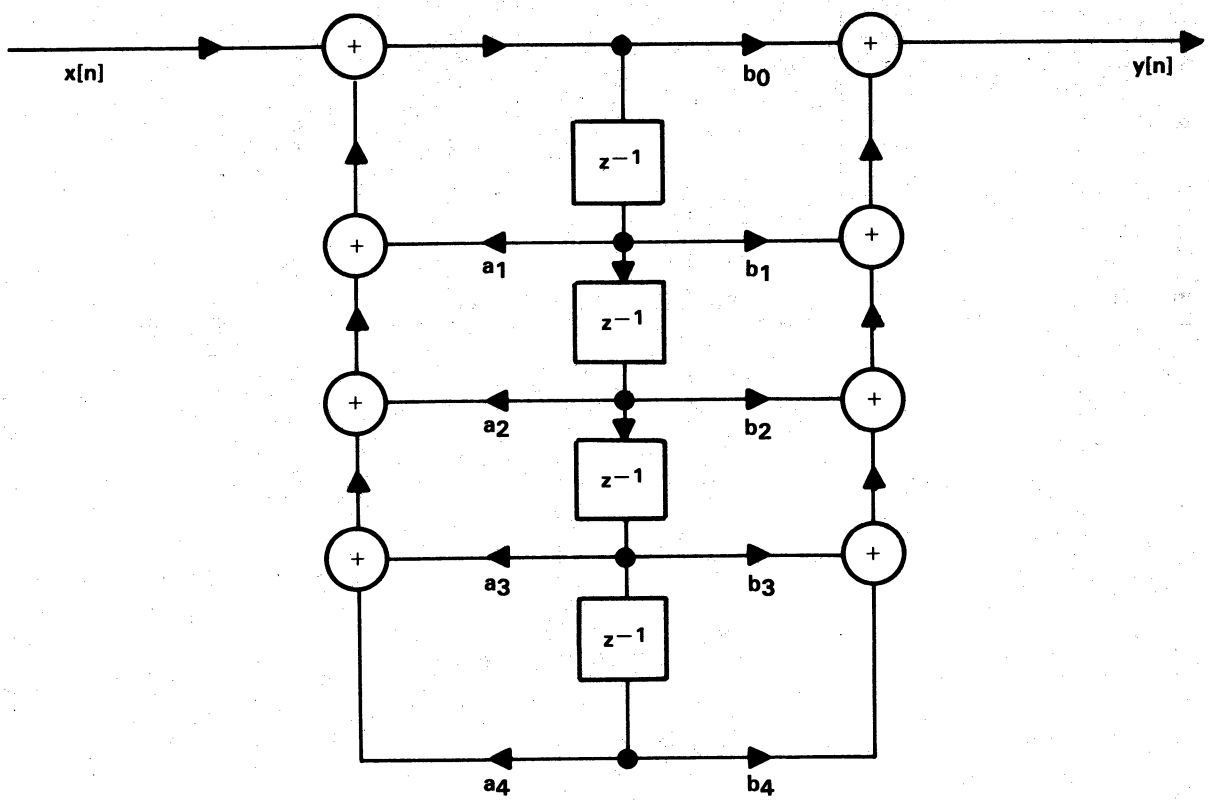


FIGURE 8-9B — DIRECT FORM II

FIGURE 8-9 — DIRECT FORMS I AND II

Figure 8-9B is obtained from Figure 8-9A. For linear time-invariant systems in cascade, the overall transfer function is the product of the individual transfer functions. Thus, the overall transfer function is the same regardless of the order in which the systems are cascaded. If the two subsystems of Figure 8-9A are interchanged, the delay chains of the two systems can be combined. This structure is often called the Direct Form II. Both forms require the same number of arithmetic operations, but the Direct Form II requires up to 50 percent fewer memory registers for storing the past values of the input and output. It is important to understand that although both forms have the same overall transfer function, they correspond to different difference equations. The difference equation for Figure 8-9A is given in (22) while the set of difference equations represented by Figure 8-9B is:

$$w[n] = \sum_{k=1}^{N} a_k w[n-k] + x[n] \tag{24A}$$

$$y[n] = \sum_{k=0}^{M} b_k w[n-k] \tag{24B}$$

Other structures (sets of difference equations) can be found for implementing a given rational transfer function such as (23). The cascade form is obtained by factoring the numerator and denominator of H(z) into second-order factors and pairing numerator and denominator factors to form:

$$H(z) = A \cdot \prod_{k=1}^{\frac{N}{2}} \left(\frac{1 + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 - a_{1k} z^{-1} - a_{2k} z^{-2}} \right) \tag{25}$$

For simplicity it is assumed that N is even. When N is odd or when $M \neq N$, some of the coefficients in (25) will be zero. The structure suggested by (25) can be implemented with a cascade of second-order sections implemented in any desired form. Figure 8-10 shows an example for $N = 4$.

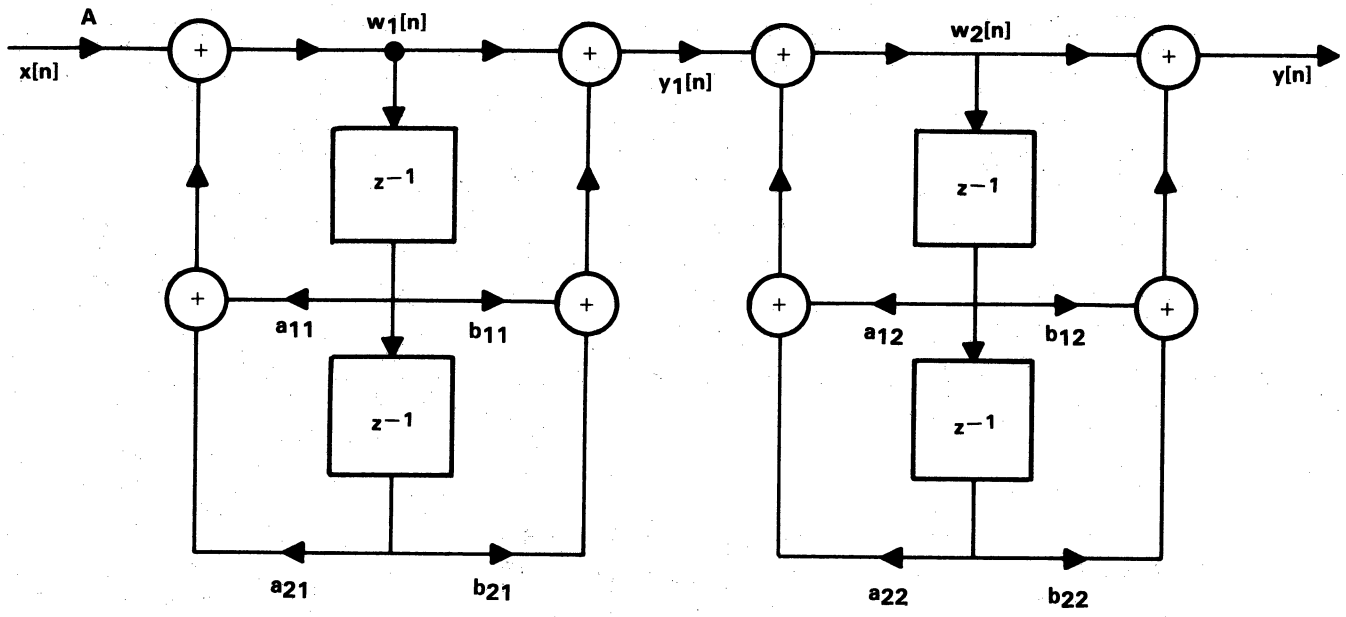


FIGURE 8-10 — CASCADE STRUCTURE FOR N = 4

The corresponding set of difference equations is:

$$y_0[n] = A \cdot x[n] \tag{26A}$$

$$w_k[n] = a_{1k}w_k[n-1] + a_{2k}w_k[n-2] + y_{k-1}[n] \qquad k = 1, 2, \ldots, N/2 \tag{26B}$$

$$y_k[n] = w_k[n] + b_{1k}w_k[n-1] + b_{2k}w_k[n-2] \qquad k = 1, 2, \ldots, N/2 \tag{26C}$$

$$y[n] = y_{\frac{N}{2}}[n] \tag{26D}$$

Still another form for the general transfer function of (25) is obtained from a partial fraction expansion of H(z) in the form of:

$$H(z) = A_0 + \sum_{k=1}^{\frac{N}{2}} \frac{b_{0k} + b_{1k}z^{-1}}{1 - a_{1k}z^{-1} - a_{2k}z^{-2}} \tag{27}$$

The set of difference equations corresponding to this form of the transfer function is:

$$w_k[n] = a_{1k}w_k[n-1] + a_{2k}w_k[n-2] + x[n] \qquad k = 1, 2, \ldots, N/2 \tag{28A}$$

$$y_k[n] = b_{0k}w_k[n] + b_{1k}w_k[n-1] \qquad k = 1, 2, \ldots, N/2 \tag{28B}$$

$$y[n] = A_0x[n] + \sum_{k=1}^{\frac{N}{2}} y_k[n] \tag{28C}$$

There is literally an infinite number of alternative structures for implementing a digital filter with a given transfer function, but the ones discussed above are the most commonly used because of the ease with which they can be obtained from the transfer function and, in the case of the cascade and parallel forms, because they are relatively insensitive to coefficient quantization and round-off errors. It is important to note that the basic arithmetic process in digital filtering is multiplication of a delayed sequence value by a fixed coefficient, followed by the accumulation of the result. This is a built-in operation of the TMS32010.

8.3.2 Digital Filter Design

A number of ways to implement a linear time-invariant system having a rational transfer function have been presented. Designing the system to meet a set of prescribed specifications is equally important. The specifications for a filter design are most frequently applied to the frequency response of the filter, i.e., to the Fourier transform of the impulse response. For example, a frequency selective filter, such as a lowpass, bandpass, highpass, or bandstop filter, may be required; or an approximation of a differentiator frequency response (i.e., $j\omega$), or a 90-degree phase shift, or in the case of compensators or equalizers, an approximation of the reciprocal of some given frequency response may be desired. In all these cases, the designer is concerned with finding the b_k's in the FIR case, or the a_k's and b_k's in the IIR case, so that the corresponding $H(e^{j\omega T})$ approximates a desired function according to some approximation error criterion. Many approximation techniques exist, and it is possible to design very accurate approximations to a wide variety of frequency responses.

A valuable collection of digital filter design programs is available from IEEE Press. [6] A reader who wants to use these programs or to write design programs is encouraged to consult the texts and reference books [1,3,7] on digital signal processing to obtain a complete understanding of each method. The following paragraphs include a survey of the important techniques, along with the advantages and limitations of each one.

The design of IIR filters has traditionally been based upon the transformation of an analog filter approximation to a digital filter. The basic approaches are impulse invariance and bilinear transformation. The former approach is based upon defining the unit sample response of the digital filter to be the sequence obtained by sampling the impulse response of an analog filter. In this case, the analog filter must be designed so that the resulting digital filter will meet its specifications. Because of the aliasing inherent in sampling, the impulse invariance method is not effective for highpass or bandstop filter types, and the detailed shape of the analog frequency response is preserved only in highly bandlimited cases, such as lowpass filters with high stopband attenuation.

In the bilinear transformation method, the system function H(z) of the digital filter is obtained by an algebraic (bilinear) transformation of the system function (Laplace transform of the impulse response) of an analog filter, i.e., the Laplace variable s is replaced by $2(1 - z^{-1})/(1 + z^{-1})$. Because the bilinear transformation causes a warping of the $j\omega$-axis of the s-plane onto the unit circle of the z-plane, the bilinear transformation method is useful primarily for the design of frequency selective filters where the frequency response consists of flat passbands and stopbands. The passband and stopband cutoff frequencies of the analog filter must be 'prewarped' so that the resulting digital filter meets its specifications. Because the bilinear transformation maps the entire jw-axis of the s-plane onto the unit circle, the equiripple amplitude response of an elliptic filter will be preserved. Thus, optimal magnitude responses can be obtained for IIR filters using bilinear transformation of analog elliptic filters.

A major reason that the above methods are widely used is the existence of a variety of approximation methods for analog frequency selective filters. That is, one can use the Butterworth, Bessel, Chebyshev, or elliptic filter approximation methods for the analog filter and then simply transform the analog filter to a digital filter by either the impulse invariance or bilinear transformation methods. As an illustration of this general method, Figure 8-11A shows the magnitude response and Figure 8-11B shows the phase response of a fourth-order elliptic filter obtained by the bilinear transformation method. The difference equations for implementation of this filter as a cascade of two second-order Direct Form II sections are:

$$y_0[n] = 0.11928 \cdot x[n] \tag{29A}$$

$$w_1[n] = 0.34863 \cdot w_1[n-1] - 0.17168 \cdot w_1[n-2] + y_0[n] \tag{29B}$$

$$y_1[n] = w_1[n] + 1.8345 \cdot w_1[n-1] + w_1[n-2] \tag{29C}$$

$$w_2[n] = -0.12362 \cdot w_2[n-1] - 0.71406 \cdot w_2[n-2] + y_1[n] \tag{29D}$$

$$y_2[n] = w_2[n] + 1.26185 \cdot w_2[n-1] + w_2[n-2] \tag{29E}$$

$$y[n] = y_2[n] \tag{29F}$$

The block diagram representation for the above set of difference equations is identical to Figure 8-10, with the appropriate identification of the coefficients.

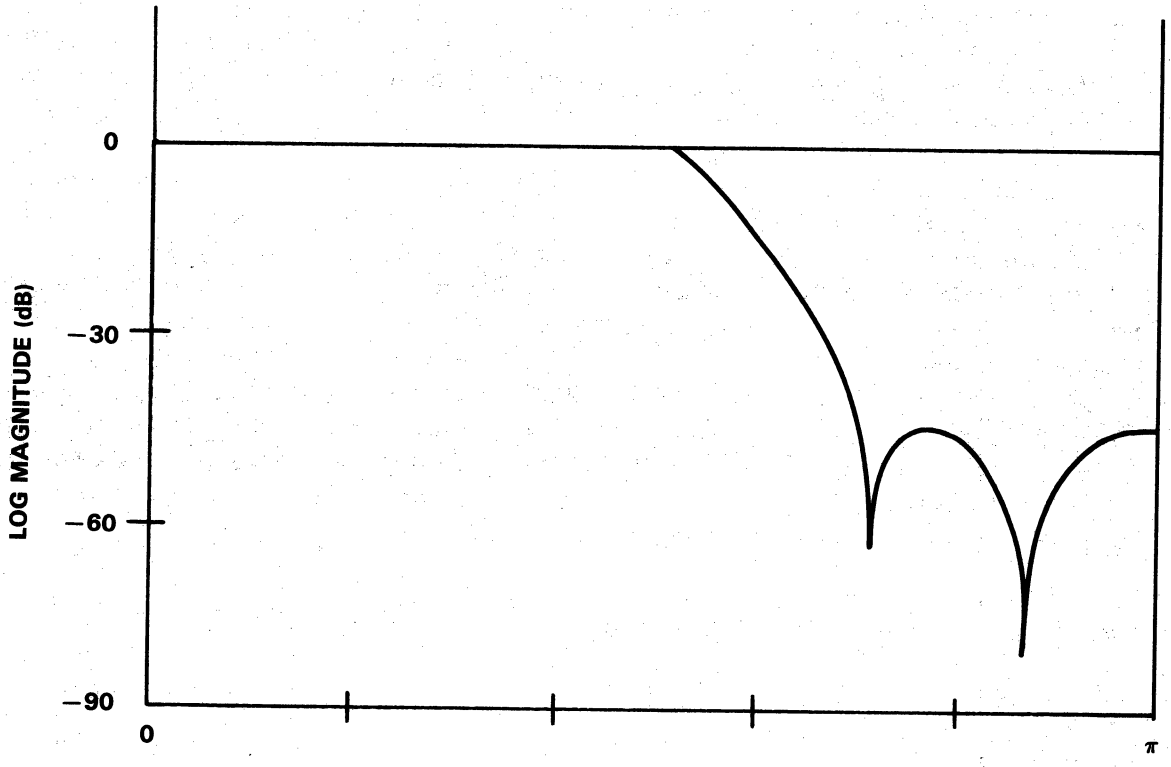


FIGURE 8-11A — LOG MAGNITUDE OF FREQUENCY RESPONSE

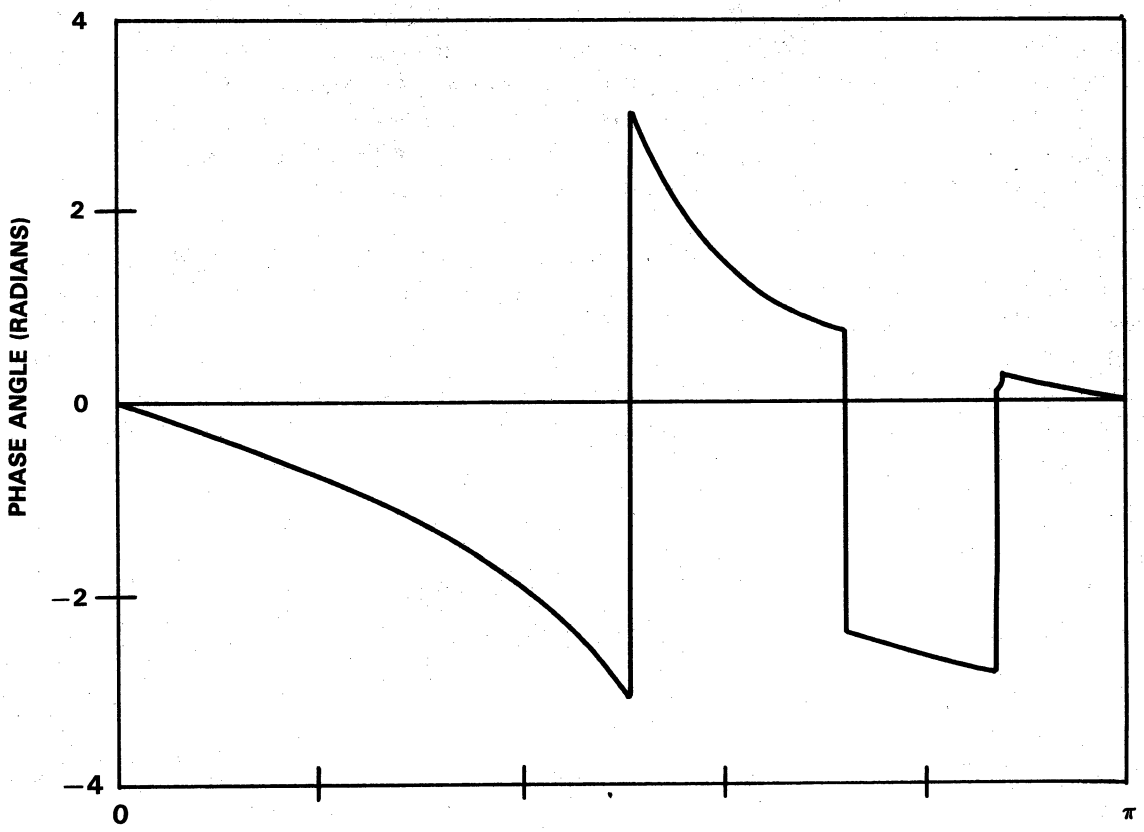


FIGURE 8-11B — PHASE ANGLE OF FREQUENCY RESPONSE

FIGURE 8-11 — FOURTH-ORDER ELLIPTIC DIGITAL FILTER

It is relatively simple to design IIR filters using tables of analog filter designs and a calculator. Alternatively, a program for designing IIR digital filters by bilinear transformation of Butterworth, Chebyshev, and elliptic filters has been given by Dehner in the IEEE Press Book. [6, Section 6.1]

The bilinear transformation method can be termed a 'closed form' solution to the IIR digital filter design problem in the sense that an analog filter can be found in a non-iterative manner to meet a set of prescribed approximation error specifications, and then the digital filter can be obtained in a straightforward way by applying the bilinear transformation.

Another approach is as follows:

1) Define an ideal frequency response function,

2) Set up an approximation error criterion,

3) Pick an implementation structure, i.e., order of numerator and denominator of H(z), cascade, parallel, or direct form,

4) Vary the filter coefficients systematically to minimize the approximation error criterion,

5) If the approximation is not good enough, increase the order of the system and repeat the design process.

A variety of such iterative design techniques have been proposed for both IIR and FIR filters. Deczky has developed a design program which minimizes a pth-order error norm. It is capable of both magnitude and group delay (negative derivative of phase with respect to frequency) approximations. [6, Section 6.2] Another optimization program for magnitude approximations only has been written by Dolan and Kaiser. [6, Section 6.3] Both this program and the Deczky program assume that the transfer function H(z) is a product of second-order factors.

Somewhat different approaches have been developed for the design of FIR filters, since there really is no counterpart of the FIR filter for the analog system. In addition, FIR discrete-time filters can have an exactly linear phase response. Since a linear phase response corresponds to only a delay, attention can be focused on approximating the desired magnitude response without concern for the phase. In most IIR design methods, the phase is ignored, and one is forced to accept whatever phase distortion is imposed by the design procedure. The condition for linear phase of a casual FIR system is the symmetry condition:

$$\begin{aligned} h[n] &= \pm h[M-n] \qquad 0 \leqslant n \leqslant M \\ &= 0 \qquad\qquad\quad \text{otherwise} \end{aligned} \tag{30}$$

In the case of the + sign in (30), the frequency response will be:

$$H(e^{j\omega T}) = R(\omega T) \cdot e^{-j\omega T \left(\frac{M}{2}\right)} \tag{31}$$

where $R(\omega T)$ is a real function of frequency. Such frequency responses are appropriate for approximating frequency selective filters. In the case of the minus sign in (30):

$$H(e^{j\omega T}) = jI(\omega T) \cdot e^{-j\omega T \left(\frac{M}{2}\right)} \tag{32}$$

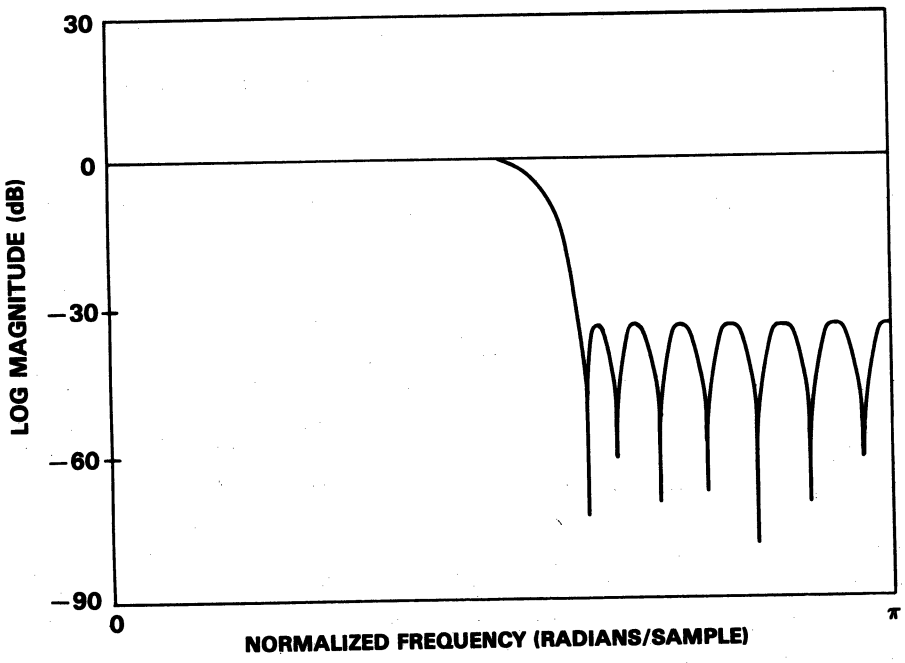
where $I(\omega T)$ is also a real function of frequency. Such frequency responses are required for approximating differentiators and Hilbert transformers (90-degree phase shifters).

The most straightforward approach to the design of FIR filters is a technique often called the 'window method.' In this approach, an ideal frequency response function is first defined. Then, the corresponding ideal impulse response is determined by evaluating the inverse Fourier transform of the ideal frequency response. (In picking the ideal frequency response, the linear phase condition may or may not be applied depending on what is most appropriate.) The ideal impulse response will in general be of infinite length. An approximate impulse response is computed by truncating the ideal impuse response to a finite number of samples and tapering the remaining samples with a window function. With appropriate choice of the window function, a smooth approximation to the ideal frequency response is obtained even at points of discontinuity. Many window functions have been proposed, but the most useful window for filter design is perhaps the one proposed by Kaiser [8] since it has a parameter which, in conjunction with the window length, can be used systematically to trade off between approximation error in slowly varying regions of the ideal response (e.g., the stopband) and sharpness of transition at discontinuities of the ideal frequency response. A program for window design of FIR frequency selective filters is given by Rabiner and McGonegal [6, Section 5.2]

FIR filters designed by the window method are not optimal, but in many cases the flexibility and simplicity of the method outweigh the relatively small cost of increased filter length. In cases where optimal designs are required for computationally efficient implementations, the Parks-McClellan algorithm can be used to obtain equiripple or Chebyshev-type approximations. Such designs are optimal in the sense of having the sharpest transitions between passbands and stopbands for a given filter length and approximation error. This iterative algorithm is based upon the principles of the Remez exchange algorithm. A program written by McClellan, Parks, and Rabiner is capable of designing frequency selective FIR filters as well as differentiators and 90-degree phase shifters. [6, Section 5.1] An example of the type of filters obtainable by this method is shown in Figure 8-12. Only the magnitude response is shown since the phase is linear. The impulse response of this system is given in Figure 8-13. With the symmetry of h[k], the difference equation for computing the filtered output is:

$$y[n] = h[16] \cdot x[n-16] + \sum_{k=0}^{15} h[k]\,[x[n-k] + x[n+k-32]] \qquad (33)$$

where h[k] is as given in Figure 8-13. (Note that M = 32.)



NOTE: This FIR lowpass filter was designed by the Parks-McClellan algorithm (M = 32). The phase is linear with slope corresponding to a delay of 16 samples.

FIGURE 8-12 – FREQUENCY RESPONSE OF FIR LOWPASS FILTER

IMPULSE RESPONSE OF EQUIRIPPLE LOWPASS FILTER

```
H(0)  = 58211200E-02 = H(32)
H(1)  = 12569420E-01 = H(31)
H(2)  = 11188270E-01 = H(30)
H(3)  = 49952310E-02 = H(29)
H(4)  = 14605940E-01 = H(28)
H(5)  = 29798820E-02 = H(27)
H(6)  = 22352550E-01 = H(26)
H(7)  = 42574740E-02 = H(25)
H(8)  = 30249490E-01 = H(24)
H(9)  = 17506790E-01 = H(23)
H(10) = 37882950E-01 = H(22)
H(11) = 41403080E-01 = H(21)
H(12) = 44224020E-01 = H(20)
H(13) = 91748770E-01 = H(19)
H(14) = 48421950E-01 = H(18)
H(15) = 31334940E-00 = H(17)
H(16) = 54989020E-00 = H(16)
```

FIGURE 8-13 — IMPULSE RESPONSE OF EQUIRIPPLE LOWPASS FILTER

8.4 QUANTIZATION EFFECTS

When digital filters are implemented on any computer, the finite precision of the machine can lead to deviations from ideal performance. Problems which arise are due to quantization of the coefficients of the difference equation and roundoff of products prior to accumulation or roundoff of accumulated products.

When a discrete system is designed to meet a certain set of specifications, the design program usually will compute the filter coefficients using floating-point arithmetic and the output of the design program will be a set of coefficients specified to at least 32-bit floating-point precision. When these coefficients are used in a fixed-point implementation, it is generally necessary to quantize the coefficients to fewer bits, e.g., 16 bits. The resulting frequency response will differ from the original design. It may not meet the original specifications and may even be unstable. This is analogous to the component tolerance problem in implementing analog active filters. Sensitivity of the frequency response to errors in a given coefficient is dependent upon the nature of the desired frequency response, and thus it is difficult to obtain theoretical results with wide generality. However, it is well established both theoretically and experimentally that the direct-form implementation structures for high-order filters are in general much more sensitive to coefficient quantization errors than the equivalent cascade or parallel-form implementations using second-order sections. Therefore, these structures are generally to be preferred in small word-length implementations.

The design program of Dehner [6, Section 6.1] has an option for optimizing filter response with constraints on word length. Steiglitz and Ladendorf have also given an iterative program for designing finite word-length IIR filters. [6, Section 6.4] A program for finite word-length design of FIR filters has been written by Heute. [6, Section 5.4]

Another source of imperfection in implementing digital filters is the 'roundoff noise' that results from quantization of intermediate computations in the difference equation. This problem is particularly acute in IIR filters, where the recursive nature of the implementation algorithm leads to a required word-length that increases linearly with time or to errors which propagate to future computations. For example, with 16-bit input samples and 16-bit coefficients, the first output value will require up to 32-bits for its representation, and in a recursive filter, the next output value will

require 32 + 16, etc. Thus, the products continually must be reduced to fit the word length of the processor. However, the TMS320 has a full 32-bit accumulator so that 16-bit by 16-bit products need not be rounded before addition. Thus, in implementing digital filters, each output value can be computed with 32-bit precision and then rounded to 16-bits for output or for storage of delayed variables.

It can be seen from (21) and (22) that in implementing digital filters, the basic operation is a multiply followed by an accumulate (addition of the product to the sum of previously computed products). An obvious additional problem is the danger of overflow of the accumulator word length. Overflow can be eliminated as a problem by using floating-point arithmetic. However, this leads to quantization of both sums and products, and implementation for floating-point arithmetic leads to much higher costs in processors like the TMS320.

Rounding in digital filter implementations leads to errors in the output of the filters. In many cases, these errors can be modeled as additive noise which is generated by noise sources in the filter structure. (This is analogous to thermal noise generated by resistors in analog active filters.) In other cases, the nonlinear nature of the quantization of products or overflow can lead to a much different effect, i.e., periodic patterns of error samples are generated in the output. These 'limit cycles' are particularly troublesome in situations where the input becomes zero for lengthy intervals. Certain structures have been found which are free of limit cycle behavior. However, these require somewhat more computation than the standard forms. [9] An important point is that limit cycles cannot exist in the output of FIR filters. Since there is no feedback, the output of a FIR system obviously becomes zero if the input is zero over an interval equal to or greater than the length of the unit sample response. [1,3,7]

8.5 SPECTRUM ANALYSIS

Spectrum analysis is another major area of digital signal processing. Spectrum analysis consists of a collection of techniques which are directed either toward the computation of the Fourier transform of a deterministic signal or toward estimation of the power spectral density of a random signal. In the following paragraphs are presented the important concepts and algorithms in discrete-time spectrum analysis.

8.5.1 Discrete Fourier Transform (DFT)

The discrete Fourier transform (DFT) of a finite length sequence is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-j2\pi kn/N} \qquad 0 \leqslant k \leqslant N-1 \tag{34}$$

The DFT is simply a sampled version of the discrete-time Fourier transform of x[n], i.e.:

$$X[k] = X(e^{j\omega k T}) \tag{35}$$

where $\omega_k = 2\pi k/(NT)$, $k = 0, 1, \ldots, N-1$. Thus, the DFT is a set of samples of the discrete-time Fourier transform at N equally spaced frequencies from zero frequency up to (but not including) the sampling frequency $w_S = 2\pi/T$.

The inverse discrete Fourier transform (IDFT) is:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k]\, e^{j2\pi kn/N} \qquad 0 \leqslant n \leqslant N-1 \tag{36}$$

The DFT (34) and its inverse (36) provide an exact Fourier representation for finite length sequences. However, an important property of the IDFT relation (36) is that if it is evaluated for values of n outside the interval $0 \leqslant n \leqslant N-1$, the result is not zero but rather a periodic repetition of x[n]. Thus, the DFT analysis and synthesis pair, (34) and (36), can also be thought of as a Fourier series representation for periodic sequences. Whether (34) and (36) represent a finite-length

sequence or a periodic sequence is only a matter of what is assumed about the sequence outside the interval $0 \leqslant n \leqslant N - 1$. Nevertheless, (36) does repeat periodically outside the interval if it is evaluated there, and it is this property that leads to a need to be careful in its use and also to efficient computational algorithms for its evaluation.[1]

8.5.2 Fast Fourier Transform (FFT)

The fast Fourier transform (FFT) is a generic term for a collection of algorithms for efficiently evaluating the DFT or IDFT. These algorithms are all based upon the general principle of breaking down the computation of the N accumulations of N products (N^2 multiplications and additions) called for by either (34) or (36) into a number of smaller DFT-like computations. Because of the periodicity and the symmetry of the quantities $e^{-j2pkn/N}$, many of the multiplications and additions can be eliminated. In fact, by increasing the control and indexing aspects of the algorithm, the amount of numerical computation can be reduced to be proportional to N.log N rather than proportional to N^2. For large N, the savings in arithmetic computation can be several orders of magnitude.

The basic arithmetic operation in a FFT algorithm is a (complex) multiply-accumulate operation, which can be easily and efficiently realized with the TMS32010. The details of many FFT algorithms can be found in references and textbooks on digital signal processing. [1,3,7]

A number of FORTRAN programs for FFT algorithms are contained in the IEEE Press Book. [6, Section 1] They range in complexity from very simple programs where N must be a power of two, to more complex (and thus more efficient) mixed radix algorithms. Although these programs cannot be run directly on the TMS32010, they do serve as a convenient and readable description of the algorithm which could be translated readily into a TMS32010 program.

8.5.3 Uses of the DFT and FFT

Since highly efficient computation of the DFT is possible, and since Fourier analysis is such a fundamental concept in signal and system theory, it is natural that many uses have been found for the DFT. One major class of applications is in the computation of convolutions or correlations. If x[n] and h[n] are convolved to produce y[n] (i.e., linear filtering), then the Fourier transforms of these sequences are related by:

$$Y(e^{j\omega T}) = H(e^{j\omega T}) \cdot X(e^{j\omega T}) \qquad (37)$$

Since the DFT is just a sampled version of the discrete-time Fourier transform, it is also true that:

$$Y[k] = H[k] \cdot X[k] \qquad 0 \leqslant k \leqslant N-1 \qquad (38)$$

and if x[n], h[n], and the y[n] resulting from their convolution are all less than or equal to N in length, then y[n] can be computed as the IDFT of Y[k] in (38). Due to the great efficiency of the FFT, it may be more efficient in some cases to compute X[k] and H[k], multiply them together, and then compute y[n] using the IFFT than to compute y[n] directly by discrete convolution. Such a scheme is depicted in Figure 8-14. Since correlations can be computed by time-reversing one of the sequences before convolution, Figure 8-14 also represents a technique for computing both auto- and cross-correlation functions.

When the lengths of the sequences are larger than the available random access memory, or when real-time operation with minimal delay is required, there are schemes whereby the output can be computed in sections. [1,3,7]

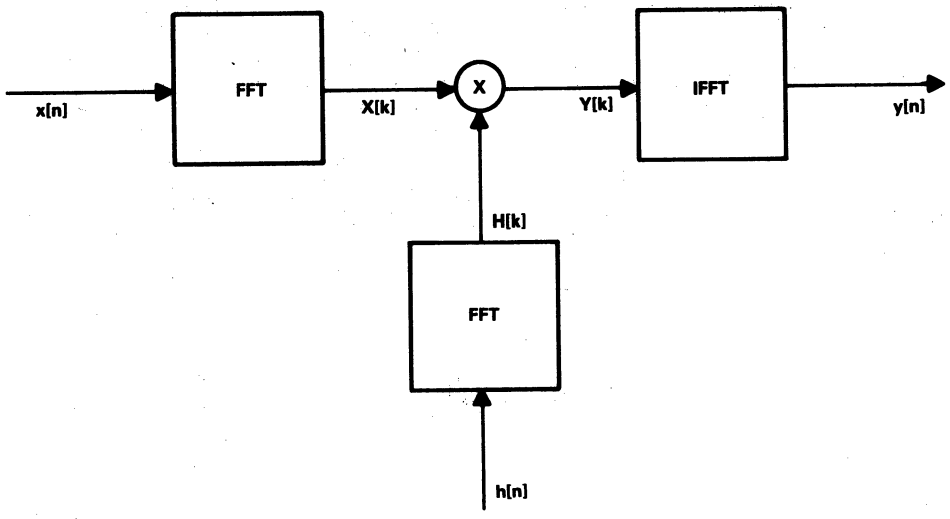


FIGURE 8-14 — A DISCRETE CONVOLUTION USING THE FFT

Another use of the DFT/FFT is in the computation of estimates of the Fourier transform or the power spectrum of an analog signal. The three basic concerns in this application are depicted in Figure 8-15. First, the analog signal $x_a(t)$ must be sampled, and thus the spectrum of $x_a(t)$ must be lowpass-filtered so as to minimize the aliasing distortion introduced by the sampling operation. The second major concern is a result of the fact that the DFT/FFT applies to finite length sequences. Thus, no matter how many samples of the input signal are available, there will always be a need to truncate the input signal to a practical length for the FFT computation. This can be represented as a windowing operation, i.e., a finite length sequence is obtained from x[n] by:

$$y[n] = w[n] \cdot x[n] \qquad 0 \leqslant n \leqslant N-1 \tag{39}$$

$$= 0 \qquad \text{otherwise}$$

Thus, the Fourier transform of y[n] is:

$$Y(e^{j\omega T}) = \frac{1}{2\pi} \int_{-\pi}^{\pi} X(e^{j\theta T}) \cdot W(e^{j(\omega-\theta)T}) d\theta \tag{40}$$

where $X(e^{j\omega T})$ is the Fourier transform of the input signal, and $W(e^{j\omega T})$ is the Fourier transform of the window. From (40), it is clear that $Y(e^{j\omega T})$ is a 'blurred' or 'smeared' version of the desired $X(e^{j\omega T})$, and that it is desirable that $W(e^{j\omega T})$ be highly concentrated around zero frequency so that it 'looks like' an impulse compared to the detailed variations of $X(e^{j\omega T})$. Then, $Y(e^{j\omega T})$ will not differ appreciably from the desired $X(e^{j\omega T})$. This can be accomplished by adjusting the length N and the shape of the window w[n]. [1-3]

In cases where the signal is modeled realistically as a stationary random process, the above procedure can be used as a basis for the estimation of the power spectrum. In order to smooth the statistical irregularities that arise in computing Fourier transforms of finite-length segments of a random signal, it is common to compute discrete Fourier transforms of windowed segments of the signal, and then average the squared magnitude of each transform. [1-3]

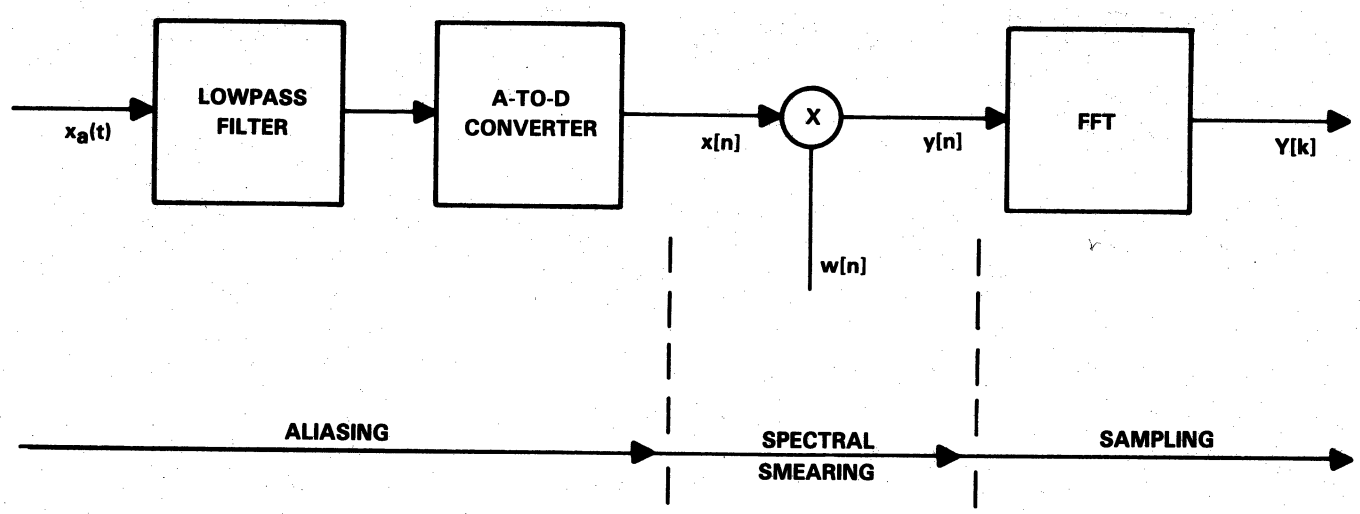


FIGURE 8-15 — ESTIMATION OF FOURIER TRANSFORM OF AN ANALOG SIGNAL

In situations where the signal is non-stationary, it is also common to compute discrete Fourier transforms of successive (either overlapping or non-overlapping) segments of the waveform, but instead of averaging the transforms, each transform is thought of as being representative of the signal in the time interval to which it corresponds. This leads to the concept of a short-time or running Fourier transform which is a function of both time and frequency. [2] This approach to spectrum analysis is widely used in speech, radar, and sonar signal processing. Figure 8-16 shows an example of a running spectrum of a doppler radar signal. The plot shows a succession of DFTs of the complex radar return signal. Evident in the plot is a strong time-varying component due to target rotation along with considerable noise. [10]

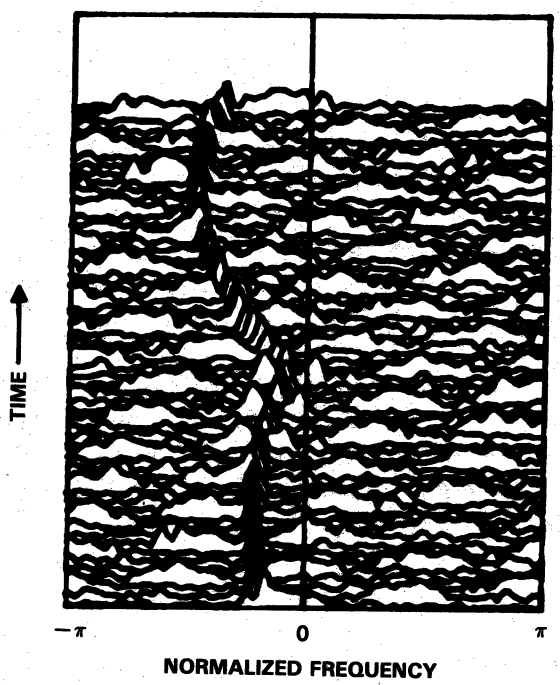


FIGURE 8-16 — SHORT-TIME FOURIER ANALYSIS OF A DOPPLER RADAR SIGNAL

8.5.4 Autoregressive Model

Another approach to spectrum analysis is based upon the assumption of a functional model for the signal, and the subsequent estimation of the parameters of the model. [6] A widely used model assumes that the signal x[n] is the output of a discrete-time linear system whose input and output satisfy a difference equation of:

$$x[n] = \sum_{k=1}^{N} a_k x[n-k] + G \cdot u[n] \qquad (41)$$

where the spectrum of the model input u[n] is flat. Estimation of the model parameters requires that an estimate be made of the filter coefficients a_k, the gain constant G, and perhaps some properties of the input to the model u[n]. The transfer function of the difference equation (41) is:

$$H(z) = \frac{G}{1 - \sum_{k=1}^{N} a_k z^{-k}} \qquad (42)$$

Thus, such models are often called all-pole models. Three basic types of excitations are generally assumed for the model. When purely transient signals consisting of damped oscillations are modeled, it is generally appropriate to use a unit impulse as the input to the model. When periodic signals (such as voiced speech) are modeled, the input is assumed to be a periodic impulse train. In cases where the signal is random and continuing in nature, the input is assumed to be white noise with unit variance. In all these cases, since the inputs all have flat spectra, the transfer function of the system determines the spectrum of the output of the model. Thus, if a given signal is assumed to be the output of the above model, then the determination of H(z) for the model is tantamount to determining the spectrum of the signal.

A number of techniques for determining the parameters a_k of H(z) have been developed. Terms, such as autoregressive modeling, linear predictive analysis, linear predictive coding (LPC), the Burg method, maximum entropy method (MEM), and maximum likelihood method (MLM), are all associated with methods of estimating the parameters of such all-pole signal models. Although the details of these methods differ, it is fair to say that most of the available methods can be shown to be equivalent to the solution of a set of N linear equations:

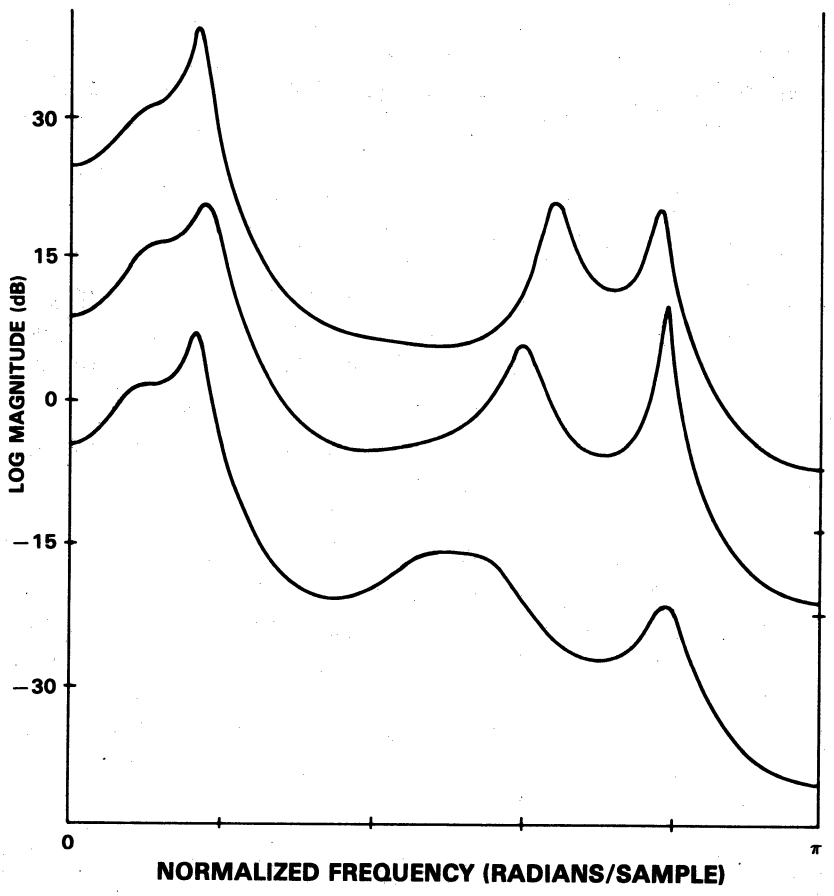
$$\sum_{k=1}^{N} a_k \cdot R[k,m] = R[0,m] \qquad m = 1, 2, \ldots, N \qquad (43)$$

where R[k,m] is a correlation-type function:

$$R[k,m] = \sum_{n} x[n-k] \cdot x[n-m] \qquad (44)$$

where the sum is carried out over a finite interval of the signal. Both the computation of R[k,m] and the solution of the set of linear equations by techniques such as the Levinson recursion [2,11,12] involve the repetitive use of the basic multiply-accumulate operation. These computations can be easily and efficiently implemented on the TMS32010.

Because the computation of the correlations R[k,m] can be based upon either a small or a large number of samples of the signal, either a short-time or a long-time estimate of the signal model (and thus of the signal spectrum) can be obtained. Thus, the autoregressive modeling approach can be applied to either stationary or nonstationary signals just as in the case of Fourier analysis. As an example, Figure 8-17 shows a spectrum estimate for several successive short segments of a speech signal. The spectral peaks, which correspond to poles of the model transfer function, result from resonances of the vocal system which produced the speech signal. These resonances are called 'formant frequencies', and they are characteristic of the sound being produced during each respective analysis interval. Spectrum analysis of this type is a cornerstone of much of the recent work in speech synthesis and speech recognition. [2,12]



NOTE: In this short-time autoregressive spectrum estimation for speech signals, the lower spectra correspond to later analysis times.

FIGURE 8-17 — SPECTRUM ESTIMATION FOR SPEECH SIGNALS

8.6 POTENTIAL DSP APPLICATIONS FOR THE TMS32010

From the discussion of the fundamentals of digital signal processing, it can be seen that the architecture of the TMS32010 is especially well suited to implementation of the basic DSP algorithms for recursive and nonrecursive linear filtering, discrete Fourier transformation, autoregressive modeling, and spectrum analysis. In the following paragraphs will be described some of the basic applications of DSP techniques and the TMS32010 in the areas of speech and audio processing and communications.

8.6.1 Speech and Audio Processing

In the field of speech and audio processing, there are three major application areas: 1) digital coding for storage and transmission, 2) automatic recognition and classification of speech and speakers, and 3) processing for enhancement and modification of speech signals.

The speech and audio coding area is very diverse, and its importance is growing rapidly as both storage (recording) and transmission systems are rapidly moving in the digital direction. In all digital coding applications, the basic concern is to encode sampled speech (or audio) signals with as low a bit-rate as possible while maintaining an acceptable level of perceived quality. Generally, this must be done within limits on the size, complexity, and cost of the encoding and decoding system.

The 'digital audio' area is rapidly becoming a major area of commercial exploitation of DSP. In this field, the emphasis is on high quality reproduction of the signal. Signals are typically sampled with 14-to-16 bit precision at sampling rates upwards of 40 kHz. Potential areas of application of DSP

techniques by the TMS32010 include the use of digital filtering together with simple A-to-D converters such as delta modulators operating at very high sampling rates to obtain high quality sampling and quantization at low cost, the use of digital filters for changing sampling rates, and high-speed coding and decoding (in the information theory sense) of samples for error protection and detection. A variety of other applications in the audio area are possible if the audio signal is available in digital form. These include delay and reverberation systems and sophisticated mixing and editing systems. Another example is in the implementation of electronic musical instruments.

The speech coding area is wide in range and diverse due to the fact that the quality of the encoded speech is not the only criterion in many applications. Often, simplicity of hardware implementation, bit-rate for transmission or storage, or robustness to errors in transmission are major concerns. This has led to the development of a multitude of coding schemes, all of which exploit one or more of the basic algorithms of DSP discussed above, and each of which has its own set of advantages and disadvantages.

Perhaps the simplest class of coders is based upon the principle of faithful reproduction of the speech waveform. Such schemes as deltamodulation, differential PCM, and nonlinear companding are examples. These systems may involve adaptive or fixed quantizers and adaptive or fixed predictors to achieve data rates ranging from about 10 kbits/s to well over 1 megabit/s. Recursive and nonrecursive digital filtering and autoregressive spectrum analysis are fundamental to most of these systems.

Another class of speech coders combines the principle of waveform replication with knowledge of the ear's lack of sensitive to certain frequency domain distortions to obtain high perceptual quality at bit rates in the 5-to-10 kbit/s range. Examples include sub-band coding, where the speech is broken up into frequency bands before quantization, and transform coding, where blocks of speech samples are transformed using the cosine transform (a close relative of the DFT) and then the transform values are quantized rather than the speech samples themselves. In the former case, the basic operations are digital filtering and adaptive quantization, and in the latter case, the basic operations are Fourier transformation and adaptive quantization. These systems may be too complex to be implemented with a single TMS32010 chip. However, several processors can be used together since it is relatively straightforward to divide the system into parts which can operate in parallel or in pipeline fashion.

In the third class of speech coding systems, there is no attempt to replicate the waveform of the speech signal. Instead, the objective is to incorporate both the physics of speech production and the psychophysics of speech perception into a system which produces speech which is intelligible and otherwise perceptually acceptable. Such systems are often called vocoders, and there are many such schemes. However, recent interest centers primarily on the class of linear predictive (LPC) vocoders. These systems are based upon an autoregressive all-pole model of the form discussed earlier. The LPC vocoder analyzer system involves the estimation of the coefficients of the digital filter in the model and the estimation of the parameters of the excitation to the model. The computation of the correlation values and the recursive solution for the filter coefficients are basic operations that can be efficiently implemented on the TMS32010. Speech is encoded in this system by quantizing the parameters of the model. Speech is decoded from these parameters by actually controlling a simulation of the model with the time-varying estimated parameters. This model consists of an all-pole digital filter excited by either white noise or a periodic impulse train. The TMS32010 is capable of generating the excitation as well as implementing the computations of the difference equation in real-time at speech sampling rates. (Alternatively, special purpose LPC speech synthesizer chips, such as the Texas Instruments TMS5100, 5200, or 5220, also can be used for speech synthesis from an LPC model.)

One of the most exciting areas of speech processing is the area of voice input to computers. This includes a wide range of considerations, such as isolated word recognition, connected speech recognition, speaker verification, and speaker identification. These systems typically break down into a 'front end' analysis or feature extraction stage, then a pattern comparison stage, followed by a classification stage. Features used to represent speech signals for pattern recognition generally are derived from an LPC spectrum analysis or a short-time Fourier spectrum analysis. Distance measures for comparing speech patterns are generally in the form of an inner product of feature vectors, which involves simply a multiply-accumulate operation. Another important operation is the time alignment of speech patterns so as to take into account differences in articulation and speaking rate. This is often accomplished using a dynamic programming algorithm. All of these operations can be readily accomplished in real-time at speech sampling rates using a system composed of several TMS32010 processors.

8.6.2 Communications

Digital signal processing has made a major impact in the general area of communications. In addition to applications such as speech waveform coding, DSP hardware is being used in the design of digital modems for communicating discrete information over voice-grade telephone channels, for signal conversion, and for the digital realization of such familiar components as filters, correlators, frequency references, and mixers.

As a specific example, a TMS32010 chip might be applied in the implementation of a digital modem operating on a voice-grade telephone line. Digital processing has had a major impact on the design of highspeed digital modems, not only because of cost, but also because these systems need to be adaptive. In fact, all modems operating over voice-grade telephone lines at data rates in excess of 1200 bits/s require some sort of adaptive channel equalization. The frequency response of such telephone lines extends from about 300 Hz to 3300 Hz. While the magnitude response is far from flat, the more serious consideration for the modem designer is the group delay response, which ranges from between 0 milliseconds at 1000 Hz to approximately 2.5 milliseconds at 3300 Hz. At a transmission rate of 2400 pulses per second, the effect of this irregular group delay is to smear each received pulse over several pulse intervals. This phenomenon is known as 'intersymbol interference.' It can be removed by convolving the received signal with a function which is the inverse of the channel impulse response. Unfortunately, the details of that response depend upon the characteristics of the line, and thus they will change every time a new connection is made and will vary during the course of a lengthy transmission. The solution is to pass the signal through an adaptive equalizer, simply a FIR filter whose coefficients bk are systematically updated.

A simplified block diagram of a digital modem, shown in Figure 8-18, will be helpful before considering the operation of the adaptive equalizer in more detial. At the transmitter, the bit stream is converted into a waveform using either phase-shift keying (PSK) or a combination of PSK and amplitude-shift keying (ASK). The resulting sequence is typically complex. This complex signal is filtered and modulated to a center frequency, which after D-to-A conversion will be centered at about 1800 Hz. These are all tasks which can be implemented easily on the TMS32010. At the receiver, the signal is demodulated, filtered, and passed through the adaptive equalizer. The output of the equalizer is decoded in order to reproduce the desired bit stream and this decision is also fed back to the adaptive equalizer.

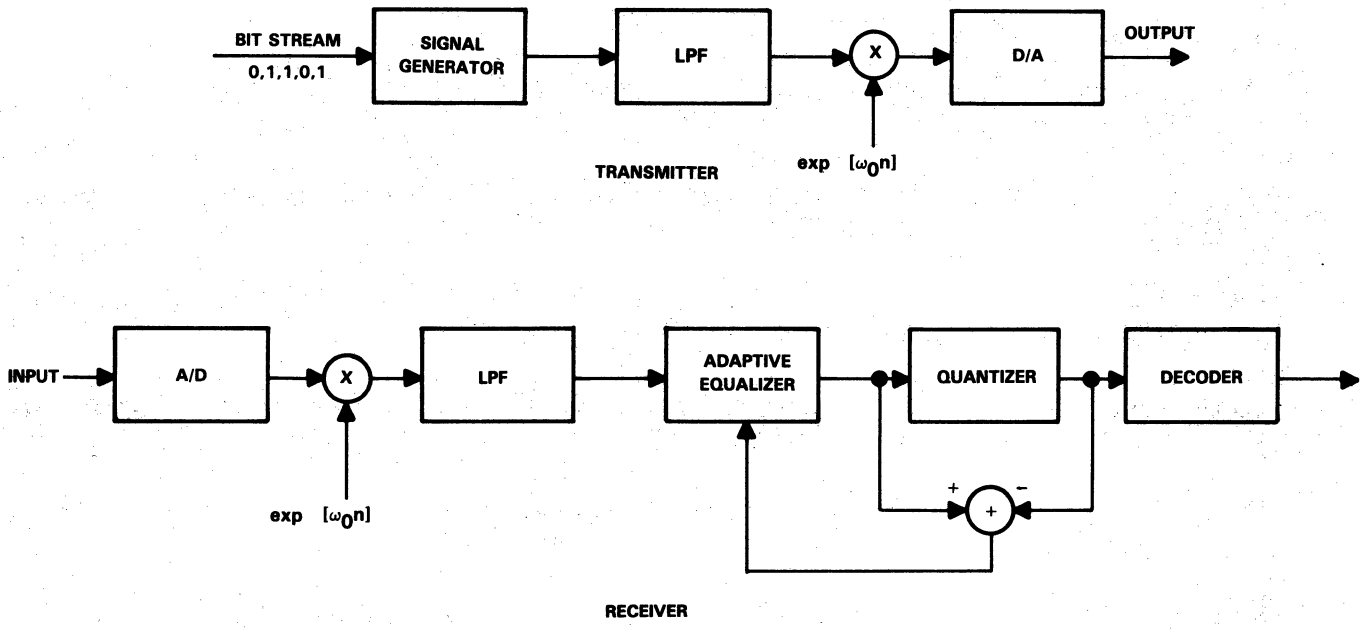


FIGURE 8-18 — BLOCK DIAGRAM OF A DIGITAL MODEM

In describing the operation of the adaptive equalizer, the k^{th} filter coefficient at time n is denoted as $b_k[n]$. Then if x[n] and y[n] denote the input and output, respectively, of the equalizer:

$$y[n] = \sum_{k=0}^{M} b_k[n] \cdot x[n-k] \qquad (45)$$

The filter coefficients are updated according to:

$$b_k[n+1] = b_k[n] + 2\mu \cdot x^*[n-k] \cdot e[n] \qquad k = 0, 1, \ldots, M \qquad (46)$$

where * denotes complex conjugation and where e[n] is the difference between the actual and the desired value for y[n]. When the connection between the transmitter and the receiver is first made, a standard preamble is transmitted, which is used to adapt the receiver coefficients. During the period of actual information transmission, the error is calculated under the assumption that the signal is being correctly received and this information is fed back to the adaptive equalizer. The stepsize parameter μ controls the rate of adaption, the stability of the equalizer, and its immunity to noise. The fundamental operation of the adaptive equalizer involves (complex) sums and products. This is a task for which the TMS32010 is ideally suited.

8.7 REFERENCES

[1] Oppenheim, A.V. and Schafer, R.W., DIGITAL SIGNAL PROCESSING. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

[2] Rabiner, L.R. and Schafer, R.W., DIGITAL PROCESSING OF SPEECH SIGNALS. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

[3] Rabiner, L.R. and Gold, B., THEORY AND APPLICATION OF DIGITAL SIGNAL PROCESSING. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

[4] Oppenheim, A.V., Willsky, A.N., with Young, I.T., SIGNALS AND SYSTEMS. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

[5] Goodman, D.J., 'The Application of Delta Modulation to Analog-to-PCM Encoding,' BELL SYSTEM TECHNICAL JOURNAL, February, 1969, 321-343.

[6] IEEE ASSP DSP Committee, ed., PROGRAMS FOR DIGITAL SIGNAL PROCESSING. New York, NY: IEEE Press, 1979.

[7] Gold, B. and Rader, C.M., DIGITAL PROCESSING OF SIGNALS. New York, NY: McGraw-Hill Book Co., 1969.

[8] Kaiser, J.F., "Nonrecursive Digital Filter Design Using The I_0-sinh Window Function," PROCEEDINGS OF THE 1974 IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, April, 1974, 20-23.

[9] Fettweis, A. and Meekotter, K., "Suppression of Parasitic Oscillations in Wave Digital Filters," IEEE TRANSACTIONS CIRCUITS AND SYSTEMS, Vol. CAS-22, March, 1975, 239-246.

[10] Schaefer, R.T., Schafer, R.W., and Mersereau, R.M., "Digial Signal Processing for Doppler Radar Signals," PROCEEDINGS OF THE 1979 INTERNATIONAL CONFERENCE OF ACOUSTICS, SPEECH, AND SIGNAL PROCESSING, April, 1979.

[11] Makhoul, J., "Linear Prediction: A Tutorial Review," PROCEEDINGS OF IEEE, Vol. 63., 1975, 561-580.

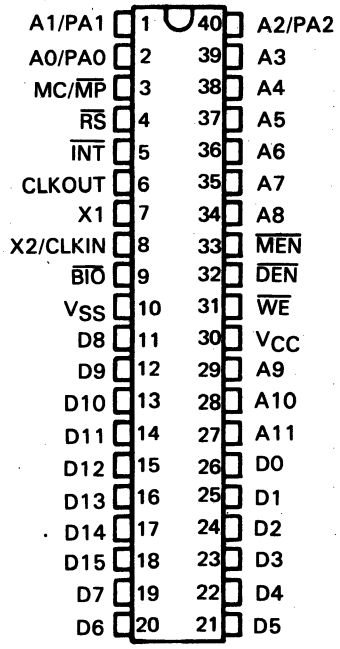
[12] Markel, J.D. and Gray, A.H., LINEAR PRODUCTION OF SPEECH. New York, NY: Springer-Verlag, 1976.

TMS32010
DIGITAL SIGNAL PROCESSOR

MAY 1983—REVISED FEBRUARY 1985

- 160-ns Instruction Cycle
- 144-Word On-Chip Data RAM
- ROMless Version - TMS32010
- 1.5K-Word On-Chip Program ROM - TMS320M10
- 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 160-ns
- 0 to 15-Bit Barrel Shifter
- Eight Input and Eight Output Channels
- 16-Bit Bidirectional Data Bus with 50-Megabits-per-Second Transfer Rate
- Interrupt with Full Context Save
- Signed Two's Complement Fixed-Point Arithmetic
- NMOS Technology
- Single 5-V Supply
- Two Versions Available
  TMS32010-20 . . . 20.5 MHz Clock
  TMS32010-25 . . . 25.0 MHz Clock

TMS32010 . . . N PACKAGE
(TOP VIEW)



PIN NOMENCLATURE

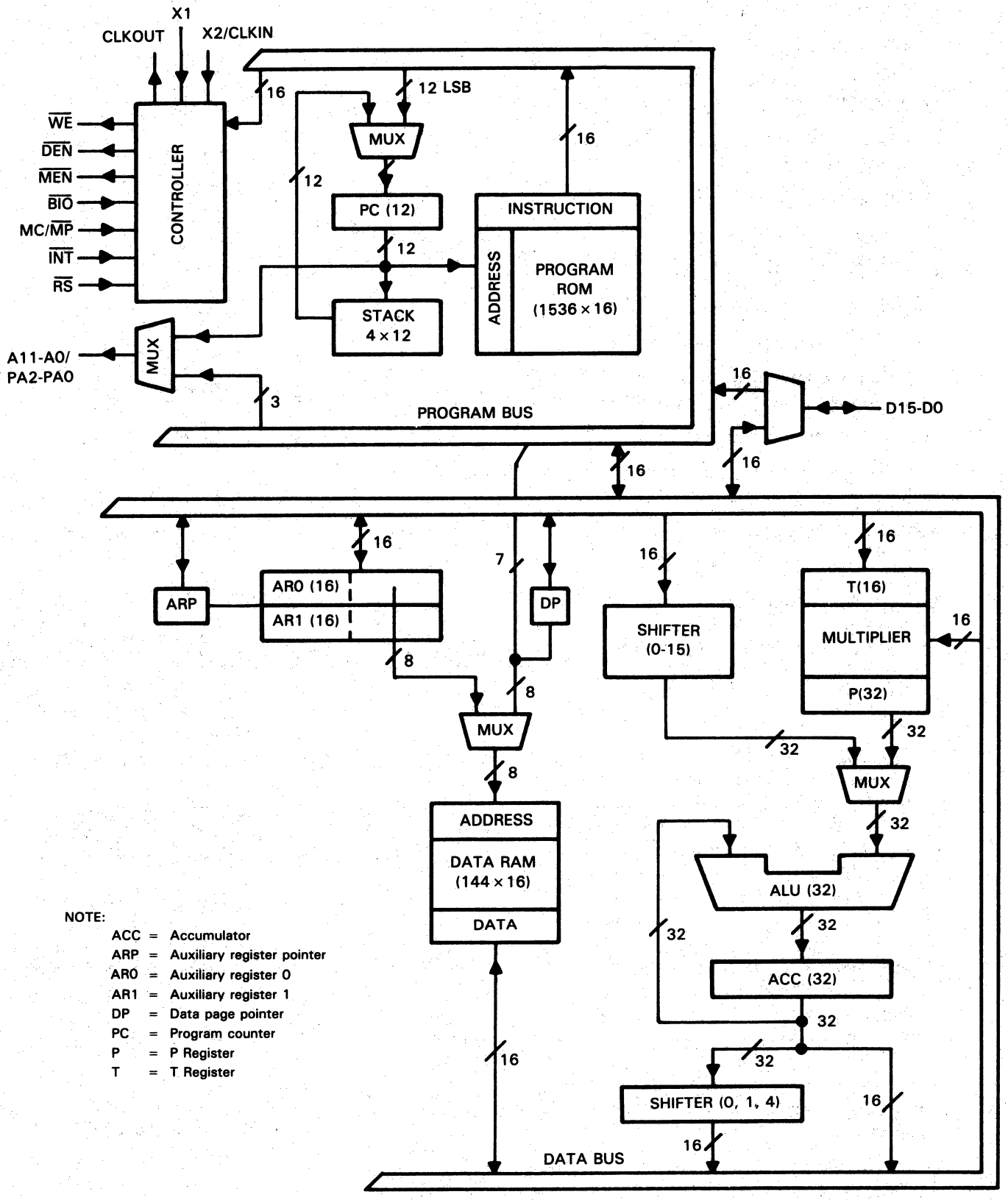
SIGNATURE	I/O	DEFINITION
A11-A0/ PA2-PA0	OUT	External address bus. I/O port address multiplexed over PA2-PA0.
$\overline{BIO}$	IN	External polling input for bit test and jump operations.
CLKOUT	OUT	System clock output, ¼ crystal/CLKIN frequency.
D15-D0	I/O	16-bit data bus.
$\overline{DEN}$	OUT	Data enable indicates the processor accepting input data on D15-D0.
$\overline{INT}$	IN	Interrupt.
MC/$\overline{MP}$	IN	Memory mode select pin. High selects microcomputer mode. Low selects microprocessor mode.
$\overline{MEN}$	OUT	Memory enable indicates that D15-D0 will accept external memory instruction.
$\overline{RS}$	IN	Reset used to initialize the device.
V_{CC}	IN	Power.
V_{SS}	IN	Ground.
$\overline{WE}$	OUT	Write enable indicates valid data on D15-D0.
X1	IN	Crystal input.
X2/CLKIN	IN	Crystal input or external clock input.

description

The TMS32010 is the first member of the new TMS320 digital signal processing family, designed to support a wide range of high-speed or numeric-intensive applications. This 16/32-bit single-chip microcomputer 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 TMS320 family contains the first MOS microcomputers capable of executing better than 6 million instructions per second. This high throughput is the result of the comprehensive, efficient, and easily programmed instruction set and of the highly pipelined architecture. Special instructions have been incorporated to speed the execution of digital signal processing (DSP) algorithms.

The TMS320 family's unique versatility and power give the design engineer a new approach to a

functional block diagram



NOTE:

ACC = Accumulator
ARP = Auxiliary register pointer
AR0 = Auxiliary register 0
AR1 = Auxiliary register 1
DP = Data page pointer
PC = Program counter
P = P Register
T = T Register

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

variety of complications. 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 TMS32010 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 160-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 TMS32010 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.

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 very useful for scaling and bit extraction.

16 × 16-bit parallel multiplier

The TMS32010's multiplier performs a 16 × 16-bit, two's complement multiplication in one 160-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 TMS32010 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 TMS320M10 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 TMS320M10 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. Those with 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 TMS32010 is identical to the TMS320M10, except that the TMS32010 operates only in the microprocessor mode. Henceforth, TMS32010 refers to both versions.

The ability of the TMS32010 to execute at full speed from off-chip memory provides the following important benefits:

- Easier prototyping and development work than is possible with a device that can address only on-chip ROM,
- Purchase of a standard off-the-shelf product rather than a semi-custom 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 TMS32010's 16-bit parallel data bus can be utilized to perform I/O functions at burst rates of 50 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 ($\overline{BIO}$) and an interrupt pin ($\overline{INT}$) have been incorporated for multi-tasking.

interrupts and subroutines

The TMS32010 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 TMS32010'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 TMS32010 are maskable.

instruction set

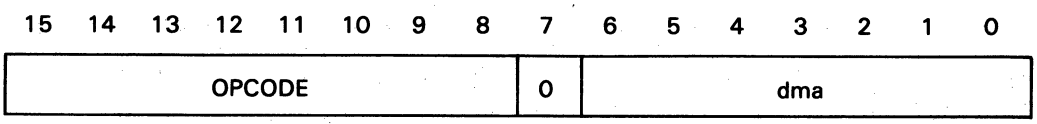
The TMS32010's comprehensive instruction set supports both numeric-intensive operations, such as signal processing, and general purpose operations, such as high-speed control. The instruction set, explained in Tables 1 and 2, consists primarily of single-cycle single-word instructions, permitting execution rates of better than 6 million instructions per second. Only infrequently used branch and I/O instructions are multicycle.

The TMS32010 also contains a number of instructions that shift data a part of an arithmetic operation. These all execute in a single cycle and are very useful for scaling data in parallel with other operations.

Three main addressing modes are available with the TMS32010 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:

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

Bit 7 = 1 defines 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 content of ARP remain 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 TMS32010 instruction set contains special "immediate" instructions. These instructions derive data from part of the instruction word rather than from the data RAM. Some very 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

TABLE 2. TMS32010 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. TMS32010 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

TABLE 2. TMS32010 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 TMS32010. 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 which can be loaded into the TMS32010 Evaluation Module, Simulator, or Emulator. The simulator provides a quick means for initially debugging TMS32010 software while the emulator provides real-time in-circuit emulation necessary to perform system level debug efficiently.

A complete list of TMS32010 software and hardware development tools are given in Table 3.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TABLE 3. TMS32010 SOFTWARE AND HARDWARE SUPPORT

HOST COMPUTER	OPERATING SYSTEM	PART NUMBER
MACRO ASSEMBLERS/LINKERS		
DEC VAX	VMS	TMDS3240210-08
DEC VAX	Berkeley UNIX 4.1	TMDS3240220-08
DEC VAX	Berkeley UNIX 4.2	TMDS3240230-08
IBM	MVS	TMDS3240310-08
IBM	CMS	TMDS3240320-08
TI/IBM PC	MS/PC-DOS	TMDS3240810-02
SIMULATORS		
DEC VAX	VMS	TMDS3240211-08
TI/IBM PC	MS/PC-DOS	TMDS3240811-02
DIGITAL FILTER DESIGN PACKAGE (DFDP)		
TI PC	MS-DOS	DFDP-TI001
IBM PC	PC-DOS	DFDP-IBM001
HARDWARE		
Evaluation Module (EVM)		RTC/EVM320A-03
Analog Interface Board (AIB)		RTC/EVM320C-06
Emulator		TMDS3262210

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 . . . . . 1.5 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		4.75	5	5.25	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

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		V_{CC} = MAX	T_A = 0°C		180	275	mA
				T_A = 70°C			235§	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 is not tested.

CLOCK CHARACTERISTICS AND TIMING

The TMS32010 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	TMS32010-20			TMS32010-25			UNIT
		MIN	NOM	MAX	MIN	NOM	MAX	
Crystal frequency f_x	0°C – 70°C	6.7		20.5	6.7		25.0	MHz
C1, C2	0°C – 70°C		10			10		pF

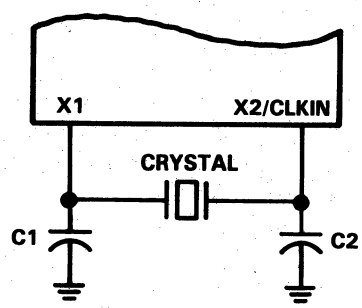


FIGURE 1. INTERNAL CLOCK OPTION

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

PARAMETER		TMS32010-20			TMS32010-25			UNIT
		MIN	NOM	MAX	MIN	NOM	MAX	
$t_{c(MC)}$	Master clock cycle time	48.78		150	40		150	ns
$t_{r(MC)}$	Rise time master clock input		5	10		5	10	ns
$t_{f(MC)}$	Fall time master clock input		5	10		5	10	ns
$t_{w(MCP)}$	Pulse duration master clock	$0.475t_{c(C)}$		$0.525t_{c(C)}$	$0.475t_{c(C)}$		$0.525t_{c(C)}$	ns
$t_{w(MCL)}$	Pulse duration master clock low, $t_{c(MC)}$ = 50 ns		20			18		ns
$t_{w(MCH)}$	Pulse duration master clock high, $t_{c(MC)}$ = 50 ns		20			18		ns

switching characteristics over recommended operating conditions

PARAMETER		TEST CONDITIONS	TMS32010-20			TMS32010-25			UNIT
			MIN	NOM	MAX	MIN	NOM	MAX	
$t_{c(C)}$	CLKOUT cycle time†	R_L = 870 Ω, C_L = 100 pF, See Figure 2	195.12			160			ns
$t_{r(C)}$	CLKOUT rise time			10			10		ns
$t_{f(C)}$	CLKOUT fall time			8			8		ns
$t_{w(CL)}$	Pulse duration, CLKOUT low			92			74		ns
$t_{w(CH)}$	Pulse duration, CLKOUT high			90			72		ns
$t_{d(MCC)}$	Delay time CLKIN↑ to CLKOUT↓‡		25		60	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.

PARAMETER MEASUREMENT INFORMATION

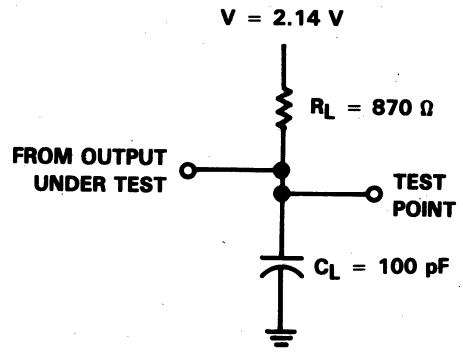
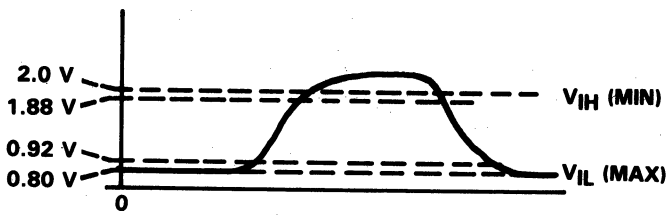
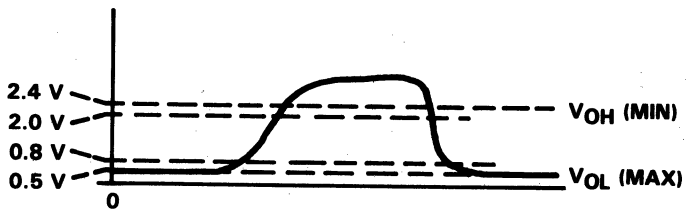


FIGURE 2. TEST LOAD CIRCUIT



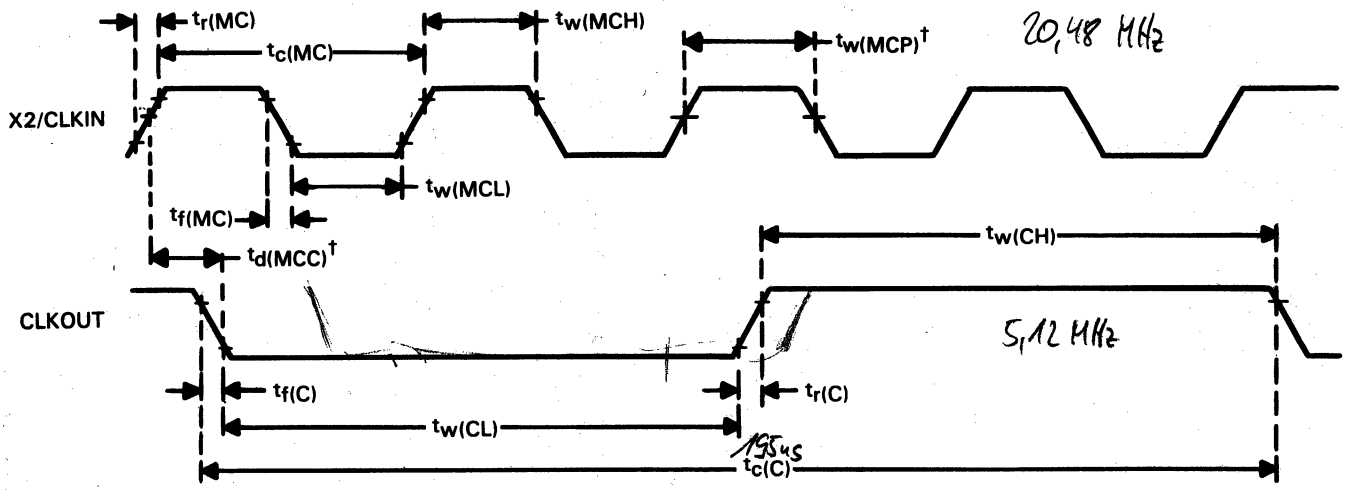
(a) INPUT



(b) OUTPUTS

FIGURE 3. VOLTAGE REFERENCE LEVELS

clock timing



NOTE: 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)	R_L = 870 Ω, C_L = 100 pF, See Figure 2	10†		50	ns
t_{d2}	Delay time CLKOUT↓ to $\overline{MEN}$↓		¼$t_{c(C)}$ − 5†		¼$t_{c(C)}$ + 15	ns
t_{d3}	Delay time CLKOUT↓ to $\overline{MEN}$↑		−10†		15	ns
t_{d4}	Delay time CLKOUT↓ to $\overline{DEN}$↓		¼$t_{c(C)}$ − 5†		¼$t_{c(C)}$ + 15	ns
t_{d5}	Delay time CLKOUT↓ to $\overline{DEN}$↑		−10†		15	ns
t_{d6}	Delay time CLKOUT↓ to $\overline{WE}$↓		½$t_{c(C)}$ − 5†		½$t_{c(C)}$ + 15	ns
t_{d7}	Delay time CLKOUT↓ to $\overline{WE}$↑		−10†		15	ns
t_{d8}	Delay time CLKOUT↓ to data bus OUT valid				¼$t_{c(C)}$ + 65	ns
t_{d9}	Time after CLKOUT↓ that data bus starts to be driven		¼$t_{c(C)}$ − 5†			ns
t_{d10}	Time after CLKOUT↓ that data bus stops being driven				¼$t_{c(C)}$ + 30†	ns
t_v	Data bus OUT valid after CLKOUT↓		¼$t_{c(C)}$ − 10			ns

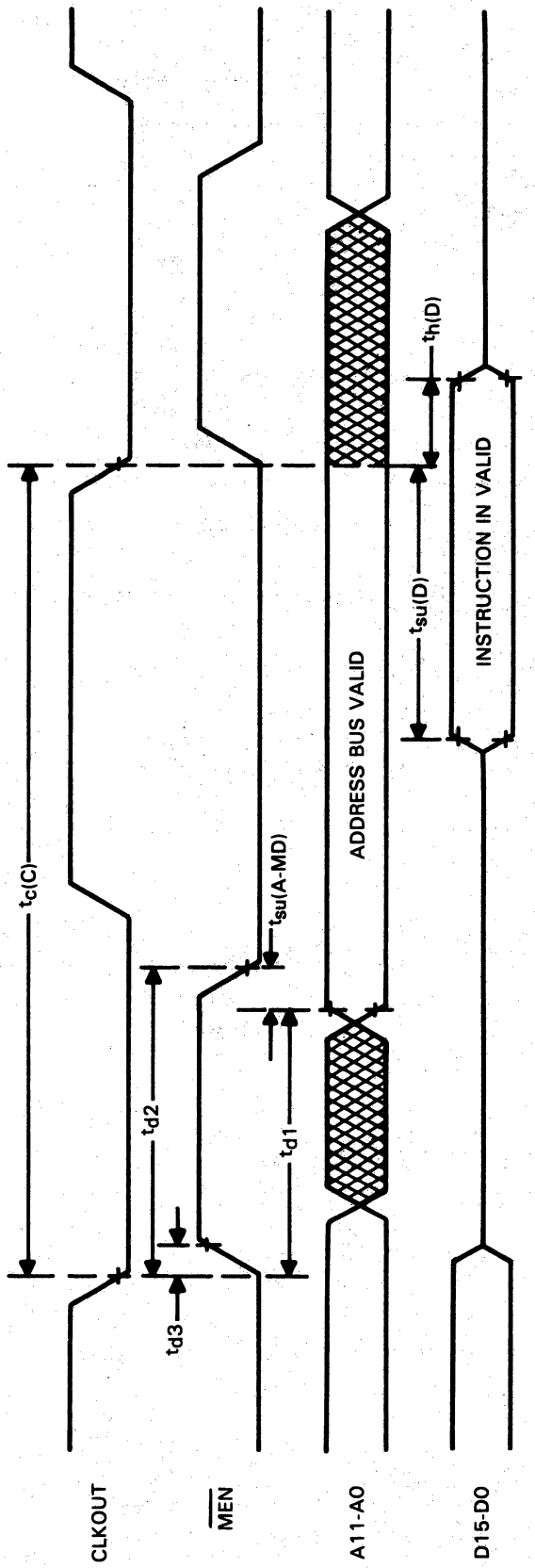
NOTE: Address bus will be valid upon $\overline{WE}$↑, $\overline{DEN}$↑, or $\overline{MEN}$↑.
† These values were derived from characterization data and are not tested.

timing requirements over recommended operating conditions

	PARAMETER	TEST CONDITIONS	MIN	TYP	MAX	UNIT
$t_{su(D)}$	Setup time data bus valid prior to CLKOUT↓	R_L = 870 Ω, C_L = 100 pF, See Figure 2	50			ns
$t_{su(A-MD)}$	Address bus setup time prior to $\overline{MEN}$↓ or $\overline{DEN}$↓		¼$t_{c(C)}$-45			ns
$t_{h(D)}$	Hold time data bus held valid after CLKOUT↓		0			ns

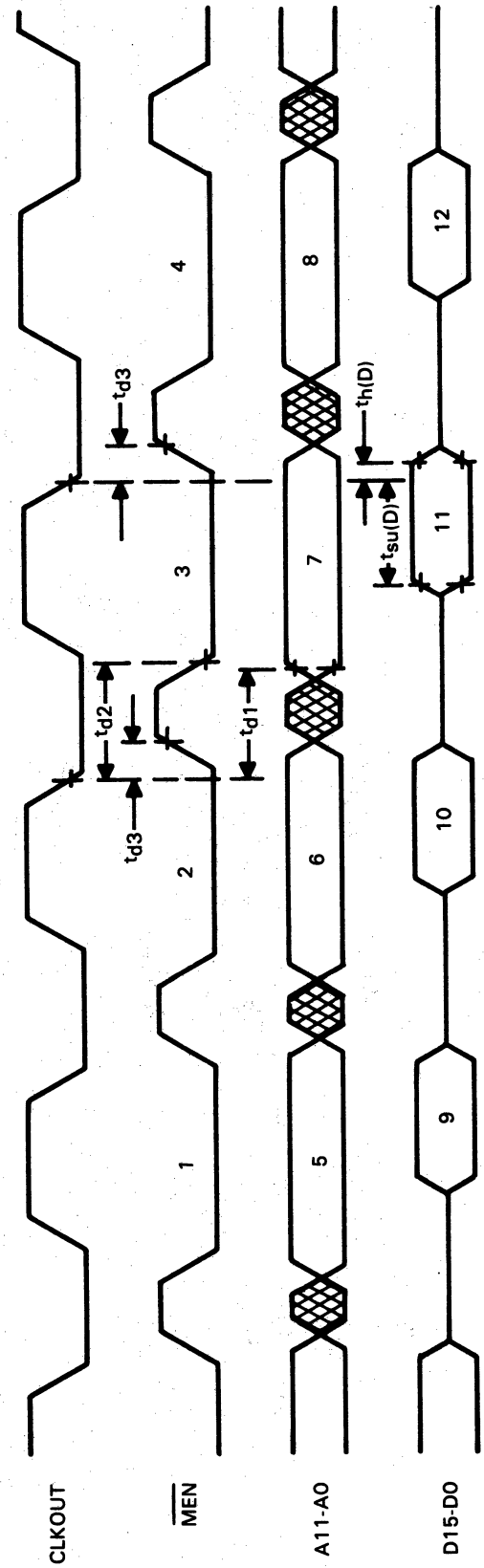
NOTE: Data may be removed from the data bus upon $\overline{MEN}$↑ or $\overline{DEN}$ ↑ preceding CLKOUT↓.

memory read



NOTE: 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

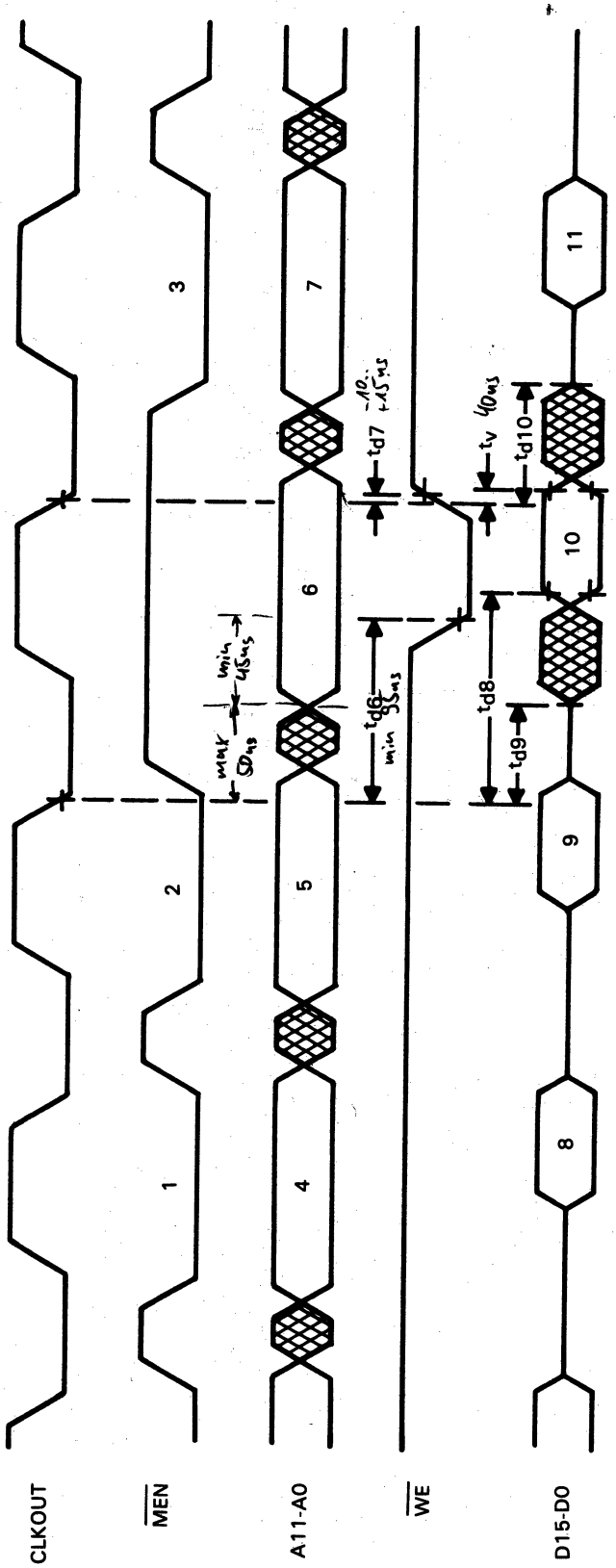


LEGEND:

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: 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.

TBLW instruction timing

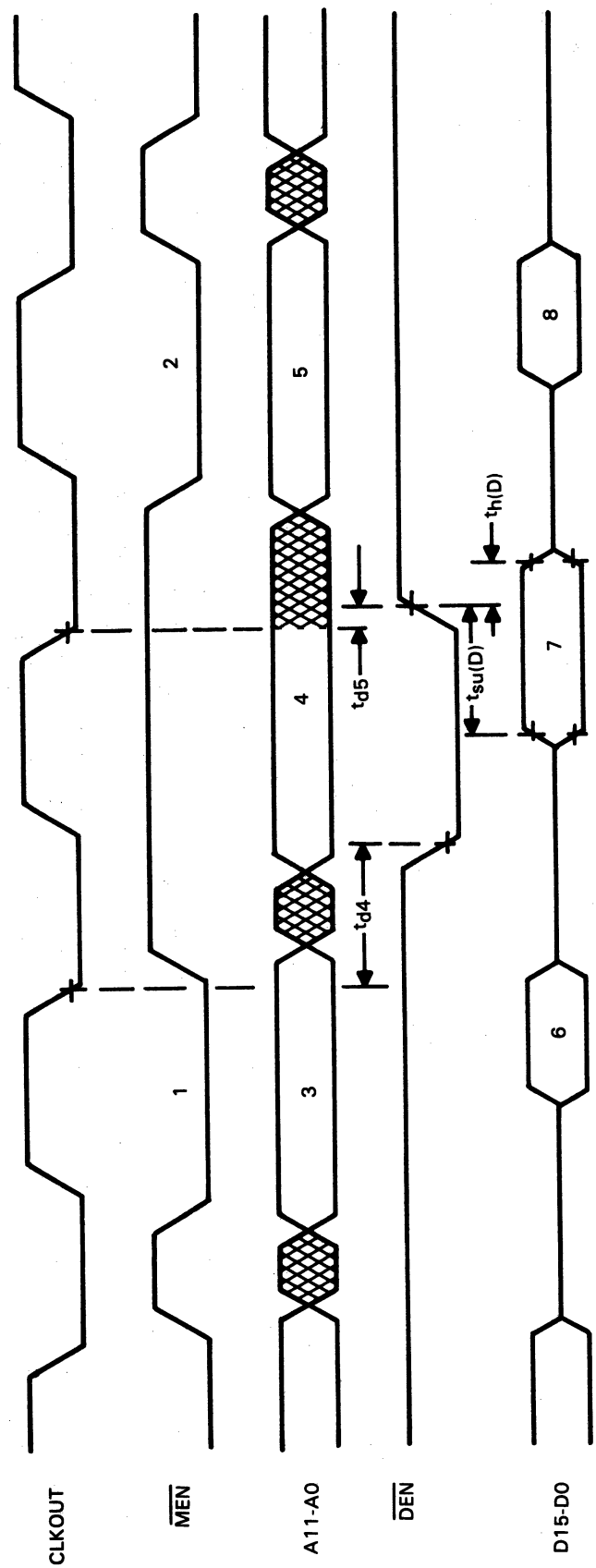


LEGEND:

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: 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

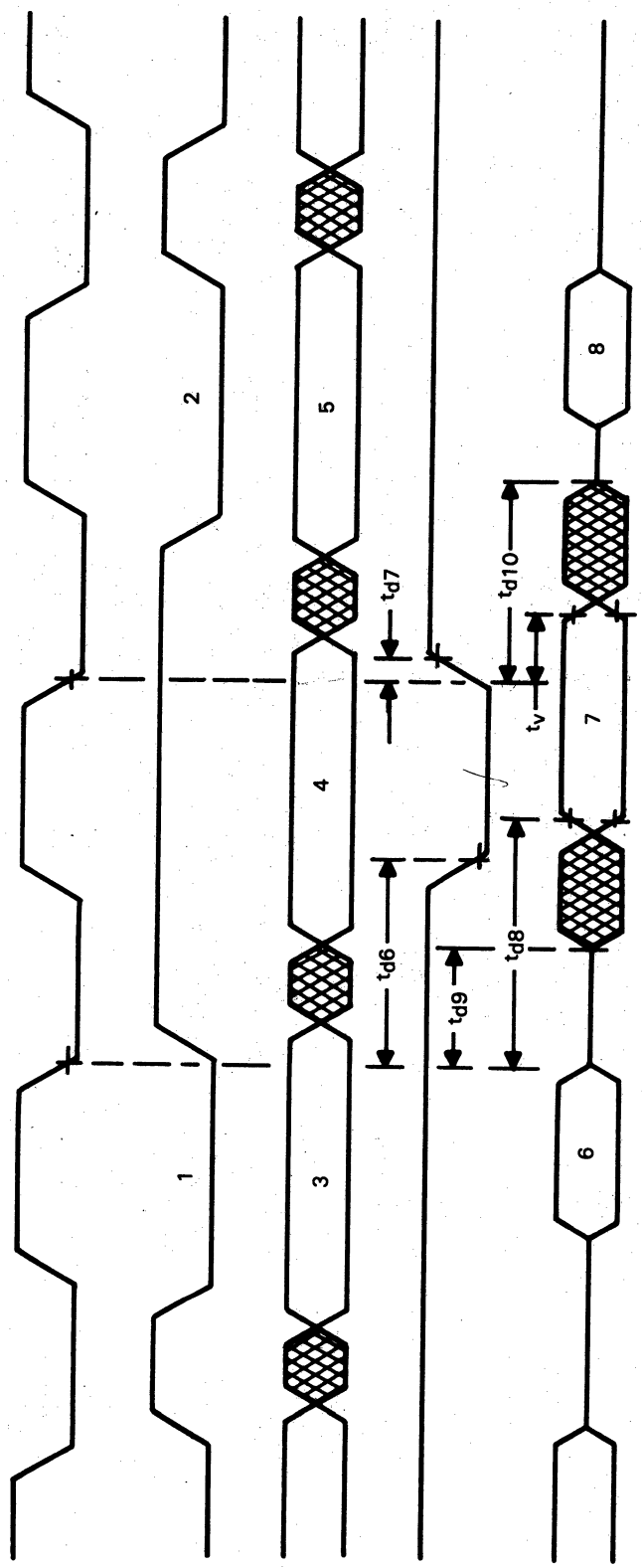


LEGEND:

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: 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.

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: 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

	PARAMETER	MIN	NOM	MAX	UNIT
$t_{su(R)}$	Reset ($\overline{RS}$) setup time prior to CLKOUT. See Note.	50			ns
$t_{w(R)}$	$\overline{RS}$ pulse duration	$5t_{c(C)}$			ns

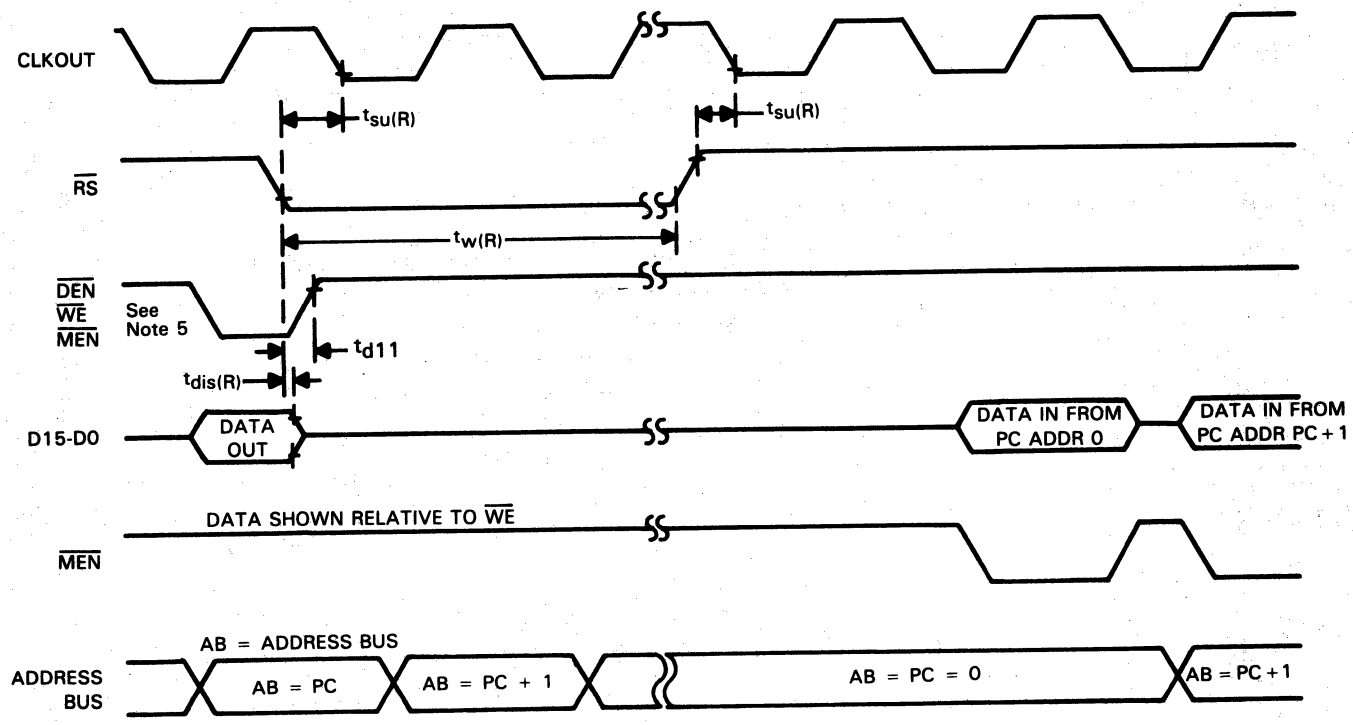
switching characteristics over recommended operating conditions

	PARAMETER	TEST CONDITIONS	MIN	TYP	MAX	UNIT
t_{d11}	Delay time $\overline{DEN}$↑, $\overline{WE}$↑, and $\overline{MEN}$↑ from $\overline{RS}$	R_L = 870 Ω, C_L = 100 pF, See Figure 2			$\frac{1}{2}t_{c(C)}+50$†	ns
$t_{dis(R)}$	Data bus disable time after $\overline{RS}$				$\frac{1}{4}t_{c(C)}+50$†	ns

NOTE: $\overline{RS}$ 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



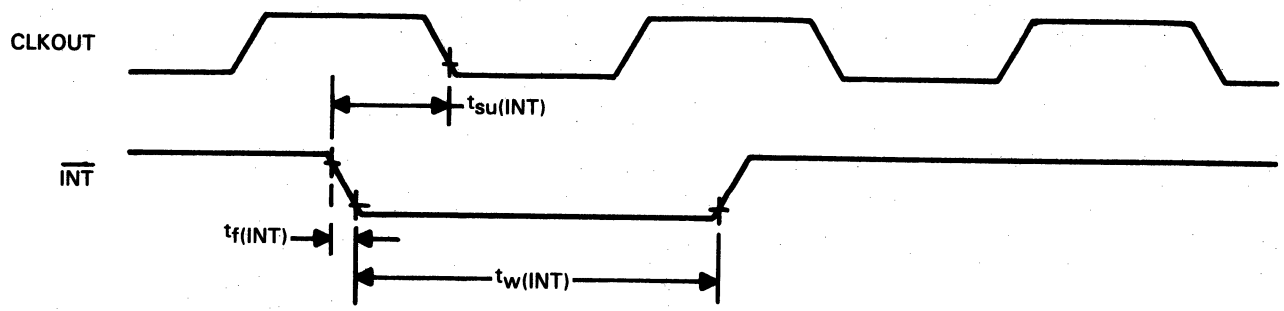
NOTES:
1. $\overline{RS}$ forces $\overline{DEN}$, $\overline{WE}$, and $\overline{MEN}$ high and tristates data bus D0 through D15. AB outputs (and program counter) are synchronously cleared to zero after the next complete CLK cycle from ↓$\overline{RS}$.
2. RS must be maintained for a minimum of five clock cycles.
3. Resumption of normal program will commence after one complete CLK cycle from ↑$\overline{RS}$.
4. 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.
5. Diagram shown is for definition purpose only. $\overline{DEN}$, $\overline{WE}$, and $\overline{MEN}$ are mutually exclusive.
6. 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.
7. During a write cycle, $\overline{RS}$ may produce an invalid write address.

INTERRUPT ($\overline{INT}$) TIMING

timing requirements over recommended operating conditions

	PARAMETER	MIN	TYP	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 $\overline{INT}$↓ before CLKOUT↓	50			ns

interrupt timing



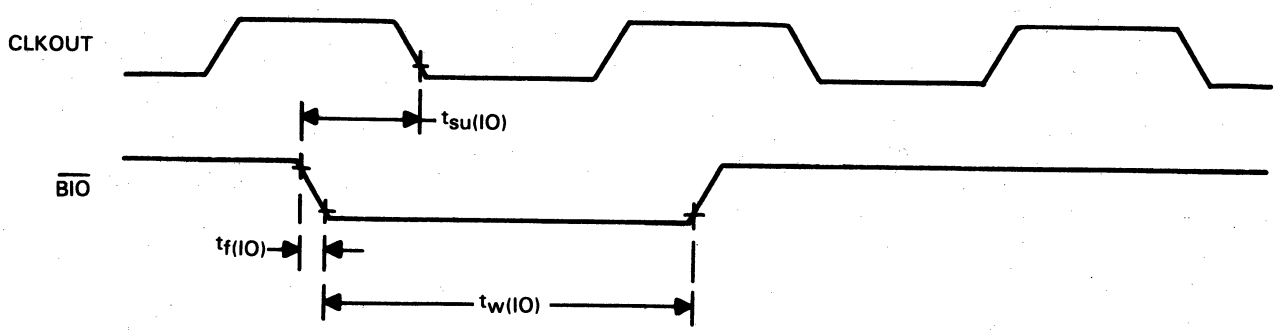
NOTE: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

I/O ($\overline{BIO}$) TIMING

timing requirements over recommended operating conditions

	PARAMETER	MIN	TYP	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 $\overline{BIO}$↓ before CLKOUT↓	50			ns

$\overline{BIO}$ timing

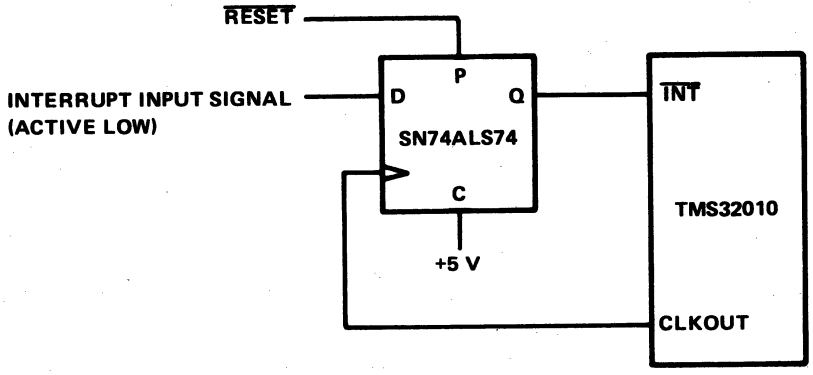


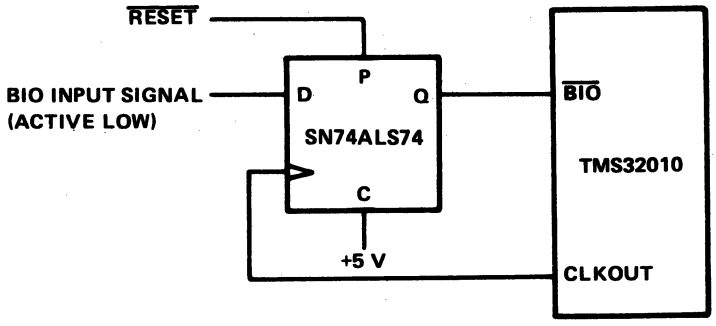
NOTE: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

input synchronization requirements

For systems using asynchronous inputs to the $\overline{INT}$ and $\overline{BIO}$ pins on the TMS32010, the external hardware shown in the diagrams below is recommended to ensure proper execution of interrupts and the BIOZ instruction. This hardware synchronizes the $\overline{INT}$ and $\overline{BIO}$ input signals with the rising edge of CLKOUT on the TMS32010. The pulse width required for these input signals is $t_{c(C)}$, which is one TMS32010 clock cycle, plus sufficient setup time for the flip-flop (dependent upon the flip-flop used).





TI standard symbolization for devices without on-chip ROM

SYMBOLIZATION

line 1: (a) (b) TMS32010NL

line 2: (c) ©1983 TI (d) DCU8327

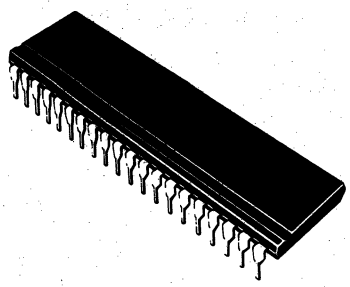
line 3: (e) 24655

MEANINGS OF SYMBOLS

(a) Texas Instruments trademark
(b) Standard device number
(c) TI design copyright
(d) Tracking mark and date code
(e) Lot code

MECHANICAL DATA

40-pin plastic dual-in-line package



NOTES: a. Each pin centerline is located within 0,254 (0.010) of its true longitudinal position.
b. All linear dimensions are in millimeters and parethetically in inches.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

APPENDIX B

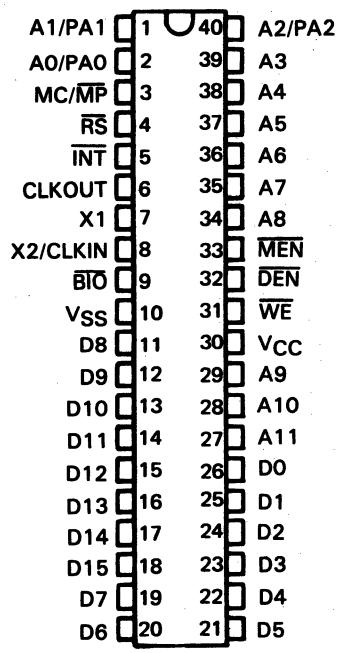
SMJ32010 DATA SHEET

SMJ32010
DIGITAL SIGNAL PROCESSOR

MAY 1983

- MIL-STD-883B Processing
- Same Features and Specifications as TMS32010 over 0°C - 70°C Temperature Range
- Currently Microprocessor Mode Only (All Program Memory Is Extended)
- Extended Temperature Version Available in Near Future
- 288-Byte On-Chip Data RAM
- External Memory Expansion to Total of 8K Bytes at Full Speed
- 16-Bit Instruction/Data Word
- 32-Bit ALU/Accumulator
- 16 × 16-Bit Multiply in One Instruction Cycle
- 0 to 15-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
- 2.7-Micron NMOS Technology
- Single 5-V Supply

SMJ32010 . . . JDL PACKAGE†
(TOP VIEW)



† Also available in a 44-pad leadless ceramic chip carrier (type FK).

description

The SMJ32010 is a member of the new TMS320 digital signal processing family, designed to support a wide range of high-speed or numeric-intensive applications. This 16/32-bit single-chip microcomputer 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 TMS320 family contains the first MOS microcomputers capable of executing five million instructions per second. This high throughput is the result of the comprehensive, efficient, and easily programmed instruction set and of the highly pipelined architecture. Special instructions have been incorporated to speed the execution of digital signal processing (DSP) algorithms.

The TMS320 family's unique versatility and power give the design engineer a new approach to a variety of complicated applications. In addition, these microcomputers are capable of providing the multiple

PIN NOMENCLATURE

SIGNATURE	I/O	DEFINITION
A11-A0/ PA2-PA0	OUT	External address bus. I/O port address multiplexed over PA2-PA0.
$\overline{BIO}$	IN	External polling input for bit test and jump operations.
CLKOUT	OUT	System clock output, ¼ crystal/CLKIN frequency.
D15-D0	I/O	16-bit data bus.
$\overline{DEN}$	OUT	Data enable indicates the processor accepting input data on D15-D0.
$\overline{INT}$	IN	Interrupt.
MC/$\overline{MP}$	IN	Memory mode select pin. High selects microcomputer mode. Low selects microprocessor mode.
$\overline{MEN}$	OUT	Memory enable indicates that D15-D0 will accept external memory instruction.
$\overline{RS}$	IN	Reset used to initialize the device.
V_{CC}	IN	Power.
V_{SS}	IN	Ground.
$\overline{WE}$	OUT	Write enable indicates valid data on D15-D0.
X1	IN	Crystal input.
X2/CLKIN	IN	Crystal input or external clock input.

B

ADVANCE INFORMATION

This document contains information on a new product. Specifications are subject to change without notice.

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.

SMJ32010 SIGNAL PROCESSOR SCREENING AND LOT PERFORMANCE

SCREEN	METHOD	RQMT
Internal Visual (Precap)	2010 Condition B See Note.	100%
Stabilization Bake	1008 Test Condition C (24 hours)	100%
Temperature Cycling	1010 Condition C (50 cycles)	100%
Constant Acceleration	2001 Condition A (MIN) in Y1 Plane	100%
Seal Fine and Gross	1014	100%
Interim Electrical	TI Data Sheet Electrical Specifications	100%
Burn-In	1015 125°C (160 hours MIN) PDA = 10%	100%
Final Electrical Tests (A) Static tests: (1) 25°C (Subgroup 1, Table 1, 5005) (2) MAX and MIN Rated Operating Temperature (Subgroups 2 and 3, Table 1, 5005) (B) Switching tests: (1) 25°C (Subgroup 9, Table 1, 5005) (2) MAX and MIN Rated Operating Temperature (Subgroups 10 and 11, Table 1, 5005) (C) Functional tests: (1) 25°C (Subgroup 7, Table 1, 5005) (2) MAX and MIN Rated Operating Temperature (Subgroup 8, Table 1, 5005)	TI Data Sheet Electrical Specifications	100%
Quality Conformance Inspection Group A (A) Static tests: (1) 25°C (Subgroup 1) (2) Temperature (Subgroups 2 and 3) (B) Switching tests: (1) 25°C (Subgroup 9) (2) Temperature (Subgroups 10 and 11) (C) Functional tests: (1) 25°C (Subgroup 7)	5005 Class B	LTPD 7% 10% 7% 10% 7%
External Visual	2009	100%

NOTE: 40x precap stress test in lieu of 100x precap per Mil-STD-883 Method 5004, Paragraph 3.3.

SMJ32010 SIGNAL PROCESSOR NOMENCLATURE

EXAMPLE: SMJ 32010 JD L

1. PREFIX
   Must contain three or four letters
   SMJ-Class B, Method 5004
   JANB – JM38510/JANB Qualified*

2. CIRCUIT DESIGNATOR
   Must contain five digits
   32010

3. PACKAGE TYPE
   Must contain two letters
   JD – Side Braze
   FK – Chip Carrier

4. TEMPERATURE RANGE
   Must contain one letter only
   L – 0°C to 70°C (extended temperature available in near future)

* Future product.

B

APPENDIX C

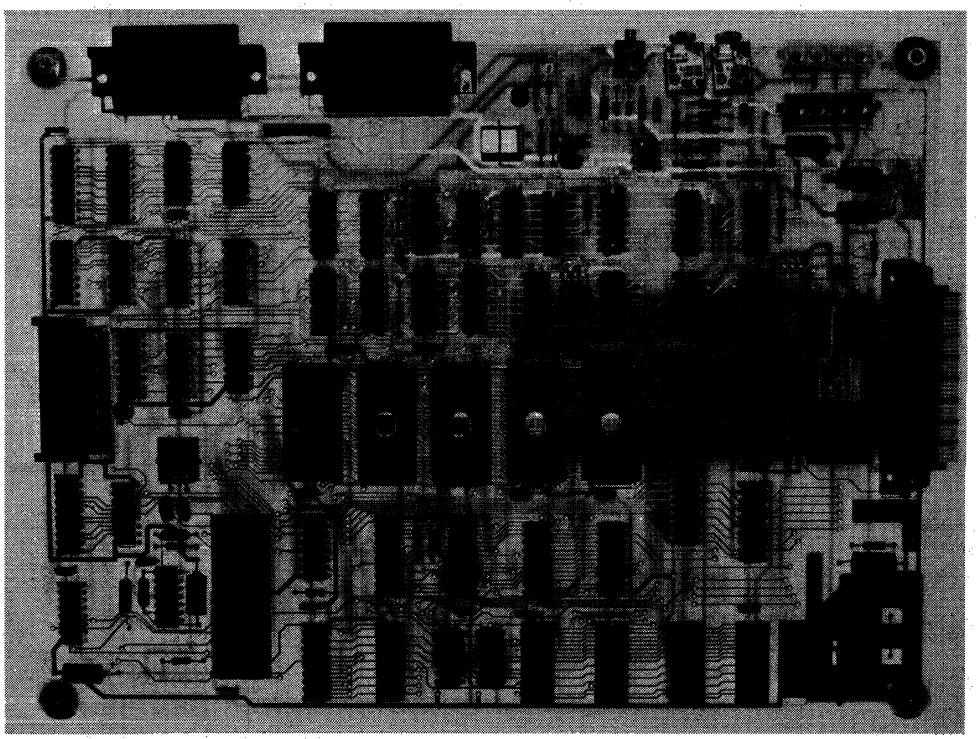
DEVELOPMENT SUPPORT/PART ORDER INFORMATION

TMS32010 EVALUATION MODULE

- Target Connector for Full In-Circuit Emulation
- Debug Monitor Including Over 60 Commands with Full Prompting
- Reverse Assembler
- Transparency Mode for Host CPU Upload/ Download
- Up to Eight Instruction Breakpoints
- Flexible Single Step with Software Trace
- Execution from EVM Program Memory or Target Memory
- Event Counter for One Breakpoint

The Evaluation Module (EVM) is a single board which enables a user to determine inexpensively if the TMS32010 meets the speed and timing requirements of the application. The EVM is a stand-alone module whch contains all the tools necessary to evaluate the TMS32010 as well as to provide full in-circuit emulation via a target connector. A powerful firmware package contains a debug monitor, editor, assembler, reverse assembler, EPROM programmer, communication software to talk to two EIA ports, and an audio cassette interface. The resident assembler will convert incoming source text into executable code in just one pass by automatically resolving labels after the first assembly pass is completed. The EVM can be configured with a dumb terminal, power supplies, and either a host computer, or an audio cassette. Either source or object code can be downloaded into the EVM via the EIA ports provided on the board.

PART NUMBER	POWER SUPPLIES (TM990/518A)	UNITS
RTC/EVM 320A-03	OUTPUT A: +5 VOC (+/– 3%) B: +12 VOC (+/– 3%) C: –12 VOC (+/– 3%)	4.0 A 0.6 A 0.4 A



C

XDS/320 MACRO ASSEMBLER/LINKER

- Macro Capabilities
- Library Functions
- Conditional Assembly
- Relocatable Modules
- Complete Error Diagnostics
- Symbol Table and Cross Reference
- Available on Several Host Computers
- Written in PASCAL

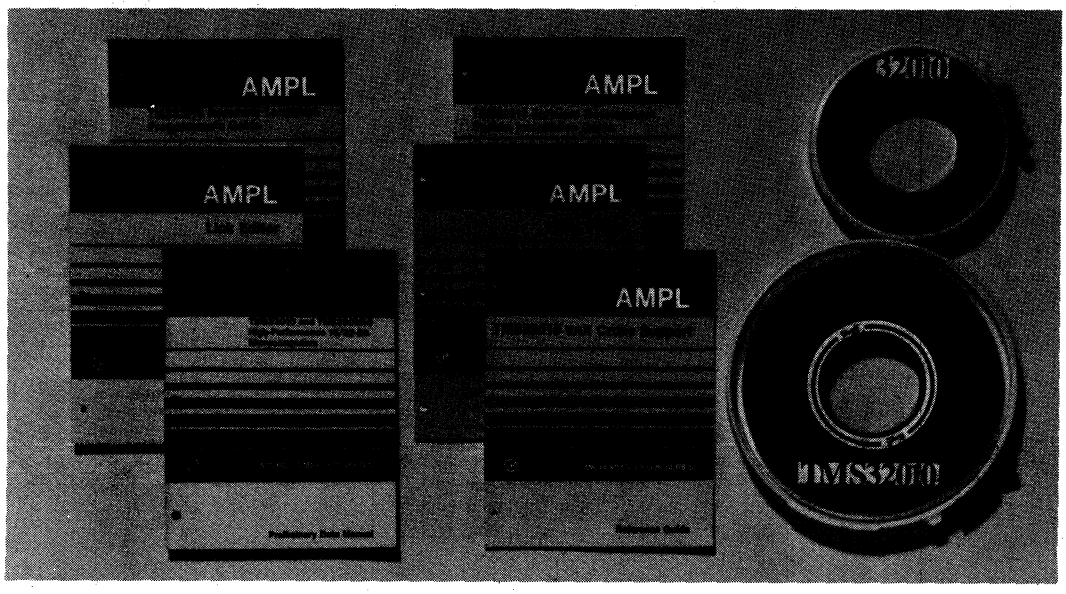
The XDS/320 Macro Assembler translates TMS32010 assembly language into executable object code. The assembler allows the programmer to work with mnemonics rather than hexidecimal machine instructions and to reference memory locations with symbolic addresses. The macro assembler supports macro calls and definitions along with conditional assembly.

The XDS/320 Linker permits a program to be designed and implemented in separate modules which will later be linked together to form the complete program. The linker assigns values to relocatable code, creating an object file which can be executed by the simulator or emulator.

The XDS/320 Macro Assembler and Linker are currently available on several host computers, including TI990(DX10), VAX(VMS), and IBM (MVS and CMS) operating systems. Currently in development is software to support the VAX(UNIX), DEC PDP11(RSX), IBM PC (DOS), and TI professional computer (DOS) operating system. Contact your local TI representative for availability or further details.

HOST	OPERATING SYSTEM	PART NUMBER	MEDIUM
TI990	DX10	TMDS3240120-08	1600 BPI MAG TAPE
DEC VAX	VMS	TMDS3240210-08	1600 BPI MAG TAPE
IBM	MVS	TMDS3240310-08	1600 BPI MAG TAPE
IBM	CMS	TMDS3240320-08	1600 BPI MAG TAPE

For additional host support, please contact your local TI Field Sales Office.



XDS/320 SIMULATOR

- Trace and Breakpoint Capabilities
- Full Access to Simulated Registers and Memories
- I/O Device Simulation
- Runs Object Code Generated by XDS/320 Macro Assembler/Linker
- Available on VAX (VMS)
- Written in FORTRAN

The XDS/320 Simulator is a software program that simulates operation of the TMS32010 to allow program verification. The debug mode enables the user to monitor the state of the simulated TMS32010 while the program is executing. The simulator program uses the TMS32010 object code, produced by the XDS/320 Macro Assembler/Linker. During program execution, the internal registers and memory of the simulated TMS32010 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 XDS/320 Simulator is currently available on the VAX (VMS).

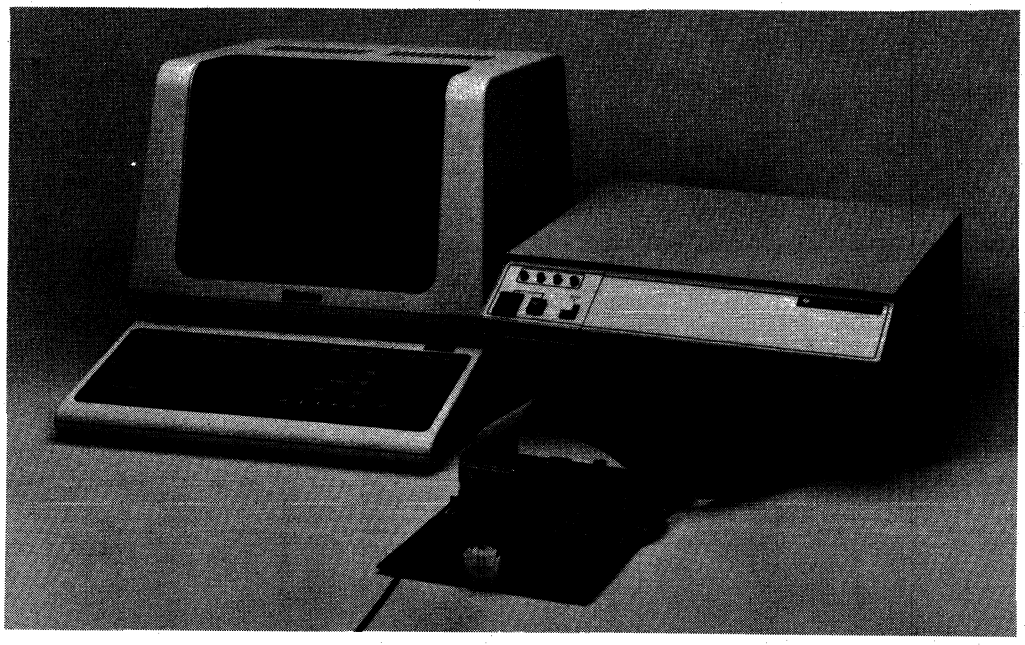
HOST	OPERATING SYSTEM	PART NUMBER	MEDIUM
DEC VAX	VMS	TMDC3240211-08	1600 BPI MAG TAPE

XDS/320 EMULATOR

- 20-MHz Operation (Full In-Circuit Emulation)
- Up to Ten Software Breakpoints
- 4K Words of Program Memory for User Code
- Full Emulation of Microcomputer or Microprocessor Modes
- Use of Target System Crystal, Internal Crystal, or External Clock Signal
- Hardware Breakpoint on Program, Data, or I/O Conditions
- 2K of Full-Speed Hardware Trace
- Single Step
- Assembler/Reverse Assembler
- Host-Independent Upload/Download Capabilities to/from Program or Data Memory
- Ability to Inspect and Modify All Internal Registers, Program and Data Memory
- Multi-Microprocessor Development

The XDS/320 Emulator is a self-contained system that has all the features necessary for real-time in-circuit emulation. This allows integration of the hardware and software 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 are also included to increase debugging productivity. The system provides three EIA ports so that the emulator can be interfaced with a host computer, terminal, printer, or PROM programmer. Using a standard EIA port, the object file produced by the macro assembler/linker can be downloaded into the emulator. The emulator then can be controlled through a terminal.

PART NUMBER
TMDS3262210



C

TMS320 NOMENCLATURE

TMS 320 10 JD L

PREFIX
TMS = standard prefix

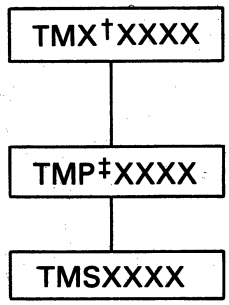
FAMILY
320 = signal processing family

DEVICE
10 = Microprocessor
M10 = Microcomputer (masked ROM)

PACKAGE TYPE
JD = ceramic side-brazed
N = plastic dual-in-line

TEMPERATURE RANGE
L = 0°C to 70°C

DEVELOPMENT FLOWCHART



Engineering prototypes that are not representative of the final device's electrical specifications

Final silicon die that conforms to device's electrical specifications but has not completed quality and reliability verification

Fully qualified production devices

†TMX units shipped against the following disclaimer:
1) Experimental product and its reliability has not been characterized.
2) Product is sold "as is."
3) Not warranted to be exemplary of final production version if or when released by Texas Instruments.

‡TMP units 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.

C

TI Sales Offices

ALABAMA: Huntsville, 500 Wynn Drive, Suite 514, Huntsville, AL 35805, (205) 837-7530.

ARIZONA: Phoenix, P.O. Box 35160, 8102 N. 23rd Ave., Suite A, Phoenix, AZ 85021, (602) 995-1007.

CALIFORNIA: El Segundo, 831 S. Douglas St., El Segundo, CA 90245, (213) 973-2571; **Irvine,** 17891 Cartwright Rd., Irvine, CA 92714, (714) 660-1200; **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, (714) 278-9600; **Santa Clara,** 5353 Betsy Ross Dr., Santa Clara, CA 95054, (408) 980-9000; **Woodland Hills,** 21220 Erwin St., Woodland Hills, CA 91367, (213) 704-7759.

COLORADO: Denver, 9725 E. Hampden St., Suite 301, Denver, CO 80231, (303) 695-2800.

CONNECTICUT: Wallingford, 9 Barnes Industrial Park Rd., Barnes Industrial Park, Wallingford, CT 06492, (203) 269-0074.

FLORIDA: Clearwater, 2280 U.S. Hwy. 19 N., Suite 232, Clearwater, FL 33515, (813) 796-1926; **Ft. Lauderdale,** 2765 N.W. 62nd St., Ft. Lauderdale, FL 33309, (305) 973-8502; **Maitland,** 2601 Maitland Center Parkway, Maitland, FL 32751, (305) 646-9600.

GEORGIA: Atlanta, 3300 Northeast Expy., Building 9, Atlanta, GA 30341, (404) 452-4600.

ILLINOIS: Arlington Heights, 515 W. Algonquin, Arlington Heights, IL 60005, (312) 640-2934.

INDIANA: Ft. Wayne, 2020 Inwood Dr., Ft. Wayne, IN 46805, (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) 890-7400.

MICHIGAN: Farmington Hills, 33737 W. 12 Mile Rd., Farmington Hills, MI 48018, (313) 553-1500.

MINNESOTA: Edina, 7625 Parklawn, Edina, MN 55435, (612) 830-1600.

MISSOURI: Kansas City, 8080 Ward Pkwy., Kansas City, MO 64114, (816) 523-2500; **St. Louis,** 11861 Westline Industrial Drive, St. Louis, MO 63141, (314) 569-7600.

NEW JERSEY: Clark, 292 Terminal Ave. West, Clark, NJ 07066, (201) 574-9800.

NEW MEXICO: Albuquerque, 5907 Alice NSE, Suite E., Albuquerque, NM 87110, (505) 265-8491.

NEW YORK: East Syracuse, 6700 Old Collamer Rd., 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; **Poughkeepsie,** 201 South Ave., Poughkeepsie, NY 12601, (914) 473-2900; **Rochester,** 1210 Jefferson Rd., Rochester, NY 14623, (716) 424-5400.

NORTH CAROLINA: Charlotte, 8 Woodlawn Green, Woodlawn Rd., Charlotte, NC 28210, (704) 527-0930; **Raleigh,** 3000 Highwoods Blvd., Suite 118, 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.

OKLAHOMA: Tulsa, 7615 East 63rd Place, 3 Memorial Place, Tulsa, OK 74133, (405) 250-0633.

OREGON: Beaverton, 6700 SW 105th St., Suite 110, Beaverton, OR 97005, (503) 643-6758.

PENNSYLVANIA: Ft. Washington, 575 Virginia Dr., Ft. Washington, PA 19034, (215) 643-6450; Coraopolis, PA 15108, 420 Rouser Rd., 3 Airport Office PK, (412) 771-8550.

TEXAS: Austin, 12501 Research Blvd., P.O. Box 2909, Austin, TX 78723, (512) 250-7655; **Dallas,** P. O. Box 1087, Richardson, TX 75080; **Houston,** 9100 Southwest Frwy., Suite 237, Houston, TX 77036, (713) 778-6592; **San Antonio,** 1000 Central Park South, San Antonio, TX 78232, (512) 496-1779.

UTAH: Salt Lake City, 3672 West 2100 South, Salt Lake City, UT 84120, (801) 973-6310.

VIRGINIA: Fairfax, 3001 Prosperity, Fairfax, VA 22031, (703) 849-1400; **Midlothian,** 13711 Sutter's Mill Circle, Midlothian, VA 23113, (804) 744-1007.

WISCONSIN: Brookfield, 205 Bishops Way, Suite 214, Brookfield, WI 53005, (414) 784-3040.

WASHINGTON: Redmond, 2723 152nd Ave., N.E. Bldg. 6, Redmond, WA 98052, (206) 881-3080.

CANADA: Ottawa, 436 Mac Laren St., Ottawa, Canada, K2P0M8,(613) 233-1177; **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) 334-3635. C

TI Distributors

ALABAMA: Hall-Mark (205) 837-8700.

ARIZONA: Phoenix, Kierulff (602) 243-4101; Marshall (602) 968-6181; Wyle (602) 249-2232; **Tucson,** Kierulff (602) 624-9986.

CALIFORNIA: Los Angeles/Orange County, Arrow (213) 701-7500, (714) 851-8961; Kierulff (213) 725-0325, (714) 731-5711; Marshall (213) 999-5001, (213) 442-7204, (714) 556-6400; R.V. Weatherford (714) 634-9600, (213) 849-3451, (714) 623-1261; Wyle (213) 322-8100, (714) 641-1600, **San Diego,** Arrow (619) 565-4800; Kierulff (619) 278-2112; Marshall (619) 578-9600; R. V. Weatherford (619) 695-1700; Wyle (619) 565-9171; **San Francisco Bay Area,** Arrow (408) 745-6600; Kierulff (415) 968-6292; Marshall (408) 732-1100; Wyle (408) 727-2500; **Santa Barbara,** R. V. Weatherford (805) 965-8551.

COLORADO: Arrow (303) 758-2100; Kierulff (303) 790-4444; Wyle (303) 457-9953.

CONNECTICUT: Arrow (203) 265-7741; Diplomat (203) 797-9674; Kierulff (203) 265-1115; Marshall (203) 265-3822; Milgray (203) 795-0714.

FLORIDA: Ft. Lauderdale, Arrow (305) 776-7790; Diplomat (305) 971-7160; Hall-Mark (305) 971-9280; Kierulff (305) 652-6950; **Orlando,** Arrow (305) 725-1480; Diplomat (305) 725-4520; Hall-Mark (305) 855-4020; Milgray (305) 647-5747; **Tampa,** Diplomat (813) 443-4514; Hall-Mark (813) 576-8691; Kierulff (813) 576-1966.

GEORGIA: Arrow (404) 449-8252; Hall-Mark (404) 447-8000; Kierulff (404) 447-5252; Marshall (404) 923-5750.

ILLINOIS: Arrow (312) 397-3440; Diplomat (312) 595-1000; Hall-Mark (312) 860-3800; Kierulff (312) 640-0200; Newark (312) 638-4411.

TEXAS INSTRUMENTS

Creating useful products and services for you.

INDIANA: Indianapolis, Arrow (317) 243-9353; Graham (317) 634-8202; **Ft. Wayne,** Graham (219) 423-3422.

IOWA: Arrow (319) 395-7230.

KANSAS: Kansas City, Component Specialties (913) 492-3555; Hall-Mark (913) 888-4747; **Wichita,** LCOMP (316) 265-9507.

MARYLAND: Arrow (301) 247-5200; Diplomat (301) 995-1226; Hall-Mark (301) 796-9300; Kierulff (301) 247-5020; Milgray (301) 468-6400.

MASSACHUSETTS: Arrow (617) 933-8130; Diplomat (617) 429-4120; Kierulff (617) 667-8331; Marshall (617) 272-8200; Time (617) 935-8080.

MICHIGAN: Detroit, Arrow (313) 971-8200; Newark (313) 967-0600; **Grand Rapids,** Arrow (616) 243-0912.

MINNESOTA: Arrow (612) 830-1800; Hall-Mark (612) 854-3223; Kierulff (612) 941-7500.

MISSOURI: Kansas City, LCOMP (816) 221-2400; **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) 235-1900; Diplomat (201) 785-1830; General Radio (609) 964-8560; Hall-Mark (201) 575-4415, (609) 424-7300, JACO (201) 778-4722, (800) 645-5130; Kierulff (201) 575-6750; Marshall (201) 882-0320; Milgray (609) 983-5010, (800) 645-3956.

NEW MEXICO: Arrow (505) 243-4566; International Electronics (505) 345-8127.

NEW YORK: Long Island, Arrow (516) 231-1000; Diplomat (516) 454-6334; Hall-Mark (516) 737-0600; JACO (516) 273-5500; Marshall (516) 273-2424; Milgray (516) 546-5600, (800) 645-3986; Hall-Mark (516) 737-0600; **Rochester,** Arrow (716) 275-0300; Marshall (716) 235-7620; Rochester Radio Supply (716) 454-7800; **Syracuse,** Arrow (315) 652-1000; Diplomat (315) 652-5000; Marshall (607) 754-1570.

NORTH CAROLINA: Arrow (919) 876-3132, (919) 725-8711; Hall-Mark (919) 872-0712; Kierulff (919) 852-9440.

OHIO: Cincinnati, Graham (513) 772-1661; Hall-Mark (513) 563-5980; **Cleveland,** Arrow (216) 248-3990; Hall-Mark (216) 473-2907; Kierulff (216) 587-6558; **Columbus,** Hall-Mark (614) 891-4555, **Dayton,** Arrow (513) 435-5563; ESCO (513) 226-1133; Marshall (513) 236-8088.

OKLAHOMA: Arrow (918) 665-7700; Component Specialties (918) 664-2820; Hall-Mark (918) 665-3200; Kierulff (918) 252-7537.

OREGON: Kierulff (503) 641-9150; Wyle (503) 640-6000.

PENNSYLVANIA: Arrow (412) 856-7000, (215) 928-1800; General Radio (215) 922-7037; Hall-Mark (215) 355-7300.

TEXAS: Austin, Arrow (512) 835-4180; Component Specialties (512) 837-8922; Hall-Mark (512) 258-8848; Kierulff (512) 835-2090; **Dallas,** Arrow (214) 386-7500; Component Specialties (214) 357-6511; Hall-Mark (214) 341-1147; International Electronics (214) 233-9323; Kierulff (214) 343-2400; **El Paso,** International Electronics (915) 778-9761; **Houston,** Arrow (713) 491-4100; Component Specialties (713) 771-7237; Hall-Mark (713) 781-6100; Harrison Equipment (713) 879-2600; Kierulff (713) 530-7030.

UTAH: Diplomat (801) 486-4134; Kierulff (801) 973-6913; Wyle (801) 974-9953.

VIRGINIA: Arrow (804) 282-0413.

WASHINGTON: Arrow (206) 643-4800; Kierulff (206) 575-4420; Wyle (206) 453-8300.

WISCONSIN: Arrow (414) 764-6600; Hall-Mark (414) 761-3000; Kierulff (414) 784-8160.

CANADA: Calgary, Future (403) 259-6408; Varah (403) 230-1235; **Hamilton,** Varah (416) 561-9311; **Montreal,** CESCO (514) 735-5511; Future (514) 694-7710; **Ottawa,** CESCO (613) 226-6905; Future (613) 820-8313; **Quebec City,** CESCO (418) 687-4231; **Toronto,** CESCO (416) 661-0220; Future (416) 663-5563; **Vancouver,** Future (604) 438-5545; Varah (604) 873-3211; **Winnipeg,** Varah (204) 633-6190. BC

TI Worldwide Sales Offices

ALABAMA: Huntsville, 500 Wynn Drive, Suite 514, Huntsville, AL 35805, (205) 837-7530.

ARIZONA: Phoenix, P.O. Box 35160, 8102 N. 23rd Ave., Suite A, Phoenix, AZ 85021, (602) 995-1007.

CALIFORNIA: El Segundo, 831 S. Douglas St., El Segundo, CA 90245, (213) 973-2571; **Irvine,** 17891 Cartwright Rd., Irvine, CA 92714, (714) 660-1200; **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, (714) 278-9600; **Santa Clara,** 5353 Betsy Ross Dr., Santa Clara, CA 95054, (408) 980-9000; **Woodland Hills,** 21220 Erwin St., Woodland Hills, CA 91367, (213) 704-7759.

COLORADO: Denver, 9725 E. Hampden St., Suite 301, Denver, CO 80231, (303) 695-2800.

CONNECTICUT: Wallingford, 9 Barnes Industrial Park Rd., Barnes Industrial Park, Wallingford, CT 06492, (203) 269-0074.

FLORIDA: Clearwater, 2280 U.S. Hwy. 19 N., Suite 232, Clearwater, FL 33515, (813) 796-1926; **Ft. Lauderdale,** 2765 N.W. 62nd St., Ft. Lauderdale, FL 33309, (305) 973-8502; **Maitland,** 2601 Maitland Center Parkway, Maitland, FL 32751, (305) 646-9600.

GEORGIA: Atlanta, 3300 Northeast Expy., Building 9, Atlanta, GA 30341, (404) 452-4600.

ILLINOIS: Arlington Heights, 515 W. Algonquin, Arlington Heights, IL 60005, (312) 640-2934.

INDIANA: Ft. Wayne, 2020 Inwood Dr., Ft. Wayne, IN 46805, (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) 890-7400.

MICHIGAN: Farmington Hills, 33737 W. 12 Mile Rd., Farmington Hills, MI 48018, (313) 553-1500.

MINNESOTA: Edina, 7625 Parklawn, Edina, MN 55435, (612) 830-1600.

MISSOURI: Kansas City, 8080 Ward Pkwy., Kansas City, MO 64114, (816) 523-2500; **St. Louis,** 11861 Westline Industrial Drive, St. Louis, MO 63141, (314) 569-7600.

NEW JERSEY: Clark, 292 Terminal Ave. West, Clark, NJ 07066, (201) 574-9800.

NEW MEXICO: Albuquerque, 5907 Alice NSE, Suite E., Albuquerque, NM 87110, (505) 265-8491.

NEW YORK: East Syracuse, 6700 Old Collamer Rd., 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; **Poughkeepsie,** 201 South Ave., Paughkeepsie, NY 12601, (914) 473-2900; **Rochester,** 1210 Jefferson Rd., Rochester, NY 14623, (716) 424-5490.

NORTH CAROLINA: Charlotte, 8 Woodlawn Green, Woodlawn Rd., Charlotte, NC 28210, (704) 527-0930; **Raleigh,** 3000 Highwoods Blvd., Suite 118, 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.

OKLAHOMA: Tulsa, 7615 East 63rd Place, 3 Memorial Place, Tulsa, OK 74133, (405) 250-0633.

OREGON: Beaverton, 6700 SW 105th St., Suite 110, Beaverton, OR 97005, (503) 643-6758.

PENNSYLVANIA: Ft. Washington, 575 Virginia Dr., Ft. Washington, PA 19034, (215) 643-6450; Coraopolis, PA 15108, 420 Rouser Rd., 3 Airport Office PK, (412) 771-8550.

TEXAS: Austin, 12501 Research Blvd., P.O. Box 2909, Austin, TX 78723, (512) 250-7655; **Dallas,** P. O. Box 1087, Richardson, TX 75080; **Houston,** 9100 Southwest Frwy., Suite 237, Houston, TX 77036, (713) 778-6592; **San Antonio,** 1000 Central Park South, San Antonio, TX 78232, (512) 496-1779.

UTAH: Salt Lake City, 3672 West 2100 South, Salt Lake City, UT 84120, (801) 973-6310.

VIRGINIA: Fairfax, 3001 Prosperity, Fairfax, VA 22031, (703) 849-1400; **Midlothian,** 13711 Sutter's Mill Circle, Midlothian, VA 23113, (804) 744-1007.

WISCONSIN: Brookfield, 205 Bishops Way, Suite 214, Brookfield, WI 53005, (414) 784-3040.

WASHINGTON: Redmond, 2723 152nd Ave., N.E. Bldg. 6, Redmond, WA 98052, (206) 881-3080.

CANADA: Ottawa, 436 Mac Laren St., Ottawa, Canada, K2P0M8,(613) 233-1177; **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) 334-3635. C

ARGENTINA, Texas Instruments Argentina S.A.I.C.F.: Esmeralda 130, 15th Floor, 1035 Buenos Aires, Argentina, 394-2963.

AUSTRALIA (& NEW ZEALAND), Texas Instruments Australia Ltd.: 6-10 Talavera Rd., North Ryde (Sydney), New South Wales, Australia 2113, 02+887-1122; 5th Floor, 418 St. Kilda Road, Melbourne, Victoria, Australia 3004, 03+267-4677; 171 Philip Highway, Elizabeth, South Australia 5112, 08+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, 02/720.80.00.

BRAZIL, Texas Instruments Electronicos do Brasil Ltda.: Av. Faria Lima, 2003, 20 0 Andar—Pinheiros, Cep-01451 Sao Paulo, Brazil, 815-6166.

DENMARK, Texas Instruments A/S, Marielundvej 46E, DK-2730 Herlev, Denmark, 2 - 91 74 00.

FINLAND, Texas Instruments Finland OY: PL 56, 00510 Helsinki 51, Finland, (90) 7013133.

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) 79-54-81; 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, Texas Instruments Deutschland GmbH: Haggerty-strasse 1, D-8050 Freising, 08161-801; Kurfuerstendamm 195/196, D-1000 Berlin 15, 030-8827365; III, Hagen 43/Kibbelstrasse, D-4300 Essen, 0201-24250; Frankfurter Allee 6-8, D-6236 Eschborn 1, 06196-43074; Hamburger Strasse 11, D-2000 Hamburg 76, 040-2201154, Kirchhorsterstrasse 2, D-3000 Hannover 51, 0511-648021; Arabellastrasse 15, D-8000 Muenchen 81, 089-92341; Maybachstrasse 11, D-7302 Ostfildern 2/Nellingen, 0711-34030.

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: 25 St. Stephens Green, Dublin 2, Eire, 01 609222.

ITALY, Texas Instruments Semiconduttori Italia Spa: Viale Delle Scienze, 1, 02015 Cittaducale (Rieti), Italy, 0746 694.1; Via Salaria KM 24 (Palazzo Cosma), Monterotondo Scalo (Rome), Italy, 06 9004395; Viale Europa, 38-44, 20093 Cologno Monzese (Milano), 02 2532541; Corso Svizzera, 185, 10100 Torino, Italy, 011 774545; Via J. Barozzi, 6, 45100 Bologna, Italy, 051 355851.

JAPAN, Texas Instruments Asia Ltd.: 4F Aoyama Fuji Bldg., 6-12, Kita Aoyama 3-Chome, Minato-ku, Tokyo, Japan 107, 03-498-2111; 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, 052-583-8691.

KOREA, Texas Instruments Supply Co.: Room 201, Kwangpoong Bldg., 24-1, Hwayand-Dong, Sung dong-ku, 133 Seoul, Korea, 02+464-6274/5.

MEXICO, Texas Instruments de Mexico S.A.: Poniente 116, No. 489, Colonia Vallejo, Mexico, D.F. 02300, 567-9200.

MIDDLE EAST, Texas Instruments: No. 13, 1st Floor Mannai Bldg., Diplomatic Area, Manama, P.O. Box 26335, Bahrain, Arabian Gulf, 973 - 72 46 81.

NETHERLANDS, Texas Instruments Holland B.V., P.O. Box 12995, (Bullewijk) 1100 AZ Amsterdam, Zuid-Oost, Holland (020) 5602911.

NORWAY, Texas Instruments Norway A/S: Kr. Augustsgt. 13, Oslo 1, Norway, (2) 20 60 40.

PHILIPPINES, Texas Instruments Asia Ltd.: 14th Floor, Ba-Lepanto Bldg., 8747 Paseo de Roxas, Makati, Metro Manila, Philippines, 882465.

PORTUGAL, Texas Instruments Equipamento Electronico (Portugal), Lda.: Rua Eng. Frederico Ulrich, 2650 Moreira Da Maia, 4470 Maia, Portugal, 2-9481003.

SINGAPORE (+ INDIA, INDONESIA, MALAYSIA, THAILAND), Texas Instruments Asia Ltd.: P.O. Box 138, Unit #02-08, Block 6, Kolam Ayer Industrial Est., Kallang Sector, Singapore 1334, Republic of Singapore, 747-2255.

SPAIN, Texas Instruments Espana, S.A.: C/Jose Lazaro Galdiano No. 6, Madrid 16, 1/458.14.58. C/Balmes, 89 Barcelona-8, 253 60 00/253 29 02.

SWEDEN, Texas Instruments International Trade Corporation (Sverigefilialen): Box 39103, 10054 Stockholm, Sweden, 08 - 235480.

SWITZERLAND, Texas Instruments, Inc. Riedstrasse 6, CH-8953 Dietikon (Zuerich) Switzerland, 1-740 2220.

TAIWAN, Texas Instruments Supply Co.: 10th Floor, Fu-Shing Bldg., 71 Sung-Kiang Road, Taipei, Taiwan, Republic of China, 02+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, 061 442-8448. BC

CROSS-ASSEMBLER DIRECTIVES (CONCLUDED)

DIGITAL SIGNAL PROCESSOR

ASCII REFERENCE TABLE

HEX-DECIMAL TABLE

RTC HOTLINE NUMBERS

TEXAS INSTRUMENTS

March 1985
Revision B
1603459-9701
Printed in the U.S.A.

TEXAS INSTRUMENTS

Creating useful products
and services for you.

SPRU001B