TEXAS INSTRUMENTS

AMPL

TMAM 6095 EVALUATION MODULE

FOR TMS 9995 MICROPROCESSOR

MICROPROCESSOR SERIES™

User's Manual

Return to Colin Hixson RTC

IMPORTANT NOTICES

Texas Instruments Reserves the right to make changes at any time in order to supply the customer with the best possible product.

TI cannot assume responsibility for any circuits shown or represent that they are free from patent infringement.

TABLE OF CONTENTS

1.0 INTRODUCTION

2.0 INSTALLATION

3.0 OPERATION

4.0 THEORY OF OPERATION

5.0 EVMBUG INTERACTIVE MONITOR

SYMBOLIC ASSEMBLER

7.0 EIA COMMUNICATIONS LINK

8.0 PROGRAMMING

APPENDICES

LIST OF ILLUSTRATIONS

LIST OF TABLES

INTRODUCTION

1.1 GENERAL

The TMAM 6095 Evaluation Module (EVM) is a self-contained, single-board microcomputer system. It is intended for use as a vehicle to provide low cost evaluation capability for the TMS 9995 microcomputer hardware/software systems. Throughout this document, the evaluation module will be referred to as the TMS 9995 EVM.

The TMS 9995 EVM contains firmware that enables programs to be assembled, edited, and executed. A powerful symbolic assembler also provides reverse assembly capability. The module will support 24K bytes of firmware without hardware expansion. Wait states are individualy selectable for each memory device.

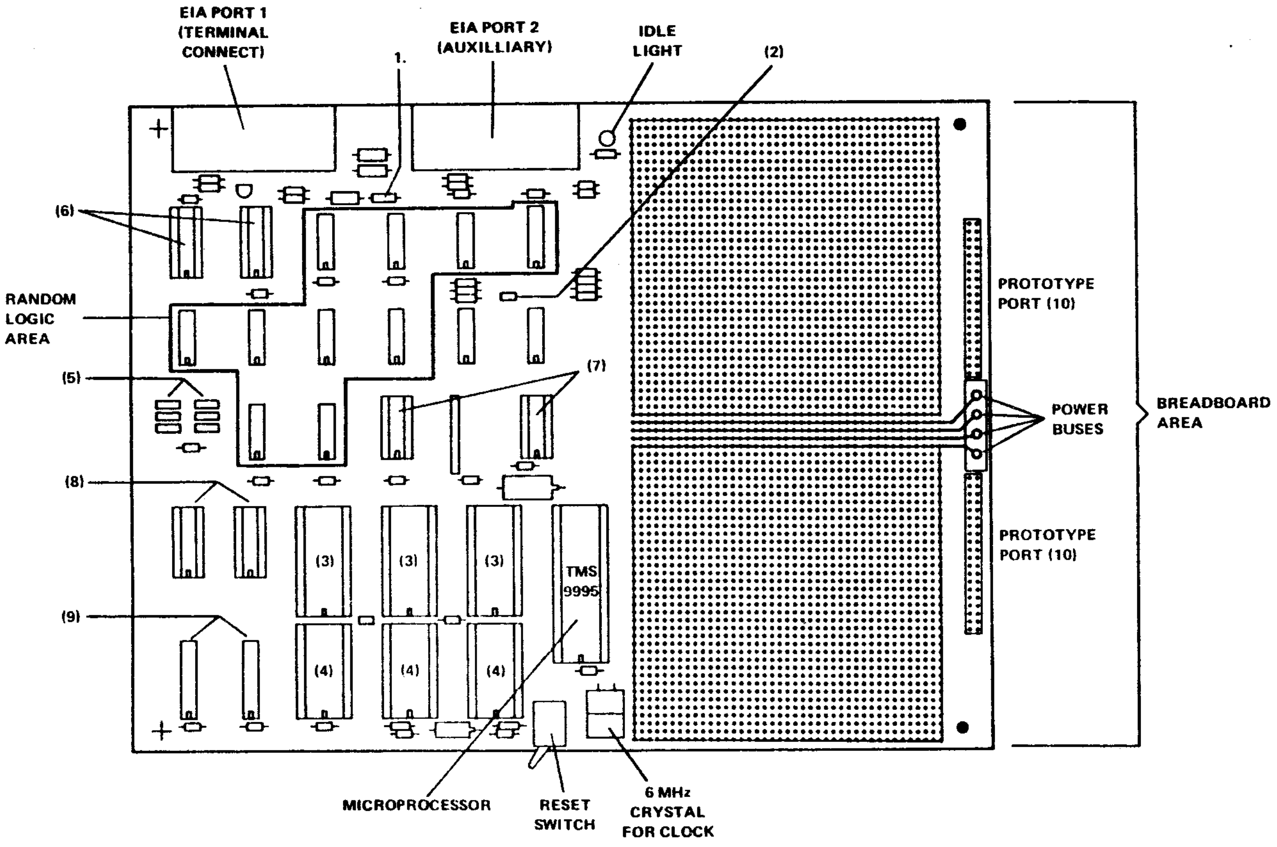


FIGURE 1-1. TMS 9995 EVALUATION MODULE.

The system´s features include:

- A debug monitor
- A symbolic assembler
- A reverse assembler
- Two EIA RS-232C data communication link ports providing interfa to a local terminal and to a host system for upload/download capability.
- Three user-configurable 28-pin memory sockets that will suport most "by 8" memory devices, i.e., 8K - 64K ROMs/EPROMs, "by 8" RAMs, and bi-polar PROMs.
- The TMS 9995 microcomputer with 256 bytes of on-chip RAM.
- 1K Bytes of external RAM populated on the board.
- 12 kilobytes of EPROM containing the supplied firmware. In addition, up to 64K of EPROM may be obtained by populating the three general-purpose sockets with "by 8" TMS memory devices.
- A large prototyping area providing ample room for breadboarding with TMS 9995 systems.
- 12 MHz crystal-controlled clock.
- Manual reset switch.
- Most signals are available at the edge of the prototyping area. Provision for off-board expansion is possible using dual ribbon cable connectors.

In addition to the basic TMS 9995 Evaluation Module, the following options are available:

- Power Supply Unit, part number TM990/519

1.2 BOARD CONFIGURATION

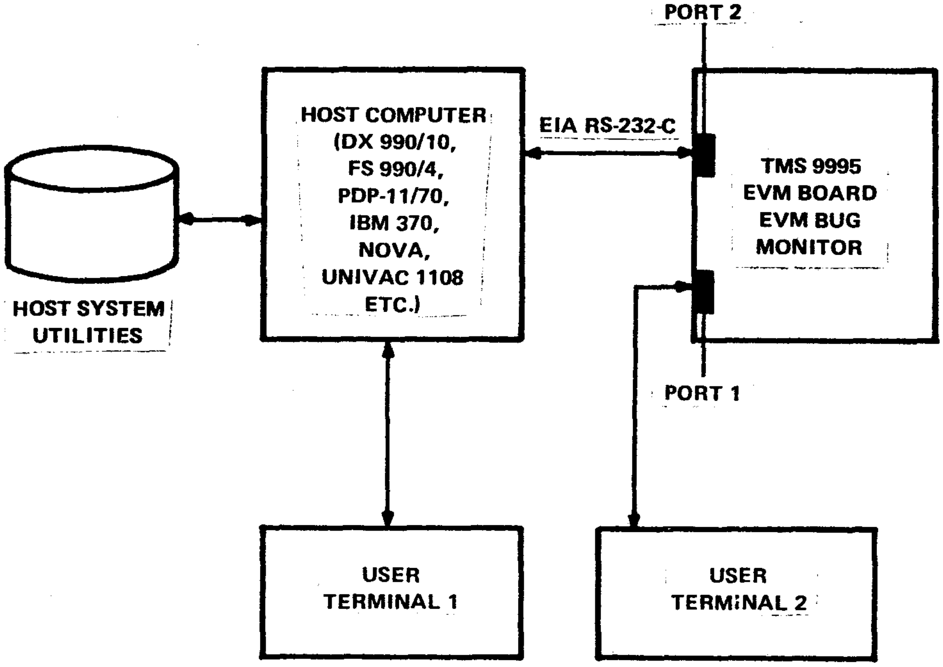


FIGURE 1-2. TMS 9995 EVALUATION MODULE CONFIGURATION.

1. Jumper Plug: allows the selection of the type of terminal to be used withthe EVM. A 3-prong plug. If prongs 1 and 2 are connected, a teletype terminal may be used; if prongs 2 and 3 a connected, an RS232C/EIA-type terminal may be used.

2. Jumper Plug: enables/disables automatic first Wait state. If prongs 1 and 2 are connected, enables; if 2 and 3 are connected Wait states are disabled.

3. General-purpose memory sockets: 28 pin.

4. Corresponding Personality Plugs: connects appropriate signals to the corresponding general-purpose memory socket.

5. Jumper Plugs (6): J1, J2, and J3 plugs determine how the memory signal to the general-purpose memory sockets is generated., and J4, J5, and J6 plugs either enable or disable a Wait state for the corresponding general-purpose memory socket.

6. The Universal Asynchronous Communications Controller (UARTS): provides interface between the processor CRU and the EIA ports.

7. Memory Decode PROMs; determine the memory map.

8. On-board Random Access Memory (RAM): 1K bytes.

9. Buffers for on-board RAM.

10. Two prototype ports, which allow the user to connect other peripheral equipment (i.e., audio cassette, VDT, additional terminals, etc.) by means of a wrap-post header ribbon cable.

1.3 GENERAL SPECIFICATIONS

Board Dimensions: 8.5" x 11"

Memory Size:

RAM: 1024 bytes (1K) on board; 256 bytes in the TMS 9995.

EPROM: 6K, expandable to 24K by populating the general-purpose memory sockets with TMS 2564 EPROMs.

Clock Rate: 3 MHz

Baud Rates: Variable, dependent upon type of terminal being used.

1.4 REFERENCE DOCUMENTS

The following is a list of documents that will provide supplementary information for the TMS 9995 EVM user:

- TMS 9900 Family System Development Manual, part number LCC4400

- TMS 9995 Microcomputer Data Manual, part number MP021

1.5 NUMERICAL REPRESENTATIONS

For the purposes of delineating between decimal, hexadecimal and binary number in this manual, hexadecimal numbers are preceeded by a greater-than sign: (>). Decimal number are unsigned. Binary number are so noted.

EXAMPLE:

>0000	HEXADECIMAL
1234	DECIMAL
1101(Binary)	BINARY

1.6 GLOSSARY

The following are definitions of terms used with the TMS 9995 EVM.

Absolute Address: the actual memory address in quantity of bytes. Memory addressing is usually represented in hexadecimal from >0000 to >FFFF.

Alphabetic Character: A to Z. On dual-printed keys, the character printed on the lower half of the key.

Alphanumeric Character: letters, numbers, and associated symbols.

ASCII Code: a seven-bit code used to represent alphanumeric characters and control characters.

Assembler: the program that translates assembly language source statement into machine usable object code.

Assembly Language: mnemonics which can be interpreted by an asembler and translated into an object program.

Bit: (Binary DI set) the smallest part of a word; it has a value of either 1 or 0.

Breakpoint: a memory address where a program is intentionally halted. This is a program debugging tool.

Byte: eight bits.

Carry: a carry occurs when the most-significant bit overflows in an arithmetic operation; i.e., when the resultant cannot be contained in only 16 bits. Same as an overflow.

Central Processing Unit (CPU): the "heart" of the computer. Responsibilities include instruction access and interpretation, arithmeitc functions, and I/O memory access. The CPU is contained in the TMS9995 microcomputer.

Command Scanner: a set of instructions in the debug monitor which takes the user´s input from the terminal and searches a table for the proper program to execute the command.

Context Switch: a change in the program execution environment. Includes the new program counter (PC) value and the new workspace pointer (WP). Usually caused by an interrupt subroutine call.

CPU: see Central Processing Unit.

Effective Address: a memory address resulting from the interpretation of an instruction; required for the execution of that instruction.

EIA: The acronym as used in this manual signifies a RS232-B or C, serial interface and implies use of the standard 25 connector as specified by the Electronic Industries Association.

EPROM: see Read Only Memory.

Hexadecimal: a numerical notation in the base 16. In this manual, denoted by ">" preceeding a number.

Indexed Addressing: the effective address is the sum of the contents of an index register and a displacement.

Indirect Addressing: a method of cross referencing in which one memory location (the indirect address) contains the address for the desired operand. The actual address is the contents of the indirect address register.

Interrupt: an externally generated context switch in which the new work- space pointer (WP) and program counter (PC) values are obtained from one of four interrupt vectors in memory addresses >0000 to >0012, or the non-maskable interrupt (NMI) vector at address >FFFC. The old PC, WP and status register (ST) values are saved so that a return to the context prior to the interrupt can be made.

I/O: input/output. I/O lines are the signals which connect an external device to the data lines of the TMS9995.

Least Significant Bit (LSB): the bit having the smallest value in a byte or word (smallest power of base 2); represented by the right-most bit.

Loader: a program that places one or more absolute or relocatable object programs into memory.

Machine Language: binary code that can be interpreted by the CPU.

Monitor: a program that assists in the real-time aspects of program execution, such as operator command interpretation and supervisor call execution. Sometimes called the Supervisor.

Most Significant Bit (MSB): The bit having the largest value in a byte. Represented by the left-most bit.

Numeric Character: numbers 1-10. On dual-printed keys, the character printed on the lower half of the key.

One´s Complement: binary representation of a number in which the negative of the number is the complement or inverse of the positive number (all ones become zeroes and vice-versa). The most significant bit (MSB) is one for a negative number and zero for positive number. Two representations exist for zero: all ones or all zeroes.

Op Code: binary operation code interpreted by the CPU to execute an instruction.

Overflow: an overflow occurs when the result of an arithmetic operation cannot be represented in two´s complement, i.e., in 15 bits plus the sign bit. Same as a carry.

Parity: the means for checking validity of a series of bits, usually a byte. Odd parity means an odd number of bits; even parity means an even number of logic one bits. A parity bit is set to make all bytes conform to the selected parity. If the parity is not as anticipated, an error flag can be set by software. The parity jump instruction can be used to determine parity.

Program Counter (PC): a hardware register that points to the next instruction to be executed.

PROM: Programmable Read Only Memory. See Read Only Memory.

Random Access Memory (RAM): memory that can be written to as well as read from (vs read only memory). Usually loses its contents when power is turned off.

Read Only Memory (ROM): memory that can only be read from (can´t change the contents). Some can be programmed (PROM) using a PROM programmer. Some PROMs can be erased (EPROMs) by exposure to ultraviolet light.

Source Program: programs written in mnemonics that can be translated into machine language by an assembler.

Status Register (ST): a hardware register that reflects the outcome of a previous instruction and the current interrupt mask.

Supervisor: see Monitor.

Utilities: routines used by different parts of the program to perform the same functions.

Wire-OR: externally connecting separate circuits/functions so that the combination of their outputs results in an "OR´ function.

Word: sixteen bits or two bytes.

Workspace Pointer (WP): a hardware register that contains the memory address of the beginning of the workspace area; points to Register 0.

Workspace Register Area: sixteen words, designated registers 0 to 15, located in RAM for use by the executing program.

XOP: Extended Operation. A software generated context switch. Can be considered as a system jump table.

SECTION 2

INSTALLATION

2.1 GENERAL

This section provides instructions for the installation of the basic TMS 9995 Evaluation Module.

The following paragraphs will enable the user to determine the power, space, and environmental requirements for the TMS 9995 EVM.

2.2 REQUIRED EQUIPMENT

- TMS 9995 EVM Board, part number 1603162
- Power Supply Cable, part number 991747
- TM 990/519 Power Supply, part number 991748
- Terminal: EIA RS-232 or 20ma current loop compatible TTY

2.3 POWER SUPPLY

The TM 990/519 power supply is plugged into a standard AC wall outlet. Fig. 2-1 shows how to connect the TM 990/519 power supply to the TMS 9995 EVM by means of the power-connect cable supplied with the board. The connections on each end of the cable are positively keyed and prohibit misconnection to the power supply. Furthermore, this cable is wired "one-for-one", and either end may be connected to the power supply or the board.

2.4 SPACE AND ENVIRONMENTAL REQUIREMENTS

The TMS 9995 EVM setup requires adequate space on a flat, non-conductive horizontal surface. The space must allow room on the side for cable connections and placement of the power supply, space to the rear for placement of the terminal cable, and clearance to the front for user access to both the module and the terminal. If desired, space should also be provided for placement of an oscilloscope. The workspace provided must be free of any material that could block the ventilation louvers on the underside of the terminal.

Environmental requirements are the same as for any microprocessor system: a reasonably open, air conditioned area. Air temperature should not exceed 80 degrees Farenheit; humidity, 80 percent.

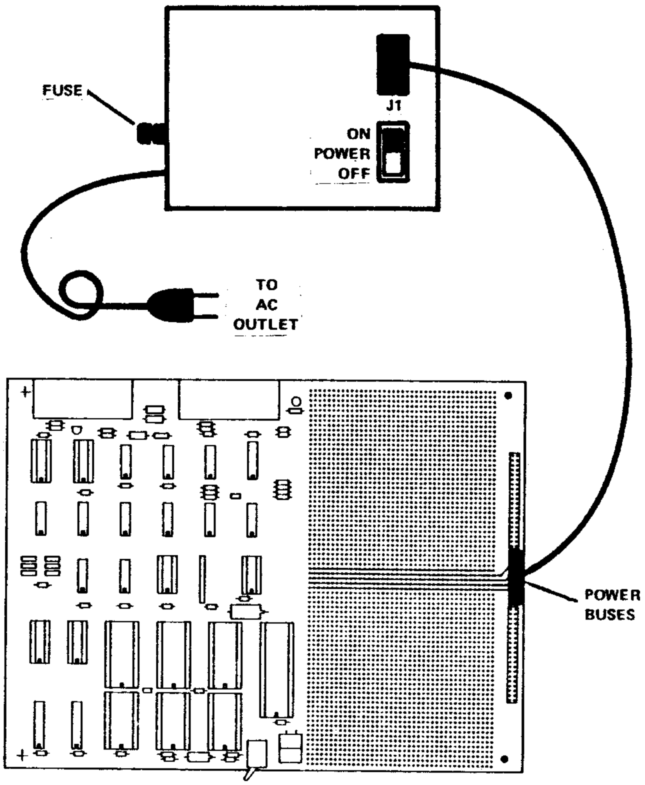


FIGURE 2-1. POWER SUPPLY HOOKUP.

2.5 UNPACKING

Lift the TMS 9995 EVM from its carton and remove the protective wrapping. Check for shipping damage. If any damage is found, notify your TI distributor.

Verify that the following components are included:

- TMS 9995 EVM

- Power-connect Cable

2.6 HOOKUP

1. Attach Power-connect cable to EVM module and power supply, as shown in Fig. 2-1.

   NOTE: If using a power supply other then the TM990/519, the user should remove one connector from the cable and attach the proper connector or plugs for the power supply to be used. The power cable conductors are color coded as follows:

   +5V - Red
   +12V - White
   -12V - Green
   Ground - Black

2. Connect terminal cable to EVM module, as shown in Fig. 2-2, below.

3. Plug power supply line into any properly grounded AC wall outlet.

CAUTION

Be very careful to apply correct voltage levels to the TMS 9995 EVM. Texas Instruments assumes no responsibility for damage caused by improper wiring or voltage applications by the user.

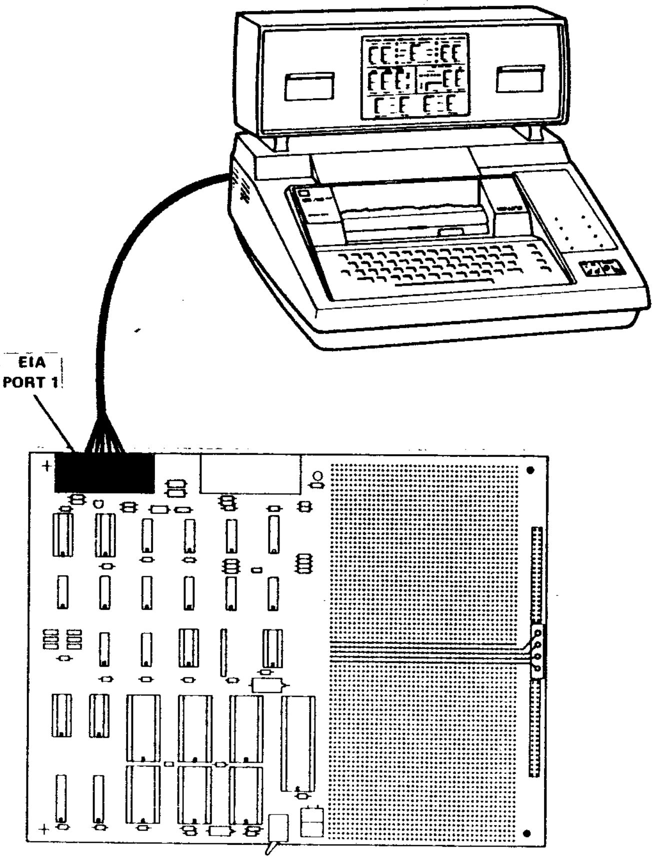


FIGURE 2-2. TERMINAL HOOKUP.

SECTION 3

OPERATION

3.1 GENERAL

This section contains a system check-out procedure to verify that the system is operational; and presents error and system malfunction correction procedures, and basic operating procedures.

3.2 VERIFICATION

Verify the following conditions before applying power:

- Power is connected to correct pins on P1 connector.
- Terminal cable is between P2 connector (NOT P3) and terminal.
- Jumpers are in correct positions:
  - J1 joins location 2 and location 3.
  - J2 joins location 1 and location 2.
  - J3 joins location 1 and location 2.
  - J4 joins location 2 and location 3.
  - J5 joins location 2 and location 3.
  - J6 joins location 2 and location 3.
  - J7 joins location 1 and location 2.
  - J8 joins location 2 and location 3.
- The baud rate and communications mode are correctly set at the terminal; terminal is ON LINE.

3.3 POWER UP/RESET

1. Apply power to the EVM and the data terminal.
2. Activate the RESET switch. This activates the EVMBUG monitor.
3. Press the "A" key on the terminal. EVMBUG measures the time of the start bit and determines the baud rate. To account for different terminals used, a carriage return time of 200 ms is provided for all baud rates at or slower than 1200 baud.

4. EVMBUG prints the EVMBUG banner message:

   EVMBUG R1.n (n represents version number)

   MON ?

   This is a request to input a command to the EVMBUG scanner. Commands are explained in detail in Section 5. The instruction set for the TMS 9995 EVM assembler is defined in Section 6.

   NOTE: If control is lost during operation, return control to the EVMBUG monitor by repeating steps 2 and 3.

3.4 SAMPLE PROGRAMS

The following sample programs can be used immediately to test the EVM.

3.4.1 Sample Program 1

This sample program uses the EVMBUG commands Inspect Memory (IM), Inspect Registers (IR), and execute (EX).

1. Enter the IM command with a hex memory address of >ED00.

2. Enter the following values into memory. After typing each value, press the space bar. Pressing the space bar opens and displays the next memory location.

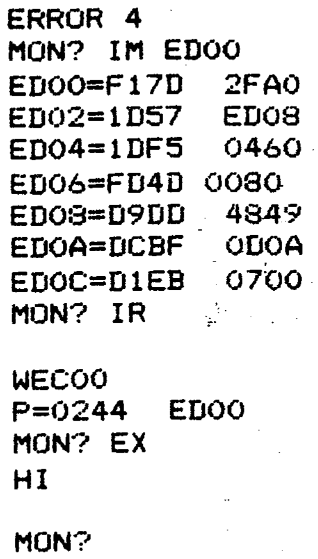
Location	Enter Value	Assembly Language	Comments
ED00	2FA0	XOP @ ED08,14	PRINT MESSAGE
ED02	ED08		
ED04	0460	B @ 0080	GO TO EVMBUG
ED06	0142		
ED08	4849	TEXT ´HI´	MESSAGE
ED0A	0D0A	DATA 0D0A	CR/LF
ED0C	0700	DATA 0700	BELL/END

   Enter a carriage return to escape the IM command. As a result, the monitor will display a question mark.

3. Use the IR command to set the program counter (PC) to the value ED00. The user must first space through the workspace pointer (WP) before the PC is displayed.

4. Use the EX command to execute the program.

5. The message "HI" will print on the printer, followed by a carriage return, line feed, and a bell. Your terminal printout should resemble the following:

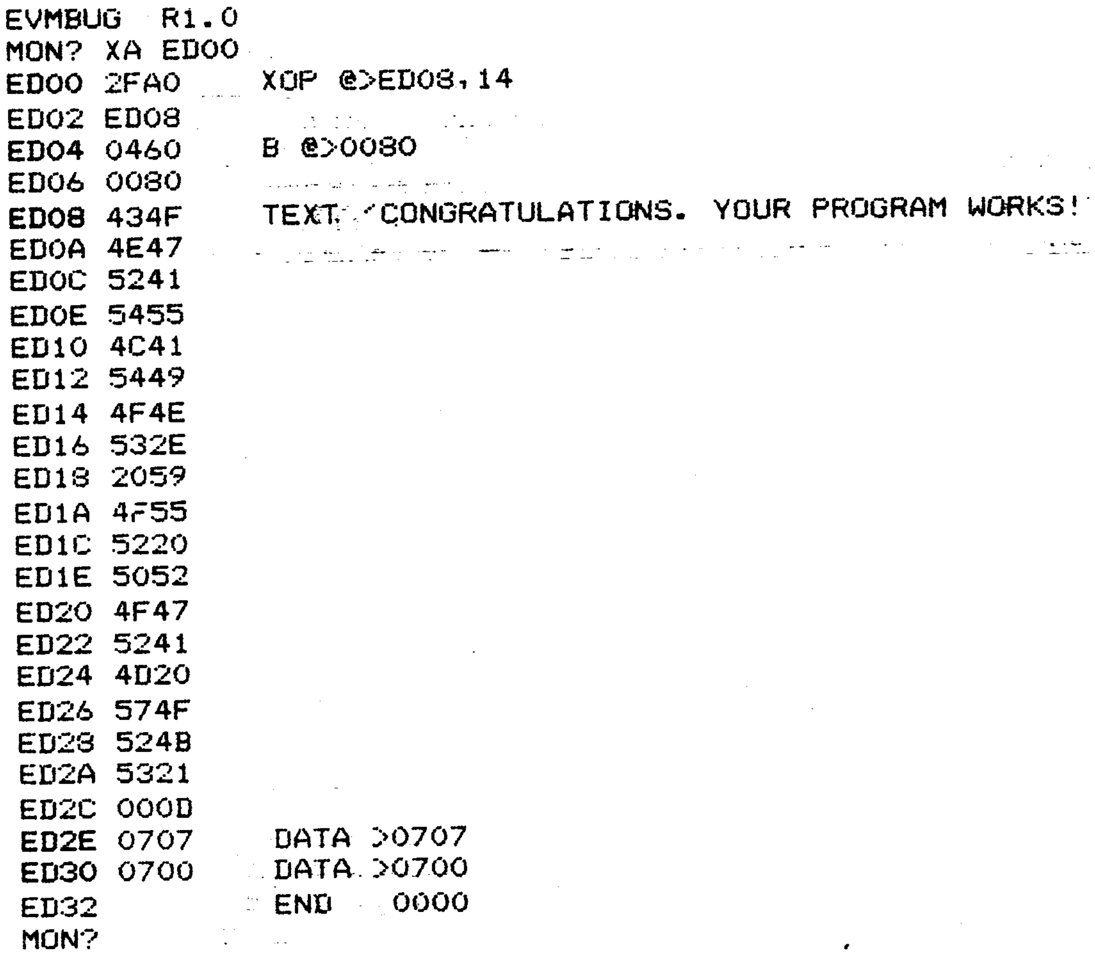


```
ERROR 4
MON? IM ED00
ED00=F17D  2FA0
ED02=1D57  ED08
ED04=1DF5  0460
ED06=FD4D 0080
ED08=D9DD  4849
ED0A=DCBF  0D0A
ED0C=D1EB  0700
MON? IR

WEC00
P=0244  ED00
MON? EX
HI

MON?
```

6. Control will then be returned to the monitor. You can re-execute your program by repeating steps 3 and 4.

3.4.2 Sample Program 2

Using steps 1 to 5 above, enter and execute the following program which has been assembled by the optional TM 990/402 line-by-line assembler.



```
EVMBUG  R1.0
MON? XA ED00
ED00 2FA0     XOP @>ED08,14
ED02 ED08
ED04 0460     B @>0080
ED06 0080
ED08 434F     TEXT 'CONGRATULATIONS. YOUR PROGRAM WORKS!'
ED0A 4E47
ED0C 5241
ED0E 5455
ED10 4C41
ED12 5449
ED14 4F4E
ED16 532E
ED18 2059
ED1A 4F55
ED1C 5220
ED1E 5052
ED20 4F47
ED22 5241
ED24 4D20
ED26 574F
ED28 524B
ED2A 5321
ED2C 000D
ED2E 0707      DATA >0707
ED30 0700      DATA >0700
ED32           END   0000
MON?
```

You can re-execute your program by repeating steps 3 and 4 above.

3.5 TROUBLESHOOTING TECHNIQUES

The following paragraphs outline suggested procedures for troubleshooting a malfunctioning TMS 9995 EVM module.

3.5.1 Test Equipment Requirements

In order to perform the necessary procedures, the user must have access to the following test equipment:

- Oscilloscope, preferably dual-trace, triggered sweep
- 10X oscilloscope probes
- VOM meter

Additional equipment which the user may find helpful includes:

- Logic Probe
- Logic Analyzer

It is suggested that the user review the theory of operation of the EVM, Section 4, before proceeding with the troubleshooting procedures.

3.5.2 Procedures

Visual checkand static check procedures are described in the paragraphs that follow.

3.5.2.1 Visual Checks

Probably the greatest source of board problems is shorts between signals caused by foreign objects and/or solder bridges between adjacent solder joints. Inspect both sides of the board carefully and remove any shorts observed. Also, brush both sides of the board with a soft dry brush (such as a drafting brush)to sweep away any loose objects which were missed in the visual inspection.

Check the jumper connections; make sure all ICs are seated properly.

3.5.2.2 Static Checks

With power applied to the board, measure the three primary supply voltages and compare the measured values to the operational limits as listed in Table 3-1. A convenient place to access those voltages is at the left edge of the prototype area.

TABLE 3-1. SUPPLY VOLTAGE OPERATIONAL LIMITS.

SUPPLY	LIMITS MIN	 MAX	CHECK AT	CURRENTS
+5V	4.5	5.5	+5 line	2A
+12V	11.64	12.36	+12 line	.25A
-12V	-11.64	-12.36	-12 line	.18A

SECTION 4

THEORY OF OPERATION

4.1 GENERAL

This section presents the theory of operation of the TMS 9995 Evaluation Module. Information from the following manuals may be used to supplement material in this section:

- TMS 9995 Microcomputer Data Manual (MP021)
- TMS 9900 Family System Design Handbook (LCC4400)
- TMS 9902 Asynchronous Communications Controller Data Manual (MP004)
- TTL Data Book, Second Edition (LCC4112)
- TTL Data Book, Second Edition Supplement (LCC4162)
- Bipolar Microcomputer Components Data Book (LCC4270)
- The MOS Memory Data Book (LCC4782)

Figure 4-1 shows the major function blocks of the TMS 9995 EVM. Included are the processing, memory and I/O portions of the system, along with the primary signal buses.

Major features of the TMS 9995 EVM are EPROM and RAM memories, two TMS 9902 EIA serial communication ports, and a prototyping area. These features are discussed in the following paragraph.

The TMS 9995 microcomputer is the central processing unit (CPU) of the EVM. The capabilities of the CPU include:

- Memory, CRU and general bus control
- Instruction acquisition, interpretation, and execution
- Timing of most control signals and data
- General system initialization

A detailed description of the TMS 9995, its signals, buses, and their operation is given in Appendix E and also in the TMS 9995 Microcomputer Data Manual. Also covered in the appendix and manual are details of the TMS 9995 on-chip RAM, Decrementer (timer/event counter), flag register, and interrupt controller.

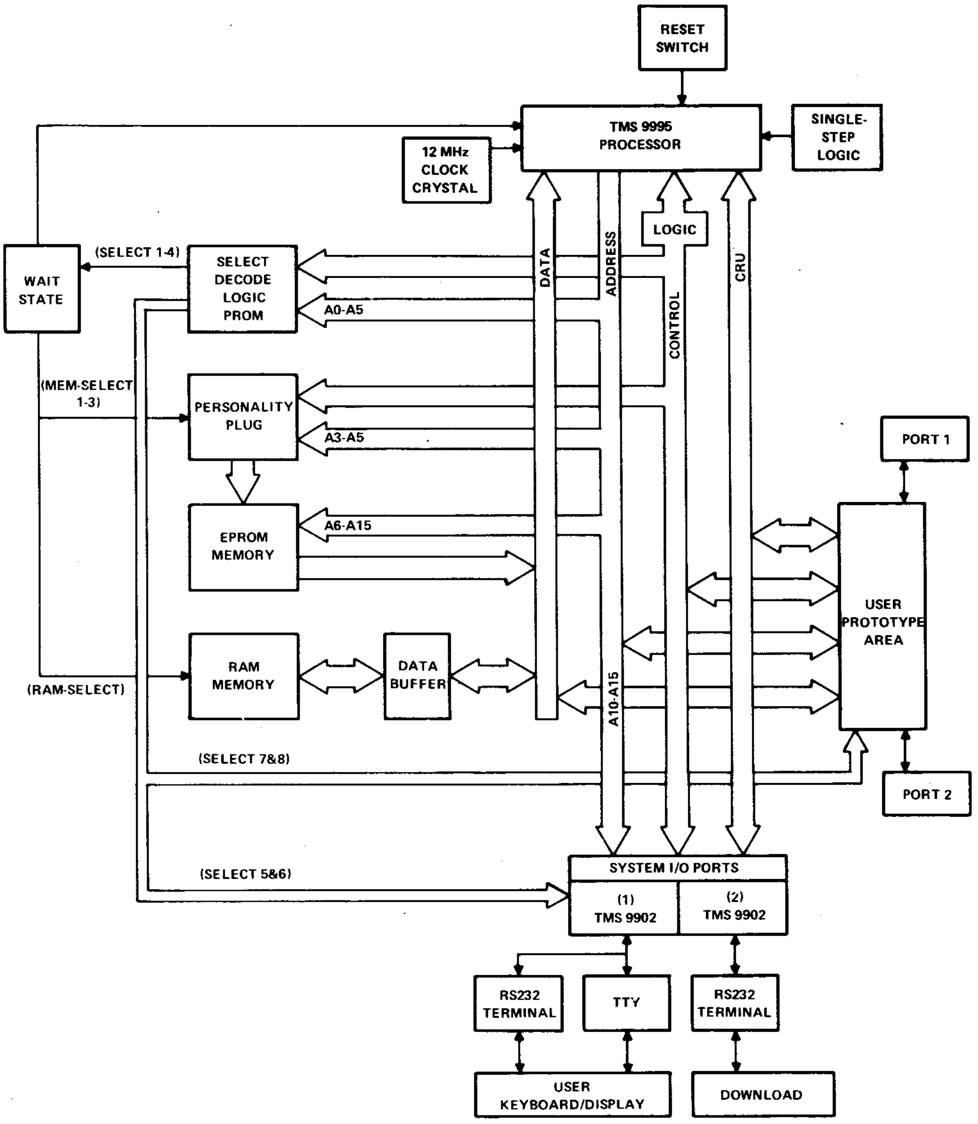


FIGURE 4-1. TMS 9995 EVM SYSTEM BLOCK DIAGRAM.

4.2 MAJOR INTERNAL SIGNALS

The signals used by TMS 9995 EVM logic are listed in Table 4-1. All system lines can also be traced by referring to the schematics in Appendix D.

TABLE 4-1. TMS 9995 EVM SIGNALS.

SIGNAL	FUNCTIONAL DEVICE CONNECTION
(Address Bus)	
A0-A2	Address Decode ROM
A3-A5	Address Decode ROM, all EPROM personality plugs
A6-A9	RAM, EPROMs
A10-A14	9902s, RAM, EPROMs
A15/CRUOUT	A15 only (in address mode): RAM, EPROMs
(Data Bus)	
D0-D2	All memory devices, external instruction decode logic (D0 = MSB)
D3-D7	All memory devices (D7 = LSB)
(CRU Bus)	
CRUIN	CRU input line, TMS 9902s
A15/CRUOUT	CRUOUT only (in CRU mode): CRU output line, TMS 9902s
CRUCLK	CRU clock, TMS 9902s
(Control Bus)	
MEMEN-	Address decode logic
DBIN-	RAM output buffer, personality plugs
WE-	Personality plugs, RAM input buffer, RAM
READY	Wait state logic, processor, Reset logic (if jumpered)
(Auxiliary Controls)	
INT1-	Processor, prototyping area
INT4-/EC-	Processor, prototyping area
HOLD-	Processor, prototyping area
IAQ	Processor, prototyping area
HOLDA-	Processor, prototyping area

Most of the signals are inputs to or outputs from the TMS 9995 microcomputer. (See Figure 4-2) Timing and other information concerning the signals are given in Appendix E and also in the TMS 9995 Microcomputer Data Manual.

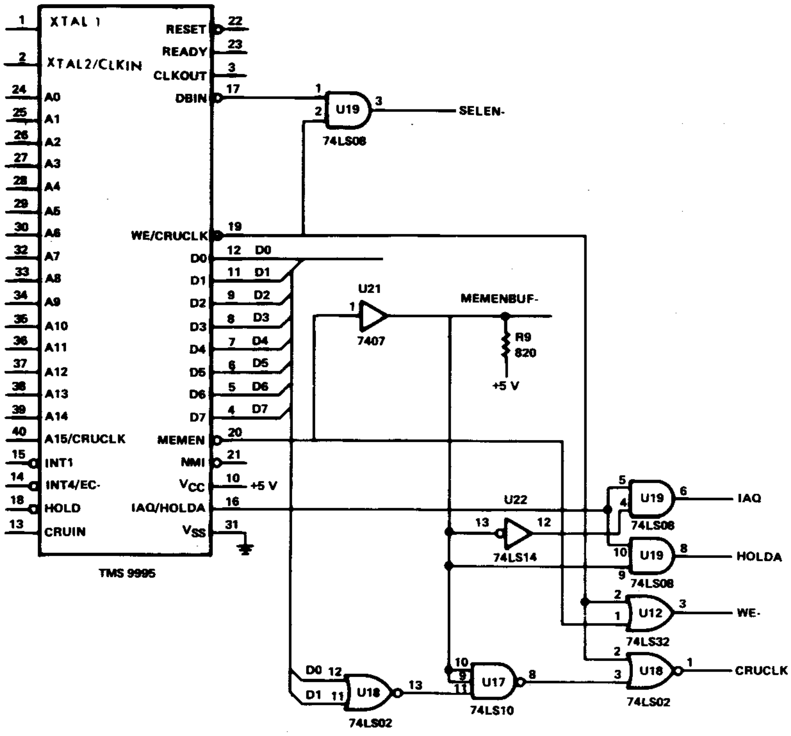


FIGURE 4-2. TMS 9995 CONTROL SIGNALS.

4.2.1 System Buses

The four major buses are subdivided by function in Table 4-1. The bus lines can also be traced by referring to the schematics in Appendix D.

4.2.1.1 Address Bus

The 16-line address bus consists of lines A0 through A15/CRUOUT. A0 through A14 are normally used for addressing memory. On-board, the address lines are routed to the address decoding PROM which selects onboard memory if the address presented lies within the limits of the memory map programmed into the PROM.

4.2.1.2 Data Bus

The data bus consists of eight bidirectional lines which are routed to and from the TMS 9995, the general-purpose memory sockets, the RAM sockets, and the prototype area. D0 is the most significant bit, and D7 is the least significant bit.

4.2.1.3 CRU Bus

The three lines in the CRU bus are CRUIN, CRUCLK, and A15/CRUOUT. Also used by CRU devices are address lines A0 to A14, logic zero on data bus lines D0, D1 and D2, and MEMEN-.

The TMS 9995 performs a CRU operation by putting the CRU address on A0 through A14, logic zero on each of D0-D2, logic one on MEMEN-, and either strobing in the addressed bit on CRUIN or by supplying the data bit on A15/CRUOUT and a pulse on WE-/CRUCLK-. (Note that CRUCLK is obtained by gating WE-/CRUCLK- with MEMEN-.)

4.2.1. Control Bus

A brief explanation of the functions of each control bus signal is given in Table 4-2.

TABLE 4-2. TMS 9995 CONTROL BUS SIGNALS.

SIGNAL	ACTIVE STATE	GROUP	PURPOSE
MEMEN-	Low & High	Memory/CRU	Indicates address on address bus is for memory (MEMEN-=0) or CRU (MEMEN-=1). Also used to demultiplex WE-/CRUCLK- and IAQ/HOLDA.
DBIN-	Low	Memory	Shows state of TMS 9995 data bus: low is input to 9995; high is output.
WE-	Low	Memory	Strobe to memory devices for writing data to memory. WE- is obtained by gating WE-/CRUCLK- with MEMEN-.
READY	High	Memory/CRU	Tells 9995 to finish memory, CRU, or external instruction cycle. Wait states are generated by pulling the line low.

NOTE: SEE APPENDIX E FOR DETAILS OF THE ABOVE OPERATIONS.

4.2.2 Auxiliary Control Signals

A brief explanation of the function of each auxiliary control signal is given in Table 4-3.

TABLE 4-3. TMS 9995 EVM AUXILIARY CONTROL SIGNALS.

SIGNAL	ACTIVE STATE	GROUP	PURPOSE
INT1-	Low	Interrupt	User defined: requests interrupt of 9995.
INT4-/EC-	Low	Interrupt	User defined: requests interrupt of 9995.
HOLD-	Low	Processor Activity	Requests 9995 to give up control of address and data buses, WE-/CRUCLK- and DBIN-.
IAQ	High	Processor Activity	Signifies this memory cycle to be an instruction fetch (MEMEM = 0).
HOLDA	High	Processor Activity	Acknowledges that 9995 has given up control of address and data buses, WE-/CRUCLK-, and DBIN- (MEMEN = 1).

4.3 CLOCK OSCILLATOR

The TMS 9995 EVM utilizes the on-chip clock oscillator of the TMS 9995 to generate the system clock signal CLKOUT. A 3 MHz CLKOUT clock is generated using the 12 MHz fundamental frequency crystal connected to the TMS 9995. This CLKOUT frequency is the machine state frequency of the TMS 9995.

4.4 RESET LOGIC

RESET initializes the EVM system and causes the following to occur:

- Clears I/O devices

- Clears single-step logic

- Inhibits memory-write and CRU operations until RESET is released

- Sets TMS 9995 Status Register interrupt mask to 0000 (Binary)
- Gets RESET interrupt vector for the TMS 9995, which activates th EVMBUG monitor.
- Decides if Auto First Wait State generation will be used or not (See paragraph 4.9)

RESET is caused by:

- Power-up
- Activating the RESET switch on the EVM

The RESET logic is shown in Figure 4-3.

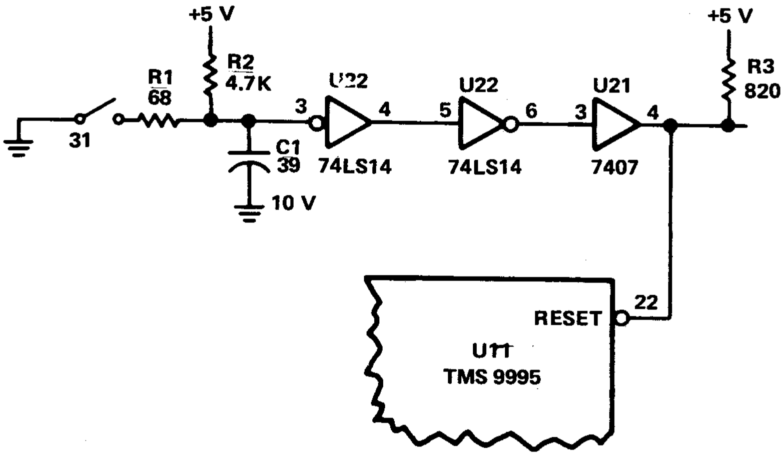


FIGURE 4-3. TMS 9995 EVM RESET LOGIC.

4.5 DEVICE SELECT LOGIC

Decoding of addresses to generate select signals for the on-board memory and CRU devices is accomplished with two 74S188 32x8-bit PROMs, as shown in Figure 4-4. Table 4-4 lists the TMS 9995 addresses assigned to each select line.

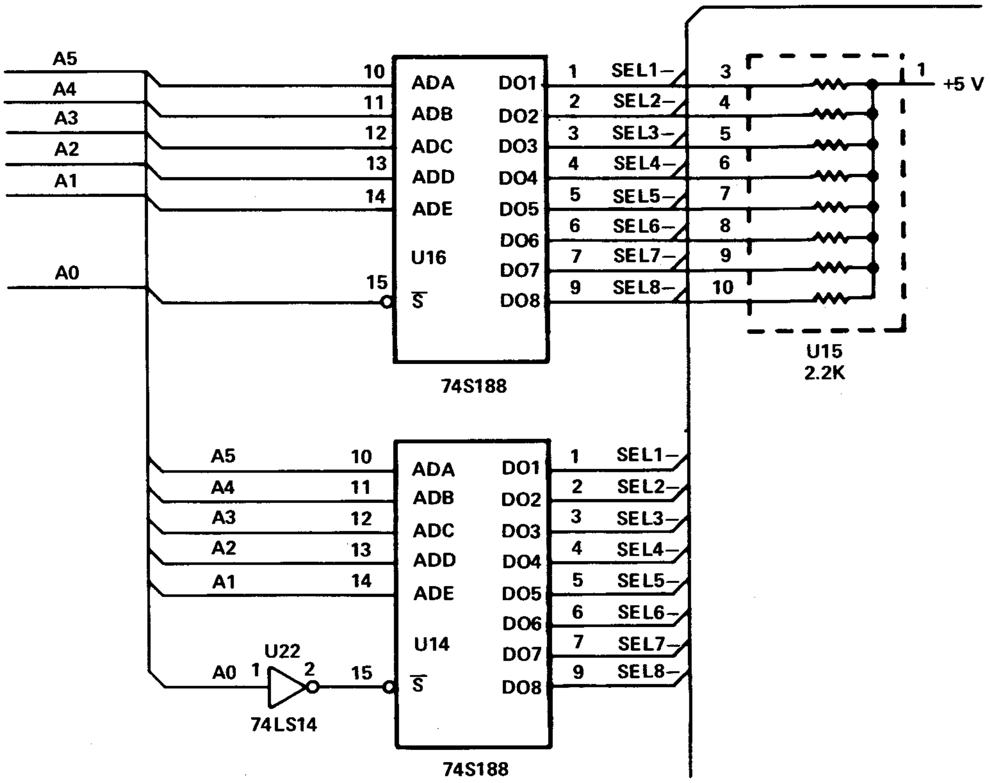


FIGURE 4-4. DEVICE SELECT LOGIC.

TABLE 4-4. SELECT LINE ADDRESS ASSIGNMENTS.

TMS 9995 ADDRESS ASSIGNMENT	PROM BIT PATTERN	SELECT LINE
0000-03FF	ED	SEL5/SEL2
0400-07FF	DD	SEL6/SEL2
0800-0FFF	FD	SEL2
1000-17FF	FB	SEL3
1800-37FF	F7	SEL4
3800-7FFF	BF	SEL7
8000-EBFF	7F	SEL8
EC00-EFFF	FE	SEL1
F000-FFFF	FF	--

4.6 MEMORY

The TMS 9995 EVM has three general-purpose memory sockets that can be used for most "by 8" memory devices, i.e., 8K to 64K ROMs/EPROMs, "by 8" RAMs and bipolar PROMs. It also has two sockets for 1Kx4 RAMs. Memory devices supplied by TI are configured according to the memory map shown in Figure 4-5.

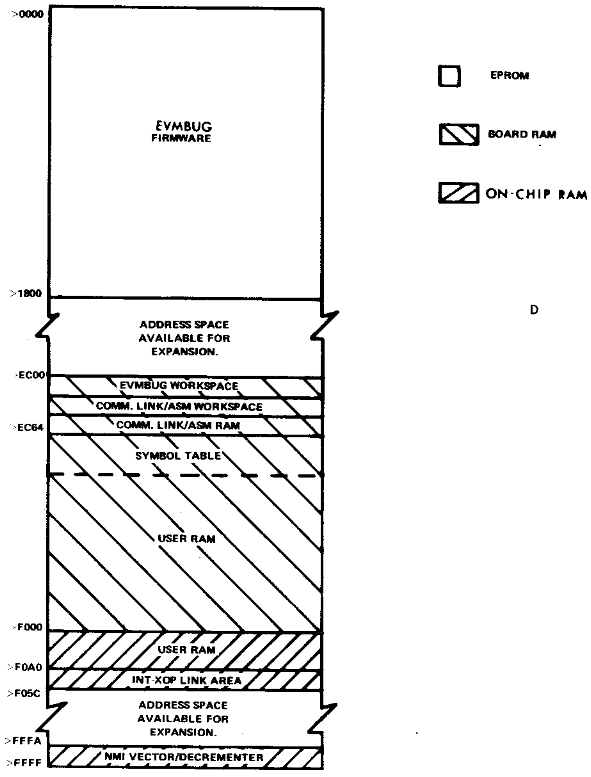


FIGURE 4-5. TMS 9995 EVM SYSTEM MEMORY MAP.

4.6.1 General-Purpose Memory Sockets.

The general-purpose memory sockets (U8, U9, U10) are able to utilize the many "by 8" memory devices through the personality plugs (U3, U4, U5) and jumpers J1 - J6. The logic associated with the general-purpose sockets is shown in Figures 4-6 and 4-7.

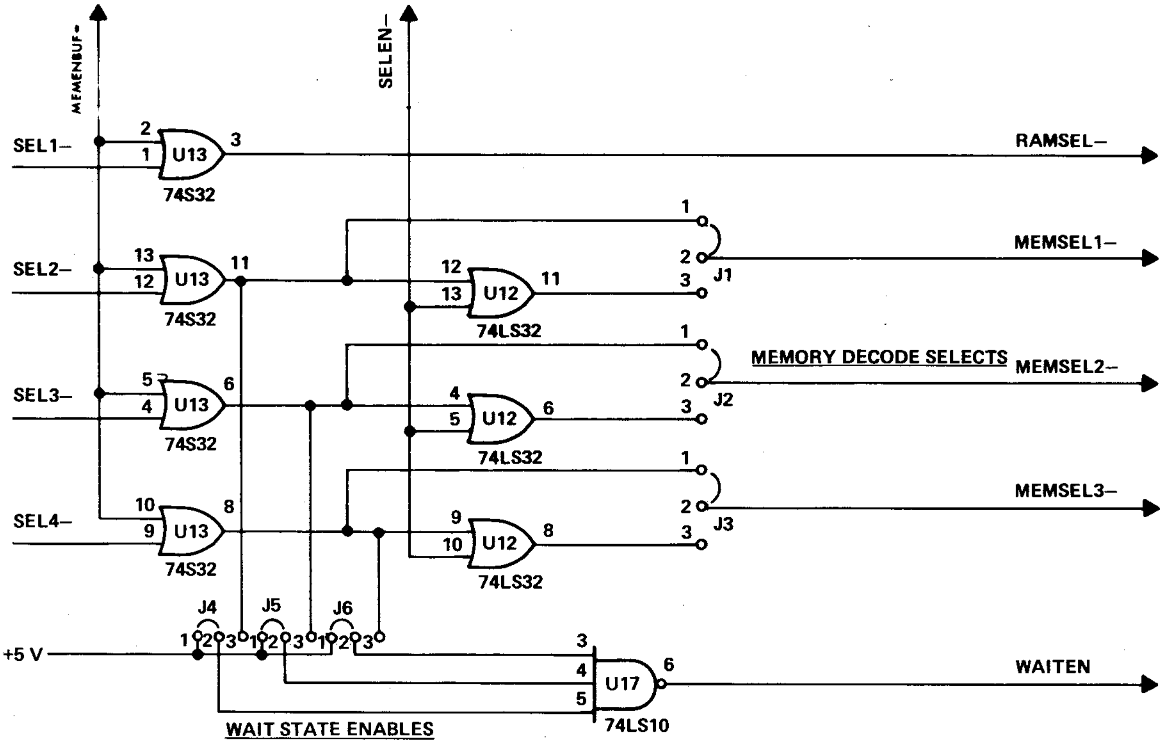


FIGURE 4-6. GENERAL-PURPOSE SOCKET JUMPER LOGIC.

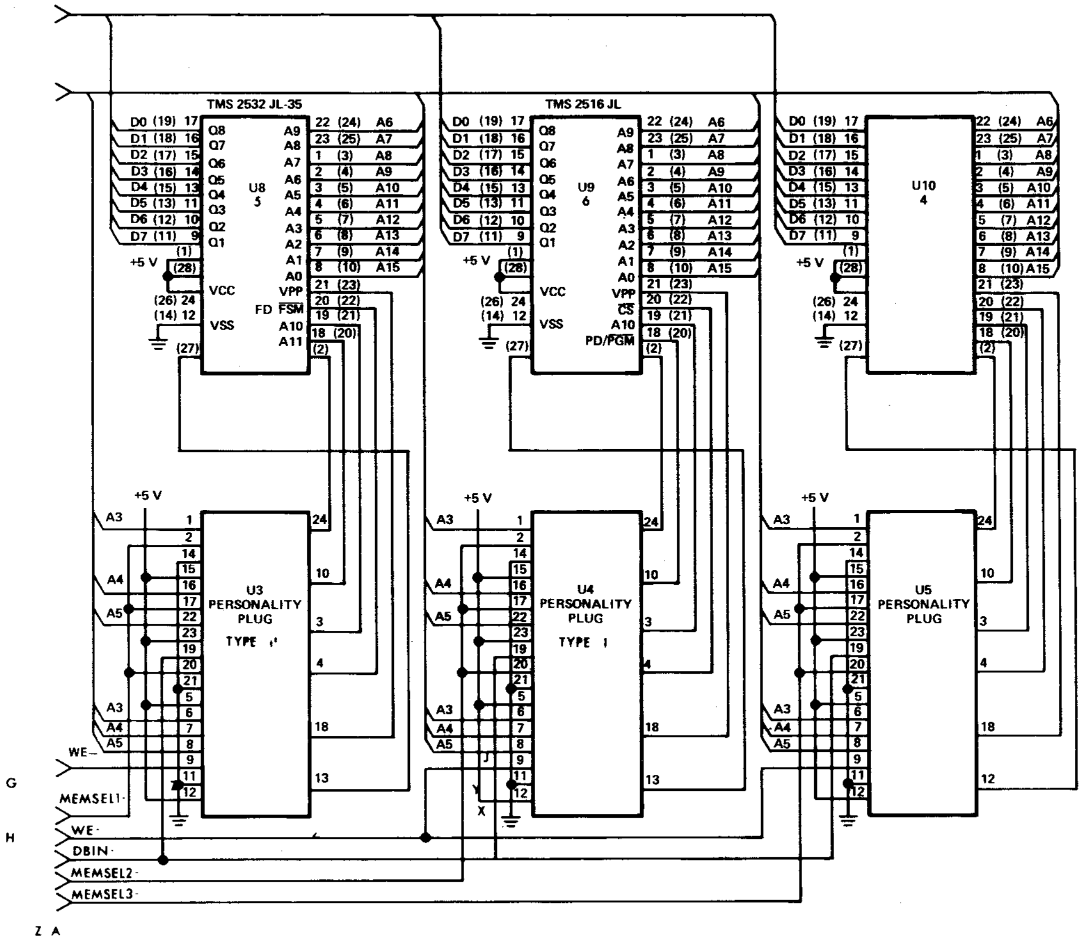


FIGURE 4-7. GENERAL-PURPOSE SOCKETS AND PERSONALITY PLUGS.

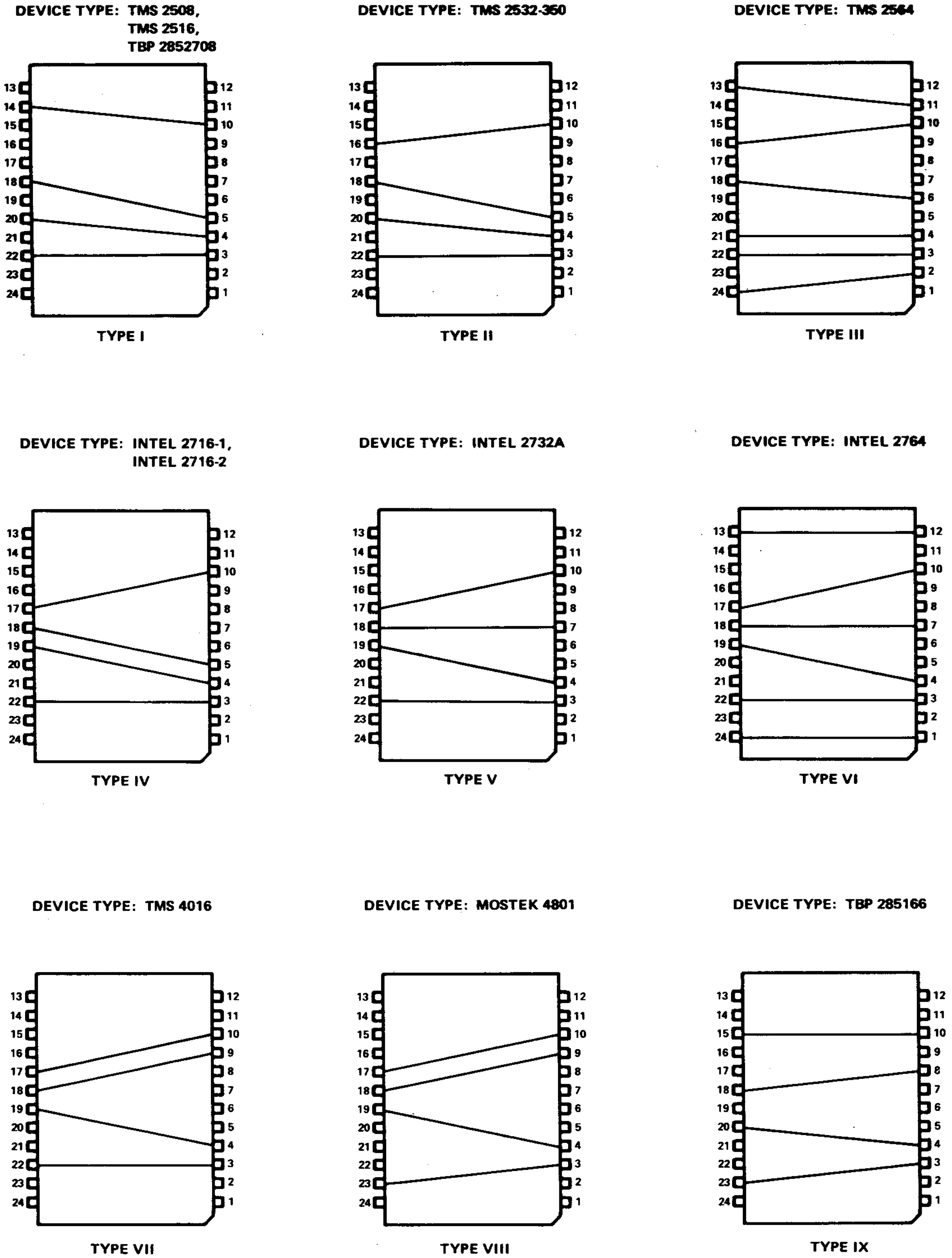
Each general-purpose memory socket has one personality plug and two jumper plugs associated with it (e.g., general-purpose socket U8: personality plug U3, and jumpers J1 and J4). The personality plugs route the appropriate signals to the memory device used.

Jumpers J1, J2 and J3 determine if the MEMSEL signal is to be generated directly from a SEL signal gated with MEMENBUF-, or if the MEMSEL signal is to be first gated with SELEN-. (Since MEMSEL is used to generate the chip select for the memory device, certain RAMs may require the additional timing information provided by SELEN- to avoid data bus conflicts.)

Jumpers J4, J5 and J6 provide for either one Wait state or no Wait states. See paragraph 4.9.

4.6.2 Personality Plugs

The wiring of the personality plugs for most of the more popular "by 8" memory devices is shown in Figure 4-8. Table 4-5 indicates the jumper connections for these devices.



DEVICE TYPE: TMS 2508, TMS 2516, TBP 2852708

TYPE I

DEVICE TYPE: TMS 2532-350

TYPE II

DEVICE TYPE: TMS 2564

TYPE III

DEVICE TYPE: INTEL 2716-1, INTEL 2716-2

TYPE IV

DEVICE TYPE: INTEL 2732A

TYPE V

DEVICE TYPE: INTEL 2764

TYPE VI

DEVICE TYPE: TMS 4016

TYPE VII

DEVICE TYPE: MOSTEK 4801

TYPE VIII

DEVICE TYPE: TBP 285166

TYPE IX

FIGURE 4-8. PERSONALITY PLUGS.

TABLE 4-5. JUMPER CONNECTIONS.

DEVICE PART NUMBER	MEMORY TYPE	JUMPER J1,2,3 CONNECTION	JUMPER J4,5,6 CONNECTION	PERSONALITY PLUG
TMS 2508	EPROM	1-2	2-3	TYPE I
TMS 2516	EPROM	1-2	2-3	TYPE I
TBP 28S2708	PROM	1-2	1-2	TYPE I
TMS 2532-35	EPROM	2-3	2-3	TYPE II
TMS 2564	EPROM	1-2	2-3	TYPE III
INTEL 2716-1&2	EPROM	1-2	2-3	TYPE IV
INTEL 2732A	EPROM	1-2	2-3	TYPE V
INTEL 2764	EPROM	1-2	2-3	TYPE VI
TMS 4016	RAM	2-3	2-3	TYPE VII
MOSTEK 4801	RAM	2-3	1-2	TYPE VIII
TBP 28S166	PROM	1-2	1-2	TYPE IX

4.6.3 Dedicated Read/Write Memory (RAM) Sockets

The dedicated RAM sockets provide 1K bytes of fast, no wait state RAM. The RAM consists of two 1Kx4 devices in U6 (MS Nybble) and U7 (LS Nybble). The dedicated RAM logic is shown in Figure 4-9. RAMSEL- signal generation is shown in Figure 4-6.

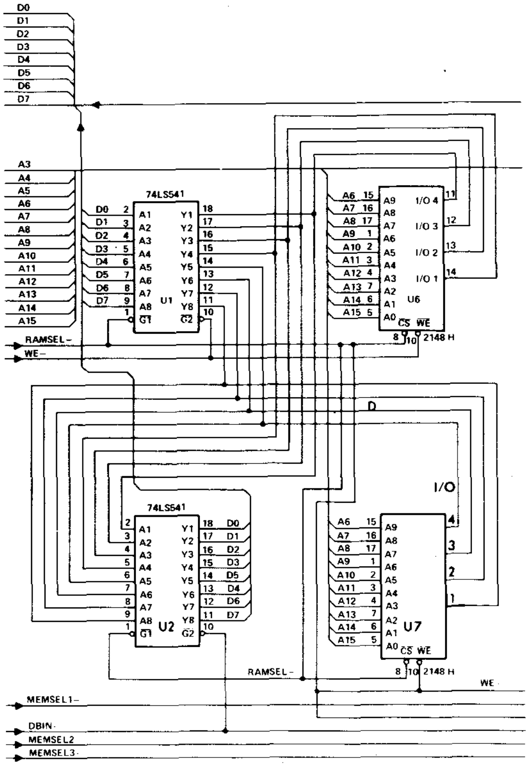


FIGURE 4-9. DEDICATED RAM LOGIC.

I/O to the RAM is buffered at Ul and U2 (either by two 74LS540s or by tw 74LS541s) in such a manner that when RAMSEL and WE- are present at the buffer, data from the data bus is passed to the RAM through Ul (input). When RAMSEL and DBIN- are present, data is passed to the data bus through U2 (output).

Note that DBIN- will be asserted while MEMEN- is low during a read cycle. In the same manner, WE- will also be asserted while MEMEN- is low. A chip select will not occur during a write cycle until after WE- drops. This is to prevent fast RAMs (which sample WE- as soon as they are selected) from sampling WE- before it goes low during a write cycle.

4.7 SERIAL COMMUNICATION PORTS

Two serial communication ports are provided on the TMS 9995 EVM. Both of these ports will support EIA RS232 communication, and one of them (Port 1) can also optionally support TTY communication.

The logic for the two ports is shown in Figure 4-10. Selection of one of the two TMS 9902 Asynchronous Communications Controller CRU devices is by SEL5- or SEL6- (See Figure 4-4.). The CRU address map of the TMS 9995 EVM is shown in Table 4-6.

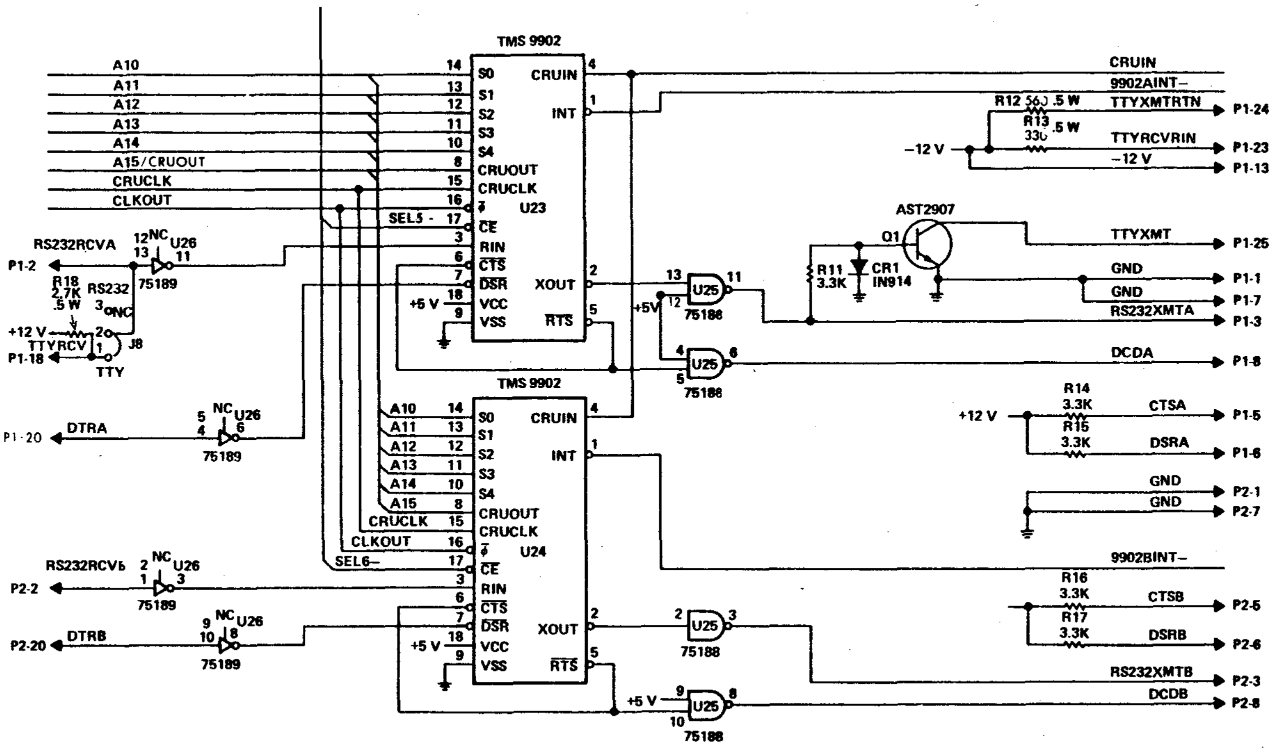


FIGURE 4-10. SERIAL COMMUNICATIONS PORTS.

TMS 9995 EVM CRU MAP

CRU ADDRESS (HEX)	FUNCTION	INPUT	OUTPUT
0000	SERIAL I/O	RBR0	DATA00
0002	PORT A	RBR1	DATA01
0004	(TMS 9902)	RBR2	DATA02
0006		RBR3	DATA03
0008		RBR4	DATA04
000A		RBR5	DATA05
000C		RBR6	DATA06
000E		RBR7	DATA07
0010		0	DATA08
0012		RCVERR	DATA09
0014		RPER	DATA10
0016		ROVER	LXDR
0018		RFER	LRDR
001A		RFDB	LDIR
001C		RSBD	LDDATA
001E		RIN	TSTMD
0020		RBINT	RTSON
0022		XBINT	BRKON
0026		0	RIENB
0028		TIMINT	XBIENB
002A		DSCINT	TIMENB
002C		RBRL	DSCENB
002E		XBRE	NOT USED
0030		XSRE	
0032		TIMERR	
0034		TIMELP	
0036		RTS	
0038		DTR	
003A		CTS	
003C		DSCH	
003E		FLAG	NOT USED
003F	PORT A	INT	RESET
0400	SERIAL I/O	RBR0	DATA00
0402	PORT B	RBR1	DATA01
0404	(TMS 9902)	RBR2	DATA02
0406		RBR3	DATA03
0408		RBR4	DATA04
040A		RBR5	DATA05
040C		RBR6	DATA06
040E		RBR7	DATA07

TABLE 4-6. CRU ADDRESS MAP (Page 1 of 2).

TMS 9995 EVM CRU MAP (Continued)

CRU ADDRESS (HEX)	FUNCTION	INPUT	OUTPUT
0410		0	DATA08
0412		RCVERR	DATA09
0414		RPER	DATA10
0416		ROVER	LXDR
0418		RFER	LRDR
041A		RFDB	LDIR
041C		RSBD	LDDATA
041E		RIN	TSTMD
0420	SERIAL I/O	RBINT	RTSON
0422	PORT B	XBINT	BRKON
0424	(TMS 9902)	0	RIENB
0426		TIMINT	XBIENB
0428		DSCINT	TIMENB
042A		RBRL	DSCENB
042C		XBRE	NOT USED
042E		XSRE	
0430		TIMERR	
0432		TIMELP	
0434		RTS	
0436		DTR	
0438		CTS	
043A		DSCH	
043C		FLAG	NOT USED
043E	PORT B	INT	RESET
1EE0	FLAG	FLAG0	FLAG0
1EE2	REGISTER	FLAG1	FLAG1
1EE4	(CRU INPUT	FLAG2	FLAG2
1EE6	AND OUTPUT)	FLAG3	FLAG3
1EE8		FLAG4	FLAG4
1EEA		FLAG5	FLAG5
1EEC		FLAG6	FLAG6
1EEE		FLAG7	FLAG7
1EF0		FLAG8	FLAG8
1EF2		FLAG9	FLAG9
1EF4		FLAGA	FLAGA
1EF6		FLAGB	FLAGB
1EF8		FLAGC	FLAGC
1EFA		FLAGD	FLAGD
1EFC		FLAGE	FLAGE
1EFE		FLAGF	FLAGF
1FDA	MID FLAG	MID FLG	MID FLG

TABLE 4-6. CRU ADDRESS MAP (Page 2 of 2).

4.7.1 EIA INTERFACE

Both serial communication ports are capable of supporting EIA communications. The two EIA links utilize one 75188 line driver and one 75189 line receiver. In addition to handling receive-data and transmit-data signals, each TMS 9902 inputs the Data-Terminal-Ready (DTR) signal from its respective connector. Also, each port provides a Data Carrier-Detect (DCD) signal for the connector terminal via the Request-To-Send (RTS) and Clear-To-Send (CTS) signal outputs of each TMS 9902.

4.7.2 TTY Interface

Port 1 has the additional circuitry to enable it to support TTY communication. A transistor and 560-ohm resistor form the transmit loop for the 20-mA current loop, TTY interface. The transistor conducts current while the line driver connected to its base is at a mark state. As the line driver goes to the space state, the positive voltage output is clamped to ground through the signal diode on the transistor base, thereby turning off the transistor and current loop. See Figure 4-10.

The receive circuit consists of a line receiver which monitors the receive loop formed by the TTY transmit circuitry and the two supply resistors. The values of these resistors is such that during a mark state, the input to the line receiver is held very close to -12 volts. When the TTY transmit circuitry cuts the loop, the receiver input is pulled up to +12 volts.

NOTE: the TTY Jumper J8 must be plugged so that the line receiver can monitor the loop voltage. Plug one and two for TTY; plug two and three for EIA. DO NOT connect an EIA terminal when Jumper 8 is plugged for TTY.

4.8 MEMORY AND CRU ADDRESS MAP CHANGES.

The memory and/or CRU address map can be changed by the user by substituting user-programmed PROMs for the TI-supplied 74S188s in the address select decoder sockets (U14 and U16). Unprogrammed 74S188 PROMs are available from your Texas Instruments distributor.

CAUTION

When planning a memory or CRU map, or when using any device in the prototyping area (such as a 2148 or 2114), the devices on the 9995 EVM must not overlap in address space either with each other or with devices in the prototyping area. On-board devices MUST be mapped into unique locations, and no other prototyping area devices may respond to addresses intended for an originally provided on-board device.

4.9 WAIT STATE LOGIC

The TMS 9995 microcomputer can generate Wait states for off-chip memory cycles, off-chip CRU cycles, and external instruction cycles. The TMS 9995 also has an Automatic First Wait State Generation feature. (See Appendix E or the TMS 9995 Microcomputer Data Manual for detailed information concerning Wait states)

The TMS 9995 EVM has logic to optionally generate a single Wait state only for memory cycles. The Wait state can be inserted into all off-chip memory cycles by invoking the Automatic First Wait State Generation feature i.e., Jumper J7 connected between posts E1 and E2. Optionally, the Wait state can be inserted into a memory cycle to any of the general-purpose memory sockets (See paragraph 4.6.1). The Wait state logic of the EVM is shown in Figure 4-11.

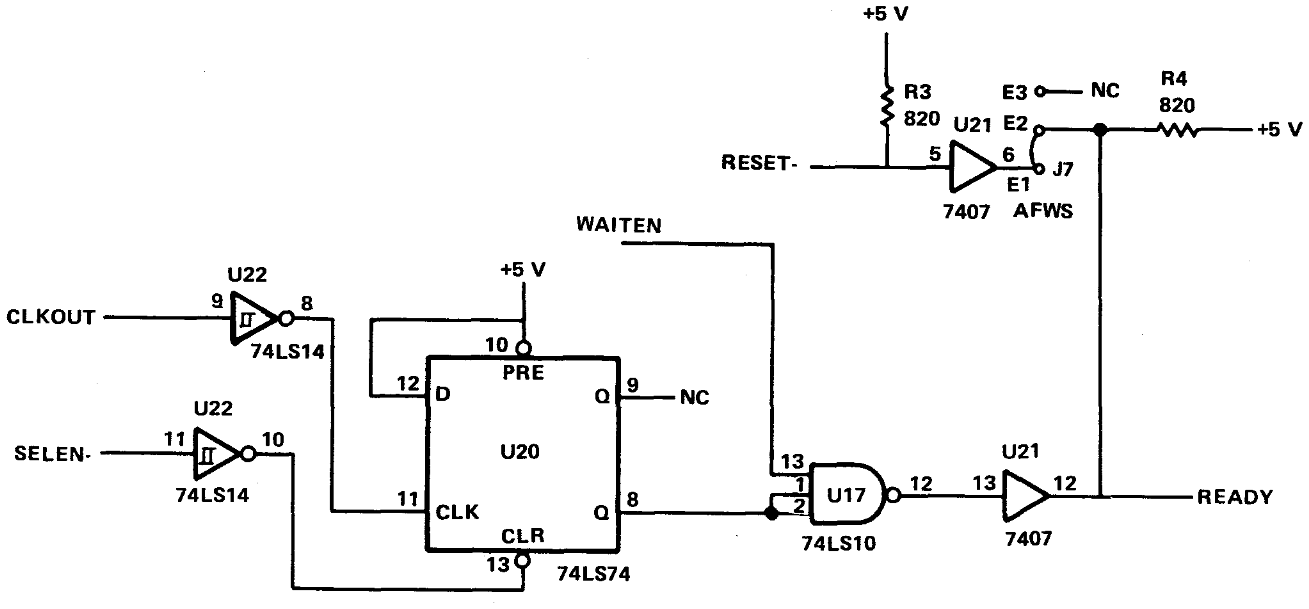


FIGURE 4-11. WAIT STATE LOGIC.

4.10 EXTERNAL INSTRUCTION LOGIC

The external instructions are those which, when executed by the TMS 9995, cause a code to be output on D0-D2 and WE-/CRUCLK- to become active. The external instructions and a description of their operation on the EVM are listed in Table 4-7. The external instruction logic is illustrated in Figure 4-12.

TABLE 4-7. EXTERNAL INSTRUCTIONS.

INSTRUCTION	OPCODE	D0	D1	D2	DESCRIPTION
IDLE	0340	0	1	0	Suspend processor until an interrupt occurs. Lights the Idle LED.
RSET	0360	0	1	1	Zeroes TMS 9995 interrupt mask, generates pulse for user-defined logic.
CKON	03A0	1	0	1	Generates pulse for user-defined logic.
CKOF	03C0	1	1	0	Generates pulse for user-defined logic.
LREX	03E0	1	1	1	Causes NMI- (single-step function).

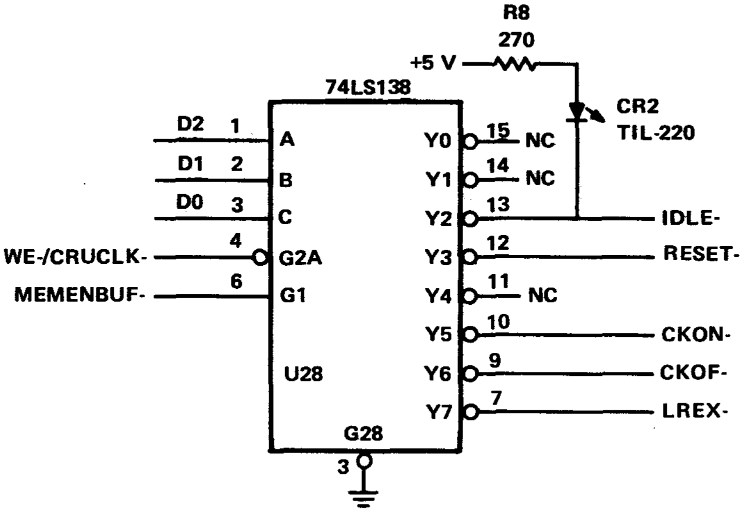


FIGURE 4-12. EXTERNAL INSTRUCTION LOGIC.

IDLE causes the TMS 9995 to suspend operation. It is, in essence, a HALT instruction. A RESET, NMI, or other interrupt terminates the idle state. When in an idle state, the Idle LED is lit.

The LREX instruction is used by the single-step capability of EVMBUG. See paragraph 4.11.

4.11 SINGLE-STEP LOGIC

The EVMBUG monitor utilizes the LREX external instruction in conjunction with the logic shown in Figure 4-13 to perform single-stepping. LREX causes a non-maskable interrupt (NMI) to be presented to the TMS 9995 after two Instruction Acquisition or IDLE pulses. This means that the NMI interrupt occurs after two instructions are executed following the LREX. EVMBUG uses this to perform single step by executing an LREX, followed by an RTWP to exit the monitor and return to the user instructions. After one user instruction is executed, the NMI interrupt is active. NMI then traps back to the monitor.

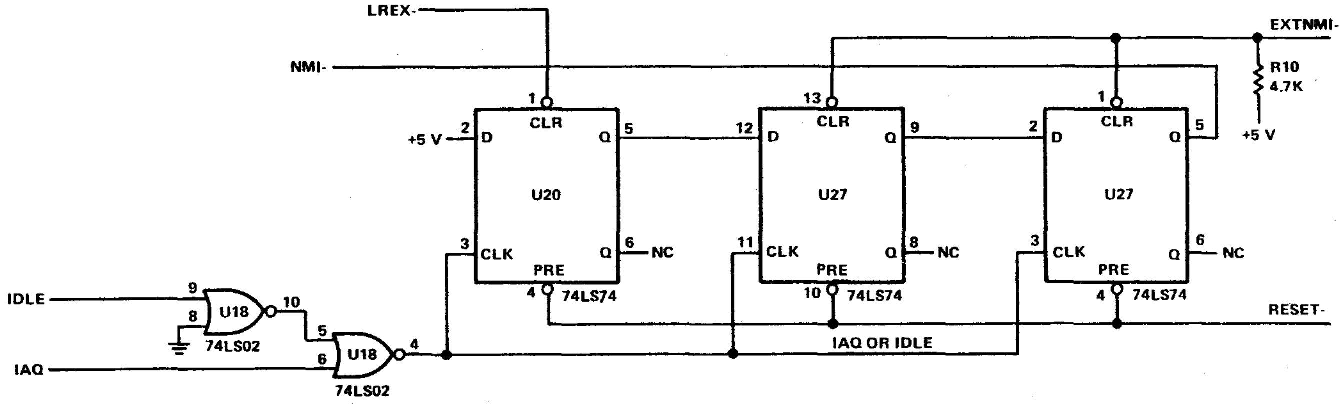


FIGURE 4-13. SINGLE-STEP LOGIC.

4.12 PROTOTYPE AREA

Capabilities of the TMS 9995 EVM may be expanded by means of the prototype area, which provides room for breadboarding of TMS 9995 systems. Most of the signals previously discussed are provided at the edge of the prototyping area for this purpose.

Two plugs, P4 and P5, located at the right side of the prototype area on either side of the power bus plug, permit the expansion of prototype capabilities off the EVM board. Off-board devices are connected to the EVM by means of a wrap-post header ribbon cable.

SECTION 5

EVMBUG INTERACTIVE DEBUG MONITOR

5.1 GENERAL

This section provides a description of the commands and subroutines available in the TMS 9995 EVM Debug Monitor (EVMBUG), including syntax conventions user-accessible utilities, and EVMBUG error messages.

EVMBUG is a debug monitor which provides an interactive interface between the user and the TMS 9995 microcomputer. It is supplied by the factory contained in one 2532-35 and one 2516 EPROM.

Initialization of the EVM Debug Monitor is described in Section 3.

5.2 USER MEMORY

The memory provided in the TMS 9995 microcomputer consists of RAM (read/write memory) and ROM (read only memory). The RAM is for user programs, while the ROM contains the monitor and assembly programs. The monitor program provides keyboard commands, I/O programs, and other user utilities.

Figure 5-1 shows the memory map for the TMS 9995. Interrupt and XOP instructions extend from >0000 to >007F. EVMBUG monitor workspaces extend from >0080 to >1800. If the assembler is used, the symbol table begins at >EC64. Four bytes are used for each label; the number of labels that are used will determine the beginning address for user RAM. As an example, if 50 labels are used, 200 bytes will be needed for for the label table. The end of the label table will be >EC64 + >C8 (>ED3C). Note that 200 = >C8. Therefore, the start of the permissible user RAM in this case would be >ED3C.

NOTE: >F0FC thru >F0FF of the address space is available for expansion.

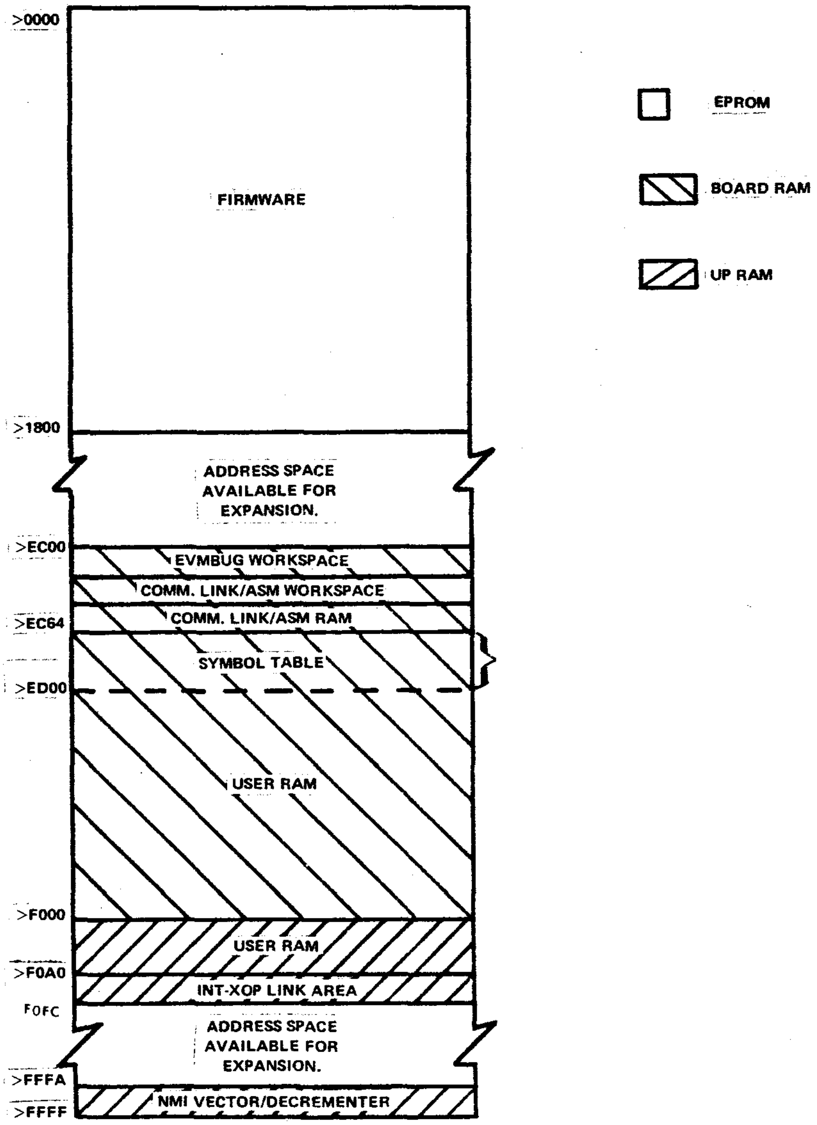


FIGURE 5-1. SYSTEM MEMORY MAP.

5.3 EVMBUG COMMANDS

The EVMBUG commands are described in subsequent paragraphs. Table 5-1 summarizes these commands. Table 5-2 presents the syntax conventions used in command definitions.

TABLE 5-1. EVMBUG COMMANDS

INPUT:	SEE SECTION NUMBER:	RESULTS:
IM	5.3.9	Inspect/Change Memory
DM	5.3.3	Dump Memory
IW	5.3.13	Inspect/Change User Workspace Registers
EX	5.3.5	Execute User Program
EX	5.3.1	Execute User Prog. To Breakpoint
SS	5.3.11	Execute Single Step
LM	5.3.8	Load Memory From Digital Cassette (ASR 733)
DM	5.3.4	Dump Memory to Digital Cassette (ASR 733)
IC	5.3.1	Inspect/Change CRU
IR	5.3.10	Inspect/Change Hardware Register (PC, WP, ST)
FD	5.3.6	Find Data In Memory (Byte/Word)
HE	5.3.7	Hex Arithmetic
TN	5.3.12	Toggle Null Flag (For ASR 733)
XA	5.3.14.2	Execute Assembler With Existing Symbol Table
XA	5.3.14.1	Execute Assembler With New Symbol Table
XR	5.3.14.3	Execute Reverse Assembler
XC	5.3.14.4	Execute Communications Link

TABLE 5-2. COMMAND SYNTAX CONVENTIONS.

CONVENTION SYMBOL	EXPLANATION
WP	Current User Workspace Pointer contents
PC	Current User Program Counter contents
ST	Current User Status Register contents
caps	Other items in capitol letters are to be entered literally
< >	Items to be supplied by the user. The term within the angle brackets is a generic term
[ ]	Optional item. May be included or omitted at the user's discretion.
{ }	One of several optional items shown inside the brackets must be chosen.
(CR)	Carriage Return
^	Space Bar
(LF)	Line Feed
RO,Rl..R15	Registers zero to fifteen

NOTE

Except where otherwise indicated, all numeric output is assumed to be hexadecimal; the last four digits input will be the value used. Thus, a mistaken numerical input can be corrected merely by making the last four digits the correct value. If fewer than four digits are input, they are right-justified.

5.3.1 Execute Under Breakpoint (EXB)

SYNTAX:

```
        EXB[{^,}<address>]<(CR)>
```

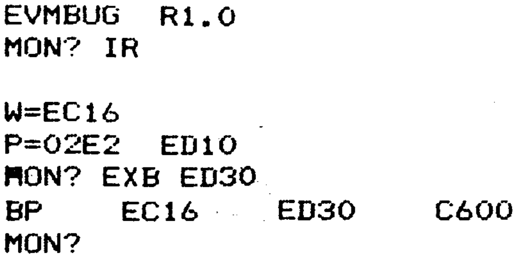
This command is used to execute instructions up to the specified stopping address. When the stopping address is reached, WP, PC, and ST register contents are displayed and control is returned to the monitor command scanner. Program execution begins at the address in the PC (set by using the IR command). Execution terminates at the address specified in the EXB command, and a banner is output showing the contents of the hardware WP, PC, and ST registers, in that order.

The address specified must be in RAM and must be the address of an instruction. The breakpoint is controlled by a software interrupt, XOP 15.

An XOP instruction takes the place of the instruction at the address specified. When this replacement is executed, the original instruction assumes its original place. If the XOP is not executed, or another EXB is specified before the XOP is executed, then the XOP will not be replaced with the original instruction or will be replaced with the wrong instruction.

If no address is specified, the EXB command defaults to an EX command, where execution continues with no halting point specified.

EXAMPLE:



```
EVMBUG  R1.0
MON? IR

W=EC16
P=02E2  ED10
MON? EXB ED30
BP    EC16    ED30    C600
MON?
```

5.3.2 Inspect/Change CRU (IC)

SYNTAX:

```
        IC{^,}<CRU address>{^,}<count><(CR)>
```

This command reads the number of bits specified by "count", beginning at the specified CRU address, and displays them, right-justified, in a

16-bit hexadecimal number. Up to 16 CRU bits may be displayed. "CRU address" is a 16-bit number stored in register twelve. (See Appendi F.)

The corresponding CRU output bits may be altered following input bit display by keying in desired hexadecimal data, right-justified.

A carriage return following data output forces a return to the command scanner. A minus sign (-) or a space reads and displays the data again.

Note well: the effective software CRU address is double the hardware CRU bit address. This is demonstrated in Fig. 5-2, in which >100 is specified in the command in order to display values beginning with CRU bit >80.

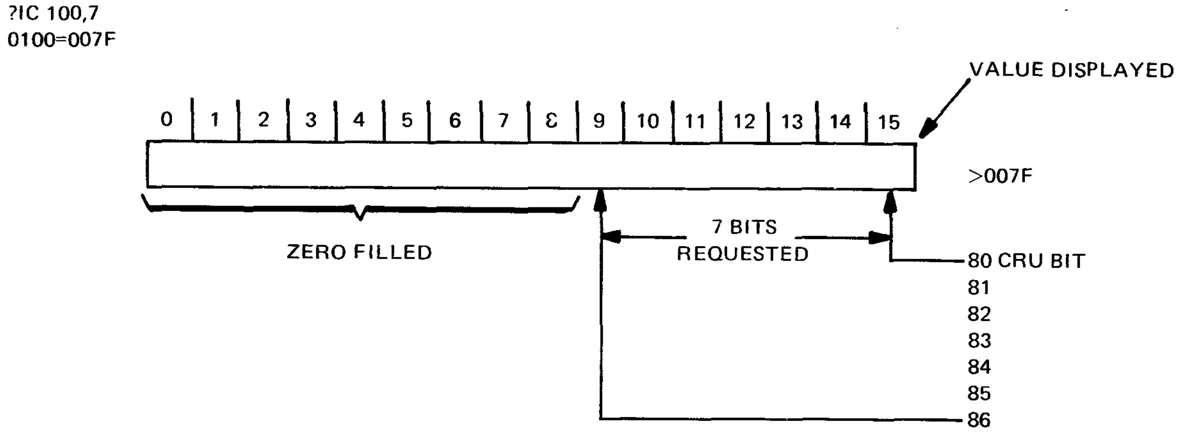

```
?IC 100,7
0100=007F
```

FIGURE 5-2. CRU BITS INSPECTED BY IC COMMAND.

EXAMPLES:

(1) Examine eight Port 2 CRU input bits. CRU address is >400.

```
EVMBUG  R1.0
MON? IC 400,8
0400=007F
MON?
```

(2) Check changes in the CRU Port 1 input buffer which result from typing commands on the terminal.

```
EVMBUG  R1.0
MON? IC 0,4
0000=000D
0000=0000  -
0000=000D
0000=0000
MON?
```

(3) Check the contents of the TMS 9995 Flag Register (Flag 0-15)

```
     EVMBUG  R1.0
MON? IC 1EE0
1EE0=7FE0
```

(4) Using the CRU, configure the TMS 9995 Decrementer as an Event Counter and start decrementing.

```
EVMBUG  R1.0
MON? IC 1EE0
1EE0=7FE0  3
1EE0=0003
MON?
```

(5) Check the contents of the MID Flag register on the TMS 9995

```
EVMBUG  R1.0
MON? IC 1FDA
1FDA=FFFE
MON?
```

5.3.3 Dump Memory (DM)

SYNTAX:

```
DM [<start address>[{^,}<stop address>]]
```

Memory is displayed, beginning and ending at the <start address> and <stop address> respectively, if specified. Each line of output begins with the address of the first memory word displayed on the line. Eight memory words follow on each line.

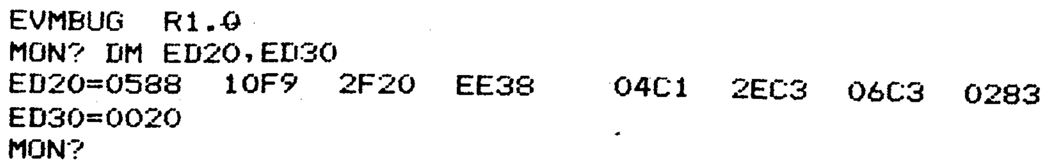
If no addresses are given, EVMBUG displays the contents of location >0000 and then returns control to EVMBUG.

If a <start address> is supplied, but no <stop address>, all memory locations from the <start address> to the end of memory will be output on the terminal before control returns to EVMBUG.

Supplying both a start and a <stop address> will cause a memory dump from the <start address> through the <stop address>.

Memory dump can be terminated at any time by typing any character on the keyboard.

EXAMPLE:



```
EVMBUG  R1.0
MON? DM ED20,ED30
ED20=0588  10F9  2F20  EE38    04C1  2EC3  06C3  0283
ED30=0020
MON?
```

5.3.4 Dump Memory To Digital Cassette/Paper Tape (DMC)

SYNTAX:
DMC{^,}<start address>{^,}<stop address>{^,}<entry address>{^,}

This command causes computer memory to be copied to digital cassette or paper tape. The memory image is stored in non-relocatable 990 object format. Object record format is explained in Appendix A. The block of memory stored begins at <start address> and ends at <stop address>. The <entry address> parameter is for use by the "LMC" command to initialize the program counter when the memory block is restored from cassette or paper tape to computer memory. Once these parameters are entered, the monitor will display the letters "IDT." The user then enters an IDT (program identifier) of up to eight characters FOLLOWED BY A SPACE OR CARRIAGE RETURN.

```
+-------- MONITOR PROMPT
|
V
IDT=<program name[ ]<(CR)>>
```

NOTE

> Termination given after IDT is a space bar or carriage return. Some other termination will cause the instruction to function incorrectly.

After the IDT prompt is answered, the monitor will display the prompt "READY Y/N". When you have readied the cassette or paper tape punch, enter "Y".

```
    +-------- MONITOR PROMPT
    |
    V
READY Y/N <Y>
```

EXAMPLE: Dump To Cassette:

The terminal is assumed to be a Texas Instruments 733 ASR or equivalent. The terminal must have automatic device control (ADC); this means that the terminal recognizes the four tape control characters DC1, DC2, DC3, and DC4.

The following procedure is carried out prior to answering the "READY Y/N" query:

(1) Load a cassette in the left (Cassette 1) transport (Figure 5-3).

(2) Place the transport in RECORD mode.

(3) Rewind the cassette.

(4) Load the cassette. If the cassette does not load, it may be write protected. The write protect hole is on the bottom right side of the cassette (Figure 5-4). Cover it with the tab provided with the cassette, then repeat Steps 1 through 4.

(5) The KEYBOARD, PLAYBACK, RECORD, and PRINTER LOCAL/OFF/LINE switches must be in the LINE position.

(6) Place the TAPE FORMAT switch in the LINE position.

(7) Answer the "READY Y/N" query with a ´Y´; the "Y" will echo.

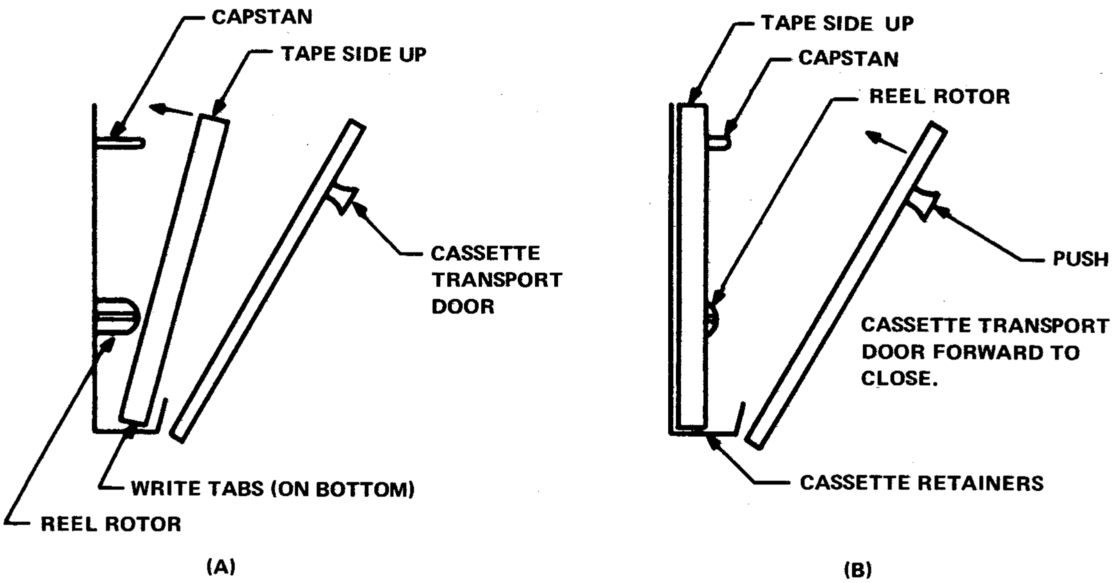


FIGURE 5-3. LOAD TAPE CASSETTE.

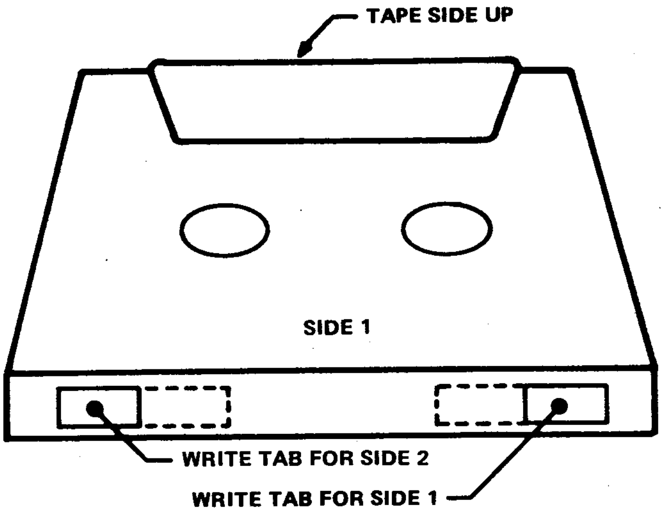


FIGURE 5-4 TAPE WRITE PROTECT TABS.

EXAMPLE: Dump To Paper Tape:

The terminal is assumed to be an ASR 33 teletypewriter. The following steps should be completed carefully to avoid punching stray characters:

(1) Enter the command:

```
DMC<start address>{^,}<stop address>{^,}<entry address>
  {^,}IDT=<name>{^,}READY Y/N<Y>
```

Do not answer the "READY Y/N" query yet.

(2) Change the teletype mode from ON LINE to LOCAL.

(3) Turn on the paper tape punch and press the RUBOUT the several times, placing rubouts at the beginning of key tape for correct-reading/program loading.

(4) Turn off the paper tape punch, and reset the teletype mode to LINE. (This is necessary to prevent punching

stray characters.)

(5) Turn on the punch and answer the "READY Y/N" query with ´Y´. The Y will not be echoed.

(6) Punching will begin. Each file is followed by sixty rubout characters. When these characters appear (identified the constant punching of all holes), the punch must be turned off.

5.3.5 Execute Command (EX)

SYNTAX:

```
        EX(CR)
```

The EX command causes task execution to begin at current values in the Workspace Pointer and Program Counter.

5.3.6 Find Data Command (FD)

SYNTAX:

```
        FD{^,}<start address>{^,}<stop address>{^,}<value>
```

The contents of memory locations from <start address> to <stop address> are compared to <value>. The memory addresses whose contents equal "value" are printed out.

If the termination character of <value> is a minus sign, the search will print the addresses of all bytes from <start address> to <stop address> whose contents are the rightmost byte in <value>. If the termination character is a carriage return (CR), then the search will print the addresses of all words from <start address> to <stop address> whose contents are <value>.

EXAMPLE:

```
MON? FD E00,EF0 400
0E40
0E74
```

```
     OR      MON? FD E00,EFO 4-
             0E01
             0E12
             0E30
             0E32
             0E40
             0E5A
             0E5C
             0E74
             0EBC
             0EC6
             0ED2
             0ED6
             0EEC
             MON?
```

5.3.7 Hexadecimal Arithmetic (HEX)

SYNTAX:

```
        HEX{^,}<number 1>{^,}<number 2><(CR)>
```

The sum and difference of two hexadecimal numbers are output.

EXAMPLE:

```
    EVMBUG  R1.0
    MON? HEX 200,100
    H1+H2=0300 H1-H2=0100
    MON?
```

5.3.8 Load Memory From Cassette Or Paper Tape (LMC)

SYNTAX:

```
        LMC{^,}<bias><(CR)>
```

Data in 990 object record format (defined in Appendix A) is loaded from paper tape or cassette into memory. <Bias> is the relocation bias (starting address in RAM). Its default is >0. Object code saved using the DMC command, however, is invariably restored using the relocation bias <starting address> specified for that command. Both relocatable and absolute data may be loaded into memory with the LMC command. After data is loaded, the module identifier (See Tag 0 in Appendix A) is printed on the next line.

LOAD FROM CASSETTE: (ASR 733):

The 733 ASR must be equipped with Automatic Device Control (ADC). The following procedure is carried out prior to executing the LMC command:

(1) Insert the cassette in one of the two transports on 733 ASR.

(2) Place the transport in the PLAYBACK mode.

(3) Rewind the cassette.

(4) Load the cassette.

(5) Set the KEYBOARD, PLAYBACK, RECORD, and PRINTER LOCAL/LINE switches to LINE.

(6) Set the TAPE FORMAT switch to LINE. Loading will be at 1200 baud.

(7) Execute the LMC command: LMC{^,}<(CR)>

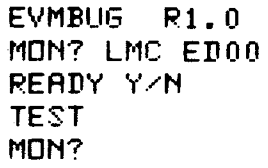
LOAD FROM PAPER TAPE: (733 teletype)

Prior to executing the LMC command, place the paper tape in the Reader and position the tape so the reader mechanism is in the Null field ahead of the file to be loaded. Enter the load command. If the 733ASR has ADC, the reader will begin to read from the tape. If the 733ASR does not have ADC, turn on the reader, and loading will begin.

Each file is terminated with 60 rubouts. When the Reader reaches this area of the tape, turn it off. The loader will then pass control to the command scanner.

The User Program Counter (P) is loaded with the entry address if a 1-tag or a 2-tag is found on the tape.

EXAMPLE:



```
EVMBUG  R1.0
MON? LMC ED00
READY Y/N
TEST
MON?
```

5.3.9 Inspect/Change Memory (IM)

SYNTAX;

```
IM{^,}<start address><(CR)>
```

Memory Inspect/Change "opens" a memory location, displays it, and gives the option of changing the data in the location. The Inspect/Change memory address directs a display of memory contents from the <start address> each time the space bar is pressed. Each line of output consis of the address of the data word followed by the data word itself. A termination character causes the following:

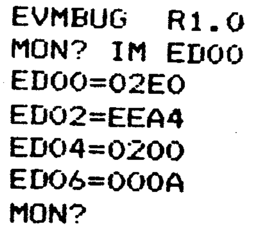
(1) If a carriage return, control is returned to the command scanner.

(2) If a space, the next memory location is opened and and displayed.

(3) If a minus sign (-), the previous location is opened and displayed.

If a hexadecimal value is entered before the termination character, the displayed memory location is updated to the value entered.

EXAMPLES:



```
EVMBUG  R1.0
MON? IM ED00
ED00=02E0
ED02=EEA4
ED04=0200
ED06=000A
MON?
```

5.3.10 Inspect/Change User WP, PC, and ST (Hardware) Registers: (IR)

SYNTAX:

```
IR<(CR)>
```

The user Workspace Pointer (WP), Program Counter (PC), and Status Register (ST) are inspected and changed with the IR command. The output letters WP, PC, and ST identify the values of the three principal hardware registers passed to the TMS 9995 microcomputer when an EXB, EX, or SS command is entered. WP points to the workspace register area, PC points to the next instruction to be executed, and ST is the Status Register contents.

The termination character causes the following:

- A carriage return causes control to return to the command scanner.

- A space causes the next register to be opened.

Order of display is: WP, PC, ST.

EXAMPLES:

(1)

```
EVMBUG  R1.0
MON? IR

W=EC16  100
P=02E2  D00
MON?
```

(2)

```
EVMBUG  R1.0
MON? IR

W=EC16
P=02E2
S=D600
MON?
```

5.3.11 Execute In Single Step Mode: (SS)

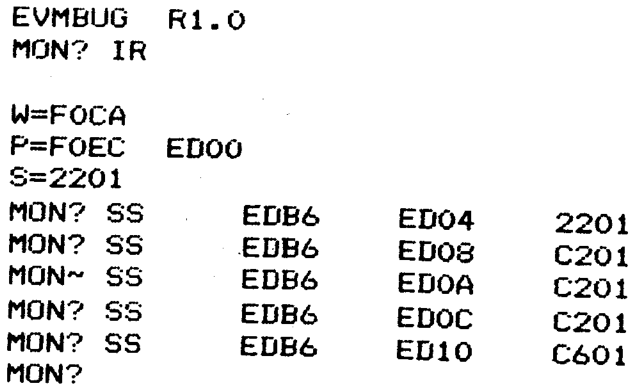
SYNTAX:

```
SS<(CR)>
```

This command executes one instruction, then returns control to the monitor.

Each time the SS command is entered, a single instruction is executed at the address in the Program Counter, then the contents of the Program Counter, Workspace Pointer, and Status Regiser (after execution) are printed out. Successive instructions can be executed by repeated SS commands.

EXAMPLE:



```
EVMBUG  R1.0
MON? IR

W=FOCA
P=FOEC  EDOO
S=2201
MON? SS       EDB6     ED04     2201
MON? SS       EDB6     ED08     C201
MON~ SS       EDB6     EDOA     C201
MON? SS       EDB6     EDOC     C201
MON? SS       EDB6     ED10     C601
MON?
```

NOTE

Incorrect results are obtained when the SS instruction causes execution of an XOP instruction in a user program. (SEE Appendix E.) To avoid these problems, the EXB command should be used to execute any XOP´s in a process, instead of the SS command.

5.3.12 Toggle Null Flag: (TNF)

SYNTAX:

```
TNF
```

The TNF command is used to alert EVMBUG that the terminal being used is a 1200 baud terminal which is not a Texas Instruments´ 733 ASR (e.g., a 1200 baud CRT). To revoke the TNF command, enter it again.

USE:

TNF is used only when operating with a true 1200 baud peripheral device. TNF is NEVER used when operating at other baud rates.

In EVMBUG, the baud rate is set by measuring the width of the character ´A´ input from a terminal. When an ´A´ of 1200 baud width is measured, EVMBUG is set up to automatically insert three nulls for every character output to the terminal. These nulls are inserted to allow correct operation of the TMS9995 with Texas Instruments´ 733ASR data terminals.

5.3.13 Inspect/Change User Workspace Registers: (IWR)

SYNTAX:

```
IWR{^,}[<register number>]<(CR)>
```

The IWR command is used to display the contents of all workspace registers or to display one register at a time, while allowing the user to change the register contents. The workspace begins at the address in the workspace pointer.

The IWR command, followed by a carriage return, causes the contents of the entire workspace to be printed. Control is then passed to the command scanner.

The IWR command followed by a register number in hexadecimal and a carriage return, causes display of the specified register´s contents. The user may then enter a new value into the register by entering a hexadecimal value. The following are valid termination characters, whether or not a new value is entered:

- A space causes display of the next register.

- A minus sign causes display of the previous register.

- A carriage return gives control to the command scanner.

EXAMPLES:

(1)

```
EVMBUG  R1.0
MON? IWR

R0=0000  R1=0000  R2=0AF5  R3=0000  R4=4AE9  R5=0A00  R6=0006  R7=EC0E
R8=0001  R9=0142  RA=4AE9  RB=04AC  RC=0000  RD=EC00  RE=0E7A  RF=9000
MON?
```

(2)

```
EVMBUG  R1.0
MON? IWR 2
R2=EC6  3456
R3=02E2  100
R4=CA01
R5=EC38  800F
R6=02A3  0
MON?
```

5.3.14 Assembler Commands: (XA, XAE, XRA, XCL)

5.3.14.1 Execute Assembler With New Symbol Table: (XA)

SYNTAX:

```
XA{^,}<assembly address><(CR)>
```

The XA command clears the existing symbol table and allows the user to establish a new symbol table.

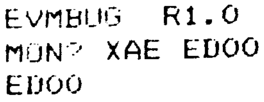
```
MON? XA ED00
ED00
```

5.3.14.2 Execute Assembler With Existing Symobl Table: (XA)

SYNTAX;

```
XAE{^,}<assembly address><(CR)>
```

The XAE command assembles using the existing symbol table.



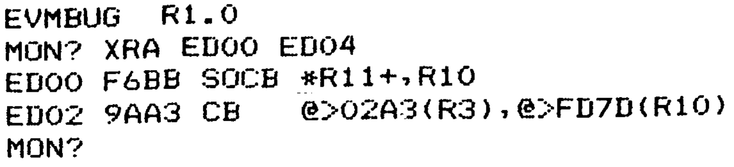
```
EVMBUG  R1.0
MON? XAE ED00
ED00
```

5.3.14.3 Execute Reverse Assembler: (XRA)

SYNTAX:

```
XRA{^,}<start address>{^,}<end address><(CR)>
```

This command allows the EVM user to inspect any memory location and see the menemonic representation of its contents. The program effectively recreates a source listing from the object code stored in memory by printing the memory address, memory data, instruction mnemonic, and operands.



```
EVMBUG  R1.0
MON? XRA ED00 ED04
ED00 F6BB SOCB *R11+,R10
ED02 9AA3 CB   @>02A3(R3),@>FD7D(R10)
MON?
```

5.3.14.4 Execute Communications Link: (XCL)

SYNTAX:

```
XCL<(CR)>
```

5.4 USER-ACCESSIBLE UTILITIES

EVMBUG contains seven utility subroutines that perform I/O functions as listed in Table 5-3, below. These subroutines are called through the XOP assembly language instruction. This instruction is covered in detail in Appendix F. Locations for XOPs 8 through 14 contain vectors for utilities that drive the TMS 9995 terminal. XOP 15 is used by the monitor for the breakpoint facility.

TABLE 5-3. USER-ACCESSIBLE UTILITIES

XOP	FUNCTION
8	Write One Hexadecimal Character
9	Read Hexadecimal Word From Terminal
10	Write 4 Hexadecimal Characters To Terminal
11	Echo Character
12	Write 1 Character To Terminal
13	Read 1 Character From Terminal
14	Write Message To Terminal

(All characters are in ASCII Code.)

NOTES

(1) Initially, EVMBUG will conduct I/O through the TMS 9902 connected to Connector P1. In this mode, >0000 is in EVMBUG´s R12, located at memory address >EC2E. To change this configuration, change the contents of >EC2E before executing the I/O XOP. For example, to use the the auxiliary TMS 9902 at P2, change the contents of location >EC2E to >0400. CRU programming is discussed in paragraph 8.5 of Section 8.

(2) The write character XOP (XOP 12) activates the REQUEST TO SEND signal of the TMS 9902. This signal is never deactivated by EVMBUG, so that modems may be used.

(3) Most of the XOP format examples herein use a register for the source address; however, all XOP´s cab also use a symbolic memory address or any of the addressing forms available for the XOP instruction.

5.4.1 Write One Hexadecimal Character to Terminal (XOP8)

FORMAT:

```
        XOP     Rn,8
```

The least significant four bits of user register Rn are converted to their ASCII coded hexadecimal equivalent (0 to F) and output on the terminal. Control returns to the instruction following the extended operation.

EXAMPLE:

Assume user register 5 contains >203C. The assembly language (A.L.) and machine language (M.L.) are shown below:

```
A.L.  XOP      R5.8          SEND 4 LSB'S OF R5 TO TERMINAL
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
M.L.	0	0	1	0	1	1	1	0	0	0	0	0	0	1	0	1	>2E05

Terminal Output: C

5.4.2 Read Hexadecimal Word From Terminal (XOP9)

```
FORMAT:
    XOP          Rn,9
    DATA         NULL              Address of continued execu-
                                     tion, if NULL is entered.
    DATA         ERROR             Address of continued execu-
                                     tion, if non-hex number
                                     is entered.
    (NEXT INSTRUCTION)             Execution continued here,
                                     invalid hex number and
                                     terminator are entered.
```

Binary representation of the last four hexadecimal digits input from the terminal is accumulated in user register Rn. (More than four digits may be input, but only the last four are used.) The termination character is returned in register Rn+1. Valid termination characters are: space, minus, comma, and a carriage return. Return to the calling task as follows:

- If a valid termination character is the only output, return is to the memory address contained in the next word following the XOP instruction (NULL, above).

- If a non-hex character or an invalid termination charact is input, control returns to the memory address contained in the second word following the XOP instruction (ERROR, above).

- If a hex string followed by a valid termination characte is input, control returns to the word following the DATA ERROR statement above.

EXAMPLE:

If the valid hexadecimal character string 12C is input from the terminal, followed by a carriage return, control returns to memory address >FFB6, with register 6 containing >012C, and register 7 containing >0D.

If the hex character string 12C is input from the terminal, followed by an ASCII plus (+) sign, control returns to location >FFC6. Registers 6 and 7 are returned to the calling program without being altered. "+" is an invalid termination character.

If the only input form the terminal is a carriage return, register 6 is returned unaltered, while register 7 contains >0D00. Control is returned to address >FFC0.

5.4.3 Write Four Hexadecimal Characters To Terminal (XOP10)

FORMAT:

```
        XOP     Rn, 10
```

The four digit hexadecimal representation of the contents of user register Rn is output to the terminal. Control returns to the instruction following the XOP call.

EXAMPLE:

Assume register 1 contains >2C46

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
M.L.	0	0	1	0	1	1	1	0	1	0	0	0	0	0	0	1	>2E81

Terminal Output: 2C46

5.4.4 Echo Character (XOP 11)

FORMAT:

```
        XOP     Rn,11
```

This is a combination of XOPs 13 (READ character) and 12(WRITE character). A character in ASCII code is read from the terminal, placed in the left byte of Rn, then echoed back to the terminal.

Control returns to the instruction following the XOP after a character is read and written. By using a code to determine a character string termination, a series of characters can be echoed and stored at a particular address:

```
        CLR       R2              Clear R2
        LI        R1,>FE00        Set Storage Address
   LOOP XOP       R2,11           Echo, Using R2
        CI        R2,>0D00        Was Character a CR?
        JEQ       EXIT            Yes, Exit Routine
        MOVB      R2,*R1+         No, Move Character to Stg
        JMP       LOOP            Repeat XOP
   EXIT NOP
```

5.4.5 Write One Character To Terminal (XOP12)

```
     FORMAT:
          XOP     Rn,12
```

The ASCII character in the left byte of user register Rn is output to the terminal. The right byte of Rn is ignored. Control is returned to the instruction following the call.

5.4.6 Read One Character From Terminal (XOP 13)

```
     FORMAT:
          XOP     Rn,13
```

The ASCII representation of the character input from the terminal is placed in the left byte of user register Rn. The right byte of register Rn is zeroed. When this utility is called, control is returned to the first instruction following the call only after a character is input.

5.4.7 Write Message To Terminal (XOP14)

```
     FORMAT:
          XOP     @MESSAGE,14
```

MESSAGE is the symbolic address of the first character of the ASCII character string to be output. The string must be terminated with a byte containing binary zeroes. After the character string is output, control is returned to the first instruction following the call.

Assuming the following program:

```
MEMORY
ADDRESS            OP CODE              A.L. MNIMONIC
 (HEX)


 ED00               2FA0                XOP>ED00,14
 ED02               EDE0
 EDO4
 .
 .
 .
 ED00               5445                 TEXT ´TEST´
 EDE2               5354
 EDE4               00                   BYTE 0
```

During the execution of this XOP, the character string "TEST" is output on the terminal, and control is then returned to the instruction at location >ED04. TEXT is an assembler directive to transcribe characters into ASCII code.

5 5 EVMBUG ERROR MESSAGES

Several error messages have been provided in the EVMBUG monitor to alert the user to incorrect operation. In the event of an error, the word ´ERROR´ is output, followed by a single digit indicating the error condition.

Table 5-4 outlines the possible error conditions.

TABLE 5-4. EVMBUG ERROR MESSAGES.

ERROR	CONDITION
0	Invalid tag detected by the loader
1	Checksum error detected by loader
2	Invalid termination character detected
3	Null input field detected by dump routine
4	Invalid command entered

NOTES

ERRORS 0/1: The program load process is terminated.

If the program is being input from a 733ASR, possible causes of the error are a faulty cassette tape or dirty Read heads in the tape transport.

If the terminal device is an ASR33, chaf may be caught in a punched hole in the paper tape.

TO CORRECT: In either case, repeat the load procedure.

ERROR 2: Invalid Termination Character. Command is terminated

TO CORRECT: Reissue the command and parameters with a Valid termination character.

ERROR 3: Incorrect Input To Dump Command. Dump command is terminated.

User either input a null field for start address, stop address, or the entry address to the dump routine.

Ending address is less than the beginning address.

TO CORRECT: Reissue the dump command and input all necessary parameters.

ERROR 4: Self explanatory.

TO CORRECT: Enter a valid command

SECTION 6

SYMBOLIC ASSEMBLER

6.1 GENERAL

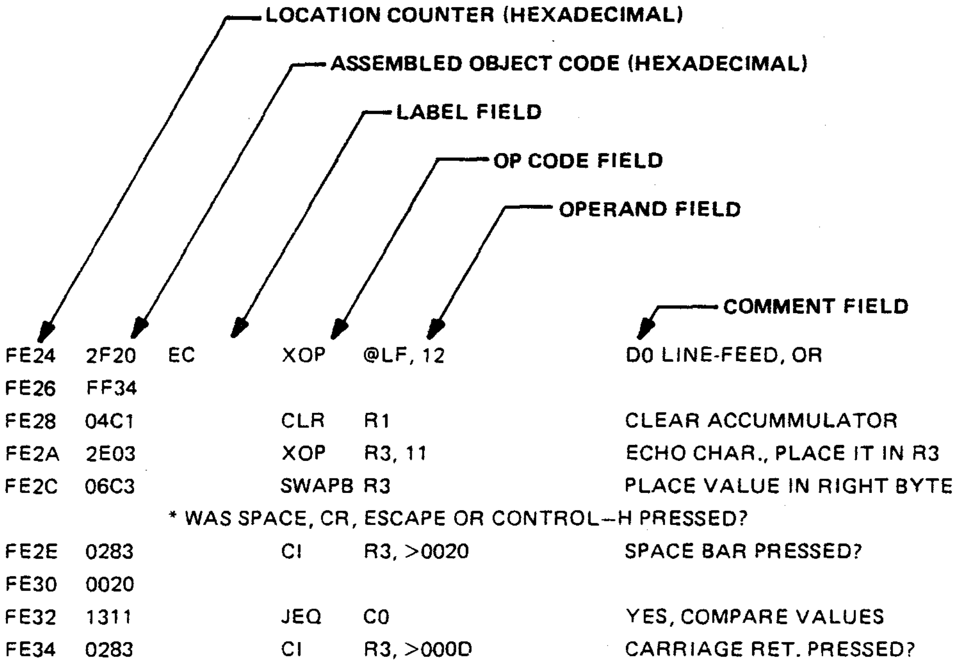
This section describes the function of the TMS 9995 EVM symbolic assembler. Also described are directions for formatting instructions and operating the assembler.

An assembler is a program that interprets assembly language source statements into object code. Assembler-directive commands allow the programmer to generate data words and values based on specific conditions at assembly time.

The TMS 9995 EVM Assembler is a one-pass symbolic line assembler designed to permit the use of comments and labels. It assembles the instructions of the TMS 9995 as well as the pseudo instruction NOP (which assembles as the instruction JMP $+2, and acts as a "no operation" or "go to next instruction"), and the following asssembler directives:

- AORG: Absolute Origin Statement (absolute start location)
- BSS: Block of memory reserved with starting symbol
- DATA: Sixteen bits of immediate data
- END: End of program, exit to monitor
- EQU: Symbol equated to value in operand
- TEXT: String of ASCII coded characters

The assembler program is contained in EPROM, along with the rest of the EVM firmware.


LOCATION COUNTER (HEXADECIMAL)
ASSEMBLED OBJECT CODE (HEXADECIMAL)
LABEL FIELD
OP CODE FIELD
OPERAND FIELD
COMMENT FIELD

```
FE24  2F20  EC     XOP   @LF, 12         D0 LINE-FEED, OR
FE26  FF34
FE28  04C1         CLR   R1              CLEAR ACCUMMULATOR
FE2A  2E03         XOP   R3, 11          ECHO CHAR., PLACE IT IN R3
FE2C  06C3         SWAPB R3              PLACE VALUE IN RIGHT BYTE
            * WAS SPACE, CR, ESCAPE OR CONTROL–H PRESSED?
FE2E  0283         CI    R3, >0020       SPACE BAR PRESSED?
FE30  0020
FE32  1311         JEQ   C0              YES, COMPARE VALUES
FE34  0283         CI    R3, >000D       CARRIAGE RET. PRESSED?
```

FIGURE 6-1. SAMPLE ASSEMBLER LISTING.

6.2. TMS 9995 SYMBOLIC ASSEMBLER LISTING

6.2.1 Listing Format

The format of the listing produced by the TMS 9995 is detailed in Figure 6-1. The elements of this listing are discussed in subsequent paragraphs.

6.2.1.1 Location Counter

This is the hexadecimal number showing the location of assembled object code.

Essentially, the value of the location counter is the address of the corresponding object code after a program has been loaded into memory. For example, in figure 6-1, the object code at memory address (MA) >FE24 is >2F20; M.A. FE26 will contain the address of the location with a label of LF when (and if) that label is defined.

6.2.1.2 Assembled Object Code

This column contains the resulting object code in hexadecimal after the source statement has been assembled.

6.2.1.3 Label Field

The two-character field contains an alphanumeric label that identifies the location of the source statement.

6.2.1.4 OP Code Field

This four-character field contains assembly language operaton code mnemonics. It is separated from the label field and operand field by one space.

6.2.1.5 Operand Field

This field contains the operands of the instruction. This field is separated from the OP code and comment fields by one space.

6.2.1.6 Comment Field

Comments are placed in the listing by the user to assist in the understanding of the instruction or the data flow. The comment field begins one space to the right of the operand field.

6.3 LABELS AND COMMENTS

Labels may consist of one or two characters. The first character must be alphabetic (but not an ´R´) and a second character must be alphanumeric. Labels may be used either as resolved (previously defined) or unresolved (to be defined in upcoming assembly statements) references. Labels may defined by entering them in the Label Field of an assembler statement. Labels used as symbolic references in an instruction that will accept both symbolic and register operands must have an (@) sign preceeding them.

Comments can be a part of the source statement. The comment field may include any printable character and is concluded by a return. A comment line is indicated by an asterisk (*) in column one.

6.3.1 Use Dollar Sign To Indicate "At This Location"

Use the dollar ($) sign to indicate a current value of the location counter (the location counter contains the next address at which object will be loaded). If the location counter contains a value of >ED00, then the following comments apply as shown in the following statements:

```
ED00        D1 EQU $
ED00        *  D1 VALUE = LOCATION COUNTER VALUE: >ED00
ED00        E1 EQU $+4
ED00        *  E1 VALUE = LOCATION COUNTER + 4 = >ED04
ED00        F1 EQU D1
ED00        *  F1 AND D1 HAVE SAME VALUE = >ED00
ED00 0207      LI R7,$    >ED00 TO R7
ED02 ED00
ED04 0208      LI R8,D1   >ED00 TO R8
ED06 ED00
ED08 0209      LI R9,$+2  >ED0A TO R9
```

```
ED0A ED0A
ED0C 020A    LI R10,E1   >ED04 TO R10
ED0E ED04
ED10
```

NOTE

In EQU (equate) directives, labels must be equated to either absolute values or defined labels.

6.3.2 Expressions

Expressions contain addition or subtraction functions. For example:

```
ED00        A1 EQU >200
ED00        *                         A1 VALUE = >200
ED00        B1 EQU A1+8
ED00        *                         B1 = A1 + 8 = >208
ED00        C1 EQU A1
ED00        *                         C1 = A1 VALUE = >200
ED00 0200     LI R0,A1                >200 TO R0
ED02 0200
ED04 0201     LI R1,A1+4              >204 TO R1
ED06 0204
ED08 0202     LI R2,A1+C1             >400 TO R2
ED0A 0400
ED0C 0203     LI R3,A1+B1+C1          >608 TO R3
ED0E 0608
ED10 0204     LI R4,A1-B1             >FFF8 TO R4
ED12 FFF8
ED14
```

6.3.3 Cancel Source Statement Being Input

If it is desired to cancel a source statement while in the process of entering it from the keyboard, press the <ESC> key. The current location counter contents will be displayed, waiting for new input. This escape MUST be executed prior to entering a return after the source statement.

6.3.4 Translate Characters Into ASCII Code Using Single Quotes

If it is desired to translate alphabetical or numerical keyboard values in ASCII code, enclose the characters in single quotes. This is the normal procedure for the TEXT assembler directive (paragraph 6.4.2.6); however, it can also apply in other situations. For example:

```
ED00      A     EQU  ´AB´
ED00      *                      ASCII FOR AB = >4142
ED00 0201       LI   R1,A        LOAD >4142 IN R1
ED02 4142
ED04 0201       LI   R1,´AB´     LOAD >4142 IN R1
ED06 4142
ED08 4142 A1    DATA ´AB´        ASSEMBLE >4142 HERE
ED0A
```

6.4 ASSEMBLER DIRECTIVES

The symbolic assembler recognizes six assembler directives. The conventions used in defining these directives are defined below.

< >: Required items to be supplied by the user

[ ]: Optional items to be supplied by the user, i.e., for example: [comment] = a space followed by any characters except <ESC> or <CR>.

EXPW: A well-defined expression (No forward references)

EXP: An expression with no forward reference, or a forward reference only.

Symbolic addresses must be preceeded by an @ sign to differentiate from a register number in an instruction that will accept both.

6.4.1 AORG Directive

```
FORMAT:
   [label]< ,><AORG>< ,><EXPW:location><CR>
```

The AORG directive places a value in the location counter and begins assembly at the location specified. The location value must be in decimal or hexadecimal. By default, the location counter for the assembler begins at >0000 and is incremented by two (bytes) for each word occupied by the instruction. When a label is used with the AORG directive, it is assigned the value in the location counter. The comment field is optional. If an odd value is input for the location, the value will be decremented to an even value.

Example:

```
ED00         AORG >200      Begin assembling source code at location
0200                        counter value of >200


ED00         AORG 200       Begin assembling source code at location
00C8                        counter value of >C8
```

6.4.2 BSS Directive

```
     FORMAT:
        [LABEL]< ,><BSS>< ,><EXPW:no. of bytes><CR>
```

The BSS (block with starting symbol) directive advances the location counter a quantity of bytes as specified in the directive. In essence, it "reserves" a block of bytes starting at the location counter value; this block will be void of object code. An optional label can be specified to identify the first location in the block. The byte count can be in decimal or hexadecimal.

6.4.3 DATA Directive

```
     FORMAT:
      [label]< ,><DATA>< ,><EXP>[,<EXPW>,...,<EXPW>][comment]<CR>
```

This directive places 16 bit values into (successive) memory locations. Data is placed at even address locations. Operand values can be chained (i.e., successive 1 to 16 bit values separated by commas). The data directive will accept multiple operands seperated by commas. It will also accept an unresolved reference, but ONLY as a first operand.

Example:

```
ED00 FFFF      DATA FFFF,1764,>BB,0,444,'AB'
ED02 06E4
ED04 00BB                                    Assembles as ASCII code
ED06 0000                                    for string AB
ED08 01BC
ED0A 4142                                  Assemble as >00BB,>0000
ED0C
```

6.4.4 END Directive

FORMAT:
[label]< ,><END>< ,>[<entry point>< ,>[comment]<CR>

This directive is mandatory for each program. It designates to the assembler that this is the final input from the source program and causes a transfer of control back to the monitor. This is the last input to the assembler, and the only means of direct transfer from the assembler to the monitor. When the optional label is used, it is assigned the current value in the location counter, but forward references to it will not be resolved. The optional load-point operand field contains a symbol or absolute memory address specifying the entry point (execution start) of the program. When the entry-point operand is used, the entry point address will be placed in the Program Counter so that the program can be executed by the EX command immediately after being loaded.

After entry of the END directive is concluded, a number indicating the number of unresolved labels will be displayed.

Example:

```
ED02          END  ST   0000
MON?                         Output by assembler: number of
                             unresolved references.

                          Location labeled ST is entry point
                          for program; places address in
                          program counter.
```

6.4.5 EQU Directive

```
FORMAT:
   <label>< ,><EQU>< ,><EXPW><CR>
```

This directive assigns a value to a label for use during assembly. The expression field can contain an absolute numeric value or expression. Expressions are further defined in paragraph 6.3.4. This directive allows the user to substitute easily remembered mnemonics for absolute values in program source lines.

Examples:

(1)

```
ED00          SM EQU 1
ED00          *    ALLOWS USING SM FOR REGISTER 1 SUCH AS
ED00 C060          MOV @>FC00,SM  MOVE QTY TO R1
ED02 FC00
ED04          *    INSTEAD OF  @>FC00,1
ED04 C060          MOV @>FC00,1  MOVE QTY TO R1
ED06 FC00
ED08
```

(2)

```
ED00          IN EQU 9681
ED00          *  ALLOWS THIS CONSTANT VALUE TO BE USED IN SUBSEQUENT
ED00          *  SOURCE LINES
ED00 0201        LI R1, IN  PLACE CONSTANT IN R1
ED02 E5D1
EDO4
```

(3) If IN has been previously defined, as above, the following will result in moving the value located four bytes beyond location IN into location OT:

```
ED00 C820      MOVE @IN+4,@OT
ED02 F004
ED04 F0C0
ED06
```

(4) A label can be equated to a string of labels being added or subtracted (expression).

```
ED00       A   EQU 4
ED00       B   EQU 10
ED00       C   EQU A+B
ED00 0014 NO   DATA C+A  EQUALS VALUE 20
ED02
      NOTE:  Value of label "NO" has value equal to sum of
             values of "C" and "A".
```

6.4.6 TEXT Directive

FORMAT:
```
   [label]< ,><TEXT>< ,><´character string´>[comment]<CR>
```

This directive, like the Data directive, is used to generate absolute data for program use. The DATA statement operand is interpreted as a numerical value. The TEXT statement operand contains an alphanumeric character string of keyboard inputs which are to be interpreted into ASCII code. Besides keyboard characters, the user can also input control characters (e.g., carriage return, line feed, DC1, DC2, etc.) which are output in ASCII code via the keyboard. ASCII code is defined in Appendix B. Character string inputs in the operand field are enclosed in single quotes. The assembler begins all character strings on an even boundary and places a zero byte after the last character that can be used as a delimiter by the XOP I/O commands. The character string may contain any characters except the single quotes (´´) and <ESC>.

The optional label field will be assigned the value in the location counter; this value will identify the location of the first character in the string. If the program counter is odd after the text and zero bytes are entered, then the program counter will be incremented to an even number.

Examples:

(1)

```
ED00 4C4F CM TEXT 'LOAD TAPE, HIT<CR><LF>'
ED02 4144
ED04 2054                                    (Followed by a Carriage Return
ED06 4150                                     and a Line feed.)
ED08 452C
ED0A 2048
ED0C 4954
ED0E 203C
ED10 4352
ED12 3E0D
ED14 0A00
ED16
```

(2)

```
ED00 4C4F CM TEXT 'LOAD TAPE, HIT <CR>.'
ED02 4144
ED04 2054
ED06 4150
ED08 452C
ED0A 2048
EDOC 4954
ED0E 203C
ED10 4352
ED12 3E2E
ED14 0000 <— This number will be that which was in memory before
ED16         the TEXT directive is assembled.
```

6.5 ASSEMBLER ACTION

The Symbolic Assembler accepts assembly language inputs from the keyboard. As each instruction is input, the assembler interprets it, places the resulting machine code in an absolute address and prints the machine code (in hexadecimal) next to its absolute address.

Example:

The user enters:

```
        LWPI >ED20  <CR>
```

The following display results:

```
        LWPI  >ED20          USER INSTRUCTION ECHOED
        ED00  >02F0          RESULTING OBJECT CODE
        ED02  >ED02
```

6.6 OPERATION

6.6.1 Calling the Assembler

1. Call up the monitor by activating RESET on the EVM and pressing the "A" key.

2. The EVMBUG monitor prints an initialization message on the terminal: MON? indicating that the command scanner is available to interpret terminal inputs.

3. Enter either the XA or XAE command and space or carriage return. If the XA command is used, the previous symbol table will be cleared.

4. Enter the hexadecimal address at which the program is to be assembled.

5. Press RETURN key. Entry to assembler is acknowledged by the display of the address. The cursor is positioned to the label entry column.

In this and following examples, the underscore marks the cursor positon within the display.

```
    Display         Enter              Comments


Move RESET Switch
                    (CR)            Monitor Entry Gained
EVMBUG
MON?
                    XA (SP)         Assembler Called
MON? XA _
                    ED00            Starting Assembly Add.
MON? XA ED00_
                    (CR)
ED00      _                         Assembler Entry Gained
```

6.6.2 Exiting To The Monitor

(SP) to the OPCODE column, then enter: END. Control returns to the monitor.

6.7 ENTERING INSTRUCTIONS

Any of the 73 instructions applicable to the TMS 9995 microcomputer can be interpreted by the Symbolic Assembler. An instruction generally consists of four fields: Label, Opcode, Operand(s), and Comment.

6.7.1 Label Field

The label field is optional and its omission is indicated by a space. It consists of a maximum of two characters; the first character must be alphabetic(BUT NOT AN ´R´) and the second must be alphanumeric. Labels may be used as either resolved or unresolved references. The label field may be followed by one or more spaces.

NOTE: the following fields will not accept unresolved references: register fields, shift count fields, CRU count fields, and CRU displacement fields. Instructions containing unresolved references should not be modified once entered until reference resolution has occurred or errors may be created.

6.7.2 Opcode Field

There should be a single space only between the opcode and the operand(s).

6.7.3 Operand Field

Operand fields generally consist of either: (1) one unresolved reference label, or (2) a succession of constants and defined symbols linked by plus and minus signs. In the case of multiple operands, a single comma should be used between the two. The operand field should be followed by a space if the comment field is desired, or a return if not.

6.7.4 Comment Field

The comment field may include any printable character and is concluded by a return.

6.7.5 Concluding The Instruction

The (CR) at the end of either the operand or the comment field concludes the instruction. Prior to entering the return, the instruction may be cancelled by use of the Escape <ESC> command, as explained in paragraph 6.3.3.

6.7.6 EXAMPLES:

```
1.  LWPI >220
        ↑---------- Single space between mnemonic and operand


2.  LI 0,33
        ↑---------- Single comma between multiple operands


3.  N1 DATA 10
      ↑----↑------- (SP) after label and opcode fields


4.     DISPLAY      ENTER                  COMMENTS


    ED00      _
                    (SP)              Omit Label Field
    ED00      _
                    LWPI >220         Enter Instruction
    ED00      LWPI >220_


5.  INSTRUCTION               TERMINATOR


    LWPI >220          (A) (CR) - comment field omitted
                       (B) (SP) - comment field to be used
```

6. The following example illustrates these functions:

   A. Calling the assembler (paragraph 6.6.1)
   B. Enter instruction one (paragraph 6.7)
   C. Enter instruction two (paragraph 6.7)
   D. Exiting to the monitor (paragraph 6.6.2)

```
Display        Enter         Comments

                    Set RESET Sw.
                    (CR)           Monitor Entry Gained
  EVMBUG
  MON? _
                    XA ((SP))      Assembler Called
  MON? XA _
                    ED00           Starting Assembly Address
  MON? XA ED00_
                    (CR)
  ED00      _                      Assembler Entry gained
                    (SP)           Skip label field
  ED00       _
                    LWPI >ED20     Enter first instruction
  ED00       LWPI >ED20_
                    (CR)
  ED00 02E0_                       Addresses and machine
                    LWPI >ED20     code for first
  ED02 ED20_                       instruction.
                    (CR)
  ED04      _
                    (SP)           Skip label field.
  ED04       _
                    LI 0,33        Enter second instruction
  ED04       LI 0,33_
                    (CR)
  ED04 0200_                       Addresses and machine
                                   code for second
  ED06 0021_                       instruction.

  ED08      _
                    (SP)           Skip label field
  ED08       _
                    END            Enter END directive
  ED08       END_
                    (CR)           First (CR)
  ED08       END 0000_
                                   Second (CR)
  MON? _
```

The following additional concepts apply to instruction entry:

1. Register numbers are in decimal or hexadecimal. Only decimal register numbers can be predefined (preceeded by an R).

```
LI R13,22
LI >D,33
```

2. Jump instruction operand can be $, $+n, $-n, or M, where n is a decimal or hexadecimal value of bytes (+256>n>-254) and M is the value of the memory address.

```
JMP $+0
JMP $-2
JMP $+2
JMP >210
```

3. Absolute numerical values can be decimal or hexidecimal hexadecimal. Decimal values have no prefix in an operand. Hexadecimal values are preceded by the greater-than sign (>).

```
LI R13,>33
LI R13,51
```

4. Where an address can be either a register or symbolic memory location, the symbolic address is preceeded by an at sign @ to differentiate a numerical memory address from a register number.

```
MOV @ST,R1          Move ST contents to R1
A @SM,@>FE00        Move SM contents to M.A.>FE00
```

NOTE:

Jump and immediate operand instructions do not use the (@) sign, before a symbol.

6.8 ERRORS

Syntax errors are indicated by an ´ERR´ message. A displacement range error (such as with jump instructions and single-bit CRU instructions) will be flagged with an ERR message.

1. Syntax error. The instruction syntax was incorrect:

```
          Display        Enter            Comments

ED00        _
                       (SP)
ED00         _
                       LDA
ED00         LDA ERR_                    Error message (ERR)
                       (CR)              Use Ret and enter
                                         the proper  mnemonic.
```

2. Range error. The operand is out of range of its field.

```
          Display        Enter              Comments

ED00        _
                       (Sp)                Skip  Label  field.
ED00         _
                       LI R44              Enter first  instr.
ED00        LI R44_

ED00        LI R44 ERROR                   Error message.
                       (CR)
ED00        _
                       (Sp)
ED00         _
                       LI R4,>ED00         Enter  proper data.
ED00        LI R4,>ED00_
                       (CR)
ED00 0204 LI R4,>ED00                      Properly  assembled
                                           code.
ED02 ED00

ED04 _
```

3. Displacement Error. The jump instruction destination is more than +256 or -254 bytes away.

Display	Enter	Comments
ED00 _	(Sp)	Skip label field.
ED00 _	JNC $+300	Displacement plus 256 bytes.
ED00 JNC $+300_	(CR)	
ED00 JNC $+300 ERROR		ERROR message.
ED00 _		

6.9 PSEUDO-INSTRUCTION

The assembler also interprets one pseudo-instruction. This pseudo-instruction is not an additional instruction, but actually is an additional mnemonic that conveniently represents a member of the instruction set. The NOP mnemonic can be used in place of a JMP $+2 instruction, which is essentially a no-op (no operation). This can be used to replace an existing instruction in memory, or it can be included in code to force additional execution time in a routine. Both NOP and JMP $+2 assemble to the machine code >1000.

```
      Display         Enter                   Comments

ED00       _
                 (Sp)
ED00        _
                 JMP $+2
ED00       JMP $+2_
                 (CR)
ED00 1000_                  <------+     Both JMP$+2 and
                                   |     NOP assemble to
                                   |     the machine code
ED00       _                       |<--  >1000
                 (Sp)              |
ED00        _                      |
                 NOP               |
ED00        NOP_                   |
                 (CR)              |
ED00 1000 _                 <------+
```

SECTION 7

EIA COMMUNICATIONS LINK

7.1 GENERAL

This section describes the use of a Software Comminications Link which allows TMS 9995 microcomputer module to communicate with a DX 990/10 minicomputer via an EIA RS-232-C interface.

This communications link is primarily intended for use as a software development aid for the programmer. It allows the programmer to create programs on a larger, more sophisticated minicomputer, taking full advantage of its utilities, i.e., text editor, macroassembler, linker, etc. The resulting program can then be downloaded into the 9995EVM for execution or debugging. The host system is needed to support the hardware and software necessary to transfer information to an EIA port using the ASCII character set. (See Appendix B)

EIA RS-232-C asynchronous serial transfer is used by this particular communications link which enables it to communicate with terminals that have EIA capabilities.

The EVM hardware supports two serial I/O ports. The local port (Port 1) is jumper selectable for either RS-232-C or teletype terminals. (See Figure 7-2) This port is controlled by a TMS 9902 asynchronous communications channel and is the port to which the main programmer´s terminal is connected for support by the EVMBUG monitor. The auxilliary port (Port 2) is RS-232-C only and is connected via EIA RS-232-C to the host computer.

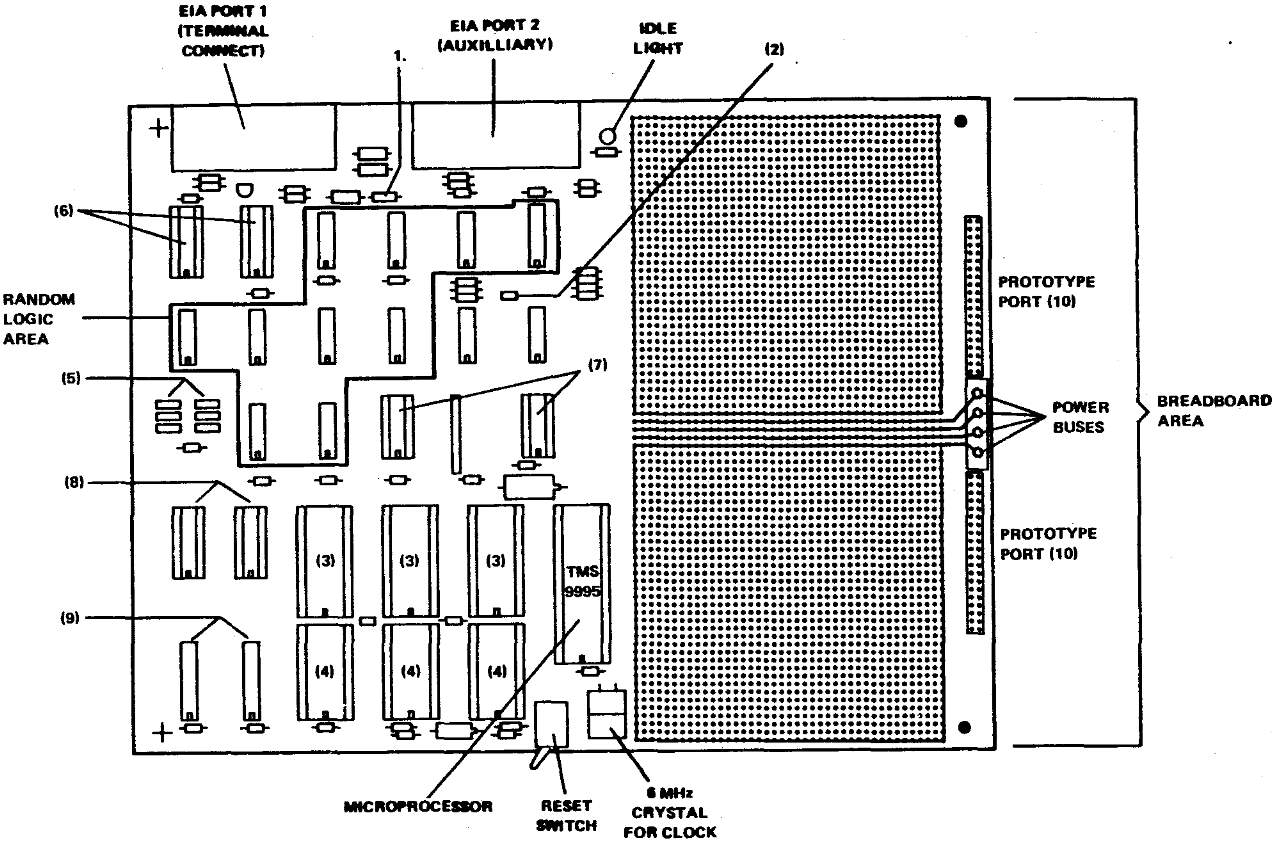


FIGURE 7-1. TMS 9995 EVALUATION MODULE BOARD.

7.2 SYSTEM DESCRIPTION

Figure 7-2 shows a typical system configuration for utilization of the communications link software. The TMS 9995 appears as another terminal to the host system. The communications link allows a user at terminal 2 to interact with the host computer in exactly the same way as if it were directly connected to the host computer. A user at either terminal may command the host computer to execute a read or write to the memory. This read or write to memory is executed by the host computer as if it were reading or writing to a cassette, paper tape, or keyboard/printer (if device support was sysgened into the host system).

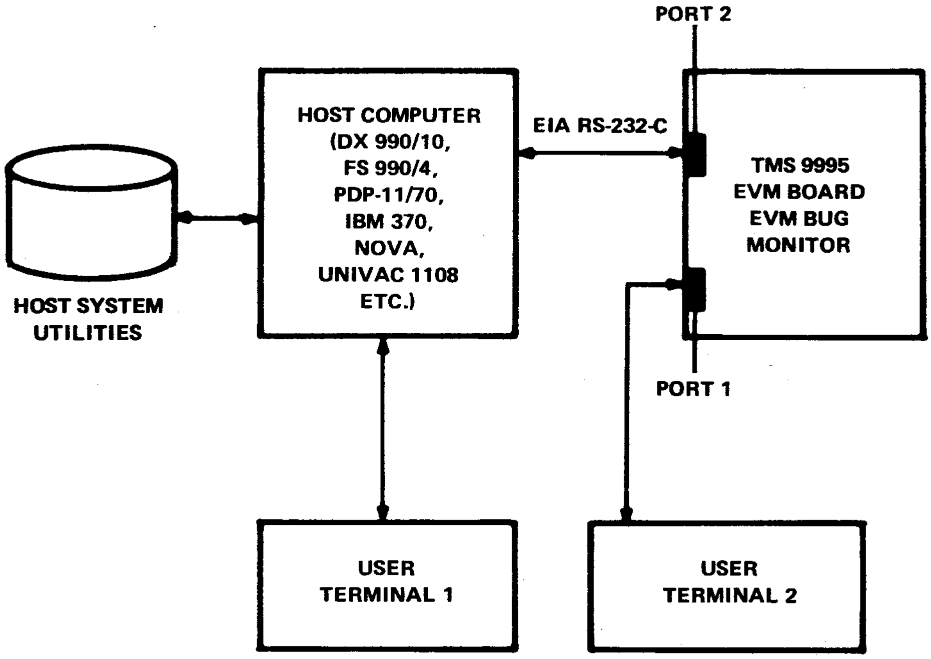

PORT 2
HOST COMPUTER (DX 990/10, FS 990/4, PDP-11/70, IBM 370, NOVA, UNIVAC 1108 ETC.)
EIA RS-232-C
TMS 9995 EVM BOARD EVM BUG MONITOR
HOST SYSTEM UTILITIES
PORT 1
USER TERMINAL 1
USER TERMINAL 2

FIGURE 7-2. TYPICAL SYSTEM CONFIGURATION.

The communications link does not require any hardware changes to the host computer, nor does the host system require any changes in device service routines, except those changes necessary to support a 733 ASR, ASR 33, or KSR protocol. Data transfers are accomplished using the ASCII character set over EIA RS-232-C levels at a variety of baud rates. Data transfers to memory are formatted in TMS 9900 object record format (see Appendix A

The following is an example of how the communications link can be implemented:

> The TMSW 101T cross support package can be installed on an already existing DEC PDP-11/70 minicomputer system. Applications software can be created on the PDP-11/70 and assembled using the cross support package. The resulting applications code can then be simulated using the cross support package, or downloaded for testing on the target system using the EIA link. Assuming the PDP-11/70 already exists, this communications link allows actual program development and test for TMS9995 software without the need for additional hardware.

Using a 990 system minicomputer allows this same opportunity without the need for the cross support package.

7.3 SYSTEM REQUIREMENTS

7.3.1 Host System Requirements.

The communications link software is capable of communicating to a wide variety of host computers, ranging from a TM990/101M microcomputer to a large time-share system. The requirements which must be met by the host computer are supplied by most existing computers.

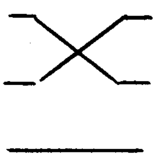
7.3.1.1 Hardware Requirements.

The EIA communications link requires at least three signals to operate:

- Transmit Data
- Receive Data
- Signal Ground

Table 7-1 illustrates how these signals must be interconnected.

TABLE 7-1. HOST SYSTEM CABLE REQUIREMENTS.

HOST INTERFACE			TMS9995 INTERFACE	
Designation	Pin		Pin	Designation
Receive Data	2		2	Receive Data
Transmit Data	3		3	Transmit Data
Signal Ground	7		7	Signal Ground

These signals, along with other control signals, may be supplied by a TM990/506 cable assembly. The other control signals are required to perform the necessary "handshaking" between the EVM board and the host computer, i.e., DSR, DCD, RTS, and may vary for different host computers.

The EVM has no baud rate limitations because the baud rate as used by the TMS 9902 asynchronous communications channel is software selected. However, the communications link software only allows baud rates of 110, 300, 600, 1200, 2400, 4800, 9600, and 19200. The baud rate of the host computer must be that of the terminal connected to Port 1, if that terminal is to be used as a remote terminal to the host system (i.e., logon identifiers, listings, etc.) The baud rates need not be the same to execute uploads or downloads, as the terminal is not involved. The baud rate to the terminal on Port 1 is automatically set by the EVMBUG monitor. The baud rate to the host computer is 1200 by default, but may be changed by the use of the communicatons link "T" command to any of the baud rates given above. (See also "T" Command, paragraph 7.4.3.)

7.3.1.2 Software Requirements.

The host system reads and writes to the TMS 9995 EVM and terminal combination as if they were a teletype, 733 ASR terminal, or keyboard/printer.

Receipt of a DC2 (ASCII Punch On) places the TMS 9995 EVM into a download mode of operation until a DC4 (ASCII Punch Off) is received.

Receipt of either a DC1 (ASCII Reader On) or DLE7 (Cassette Block Forward) command places the TMS 9995 EVM into an upload mode. This mode is continued until a complete record is output in 733 ASR protocol, or until the upload is complete in ASR 733 protocol. When not in either of the above modes, any characters received on EVM Port 2 are echoed to the terminal (which may or may not be present) at EVM Port 1.

A provision is provided to operate the communications link with a host computer that cannot supply the DC2 and DC4 commands necessary for downloads. This provision is described in paragraph 7.4.5 "Use Without Cassette Or Paper Tape Support".

Control characters entered at EVM Port 1 are recognized at all times by the TMS9995 EVM, with the appropriate response taken. Noncontrol characters are ignored during uploads and downloads, but are echoed to the host system otherwise.

To accomplish downloads, the host system is required to supply standard TMS 9900 machine code in object record format (see Appendix A). The code can be either copied from storage media (magnetic tape, disk, etc.) or actually created by the host computer. This machine code is the same as that of the host (if the host is a 990 family minicomputer), due to the software compatibility between all members of the 9900 family. Cross assemblers are also available to produce TMS 9900 machine code on non-9900 family computers (IBM, DEC, etc.)..

7.3.2 Terminal Requirements.

The terminal connected to Port 1 of the TMS 9995 EVM Board must be either EIA RS-232-C or 20-mA current loop (jumper selectable), and communicate via the ASCII character set.

The allowable baud rates are the same as those listed for the host computer and are automatically set by the EVMBUG monitor. If this terminal is intended for use as a remote terminal for the host system, it must be the same baud rate as the host system and use the same protocol.

7.4 COMMUNICATIONS LINK USAGE

This section details how the communications link operates from a user´s point of view. Table 7-2 lists the communication link commands available to the user. The functions which require use of the Control key are available only in the terminal mode; other functions are available only in the command mode.

Table 7-3 is a list of error messages which the communications link produces under certain error conditions. Each mode is described in the following paragraphs.

TABLE 7-2. SUMMARY OF COMMUNICATIONS LINK COMMANDS.

INPUT	RESULTS
Control C	Enter Command Mode
T	Change Port 2 Baud Rate
D	Set Download Bias
U	Set Upload Limits
Q	Return To Terminal Mode
Control Z	Return To EVMBUG Monitor
Control R	Initiate Download
Control T	Terminate Download

TABLE 7-3. SUMMARY OF COMMUNICATIONS LINK ERROR MESSAGES

ERROR MESSAGE	MEANING
CMD ERR	Invalid command entered. Reenter the correct command code, i.e., T, U, D, or Q.
PARM ERR	Invalid parameter entered. Reenter a valid parameter.
CKSM ERR	Checksum error occurred during download.
TAG ERR	Invalid obj. record tag encounter during download.
UPLD ERR	Error occurred during upload attempt or, upload end limit is smaller than start limit or, upload aborted.

7.4.1 Starting The Link.

On-board RAM provides one workspace area, two flags, and a link area for handling the communications link software. The location of this area is from >EC00 to >EC56.

The link is entered by executing the XCL command. At this time, the EVM is in the terminal mode, and an entry banner: TERMINAL MODE, is printed on the terminal. This entry banner will be printed every time the terminal mode is reentered.

7.4.2 Terminal Mode.

Once in terminal mode, the communications link is in its active mode. The program constantly scans both ports until a character is received on one or the other. It then takes the appropriate action, depending on the character received and the function currently being executed. Downloads, uploads, and listings can be executed under control of the host computer and all host commands entered at the terminal are echoed to the host computer.

If an error occurs on the EVM side of the communication link during a download, the link will wait until the current input from the host computer ceases and then output an error message. At this time, the download may or may not be completed. This must be verified by use of the EVMBUG monitor Inspect Memory Command. If an error occurs during an upload, the link will output an end-of-file, discontinue output, and output an error message. (See Table 7-3) These download and upload error conditions also set the download bias to >FFFF.

7.4.3 Command Mode.

The communications link also supports a command mode of operation, if a terminal is present on EVM Port 1. In this mode, commands are entered from the terminal to:

- Change Port 2 baud rate
- Set upload limits
- Set bias for downloads of reloacatable object files

This mode can be entered from the terminal mode at any time (even during up or downloads) by simultaneously pressing the Control and "C" keys. Once in the Command mode, a question mark prompt is displayed at the terminal.

The commands supported while in the command mode are described below.

"T" COMMAND: A "T" is input to change the baud rate of Port 2. The link will echo the "T" to the printer, followed by a space, and will await the entry of a parameter. This parameter must be a valid decimal digit between 1 and 8, followed by a space, comma, minus sign, or carriage return. Otherwise, a parameter error will be generated. According to the value entered, the baud rate of Port 2 (to the host computer) will be set to the value indicated in Table 7-4, below. If no parameter is entered, the baud rate will remain unchanged.

TABLE 7-4. BAUD RATE SELECTION PARAMETERS

PARAMETER RATE	BAUD RATE
1	19200
2	9600
3	4800
4	2400
5	1200
6	600
7	300
8	110

"D" COMMAND: A "D" is entered to set the bias of any relocatable object code received by the downloader. The "D" will be echoed to the printer, followed by a space. The user may then enter a valid hexadecimal address, followed by a space, comma, minus sign, or carriage return. This address will be the download bias until changed. The default bias address is >ED00. If no address is entered, the download bias will remain unchanged. The download bias only applies to relocatable code, as any absolute code will be loaded wherever its object record tags indicate.

"U" COMMAND: A "U" is entered to set the upload limits. The "U" is echoed to the printer, followed by a space. The user may then enter the upload starting address, followed by the upload ending address. Both must be valid hexadecimal numbers, followed by a space, carriage return, comma, or minus sign. Either or both may be omitted to leave the corresponding upload limit unchanged. An invalid address results in the printing of an error message with no change in upload limits. Default upload limits are from >ED00 to >EFF0 inclusive. Limits must be reestablished before each upload, as the starting address is changed by the uploader to equal the ending address at the end of the upload. An error will result at any time during the upload if the starting address is greater than the ending address. The host computer may also set these limits by writing to the appropriate memory locations (>EC50 and >EC48), using the downloader.

"Q" COMMAND: To exit from the command mode back to the active terminal mode, a "Q" must be entered. The "Q" will be echoed to the printer, followed by the terminal mode entry banner.

7.4.4 Returning Control To EVMBUG Monitor.

The communications link execution can be terminated in the terminal mode by simultaneously pressing the Control and "Z" keys. This returns control to the EVMBUG monitor. The EVMBUG entry banner will be displayed on the printer at this time.

7.4.5 Link Use Without Cassette Or Paper Tape Support.

A provision exists which allows the user to initiate and terminate downloads from the terminal. This provision is necessary if the host computer software does not support cassettes or paper tape. Downloads can be accomplished by listing the object file to the TMS9995 EVM as if it were a printer. Pressing the Control and "R" keys sets the communication link to download mode. All input on Port 2 is then transferred to the downloader instead of the terminal.

The Control ´T´ command allows the user to exit this download mode at any time and to return to the terminal mode.

The Control and "T" keys must be simultaneously pressed to return to the terminal mode. The two control keys generate the required DC2 (Punch ON) and DC4 (Punch Off), which are normally generated for a paper tape punch or cassette by the device service routine.

7.5 SAMPLE SOFTWARE DEVELOPMENT SESSION

With a terminal or TTY connected to Port 1 (See paragraph 7.3.2), and a host computer connected by an EIA RS232-C communication link to Port 2 (See paragraph 7.4), the TMS 9995 EVM may be used for software development. See Figures 7-1 and 7-2.

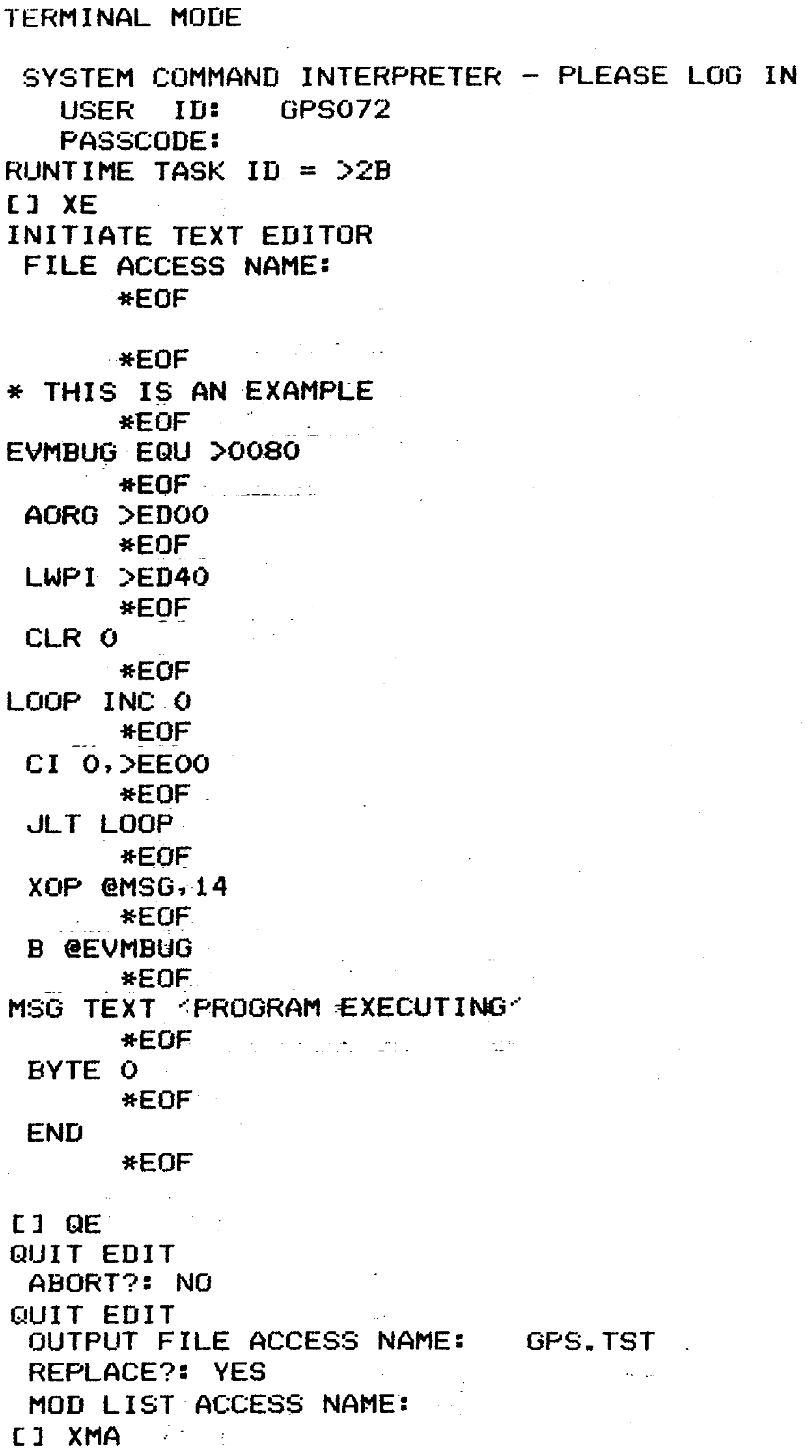
If the 990/10 (or 990/12) has not been sysgened to include the use of a 733 ASR terminal, then a 733 should be installed and the sysgen executed as follows before a software session begins. Brief details of a sysgen follow (for experienced programmers only). For full details of sysgen, refer to the Model 990 Computer DX10 Operating System Programming Guide, Volume V, part number 946250-9705.

SAMPLE SYSGEN:

```
NAME = ST01
DEVICE TYPE = ASR
CRU = >0000
ACCESS TYPE = RECORD
TIME OUT = --- SECONDS
CASSETTE TIME OUT = 3 SECONDS
CASSETTE ACCESS TYPE = FILE
CHARACTER QUEUE SIZE = 6
INTERRUPT = 6
```

The following is a sample software development session using a DX990/10 as a host computer. This sample shows the basic steps necessary to load the communications link and begin execution. Then the same terminal is used as a user terminal to the DX990/10 system. A program is created using the text editor and macroassembler. The program listing is printed at the user terminal and the object code is then downloaded into the TMS 9995 EVM memory. After logging off the DX990/10, control is returned to the EVMBUG monitor and the sample program is executed. For this example, the TMS 9995 EVM is connected to the DX990/10 as a 733 ASR terminal.

Commands preceeded by brackets [ ] are DX10 commands to the TMS 9995 microcomputer.



```
TERMINAL MODE

 SYSTEM COMMAND INTERPRETER - PLEASE LOG IN
   USER  ID:   GPS072
   PASSCODE:
RUNTIME TASK ID = >2B
[] XE
INITIATE TEXT EDITOR
 FILE ACCESS NAME:
          *EOF

          *EOF
* THIS IS AN EXAMPLE
          *EOF
EVMBUG EQU >0080
          *EOF
 AORG >ED00
          *EOF
 LWPI >ED40
          *EOF
 CLR 0
          *EOF
LOOP INC 0
          *EOF
 CI 0,>EE00
          *EOF
 JLT LOOP
          *EOF
 XOP @MSG,14
          *EOF
 B @EVMBUG
          *EOF
MSG TEXT 'PROGRAM EXECUTING'
          *EOF
 BYTE 0
          *EOF
 END
          *EOF

[] QE
QUIT EDIT
 ABORT?: NO
QUIT EDIT
 OUTPUT FILE ACCESS NAME:   GPS.TST
 REPLACE?: YES
 MOD LIST ACCESS NAME:
[] XMA
```

SAMPLE SOFTWARE DEVELOPMENT SESSION

```
EXECUTE MACRO ASSEMBLER
 SOURCE ACCESS NAME: GPS.TST
 OBJECT ACCESS NAME:   GPS.TSTO
 LISTING ACCESS NAME:   GPS.TSTL
 ERROR ACCESS NAME:
 OPTIONS:
 MACRO LIBRARY PATHNAME:
[] WAIT
--WAITING FOR BACKGROUND TASK TO COMPLETE--
MACRO ASSEMBLY COMPLETE, 0000 ERRORS, 0000 WARNINGS
[] CC
COPY/CONCATENATE
 INPUT ACCESS NAME(S): PRO.TIMERO  GPS.TSTL
 OUTPUT ACCESS NAME: CS03  ST17
 REPLACE?: NO
 MAXIMUM RECORD LENGTH:   60


 [] CC
 COPY/CONCATENATE
  INPUT ACCESS NAME(S): GPS.TSTL
  OUTPUT ACCESS NAME: ST17
  REPLACE?: NO
  MAXIMUM RECORD LENGTH:   60




           SDSMAC 3.2.0 78.274   15:41:31 THURSDAY, MAR 19
 ACCESS NAMES TABLE
 SOURCE ACCESS NAME=     R32USR.GPS.TST
 OBJECT ACCESS NAME=     R32USR.GPS.TSTO
 LISTING ACCESS NAME=    R32USR.GPS.TSTL
 ERROR ACCESS NAME=
 OPTIONS=
 MACRO LIBRARY PATHNAME=
```

SAMPLE SESSION (Continued)

```
           SDSMAC 3.2.0 78.274   15:41:31 THURSDAY, MAR 19

 0001                * THIS IS AN EXAMPLE
 0002        0080    EVMBUG EQU >0080
 0003 ED00            AORG >ED00
 0004 ED00 02E0       LWPI >ED40
      ED02 ED40
 0005 ED04 04C0       CLR 0
 0006 ED06 0580      LOOP INC 0
 0007 ED08 0280       CI 0,>EE00
      ED0A EE00
 0008 ED0C 11FC       JLT LOOP
 0009 ED0E 2FA0       XOP @MSG,14
      ED10 ED16
 0010 ED12 0460       B @EVMBUG
      ED14 0080
 0011 ED16   50      MSG TEXT 'PROGRAM EXECUTING'
      ED17   52
      ED18   4F
      ED19   47
      ED1A   52
      ED1B   41
      ED1C   4D
      ED1D   20
      ED1E   45
      ED1F   58
      ED20   45
      ED21   43
      ED22   55
      ED23   54
      ED24   49
      ED25   4E
      ED26   47
 0012 ED27   00       BYTE 0
 0013                 END
NO ERRORS,      NO WARNINGS

  [] CC
  COPY/CONCATENATE
   INPUT ACCESS NAME(S): GPS.TSTL  GPS.TSTO
   OUTPUT ACCESS NAME: ST17  CS03
   REPLACE?: NO
   MAXIMUM RECORD LENGTH:
  [] Q
  QUIT
  RUNTIME TASK ID = >5E
  MON? IR
```

SAMPLE SESSION (Continued)

```
W=0430
P=EC00  EC00
MON? DM ED00 ED28
ED00=02E0  ED40  04C0 0580    0280  EE00  11FC  2FA0
ED10=ED16  0460  0080  5052    4F47  5241  4D20  4558
ED20=4543  5554  494E  4700    0003
MON? EX


MON? DM ED00 ED28
ED00=02E0  ED40  04C0  0580    0280  EE00  11FC  2FA0
ED10=ED16  0460  0080  5052    4F47  5241  4D20  4558
ED20=4543  5554  494E  4700    0003
MON? IR

W=5554
P=556C  ED00
MON? EX
PROGRAM EXECUTING
```

SECTION 8

PROGRAMMING

8.1 GENERAL

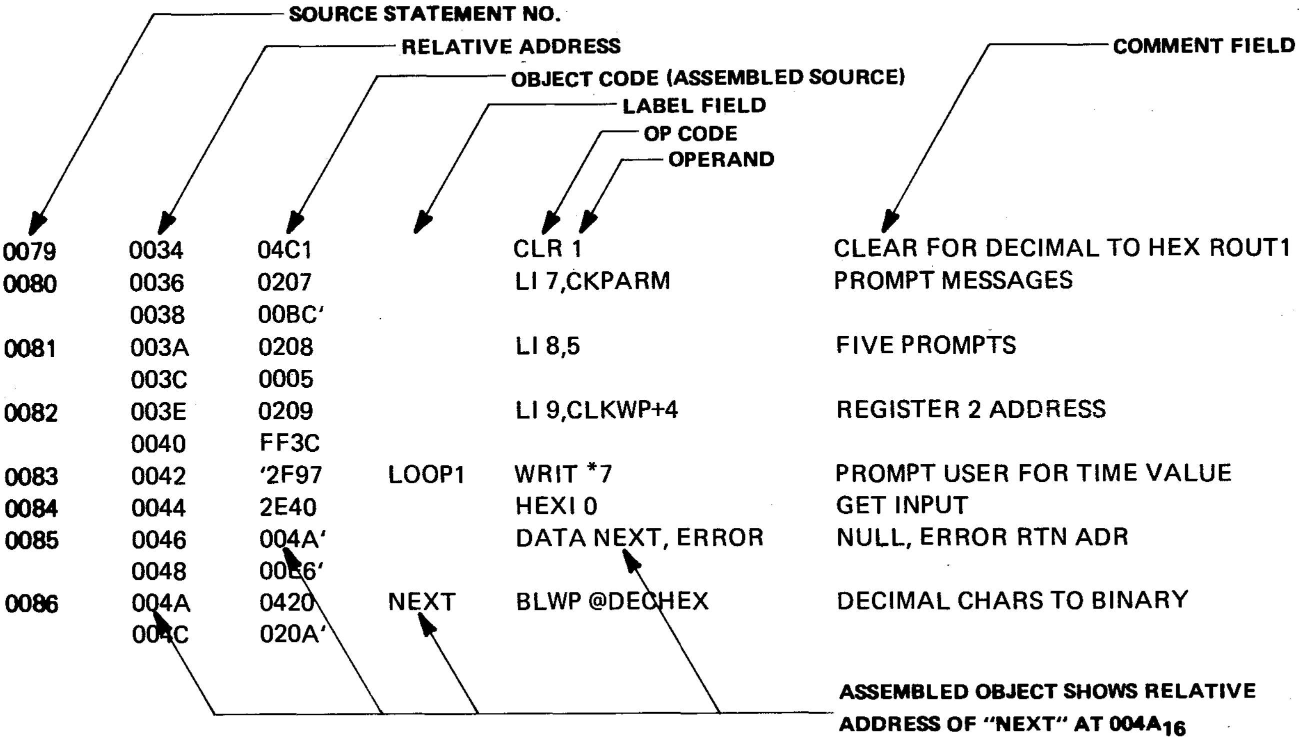
This section is designed to familiarize the user with programming the TMS 9995. Explanations of the programming environment, using EVMBUG XOPs, supporting special features of the hardware, and certain programming practices are included. Programs are provided as examples for the the user to analyze and follow, and possibly to combine into the user´s system. This section is divided into two general areas: the first gives background information on the programming environment and shows suggested coding practices for a variety of situations. The second part gives specific program examples using special features of the hardware.

For clarity, source listing examples in this section use assembler directives recognized by larger assemblers, but not recognized by the TMS 9995 Symbolic Assembler. These directives are explained in detail in the "Model 990 Microprocessor Assembly Language Programmer´s Guide". A synopsis of the definitions is presented in Table 8-1.

TABLE 8-1. ASSEMBLER DIRECTIVES USED IN EXAMPLES.

Label	Opcode	Operand	Meaning
	AORG	XXXX	Assemble code that follows so that it is loaded beginning at memory address XXXX this is similar to the absolute load / request of the symbolic assembler.
	DATA	YYYY	Place the value YYYY in this location (if preceeded by the greater-than sign (>), the quantity is a hex representation.
	DATA	LABEL	If LABEL represents a memory address, the memory address value is placed at this location, aligned on an even address (word boundary).
	END		Signifies end of program for assembler.
AAAA	EQU	BBBB	Wherever the symbol AAAA is found, substitute the value BBBB.
	IDT	´NAME´	Program will be identified by NAME.
	TEXT	´ABCD123´	The ASCII value of the specified character string is assembled in successive bytes.

Figure 8-1 is part of a source listing used in this section, as assembled by TI´s TXMIRA assembler. Unless specified otherwise by directive, the TXMIRA assembler will begin assembling code relative to memory address >0000 (second column). When resolving an address for an instruction, as shown at the bottom of Figure 8-1, the instruction address operator is the same as the relative address in column two of the listing. Thus, for the label NEXT, the address >004A is assembled, which is the relative address within the listing. This is useful when determining such addresses as the destination of a labelled BLWP instruction. Note that the symbolic assembler does not use labelled addressing, but assembles the absolute address given.



SOURCE STATEMENT NO.

RELATIVE ADDRESS

OBJECT CODE (ASSEMBLED SOURCE)

LABEL FIELD

OP CODE

OPERAND

COMMENT FIELD

```
0079   0034   04C1            CLR 1                CLEAR FOR DECIMAL TO HEX ROUT1
0080   0036   0207            LI 7,CKPARM          PROMPT MESSAGES
       0038   00BC'
0081   003A   0208            LI 8,5               FIVE PROMPTS
       003C   0005
0082   003E   0209            LI 9,CLKWP+4         REGISTER 2 ADDRESS
       0040   FF3C
0083   0042   '2F97   LOOP1   WRIT *7              PROMPT USER FOR TIME VALUE
0084   0044   2E40            HEXI 0               GET INPUT
0085   0046   004A'           DATA NEXT, ERROR     NULL, ERROR RTN ADR
       0048   00E6'
0086   004A   0420    NEXT    BLWP @DECHEX         DECIMAL CHARS TO BINARY
       004C   020A'
```

ASSEMBLED OBJECT SHOWS RELATIVE ADDRESS OF "NEXT" AT $004A_{16}$

FIGURE 8-1. SOURCE LISTING.

8.2 PROGRAMMING CONSIDERATIONS.

8.2.1 Program Organization

Programs should be organized into two major areas:

- Prodecure area of executable code and data constants (never modified)

- Data area of program data and work areas whose contents will be modified.

The executable code and constant data section can be debugged as a separate entity, and then programmed into EPROM. The work area can be placed at any address in RAM, and that address does not have to be contiguous with the program code area, and can even be dynamically allocated by a Get Memory supervisor call of some kind. Even if the program parts are loaded and executed together, the organization and debug ease are enhanced.

In this programming section all example programs are coded, with one exception, in this manner: the work area is the register set, which is arbitrarily fixed to a RAM address. The one exception, the Two-Terminal routine, is coded to reside entirely in RAM because the workspace is a part of the contiguous extent of code. This method of coding is used in RAM-intensive systems because the operating system need not manage workspaces as might be necessary in a system with very little RAM.

8.2.2 Executing TMS 9995 System Programs On the TMS 9995 EVM

On the TMS 9995 EVM, all interrupt and XOP vectors are programmed, and a linking scheme in RAM is used as detailed in subsection 8.9.

8.2.3 Required Use Of RAM In Programs

All memory locations that will be written to must be in RAM-type memory (this is important to consider when the program is to be programmed into EPROM. Areas to be located in RAM include all registers, as well as the destination operands of Format 1 instructions and the source operands of most Format 6 instructions.

For example, in the following source lines:

```
          MOV     @>0700,@>ED00      MOVE DATA
          CLR     @>ED00             CLEAR MEMORY ADDRESS
          ABS     @>ED00             SET TO ABSOLUTE VALUE
          INCT    @>ED00             INCREMENT BY TWO
          S       R1,@>ED00          >ED00 - R1, ANSWER IN >ED00
```

the address >ED00 will be written to; thus, it has to be in RAM.

8.3 PROGRAMMING ENVIRONMENT

The programming environment of a computer is loosely defined as the set of conditions imposed on a programmer by either the hardware or the system software or both, and the facilities available to the programmer because of the design of the hardware and software. The environment in which a program resides usually determines how that program is coded. The following paragraphs give explanations of the major areas of the TMS 9995 EVM from a programmer's point of view. Note that all program examples given are for a full assembler (e.g. SDSMAC) and not necessarily for the symbolic assembler. Thus labels can be used for reader comprehension.

8.3.1 Hardware Registers

The TMS 9900 family of processors are designed around a memory-to-memory architecture philosophy; consequently, the only hardware registers inside the processor affecting the programmer are the Workspace Pointer (WP) Register, the Program Counter (PC) Register, and the Status (ST) Register There are no dedicated accumulators or general purpose registers physically residing inside the microprocessor. All manipulation of data is accomplished by using these three registers as described below.

8.3.1.1 Workspace Pointer Register (Wp)

The Workspace Pointer is the register that holds the address of a sixteen-word area in memory; this memory area serves as a general-purpose register set. A memory area is designated as a workspace or general-purpose register set by loading the address of the first word (Register 0) of the 16-word space into the WP Register. Thus, the programmer´s register set is in memory, and can be referred to with register addressing, or if the WP value is known, with memory addressing. The registers are simply a d area in a program with the special privileges usually given to processor registers. This approach has several advantages for the programmer:

1. Register save areas need no longer be kept in programs, since the actual program registers are already in memory, and are maintained by the hardware during program linking by the use of a special class of instructions.

2. Program debugging is greatly enhanced, since the registers of questionable program remain intact in memory during debugging. The debug monitor has its own set of registers in memory, and there is no question of which of many program modules has tampered with the processor registers, since each program in question can have its own registers.

3. Recursive, re-entrant, and EPROM resident code is much easier to write, since program calls are handled by special instructions and new workspace areas, linked together by hardware, are available for use at each program call.

4. Linked-list structuring of workspaces is automatically done by hardware, reducing system software overhead.

5. Very fast interrupt handling is possible, since only three processor registers (WP,PC,ST) rather than a whole register set, are stored by the hardware during the interrupt, usually by a software instruction or routine.

8.3.1.2 Program Counter Register (PC)

The Program Counter (PC) Register holds the address of the next instruction to be executed by the processor. As such, it is no different than the PC in any other processor and is incremented while fetching instructions, unless modified by a program branch or jump, or during an interrupt sequence.

8.3.1.3 Status Register (ST)

The Status Register holds the processor status and is the only one of the three processor registers which has nothing to do with memory directly. It is divided into two parts:

(1) The status bits, which are set to reflect the attributes of data being handled by the processor.

(2) Interrupt mask, which governs the priority structure of interrupt processing.

The ST is organized as shown in Figure 8-2.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ST0	ST1	ST2	ST3	ST4	ST5	ST6	ST7	ST8	ST9	ST10	ST11	ST12	ST13	ST14	ST15
L>	A>	EQ	C	OV	OP	X	*	*	*	OVEN	*	INTERRUPT MASK			

L>	LOGICALLY GREATER THAN	OV	OVERFLOW
A>	ARITHMETICALLY GREATER THAN	OP	ODD PARITY
EQ	EQUAL	X	XOP BEING EXECUTED
C	CARRY	OVEN	OVERFLOW ENABLE

FIGURE 8-2. STATUS REGISTER.

8.3.2 Address Space

The TMS 9995 microcomputer addresses 65,536 (64K) bytes of 8-bits each. Although the data bus is 16 bits wide, and the instruction set is mainly word (16-bit) oriented, the basic unit of address is a byte. The actual memory architecture is 32,768 (32K) words of two bytes each, and byte processing is accomplished within the processor after fetching a word from memory. Because the instruction set is mainly arithmetically oriented and usually operates on 16-bit words, view the address space as a collection of words, each containing two bytes.

8.3.3 Vectors (Interrupt and XOP)

Interrupt and XOP vectors are located beginning with address >0000, and extend through >007F. The first part, addresses >0000 through >0013, contain the interrupt vectors. There are 7 prioritized interrupts. Level 0 is the highest priority, with a vector pair at >0000 and >0002. Level 4 is the lowest priority, with its vector pair at >0010 and >0012. Level 0 interrupt is synonomous with the RESET function. A vector pair consists of a workspace pointer and a program counter, both values identifying the interrupt program environment.

Before an interrupt can occur, the processor must recognize it as having an equal or higher priority than the interrupt mask in the Status Register. After a valid interrupt has occurred, the interrupt vector values are retrieved from memory, and the hardware equivalent of a BLWP instruction takes place.

There is one additional vector pair, at >FFFC and >FFFE, for the NMI interrupt. When signaled, this interrupt always occurs and cannot be disabled by the Status Register interrupt mask. Note also that RESET being level zero, cannot be disabled, since its Status Register priority value of zero is always equal to or higher than any value in the interrupt mask field.

The XOP vectors work in a similar manner. Vector location begins at >0040 and extend through >007F.These vectors are triggered by execution of the XOP instruction, with a number from 0 to 15. There are no priority-setting interrupts, and XOP service routines may freely execute other XOPs. One additional event occurs during the vector action: the source operand of the XOP instruction is evaluated as an address and placed in the new workspace Register 11. This provides a parameter to the XOP routine.

The EVMBUG monitor uses several XOPs for I/O service from the terminal; some of these are available for the user, as explained in paragraph 8.2. In addition, the programmer may wish to program interrupt and XOP vectors for special functions.

8.3.4 Workspace Registers

The actual workspace registers, in memory, provide general working areas for a program. Some registers can also be used for special purposes; these are listed in Table 8-2.

In general, Registers 2 to 10 are available for unrestricted use, although the programmer can use the reserved registers for other purposes if proper consideration is given.

One advantage of the workspace concept is that one program can request an almost unlimited number of register sets, or alternatively, every module in a program system can have at least one set of its own registers. Programs are usually written to take advantage of the benefits associated with program operands in registers.

TABLE 8-2. REGISTER RESERVED APPLICATIONS.

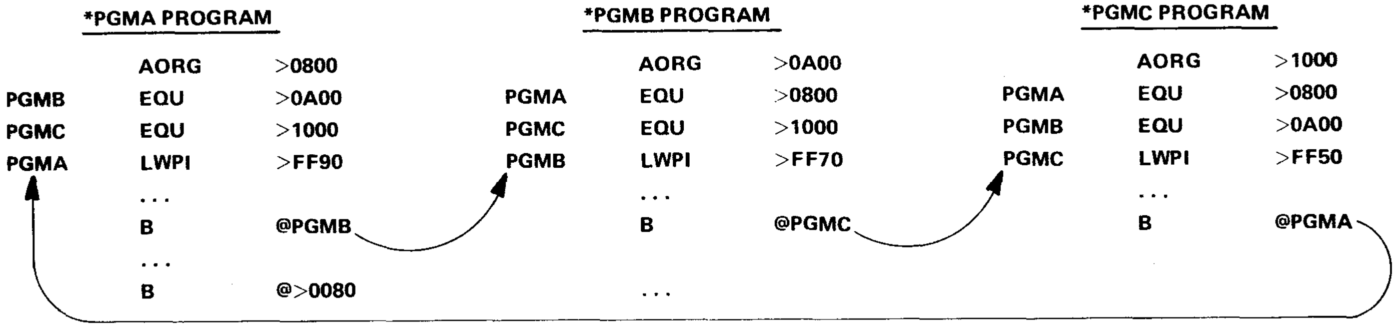
Register	Application
0	Bits 12-15 (Least significant nibble) provide the shift count for shift instructions coded to refer to this register. Register 0 is also used for operands signed multiply and signed divide instructions. This register cannot be used for indexed addressing.
1	Used for operands of signed multiply and signed divide instructions.
11	Holds return address following execution of a BL instruction. During XOP service routine, it holds the resolved memory address of argument in XOP instructon.
12	CRU base address.
13	During BLWP, RTWP, interrupts and XOPs holds old WP contents.
14	During BLWP, RTWP, interrupts and XOPs holds old PC contents.
15	During BLWP, RTWP, interrupts and XOPs holds old ST contents.

8.4 LINKING INSTRUCTIONS

These instructions are of vital interest to a programmer, since they solve the problem of how to get in and out of a program. These instructions are:

- B	BRANCH
- BL	BRANCH with return link in R11
- BLWP	BRANCH, new workspace, return link in R13 to R15
- RTWP	RETURN, uses vectors in R13 and R14
- XOP	BRANCH, new workspace, vectors in low memory

Though not normally considered a program linking instruction, the BRANCH instruction can be used to link programs in a specific location, such as the start of EVMBUG. Since the Workspace Pointer is not affected by the instruction, program systems using this convention usually delegate the responsibility for establishing workspaces to each program. Thus, we may have branches to various programs, as shown in Figure 8-3. Note that each program sets up its own WP (LWPI instruction). The AORG and EQU directives are explained in paragraph 8.1.



```
        *PGMA PROGRAM

        AORG    >0800
PGMB    EQU     >0A00
PGMC    EQU     >1000
PGMA    LWPI    >FF90
        ...
        B       @PGMB
        ...
        B       @>0080
```

```
        *PGMB PROGRAM

        AORG    >0A00
PGMA    EQU     >0800
PGMC    EQU     >1000
PGMB    LWPI    >FF70
        ...
        B       @PGMC
        ...
```

```
        *PGMC PROGRAM

        AORG    >1000
PGMA    EQU     >0800
PGMB    EQU     >0A00
PGMC    LWPI    >FF50
        ...
        B       @PGMA
```

FIGURE 8-3. EXAMPLE OF SEPARATE PROGRAMS JOINED BY BRANCHES TO BRANCHES TO ABSOLUTE ADDRESSES.

8.4.1 BL (Branch and Link) Instruction

The BL instruction is designed mainly for the calling of subprograms with a convenient means of returning back to the calling program. Since the processor puts the address of the next instruction in Register 11 (it effectively transfers the PC to R11) before branching, the return path is established. To return (using the same workspace), simply execute a B *R11 (or RT instruction).

Note, however, that only one level of subroutine call is possible if only one workspace area is used, unless Register 11 is saved by the first subroutine wishing to branch and link to a second routine.

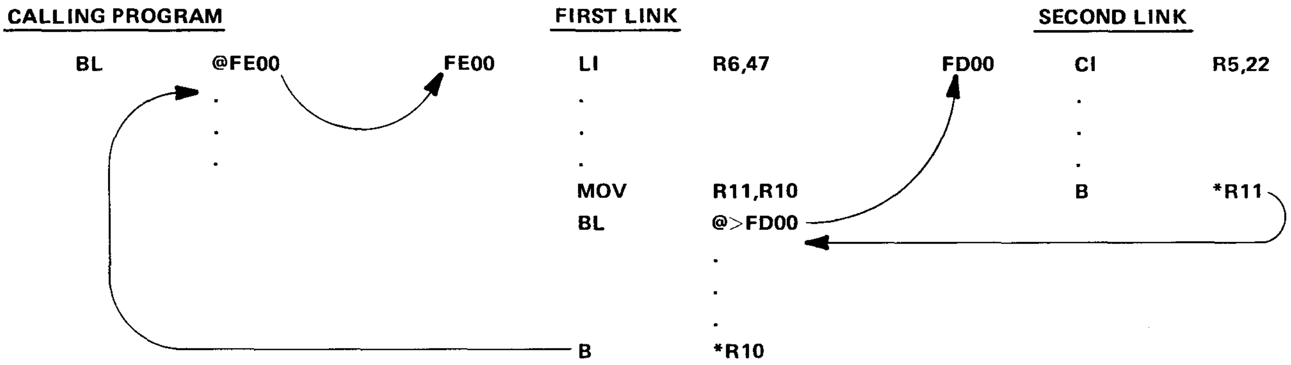


FIGURE 8-4. BRANCH AND LINK SUBROUTINE.

The BL subroutine can include XOP instructions to provide special services needed to accomplish the subroutine function, as in the following example :

```
 CALLING PROGRAM                         SUBROUTINE

                          RDNUM   XOP    R1,13          READ A CHARACTER
    BL @RDNUM                     CI     R1,>3000       IS IT BELOW A ZERO?
         .                        JL     RDNUM          YES, GO BACK
         .                        CI     R1,>3900       IS IT ABOVE A NINE?
         .                        JH     RDNUM          YES, GO BACK
         .                        XOP    41,12          ECHO THE CHARACTER
                                  B      *11            RETURN
```

The very simple routine shown above reads a character from the terminal and checks for a decimal digit 0-9. If the character is acceptable, it is echoed back to the terminal, and then control is returned to the calling program. If the character is unacceptable, the routine drops it and requests another; the bad character is not echoed to show the user that another character must be typed.

8.4.2 BLWP (Branch and Load Workspace Pointer) Instruction

This is the most sophisticated linking instruction. It causes a complete program environment change (context switch), automatically links the old workspace to the new, and also preserves the old processor status. As such, BLWP behaves in the same way as the interrupt sequence or XOP sequence, and it is therefore possible to vector to an interrupt or XOP service routine without actually causing an interrupt or executing an XOP. For example, executing a BLWP @0 will vector to the RESET interrupt handler, which if EVMBUG is resident, causes the user to set the baud rate and start EVMBUG.

The TMS 9995 is a linked-list rather than a stack machine. Programmers used to a stack for systems programming may need some readjustment of thinking, but the superior flexibility of linked-lists is simplified by the fact that the programmer can move nodes around, whereas in a stack, the nodes are fixed in Last-In First-Out (LIFO) order. The transition can be made easily, since the hardware completes program linking with the execution of one instruction, and very little effort is required on the part of the programmer.

There are two immediate possibilities to discuss in using the BLWP instruction. For simple subroutine linking the following is an example:

```
         CALLING PROGRAM                         SUBROUTINE

    ENTRY           .
                    .
                    .                                .
              BLWP @SUBA          PCSUBA             .          ENTRY POINT
                                                     .
                                                   RTWP
    SUBA      DATA WPSUBA         WPSUBA             .
              DATA PCSUBA                            .
                                                     .
```

Note the double word vector pointed to by the BLWP operand, the values of WPSUBA and PCSUBA. These two Data statements provide the memory addresses of these vectors. The latter (PCSUBA) is the entry point, and is well defined. However, the WP value is shown here without a definition. This raises a fundamental question: if there are many programs operating together (such as EVMBUG, possibly a user-written monitor, and a collection of application programs and subroutines), who is responsible for managing the workspaces? If each individual program is responsible, then the following definition would be added to the above subroutine:

```
     WPSUBA     EQU             >ED70
```

Note that this defines WPSUBA as M.A. >ED70, and ties down one area of memory to the subroutine; no other program in the system can call this subroutine without chancing some conflict by using the same workspace. Thus, the memory area is reserved for one subroutine.

A second approach is to code a value which is designated as a common workspace for whichever program is in control at the time. In the EQU statement above, the value could be (by agreement) the common workspace. This implies that there are now two entities:

(1) The reserved workspace, which must be carefully mapped out ahead of time so that there is no overlap.

(2) The common workspace (of which there may be more than one), whose status is such that any program can use it when it is not already in use.

NOTE: The previous discussion assumes that the program code is in EPROM. If the code is to be executed from RAM, then writing the program is simple: put the workspace at the end of the program as a data area.

In either case, the user is responsible for partitioning his memory so that user-defined workspaces do not overlap or interfere with EVMBUG or the XOPs defined by EVMBUG, or with each other.

8.4.3 RTWP (Return With Workspace Pointer) Instruction

The RTWP instruction can be used to both return from a program, and to link to a program. Because the instruction reloads the processor WP, PC, and ST Registers from Workspace Registers 13, 14, and 15, the contents of these registers govern where control will go. If those registers were initialized by a BLWP instruction, then the action can be seen as a Return; if special values are placed in these registers, the action can be viewed as a subroutine call. Program calls are not limited to a nesting structure, as in stack architecture, but are generalized so that chains and even rings may be formed. The EVMBUG monitor uses the RTWP instruction in this manner: using the "IR" command, the user fills EVMBUG's registers 13, 14, and 15. Using the "EX" command causes EVMBUG to execute a RTWP instruction using the values in these registers.

Since the RTWP does not affect the new workspace at all, there is no way for the called program to return to the caller, unless the caller had initalized the new workspace registers before executing the RTWP. This type of program transfer is in a "forward" direction only, and is usually suitable only for a monitor program in a fixed location, such as EVMBUG.

8.4.4 XOP (Extended Operation) Instruction

The XOP instruction works almost like a BLWP instruction, except that the address containing the double-word vector area is between >0040 and >007F, and is selected by an argument of from 0 to 15, and that register 11 of the new workspace is initialized with the fully resolved address of the first operand of the XOP instruction. This means that if the operand is a register, the actual memory address is computed and placed in the new register 11.

The XOP instruction is meant as a "supervisor call" or special function operation. As such, a programmer might wish to implement routines which perform some standard process, such as a character string search.

EVMBUG supplies definitions for XOPs 8 through 15, leaving 0 through 7 available for the user. XOPs 0 through 7 are programmed as described in paragraph 8.9.

8.4.5 Linked-Lists

A linked-list is a system of data organization wherein a collection of related data, called a node, contains information which links it to other nodes. A prime example is a workspace register set. It contains 16 words of data. If there are many workspaces present at one time connected by BLWP instructions, then every register 13 will contain the address of the previous workspace, forming a linked list. At the same time, the BLWP also places the previous Program Counter value in Register 14, providing a means of returning back to the previous program environment.

For example, the "XE" or execute EVMBUG command uses the RTWP instruction to begin program execution of the WP, PC and St Registers values in current Registers 13, 14 and 15. The "IR" or Register Inspect/Change EVMBUG command can be used to set up these registers prior to the execute command. In the example in Figure 8-5, program PGMA is executed using the EVMBUG "EX" command; it later gives control to program PGMB using the BLWP command. In doing so, the processor forges links back to PGMA by placing return WP, PC and ST values in Registers 13, 14 and 15 of PGMB. Likewise, PGMB branches to PGMC with return links to PGMB forged into R13 to R15 of PGMC. Each can return to the previous program by executing an RTWP instruction, and the processor can travel up the linked list until PGMA is reached again.

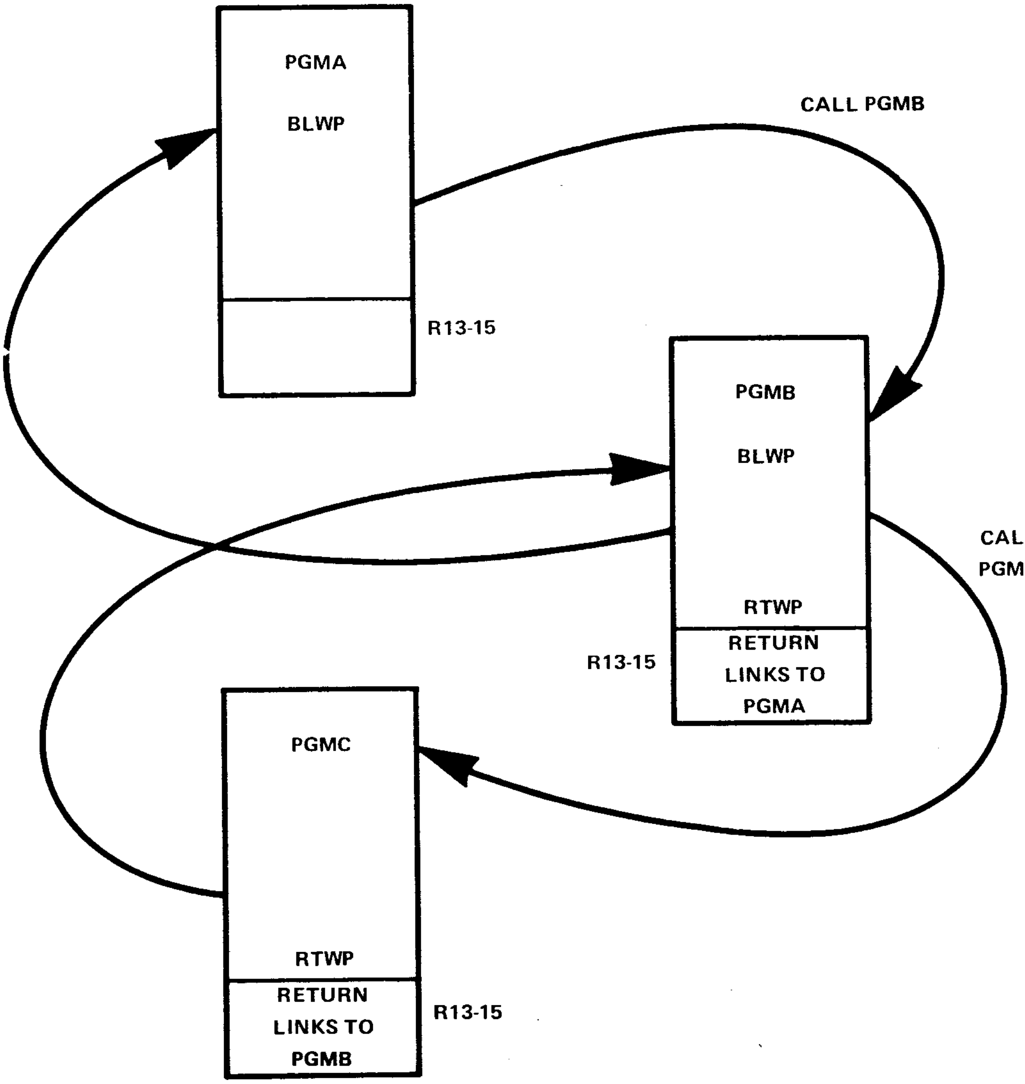


FIGURE 8-5. LINKED-LIST EXAMPLE.

8.5 COMMUNICATIONS REGISTER UNIT (CRU)

The CRU is an instruction (software) driven bit-oriented I/O interface that is separate from the memory interface. The CRU of the TMS 9995 can directly address, in bit fields of one to sixteen bits, up to 32768 input bits and 32768 output bits.

8.5.1. CRU Addressing

The CRU bit address is the value as seen on address lines A0 to A14. These 15 lines allow addresses from 0 to 32767. In other words, the CRU bit addressing scheme allows the user to addresss up 32768 distinct CRU entities (CRU "bits"). For example, the large address decoder monitoring these lines could enable up to 32768 devices through the Address Bus. CRU bit addresses for CRU devices on the TMS 9995 EVM are listed in Table 8.3.

TABLE 8-3. TMS 9995 EVM BOARD PREDEFINED CRU ADDRESSES.

Function	CRU Bit Address (Address Lines) (A0 to A14)	CRU Base Address (R12 Bits 0-15)
TMS 9902, Main I/O (lower half)	0000	0000
TMS 9902, Main I/O (upper half)	0010	0020
TMS 9902, Auxiliary I/O (lower half)	0200	0400
TMS 9902, Auxiliary I/O (upper half)	0210	0420
9995 CRU Flag Register	0F78	1EF0
9995 MID Flag	0FED	1FDA

8.5.1.1 CRU Bit Address And Register 12

When any of the five CRU instructions is executed, the CRU bit address plus a displacement (TB, SBO and SBZ only) are active on address lines A0 to A14. This address is obtained from 15 bits of Register 12 (R12), bits 0 through 14. Note that only 15 of the 16 bits of R12 are used, with bit 15 ignored.

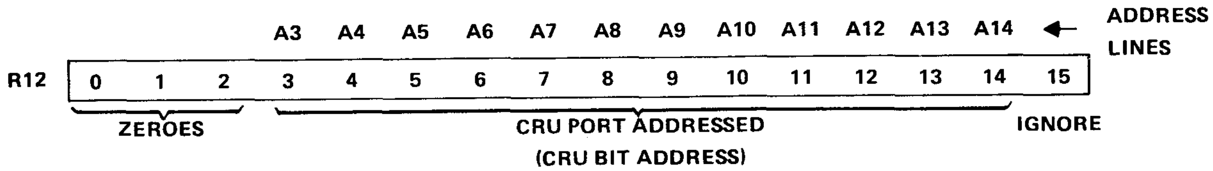


FIGURE 8-6. CRU ADDRESS IN REGISTER 12 vs ADDRESS BUS LINES.

Because bit 15 of R12 is not used, some confusion can result while programming. Instead of loading the CRU address in bits 1 to 15 of Register 12, e.g., LI R12,>0200 to address the Port 2 TMS 9902 at CRU address >0200, the programmer must shift the base address value one bit to the left so that it is in bits 0 to 14 instead of bits 1 to 15. Several programming methods can be used to ensure this correct placement, and all of the following examples place the TMS 9902 bit address of >0200 correctly in R12:

```
(1)  LI     R12,>0400        PLACES 200 IN BITS 0 TO 14

(2)  LI     R12,>0200*2      MULTIPLY BASE ADDRESS BY 2 (NOT
                             RECOGNIZED BY LINE-BY-LINE ASSEMBLER)

(3)  LI     R12,>0200        BASE ADDRESS IN BITS 1 TO 15
     SLA    R12,1            SHIFT BASE ADDRESS ONE BIT TO LEFT
```

From a programming standpoint, it may be best to view addressing of the CRU through the entire 16 bits of R12. In this context, blocks of a maximum of 16 CRU bits can be addressed, and in order to address an adjacent 16-bit block, a value of >20 must be added or subtracted from R12. For example, with R12 containing >0000, CRU bits >10 to >1F can be addressed by adding >20 to R12.

8.5.2 CRU Instructions

The five instruction that use the CRU interface are:

- LDCR — Place the CRU bit address on address lines A0 to A14. Load from memory a pattern of 1 to 16 bits and serially transmit this pattern through the CRUOUT pin of the TMS 9995. Increment the address on A0 to A14 after each CRUOUT transmission.
- STCR — Place the CRU bit address on address lines A0 to A14. Store into memory a pattern of 1 to 16 bits obtained serially at the CRUIN pin of the TMS 9995. Increment the address on A0 to A14 after each CRUIN sampling.
- SBO — Place the CRU bit address plus the instruction´s signed displacement on address lines A0 to A14. Send a logical one through the CRUOUT pin of the TMS 9995.
- SBZ — Place the CRU bit address plus the instruction´s signed displacement on address lines A0 to A14. Send a logical zero through the CRUOUT pin of the TMS 9995.
- TB — Place the CRU bit address plus the instruction´s signed displacement on address lines A0 to A14. Sample the CRUIN pin of the TMS 9995 and place the bit read into ST2, the equal bit of the Status Register.

8.5.3.1 CRU Multibit Instructions

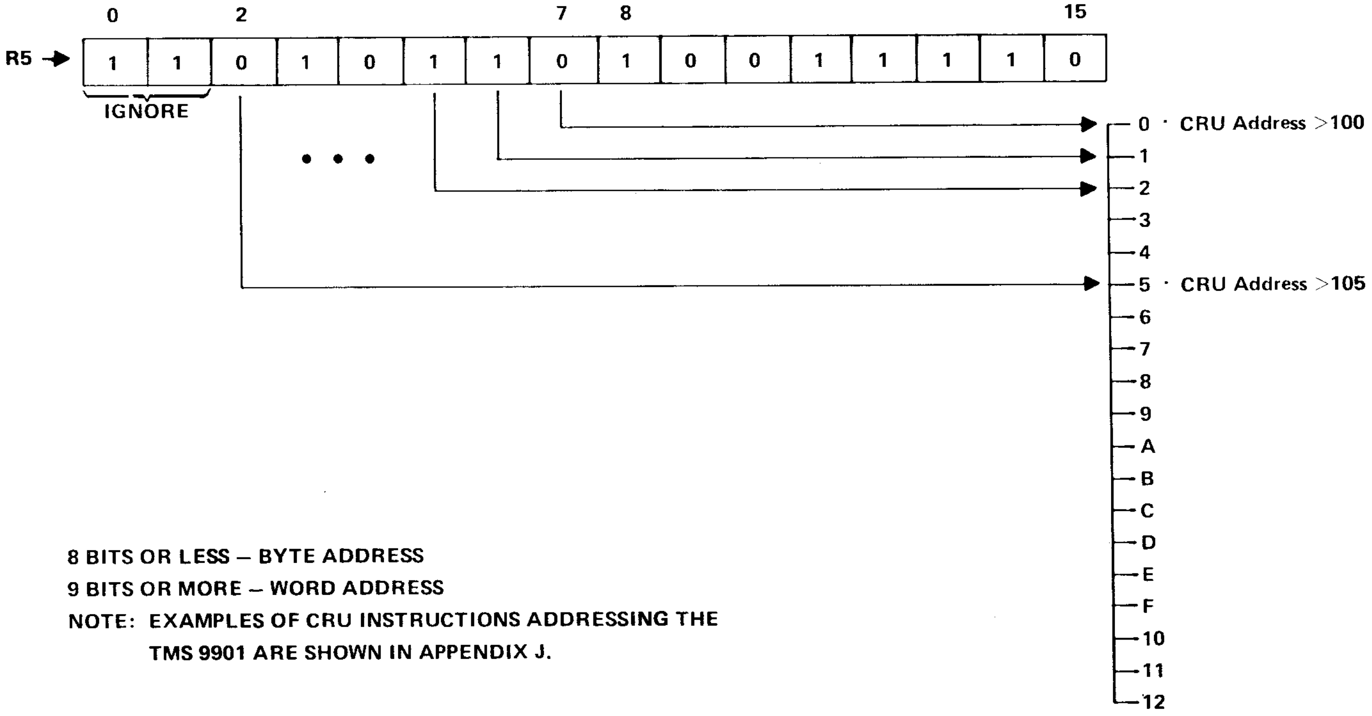
The two multibit instructions, Load Communications Register (LDCR) and Store Communications Register (STCR) address the CRU devices by placing bits 0 through 14 of CRU bit address Register 12 on address lines A0 through A14. The first operand addresses the source field or receiving field, and the second operand supplies the length of the operation.

If the length is coded as from 1 through 8 bits, only the left byte of the source or receiving field takes part in the operation, and bits are shifted in or out from the least significant bit of that left byte. Thus, an instruction: LDCR R2,1: outputs bit 7 of R2 to the CRU at the address derived from Register 12. An instruction : STRC R5,2: would receive two bits of data serially and insert them into bit 7 and then bit 6 of Register 5. The CRU address lines are automatically incremented to address each new CRU bit, until the required number of bits are transferred. In an STCR instruction, unused bits of the byte or word are zeroed. In this last example, bits 0-5 are zeroed; the right bit is unaffected.

An LDCR loads the CRU device serially from memory. An STCR stores data into memory obtained serially from the addressed CRU device. Figures 8-7 and 8-8 show this operation graphically.

```
LI      R12,>200        LOAD CRU BASE ADDRESS >100 IN BITS 4 TO 14 OF R12

LDCR        R5,6        6 BITS TO CRU
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	>020C
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	>0200
0	0	1	1	0	0	0	1	1	0	0	0	0	1	0	1	>3185



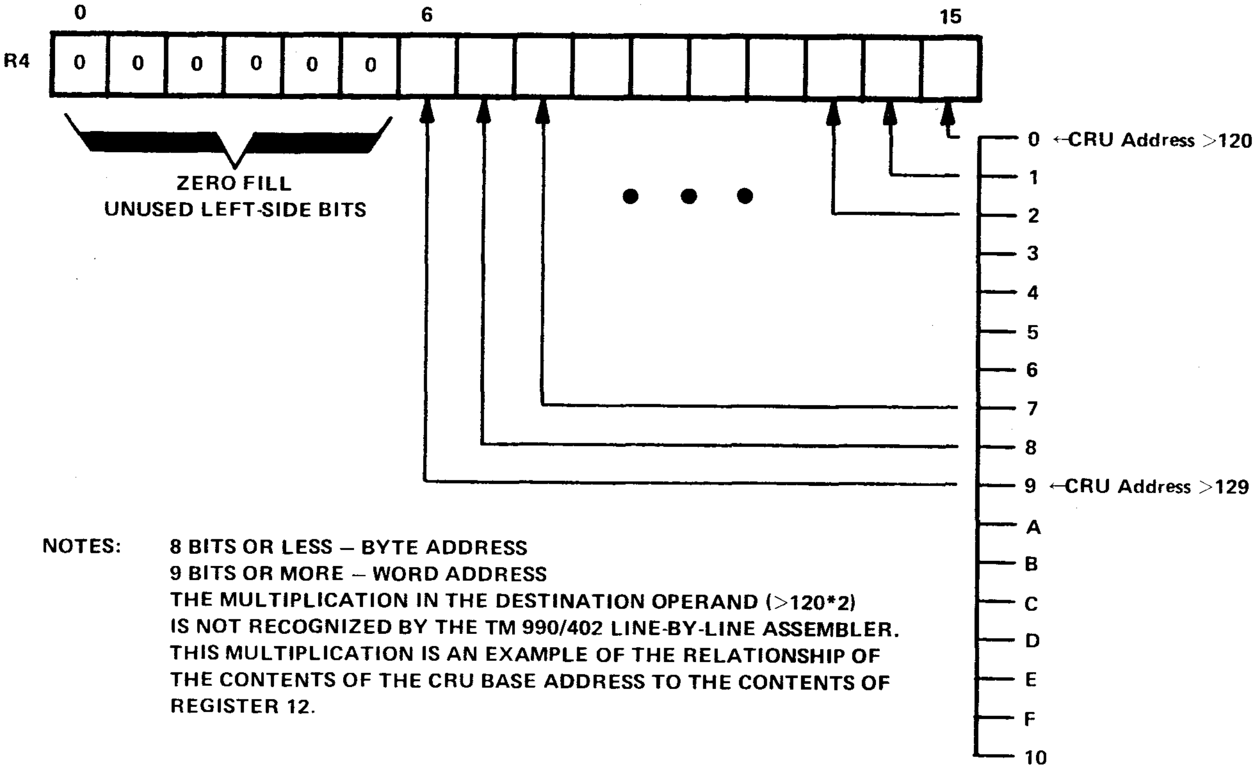
	0		2					7	8							15
R5 →	1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	0

8 BITS OR LESS – BYTE ADDRESS
9 BITS OR MORE – WORD ADDRESS
NOTE: EXAMPLES OF CRU INSTRUCTIONS ADDRESSING THE
TMS 9901 ARE SHOWN IN APPENDIX J.

FIGURE 8-7. LDCR INSTRUCTION.

```
LI
R12,>120*2
LOAD CRU BASE ADDRESS >120 IN BITS 4 TO 14 OF R12
STCR
R4,10
10 BITS FROM CRU TO R4
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	>020C
0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	>0240
0	0	1	1	0	1	1	0	1	0	0	0	0	1	0	0	>3684



NOTES: 8 BITS OR LESS – BYTE ADDRESS
9 BITS OR MORE – WORD ADDRESS
THE MULTIPLICATION IN THE DESTINATION OPERAND (>120*2) IS NOT RECOGNIZED BY THE TM 990/402 LINE-BY-LINE ASSEMBLER. THIS MULTIPLICATION IS AN EXAMPLE OF THE RELATIONSHIP OF THE CONTENTS OF THE CRU BASE ADDRESS TO THE CONTENTS OF REGISTER 12.

FIGURE 8-8. STCR INSTRUCTION.

8.5.3.2 CRU Single-bit Instructions

The three single-bit instructions are (1) Set Bit To Zero (SBZ), (2) Set Bit To One (SBO), and Test Bit (TB). The first two are output instructions, and the last is an input instruction. All three instructions have only one operand, which is an eight-bit displacement to be added to the contents of R12 to provide the address for the desired bit. The SBZ instruction sets the addressed bit to zero (CRUOUT of zero), and the SBO instruction sets the addressed bit to one (CRUOUT of one). The TB instruction reads the addressed bit (samples CRUIN) and places it directly into bit 2 (EQ) of the Status Register for testing with JEQ and JNE instructions.

The displacement is treated as a signed, eight-bit number, and thus, has a range of values of -128 to +127. This number is added to the CRU bit address derived from bits 0 to 14 of Register 12, and the result is placed on the address lines. This process is illustrated in Figure 8-9.

Notice that after execution of a TB instruction, A JEQ instruction will cause a jump, if the bit value is a one, and the JNE will cause a jump if a zero.

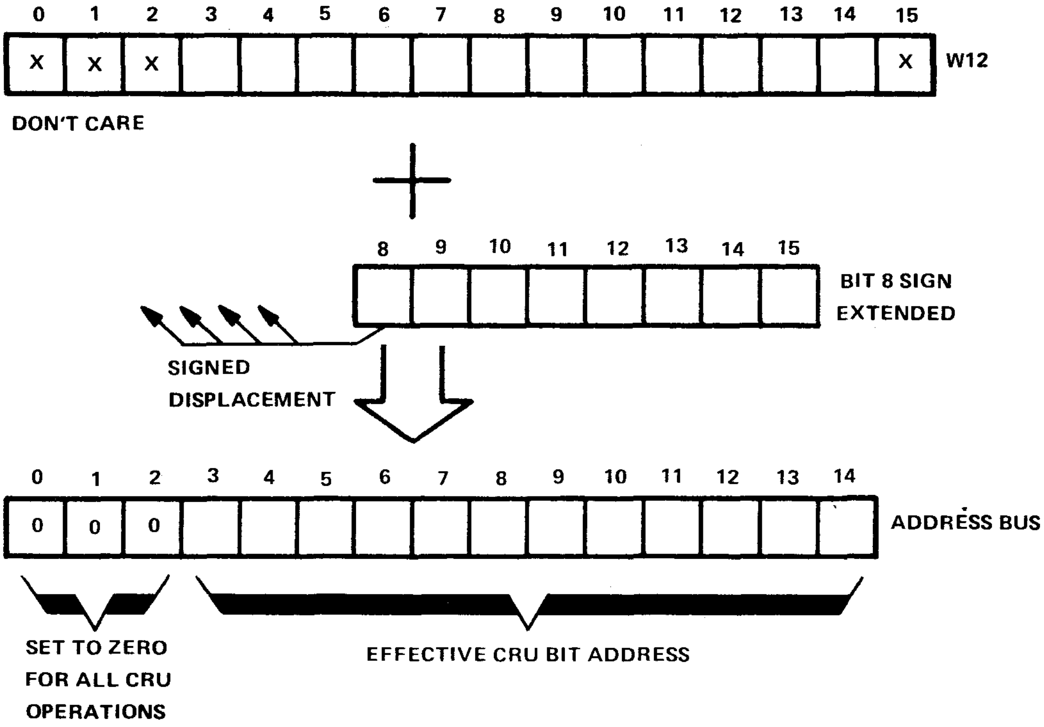


FIGURE 8-9. ADDITION OF DISPLACEMENT AND R12 CONTENTS TO CRU BIT ADDRESS.

8.6 DYNAMICALLY RELOCATABLE CODE

Most programs written for the TMS 9995 will contain references in memory. These references are given by means of a symbolic name preceeded by an at (@) sign. Examples are: @>ED00 (memory address >ED00, recognized by the LBLA) or, @SUM (recognized by a symbol-reading assembler, not the LBLA).

For example, a short program, located at M.A. >0900 to >090F, adds two memory addresses, then branches to the monitor:

```
0900     MOV      @>090C,R1     MOVE VALUE AT M.A. 090C TO R1
0904     A        @>090E, R1    ADD VALUE AT M.A. 090E TO R1
                                  (R1=ANSW)
0908     B        @>0080        RETURN TO MONITOR
090C     DATA     100           FIRST NUMBER
090E     DATA     200           SECOND NUMBER
```

In this program, a number in EPROM is moved to a register in RAM, and another number in EPROM is added to that register (the destination of an add must be in RAM in order for the sum to be written into it). If it is desired to move this entire program to another address (such as to RAM for debugging purposes to allow data changes as desired), then the locations in the code must be changed to reflect the new addresses. For example, to relocate the above example to start at address >ED00, each of the addresses of the numbers must be changed before the program can execute; otherwise, the program will try to access numbers in M.A. >090C and >090E When they have been relocated to M.A. >ED0C and >ED0E respectively.

For a variety of reasons, it may be advantageous to have code that is "self-relocating"; that is, it can be relocated anywhere in memory and execute correctly. Such "position-independent" or "dynamic-relocating" code is of great advantage when the code is programmed into EPROM, since the EPROMs can be installed in any socket, responding to any address, and the program will still execute correctly. Such programs are possible with the TMS 9995 EVM by merely beginning the program with the code segment shown below (Register 10 is used in the following examples) Thereafter, memory addresses can be indexed, relative to the beginning of the program (using R10 at the Index Register, in this case). This code is shown in Figure 8-10.

```
          Memory Address       Opcode/Operands             Comments

               0000   START     LWPI    FE00              R0 AT M.A. FE00.
               0004             LI      R10,START         LOOK AT START ADDRESS.
               0008             JEQ     RELOC             IF NOT BIASED, NEED
                                                            RELOCATING.
Base           000A             CLR     R10               LOADER HAS BIAS, CLEAR
Register                                                    BASE REGISTER.
Setup          000C             JMP     STARTX            GO TO PROGRAM.
               000E   RELOC     LI      R10,>045B         B *R11 OPCODE IN R10.
               0012             BL      R10               PC VALUE TO R11.
               0014   RELOCX    AI      R11,START-
                                          RELOCX          PC-10 = PROGRAM START
               0018             MOV     R11,R10           PROGRAM START TO R10


               001E   STARTX    MOV     @>001A(R10),      MOVE FIRST NUMBER TO R1.
                                          R1
Reloca-        0012             A       @>001C(R10,       ADD 2ND NUMBER TO R1,
table                                     R2                ANSWER IN R1.
Program        0016             B       @>0080            RETURN TO MONITOR.
               001A             DATA    100               FIRST NUMBER.
               001C             DATA    200               SECOND NUMBER.
```

FIGURE 8-10. EXAMPLE OF PROGRAM CODING ADDED TO MAKE (CODING) RELOCATABLE.

This coding first sets up a program base register which computes the address of the beginning of the program. This is accomplished by:

- Establishing the beginning workspace register address with LWPI.

- Placing the opcode for the instruction: B *R11 in the designated index register address (R10 above).

- Executing a branch and link to R10; this places the address of the next instruction following BL R10 into Register 11; a branch of R10 means a return indirect through R11.

- Computing the beginning address of the program by subtracting >10 from the address in Register 11.

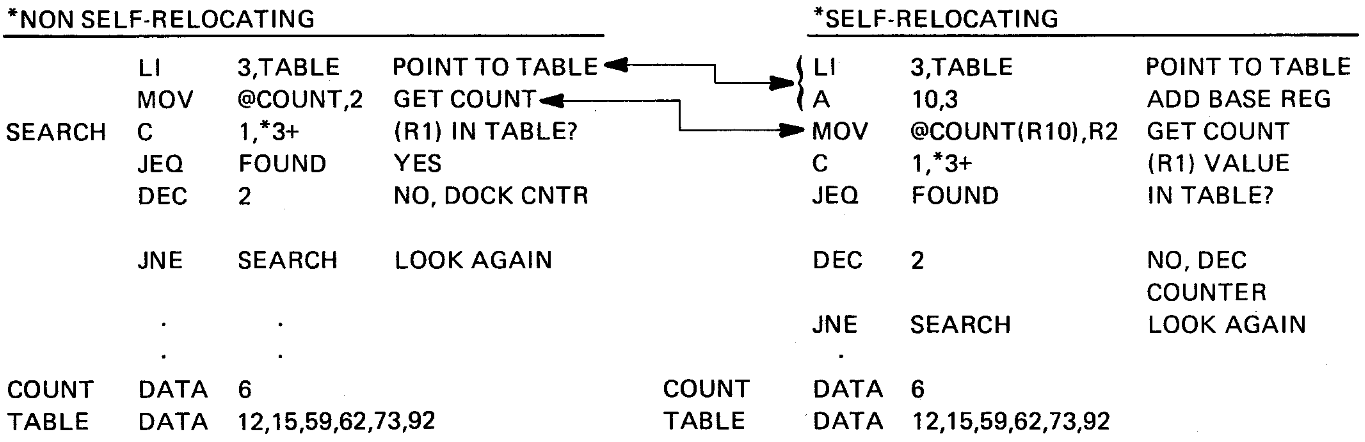
- Moving this beginning address to R10, allowing R11 to be further used as a linking register.

- Indexing all future relocatable addresses using R10.

There are several considerations. Absolute addresses (e.g., beginning of monitor at >0080) need not be indexed, and other types of memory indexing should consider the contents of the base register; it may be necessary t0 add the contents of the base register to another indexing register. Also, an immediate load of an address into a register will require that the base address in the index register be added to the register. For example:

```
          LI      R2,>0980             ADDRESS OF VALUES IN R2
          A       R10,R2               ADD BASE ADDRESS
```

Figure 8-11 is an example of a program that searches a table of numbers for a value. The example shows both relocatable and non-relocatable code for comparison. Symbolic addressing is used.



```
*NON SELF-RELOCATING                                   *SELF-RELOCATING
        LI     3,TABLE     POINT TO TABLE                     LI     3,TABLE          POINT TO TABLE
        MOV    @COUNT,2    GET COUNT                          A      10,3             ADD BASE REG
SEARCH  C      1,*3+       (R1) IN TABLE?                     MOV    @COUNT(R10),R2   GET COUNT
        JEQ    FOUND       YES                                C      1,*3+            (R1) VALUE
        DEC    2           NO, DOCK CNTR                      JEQ    FOUND            IN TABLE?

        JNE    SEARCH      LOOK AGAIN                         DEC    2                NO, DEC
                                                                                      COUNTER
        .      .                                              JNE    SEARCH           LOOK AGAIN
        .      .                                              .
COUNT   DATA   6                                      COUNT   DATA   6
TABLE   DATA   12,15,59,62,73,92                      TABLE   DATA   12,15,59,62,73,92
```

FIGURE 8-11. EXAMPLES OF NON-RELOCATING CODE AND SELF-RELOCATING CODE.

Great care must be taken with B, BL, and BLWP. If linking to other modules is needed, these modules must be part of a system which is linked together by the linker program (e.g., TXLINK on the FS990 system, for example), and all modules must be coded as self-relocating.

When programming the EPROMs, the code must be loaded so that the address START has the value zero, i.e., the code must appear biased at location >0000.

8.7 PROGRAMMING HINTS

In any programming environment there are several ways to accomplish a task. Table 8-4 contains alternate coding practices; some have an advantage over conventional coding.

TABLE 8-4. ALTERNATE PROGRAMMING CONVENTIONS.

	Conventional	Alternate	Alternate Code
Purpose	Code	Code	Advantages
Compare register contents with 0	CI RX,0	MOV RX,RX	Saves one word.
Increment a register by 4	INCT RX INCT RX	C *RX+,*RX+	Saves one word.
Access old workspace registers		MOV @N(R13),R1	N is twice the number of the old register wanted.
Swap two registers	MOV RX,RHOLD MOV RY,RX MOV RHOLD,RY	XOR RX,RY XOR RY,RX XOR RX,RY	Saves a register ("RHOLD" not needed).
Clear a register	CLR RX	XOR RX,RX	None.

8 8 INTERFACING WITH EVMBUG

The EMVMBUG monitor provides a starting point for the programmer to consider when looking for program examples. The monitor contains some basic user facilities, and the user will probably enter and exit programs through EVMBUG.

8.8.1 Program Entry and Exit

To execute a program under EVMBUG, use the "IR" and "EX" commands, as explained in Section 5 of this manual.

Exit from a program to EVMBUG can be through: B @>0080. EVMBUG will print the prompting question mark. Note that the power-up initialization routine is not entered; instead, control goes directly to EVMBUG´s command scanner.

8.8.2 I/O Using Monitor XOPs

8.8.2.1 Character I/O

Four XOPs deal specifically with character I/O:

-	Echo Character	XOP 11
-	Write Character	XOP 12
-	Read Character	XOP 13
-	Write Message	XOP 14

The echo (XOP 11) is a read character (XOP 13), followed by a write character (XOP 12). The following code reads in a character from a terminal. If an A or E is found, the character is written back to the terminal and program execution continues; otherwise, the program loops back, waiting for another keyboard entry:

```
GETCHR      XOP     R1,13       READ CHARACTER
            CI      R1,>4100    COMPARE R1 TO ASCII "A"
            JEQ     OK          IF "A" FOUND, JUMP
            CI      R1,>4500    COMPARE R1 TO ASCII "E"
            JEQ     OK          IF "E" FOUND, JUMP
            JMP     GETCHR      RETURN TO READ ANOTHER CHAR
OK          XOP     R1,12       WRITE CHARACTER AS ECHO
```

XOP 14 causes a string of characters to be written to the terminal. Characters are written until a byte of all zeroes is found.

XOP 13 reads one character and stores it into the left byte of a word; the right byte is zero filled. The previous coding example could also have been completed with the following: OK   XOP   R1.14

Instructions are written in uninterrupted form; thus, messages should be grouped in a block separated from the continuous executable code. Each message must be delimited by a byte of all zeroes:

```
**MESSAGES
CRLF           BYTE           >0D
LF             BYTE           >0A,>00
MSG1           TEXT           'BEGIN PGMA'
               BYTE           0
MSG2           TEXT           'END PGMA'
               BYTE           0
MSG3           TEXT           '# ERRORS (IN HEX):'
               BYTE           0
MSG4           TEXT           'ERROR EXP VALUE='
               BYTE           0
MSG5           TEXT           ',RCV VALUE='
               BYTE           0
```

Note in the preceeding example, that if it is desired to send a carriage return and a line feed, use the following: XOP @CRLF,14. If only a line feed is wanted, use: XOP @LF,14.

8.8.2.2 Hexadecimal I/O

Three XOPs handle hexadecimal numbers:

- Write one hexadecimal character	XOP 8
- Read a four-digit hexadecimal word	XOP 9
- Write four hexadecimal characters	XOP 10

Using the message block in paragraph 8.8.2.1, an example code segment might be:

```
*ERROR ROUTINE
ERROR      XOP      @MSG4,14       START ERROR LINE
           XOP      R1,10          PRINT CORRECT EXPECTED VALUE
           XOP      @MSG5,14       MORE ERROR LINE
           XOP      R2,10          PRINT ERRORED RCV VALUE
           XOP      @CRLF,14       DO CARRIAGE RETURN/LINE FEED
           XOP      @LF,14         ONE MORE LF FOR DOUBLE SPACE
```

XOP 8 is actually called four times by XOP 10, after positioning the next digit to be written into the least significant four bits of the Work Register.

The following shows how to input values to a program by asking for inputs from the terminal:

```
GET        XOP      R4,9              CALL TO GET HEX # ROUTINE
           DATA     NULL,ERROR        NO INPUT/BAD INPUT ADDRESSES
OK         A        R3,R4             ADD OLD NUMBER IN
           JMP      XXX               CONTINUE PROGRAM
NULL       LI       R4,>3AF1          LOAD DEFAULT VALUE
           XOP      @DEFMSG,14        PRINT DEFAULT MESSAGE
           JMP      OK
ERROR      XOP      @ERRMSG,14        PRINT ERROR MSG
           JMP      GET               TRY AGAIN
           . .      . .
DEFMSG     TEXT     'DEFAULT USED'
           BYTE     0
ERRMSG     TEXT     'ERROR: USE 0-9, A-F ONLY'
           BYTE     0
```

Note that the XOP 9 routine stores only the last four digits typed before the termination character (delimiter) is typed. This means if a wrong number is entered, continue typing until four correct digits are entered; then type a delimiter (space, carriage return, or minus sign). Typing fewer than four digits total (but at least one digit) causes leading zeroes to be inserted. Typing only a delimiter gives control to the first address following the XOP, and typing an illegal character at any time causes control to go to the address specified in the second word following the XOP call.

8.9 INTERRUPTS AND XOPs

8.9.1 Interrupt and XOP Linking Areas

When an interrupt or XOP instruction is executed, program control is transferred using the WP and PC vectors located in lower memory. Interrupt vectors are contained in memory addresses >0000 to >0013; and XOP vectors are contained in memory addresses >0040 to >007F. User-available interrupt and XOP vectors are preprogrammed in the EPROM chip with WP and PC values that allow the user to implement interrupt service routines (ISRs) and XOP service routines (XSRs). This includes programming an intermediate linking area as well as the ISR or XSR code.

When an interrupt or XOP is executed, it first passes control to the vectors which point to the linking area. The linking area directs execution to the actual ISR or XSR. The linking areas are shown in Table 8-5. The linking area is designed to leave as much space free as possible when not using all the interrupts. that is, the most frequently used areas are butted up against EVMBUG area, while the least frequently used areas extend downward into RAM.

Return from the ISR or XSR is through return vectors in R13, R14, and R15 at the ISR or XSR workspace and at the linking area workspace.

How to program these linking areas is explained in the following paragraphs.

8.9.1.1 Interrupt Linking Areas

When one of the programmable interrupts (INT1 - INT4) is executed, it traps to an interrupt linking area in RAM. Each linking area consists of six words (12 bytes) as shown in Figures 8-11 and 8-12. The first three words contain the last three registers of the called interrupt vector workspace (R13, R14, and R15). The second three words, located at the interrupt vector PC address, are intended to be programmed by the user to contain code for a BLWP instruction, a second word for the BLWP destination address, and an RTWP instruction code (all three words to be entered by the user). When the ISR is completed, control returns to this linking area's three registers (R13-R15), then the BLWP instruction (at the PC vector address) is executed using the M.A. provided by the user. The BLWP instruction consists of two words, the BLWP operator and the destination address; the destination address points to a two-word area also programmed by the user.

TABLE 8-5. PREPROGRAMMED INTERRUPT AND USER XOP TRAP VECTORS.

NOTE: Interrupt 4 is used by the timers at the TMS 9902.

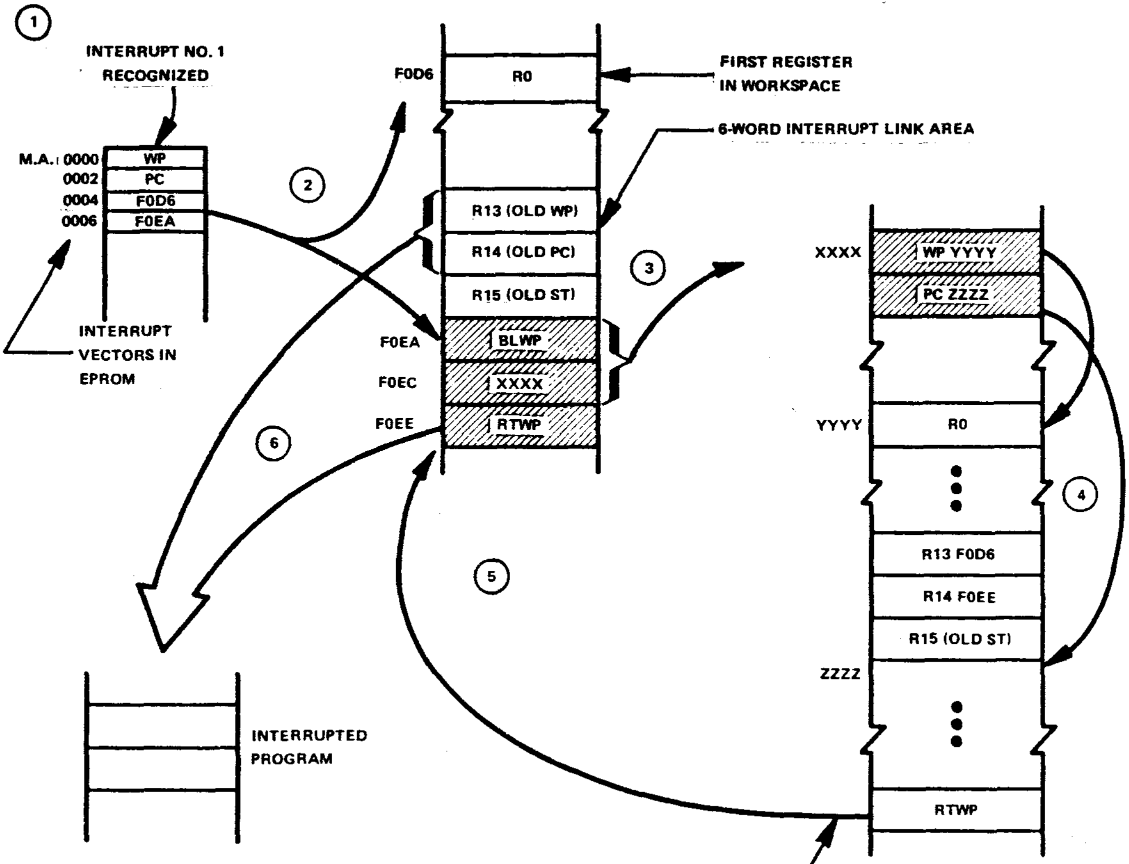
Memory Address.	Interrupt	WP	PC
0000	INT0	EC00	022E
0004	INT1	F0D6	F0F0
0008	INT2	F0CA	F0EA
000C	INT3	F0BE	F0DE
0010	INT4	FOB2	FOD2

Memory Address	Interrupt	WP	PC
0040	XOP0	F0AC	F0BE
0044	XOP1	F09E	F0B0
0048	XOP2	F090	F0A2
004C	XOP3	F082	F094
0050	XOP4	F074	F086
0054	XOP5	F066	F078
0058	XOP6	F058	F06A
005C	XOP7	F04A	F05C

TABLE 8-6. INTERRUPT AND USER XOP LINKING AREAS.

MEMORY ADDRESS	0-1	2-3	4-5	6-7	8-9	A-B	C-D	E-F
F050							XOP7	XOP7
F060	XOP7	XOP7	XOP7	XOP7	XOP7	XOP6	XOP6	XOP6
F070	XOP6	XOP6	XOP6	XOP6	XOP5	XOP5	XOP5	XOP5
F080	XOP5	XOP5	XOP5	XOP4	XOP4	XOP4	XOP4	XOP4
F090	XOP4	XOP4	XOP3	XOP3	XOP3	XOP3	XOP3	XOP3
F0A0	XOP3	XOP2	XOP2	XOP2	XOP2	XOP2	XOP2	XOP2
F0B0	XOP1	XOP1	XOP1	XOP1	XOP1	XOP1	XOP1	XOP0
F0C0	XOP0	XOP0	XOP0	XOP0	XOP0	XOP0	INT4	INT4
F0D0	INT4	INT4	INT4	INT4	INT3	INT3	INT3	INT3
F0E0	INT3	INT3	INT2	INT2	INT2	INT2	INT2	INT2
F0F0	INT1	INT1	INT1	INT1	INT1	INT1		

Return from the interrupt service routine is through the RTWP instruction (routines´s last instruction). This places the (previous) WP and PC values at the time of the BLWP instruction (in the six-word linking area) into the WP and PC registers. The RTWP code that follows the BLWP instruction will now be executed, causing a second return routine to occur, this time to the interrupted program using the return values in R13, R14, and R15 of the interrupt link area. This is shown graphically in Figure 8-12.



1,2 INTERRUPT EXECUTION TRAPS TO 6-WORD INTERRUPT LINK AREA.
3,4 BLWP EXECUTED TO 2-WORD VECTORS TO INTERRUPT SERVICE ROUTINE (ISR)
5 RTWP FROM ISR TRAPS BACK TO 6-WORD LINK AREA.
6 RTWP FROM LINK AREA RETURNS BACK TO INTERRUPTED PROGRAM.

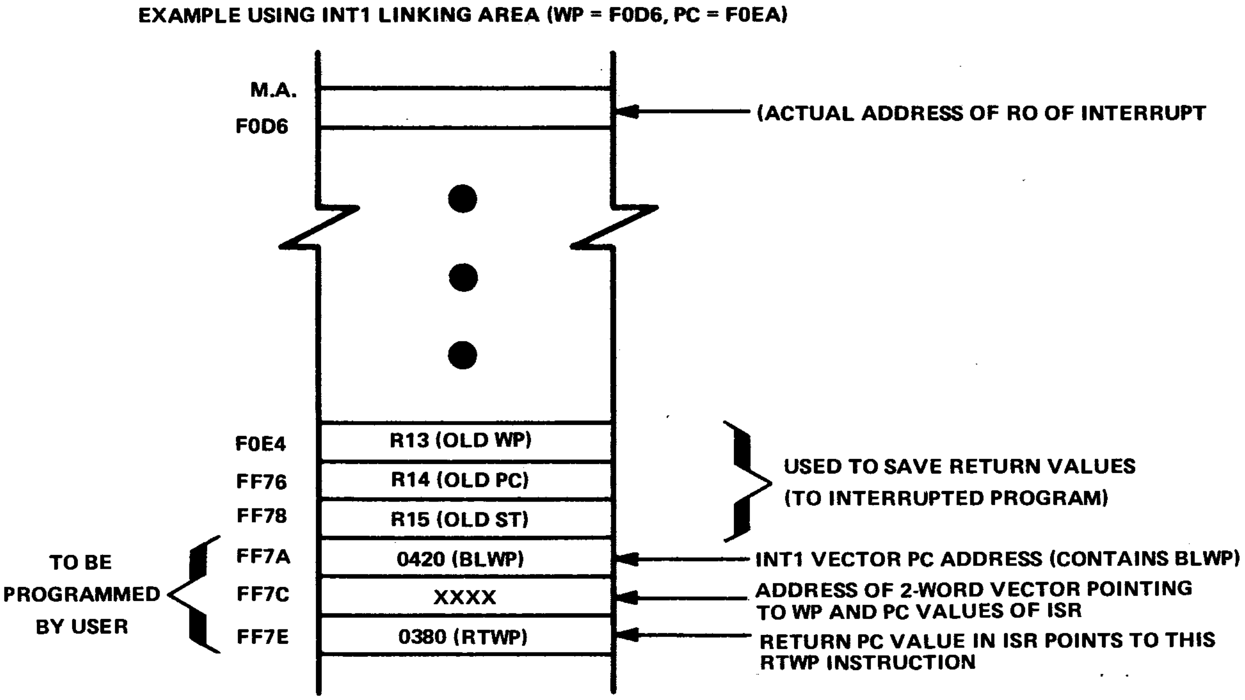
FIGURE 8-12. INTERRUPT SEQUENCE.

Each interrupt linking area is set up so that it can be programmed in this manner. In summary, each six-word linking area can be programmed as follows:

- Determine the location of the linking area, as shown by the WP and PC vectors in Table 8-4.

- The PC vector will point to the last three words of the six-word area. The user must program these three words respectively, with >0420 for a BLWP instruction, the address (BLWP operand) of the 2-word vector pointing to the interrupt service routine, and >0380 for an RTWP instruction, as shown in Figure 8-13.

- At the vector address for the BLWP operand, place the WP and PC values respectively of the interrupt handler.


EXAMPLE USING INT1 LINKING AREA (WP = F0D6, PC = F0EA)
M.A.
F0D6
(ACTUAL ADDRESS OF R0 OF INTERRUPT
F0E4
R13 (OLD WP)
FF76
R14 (OLD PC)
FF78
R15 (OLD ST)
USED TO SAVE RETURN VALUES (TO INTERRUPTED PROGRAM)
TO BE PROGRAMMED BY USER
FF7A
0420 (BLWP)
INT1 VECTOR PC ADDRESS (CONTAINS BLWP)
FF7C
XXXX
ADDRESS OF 2-WORD VECTOR POINTING TO WP AND PC VALUES OF ISR
FF7E
0380 (RTWP)
RETURN PC VALUE IN ISR POINTS TO THIS RTWP INSTRUCTION

NOTE

DO NOT USE R0-R12 OF THE LINKING AREA WORKSPACE, BECAUSE THE OVERLAPPING STRUCTURE WILL DESTROY THE CONTENTS OF A LINKING AREA FOR ANOTHER INTERRUPT OR XOP.

FIGURE 8-13. SIX-WORD INTERRUPT LINKING AREA.

Coding to program the linkage to the interrupt service routine is as follows (sample only):

```
   *PROGRAM POINTER TO INT1 SERVICE ROUTINE FOLLOWING BLWP INSTRUCTION
        AORG      >FFEA      INT1 PC VECTOR ADDRESS
        DATA      >0420      HEX VALUE OF BLWP OP CODE
        DATA      >ED00      LOCATION OF 2-WORD VECTORS TO ISR (EXAMPLE)
        DATA      >0380      HEX VALUE OF RTWP OP CODE


   *PROGRAM POINTER TO 2-WORD VECTORS TO INTERRUPT SERVICE ROUTINE
     (EXAMPLE)
        AORG      >ED00
        DATA      >EE00       WP OF INTERRUPT SERVICE ROUTINE (EXAMPLE)
        DATA      >ED04       PC OF INTERRUPT SERVICE ROUTINE (EXAMPLE)


     *INT1 ISR FOLLOWS (BEGINS AT M.A. >ED04)
```

The interrupt service routine which begins at M.A. >ED04 will terminate with an RTWP instruction.

8.9.1.2 XOP Linking Area

The XOP linking area contains seven two-byte words. The first, second, and the fourth words must be programmed by the user. Each XOP vector pair contains the pointer to the new WP in the first word, and a pointer to the new PC in the second word. These point to the first instruction to be executed..

In the seven-word XOP linking area, the first word is the destination of the XOP PC vector. The last three words are the final three registers (R13, R14, and R15) of the linking area workspace which will contain the return vectors back to the program that called the XOP. The third word of the seven-word area is R11, which contains the parameter being passed to the XOP service routine. This is shown in Figure 8-14.

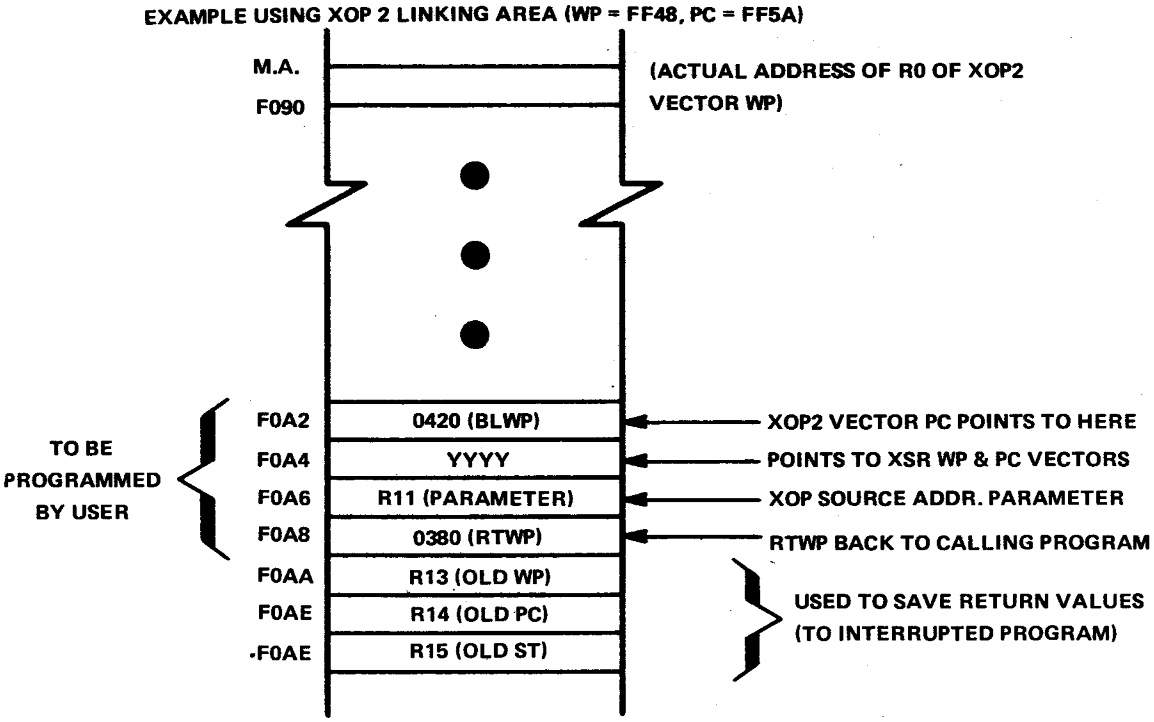

EXAMPLE USING XOP 2 LINKING AREA (WP = FF48, PC = FF5A)
M.A.
F090
(ACTUAL ADDRESS OF R0 OF XOP2 VECTOR WP)
TO BE PROGRAMMED BY USER
F0A2
0420 (BLWP)
XOP2 VECTOR PC POINTS TO HERE
F0A4
YYYY
POINTS TO XSR WP & PC VECTORS
F0A6
R11 (PARAMETER)
XOP SOURCE ADDR. PARAMETER
F0A8
0380 (RTWP)
RTWP BACK TO CALLING PROGRAM
F0AA
R13 (OLD WP)
F0AE
R14 (OLD PC)
•F0AE
R15 (OLD ST)
USED TO SAVE RETURN VALUES (TO INTERRUPTED PROGRAM)

FIGURE 8-14. SEVEN-WORD XOP INTERRUPT LINKING AREA.

For example, when XOP2 is executed, the PC vector points to the BLWP instruction shown at M.A. >F0A2 in Figure 8-14. This executes, transferring control to the pre-programmed WP and PC values at the address in the next word (YYYY, as shown in Figure 8-14). To obtain the parameter passed to R11 of the vector WP (M.A. >F0A6 in Figure 8-13), use the following code in the XOP service routine:

```
MOV *R14+,R1  MOVE PARAMETER TO R1
```

This moves the parameter to R1 from the old R11 (the old PC value in R14 was pointing to this address following the BLWP instruction immediately above it, effectively to R11), and increments the XOP service routine PC value in its R14 to the RTWP instruction at M.A. >F0A8. Thus, an RTWP return from the XOP service routine will branch back to the RTWP instruction at memory address >F0A8, which returns control back to the instruction following the XOP.

In summary, the seven-word XOP linking area can be programmed as follows :

- Determine the value of the PC vector for the XOP, as shown in Table 8-4.

- The PC value will point to the first word of the seven-word linkage area. The user must program three of the first four words of this area as follows: >0420 for a BLWP instruction, the address of the two-word vector that points to the XOP service routine, ignore the third word, and, >0380 for an RTWP instruction in the fourth word.

- At the address of the BLWP destination in the second word, place the WP and PC values respectively to the XOP service routine.

An example of coding to program the XOP linkage for XOP2, as shown in Figure 8-14, is as follows:

```
*  PROGRAM POINTER TO XOP SERVICE ROUTINE AT XOP2 LINK AREA
     AORG      >FF5A      XOP2 PC VECTOR ADDRESS
     DATA      >0420      HEX VALUE OF BLWP ADDRESS
     DATA      >FA00      LOCATION OF 2-WORD VECTORS TO XSR (EXAMPLE)
     DATA       0         IGNORE
     DATA      >0380      HEX VALUE OF RTWP CODE


*  PROGRAM POINTER TO 2-WORD VECOTRS TO XOP2
*  SERVICE ROUTINE (EXAMPLE)
     AORG      >FA00      LOCATION OF VECTORS
     DATA      >FB00      WP OF XOP SERVICE ROUTINE (EXAMPLE)
     DATA      >FA04      PC OF XOP SERVICE ROUTINE (EXAMPLE)


*  XSR CODE FOLLOWS (BEGINS AT M.A. >FA04)
```

At the XOP service routine, the following code uses the PC return value (in R14 of the XOP service routine workspace) to obtain the parameter in R11 (in the link area), as well as set the return PC value in R14 (in the XOP service routine workspace) to the RTWP in the link area:

```
        MOV      *R14+,R1  MOVE OLD R11 CONTENTS
                           TO R1 OF XOP SERVICE ROUTINE
```

Now, R14 points to the RTWP instruction in the link area. The last instruction in the XOP service routine is RTWP. RTWP execution causes a return to the link area, where a second RTWP executes, returning control to the next instruction following the XOP.

8.10 TMS 9995 INTERVAL TIMER INTERRUPT PROGRAM

A Detailed discussion as to how the TMS 9995 Decrementer is configured to act an interval timer can be found in the TMS 9995 Data Manual.

There are several possible sequences of coding that can program and enable the Interrupt 3 interval timer, and since the timer has a maximum period of 87.25 milliseconds before issuing an interrupt, the programmer must decide whether to set the interval period in the calling program or in the code handling the interrupt. If the interrupt period desired is longer the 87.25 milliseconds then it may be advantageous to reset the timer in the interrupt subroutine, which also triggers the interrupt and returns control back to the interrupted program. In any case, the timer must be initially set and triggered following the general sequence below.

1. Set Flag Register CRU address for the TMS 9995 in bits 0 to 14 of R12.

2. Set up the Interrupt 3 linking area.

3. Set the Status Register interrupt mask to a value of 3 or greater.

4. Set the 9995 RAM address for the Decrementer to the value of the interval desired (bits 0 to 15).

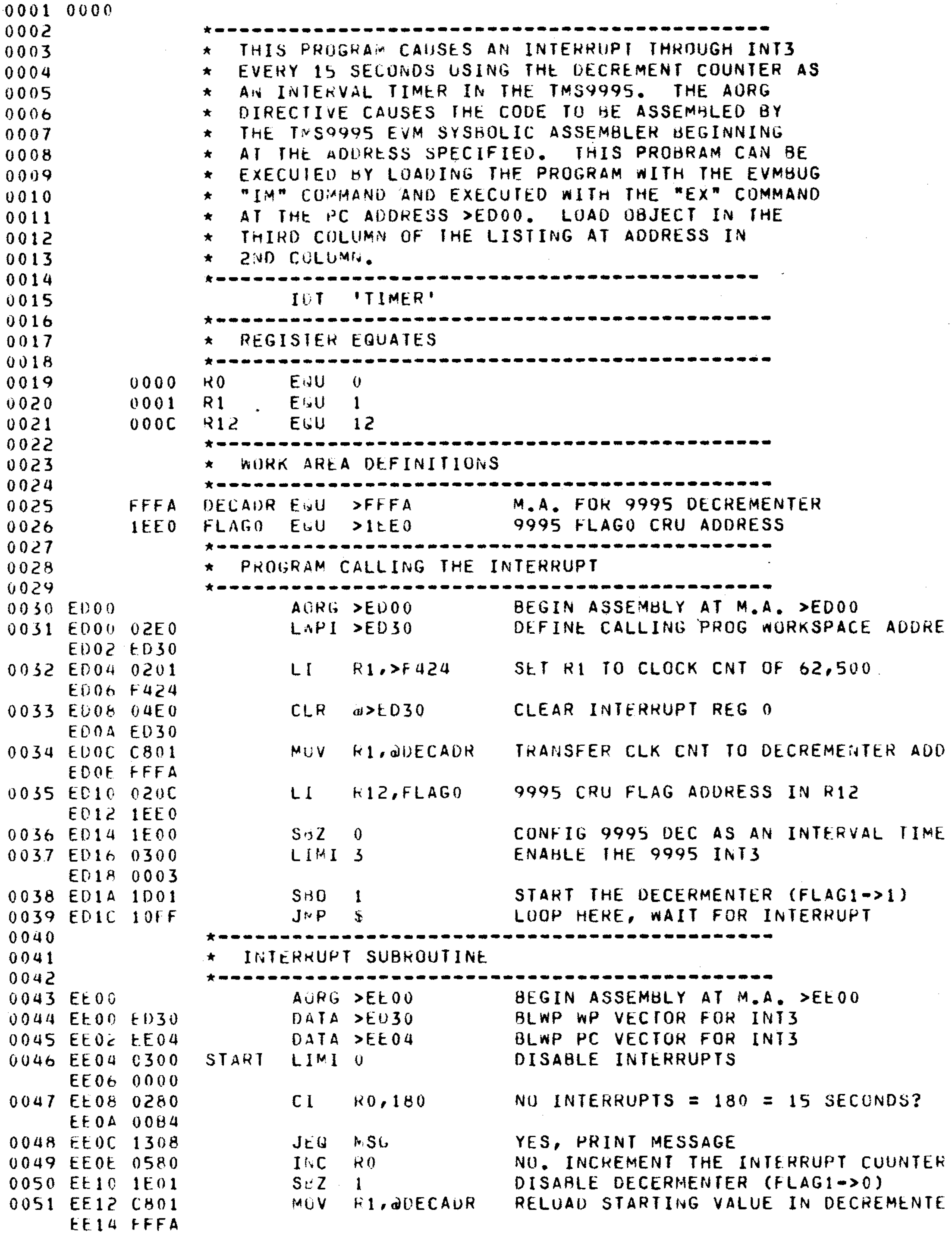
5. Set the 9995 FLAG1 to 1 to enable the Decrementer countdown.

The TMS 9995 Decrementer decrements the value set in Step 4 at the rate of 1 every 4 clock cycles (approximately 750 K Hz with a 3 MHz clock). The maximum interval register value of all ones in 16 bits (65,535) takes approximately 87.25 milliseconds to decrement to zero.

The code in Figure 8-15 is an example of a code to set up and call the TMS 9995 interval timer and also the code of the interrupt handling subroutine. Note that the calling program first clears the counting register (RO) of the interrupt workspace, then it sets up the interrupt masks at the 9995 after setting the CRU address of the 9995 Flag Register in R12. Then the calling program sets an initial value in the timer register. Because the desired output on the terminal is a message every 15 seconds, the miminum 9995 decrement count program is set up in the calling program while the interrupt handler routine is responsible for tabulation and clearing of interrupts after they occur. The handler keeps track of the number of intervals to determine the 15 second count.

At the bottom of the figure is the interrupt linking area. Since all the code in this figure is loaded as if at absolute memory address values (using the AORG assembler directive), data statements are used here at the appropriate memory address. This program can be loaded and executed by placing the machine-language assembler output in the third column at the address shown in the second column. Then execute with the program start at memory address >ED00.

The TMS 9995 can also be used as an event timer by starting the counter at the beginning of an interval and reading the counter after the event has occurred.



```
0000
               *--------------------------------------------
               *  THIS PROGRAM CAUSES AN INTERRUPT THROUGH INT3
               *  EVERY 15 SECONDS USING THE DECREMENT COUNTER AS
               *  AN INTERVAL TIMER IN THE TMS9995.  THE AORG
               *  DIRECTIVE CAUSES THE CODE TO BE ASSEMBLED BY
               *  THE TMS9995 EVM SYSBOLIC ASSEMBLER BEGINNING
               *  AT THE ADDRESS SPECIFIED.  THIS PROBRAM CAN BE
               *  EXECUTED BY LOADING THE PROGRAM WITH THE EVMBUG
               *  "IM" COMMAND AND EXECUTED WITH THE "EX" COMMAND
               *  AT THE PC ADDRESS >ED00.  LOAD OBJECT IN THE
               *  THIRD COLUMN OF THE LISTING AT ADDRESS IN
               *  2ND COLUMN.
               *-------------------------------------------
                      IDT  'TIMER'
               *--------------------------------------------
               *  REGISTER EQUATES
               *--------------------------------------------
          0000 R0     EQU  0
          0001 R1     EQU  1
          000C R12    EQU  12
               *--------------------------------------------
               *  WORK AREA DEFINITIONS
               *--------------------------------------------
          FFFA DECADR EQU  >FFFA          M.A. FOR 9995 DECREMENTER
          1EE0 FLAG0  EQU  >1EE0          9995 FLAG0 CRU ADDRESS
               *--------------------------------------------
               *  PROGRAM CALLING THE INTERRUPT
               *--------------------------------------------
ED00                  AORG >ED00          BEGIN ASSEMBLY AT M.A. >ED00
ED00 02E0             LWPI >ED30          DEFINE CALLING PROG WORKSPACE ADDRE
ED02 ED30
ED04 0201             LI   R1,>F424       SET R1 TO CLOCK CNT OF 62,500
ED06 F424
ED08 04E0             CLR  @>ED30         CLEAR INTERRUPT REG 0
ED0A ED30
ED0C C801             MOV  R1,@DECADR     TRANSFER CLK CNT TO DECREMENTER ADD
ED0E FFFA
ED10 020C             LI   R12,FLAG0      9995 CRU FLAG ADDRESS IN R12
ED12 1EE0
ED14 1E00             SBZ  0              CONFIG 9995 DEC AS AN INTERVAL TIME
ED16 0300             LIMI 3              ENABLE THE 9995 INT3
ED18 0003
ED1A 1D01             SBO  1              START THE DECERMENTER (FLAG1->1)
ED1C 10FF             JMP  $              LOOP HERE, WAIT FOR INTERRUPT
               *--------------------------------------------
               *  INTERRUPT SUBROUTINE
               *--------------------------------------------
EE00                  AORG >EE00          BEGIN ASSEMBLY AT M.A. >EE00
EE00 ED30             DATA >ED30          BLWP WP VECTOR FOR INT3
EE02 EE04             DATA >EE04          BLWP PC VECTOR FOR INT3
EE04 0300      START  LIMI 0              DISABLE INTERRUPTS
EE06 0000
EE08 0280             CI   R0,180         NO INTERRUPTS = 180 = 15 SECONDS?
EE0A 00B4
EE0C 1308             JEQ  MSG            YES, PRINT MESSAGE
EE0E 0580             INC  R0             NO. INCREMENT THE INTERRUPT COUNTER
EE10 1E01             SBZ  1              DISABLE DECERMENTER (FLAG1->0)
EE12 C801             MOV  R1,@DECADR     RELOAD STARTING VALUE IN DECREMENTE
EE14 FFFA
```

FIGURE 8-15. EXAMPLE OF CODE TO RUN TMS 9995 INTERVAL TIMER. (1 of 2)

```
0052 EE16 0300         LIMI 3             REENABLE THE 9995 INT3
     EE18 0003
0053 EE1A 1D01         SBO  1             RESTART THE DECREMENTER (FLAG1->1)
0054 EE1C 0380         RTWP               RETURN TO CALLING PROGRAM AND WAIT
0055 EE1E 2FA0  MSG    XOP  @MTEXT,14     WRITE MESSAGE
     EE20 EE28
0056 EE22 04C0         CLR  R0            RESET INTERRUPT COUNTER
0057 EE24 0460         B    @START        RETURN TO START OF INTERRUPT ROUTIN
     EE26 EE04
0058 EE28   31  MTEXT  TEXT '15 SECONDS HAVE ELAPSED.'
     EE29   35
     EE2A   20
     EE2B   53
     EE2C   45
     EE2D   43
     EE2E   4F
     EE2F   4E
     EE30   44
     EE31   53
     EE32   20
     EE33   48
     EE34   41
     EE35   56
     EE36   45
     EE37   20
     EE38   45
     EE39   4C
     EE3A   41
     EE3B   50
     EE3C   53
     EE3D   45
     EE3E   44
     EE3F   2E
0059 EE40 0707         DATA >0707,>0707
     EE42 0707
0060 EE44   0D         BYTE >D,>A,0,0
     EE45   0A
     EE46   00
     EE47   00
0061            *---------------------------------------------
0062            *  INTERUPT LINK AREA PROGRAMMING
0063            *---------------------------------------------
0064 F0DE              AORG >F0DE         BEGIN ASSEMBLY AT M.A. >F0DE
0065 F0DE 0420         DATA >0420         BLWP INSTRUCTION CODE
0066 F0E0 EE00         DATA >EE00         BLWP VECTORS LOCATION
0067 F0E2 0380         DATA >0380         RTWP INSTRUCTION CODE
0068                   END
NO ERRORS,      NO WARNINGS
```

FIGURE 8-15. EXAMPLE OF CODE TO RUN TMS 9995 INTERVAL TIMER. (2 of 2)

8.11 MOVE BLOCK FOLLOWING PASSING OF PARAMETERS

The coding in Figure 8-16 is an example of a called subroutine that will move a block of data from one location to another. The three parameters of (1) Move-From-Address, (2) Move-To-Address, and (3) Length-Of-Block, are provided to the subroutine either through Registers 0 to 2 or by the three words following the calling program´s BLWP instruction, or by a combination of both. The block move subroutine first interrogates the words following the calling program´s BLWP instruction; if a zero is found, it looks in a register for the parameter. In Figure 8-15, the calling program provides the Move-From and Block-Length parameters in Register, and the Move-To parameter in the second word following the BLWP instruction.

```
            LI          R0,>F100            MOVE-FROM ADDRESS
            LI          R2,125              MOVE 125 BYTES
            BLWP        @MOVBLK             BRANCH TO SUBROUTINE
            DATA        0                   MOVE-FROM ADDR IN R0
            DATA        >F200               MOVE-TO ADDRESS
            DATA        0                   BYTE COUNT IN R2
              .
```

(A) CALLING PROGRAM

```
MVBLK       DATA        >FF90,MVBLK1        WP, PC OF SUBROUTINE
MVBLK1      MOV         13,12               SAVE WP
            MOV         *14+,1              GET "FROM" ADR
            JNE         MVBLK2              NON-ZERO:  PARM IN-LINE
            MOV         *13+,1              PICK UP FROM REG INSTEAD
MVBLK2      MOV         *14+,2              GET "TO" ADR
            JNE         MVBLK3              PARM IN IN-LINE CODE
            MOV         *13+,2              GET FROM REGS
MVBLK3      MOV         +14+,3              GET LENGTH
            JNE         MVBLK4              IN-LINE PARM
            MOV         *13,3               GET FROM REGS
MVBLK4      MOVB        *1+,*2+             MOVE BYTE
            DEC         3                   ONE LESS TO GO
            JNE         MVBLK4              NOT DONE YET
            MOV         12,13               RESTORE WP
            RTWP                            RETURN TO CALLING PROGRAM
              .
              .
              .
```

(B) MOVE BLOCK SUBROUTINE

FIGURE 8-16. MOVE BLOCK OF BYTES SAMPLE ROUTINE.

8.12 BLOCK-COMPARE SUBROUTINE

Figure 8-17 shows a sample block-compare subroutine which accepts three parameters from the calling program in the same manner as the block-move subroutine, Figure 8-15. This compare subroutine inspects two strings, comparing successive bytes until an unequal byte is found or until the specified string length is exhausted. The Status Register bits in Register 15 are updated accordingly, and the subroutine returns to the calling routine with the altered status bits, which may be used immediately for conditional jump.

The sample calling program is at the top of Figure 8-17. Note that the conditional jumps follow directly after the calling code, so the calling program simply compares (through the subroutine) and jumps, in the normal programming manner.

```
            .
            .
            LI          R0,>100         FIRST BLOCK START ADDRESS
            LI          R1,>F200        SECOND BLOCK START ADDRESS
            BLWP        @CMBLK          BRANCH TO SUBROUTINE
            DATA        0               START ADDR IN R0 (1ST BLOCK)
            DATA        0               START ADDR IN R1 (2ND BLOCK)
            DATA        100             COMPARE 100 BYTES
            JLE         $+10            IF LESS THAN OR EQUAL, JUMP
            JGT                         IF GREATER THAN, JUMP
```

(A) CALLING PROGRAM

```
            .
            .
            .
CMBLK       DATA        >FF90,CMBLK1    WP, PC OF SUBROUTINE
CMBLK1      MOV         13,12           SAVE WP
            MOV         *14+,1          GET "A" ADR
            JNE         CMBLK2
            MOV         *13+,1          GET IN CALLER REG
CMBLK2      MOV         *14+,2          GET "B" ADR
            JNE         CMBLK3
            MOV         *13+,2          GET FROM IN CALLER REG
CMBLK3      MOV         *14+,3          GET LENGTH
            JNE         CMBLK4
            MOV         *13,3           GET FROM REG
CMBLK4      CB          *1+,*2+         LOOK AT STRINGS
            JNE         CMBLK5          FOUND UNEQUAL
            DEC         3               ONE LESS BYTE
            JNE         CMBLK4          STILL MORE TO LOOK AT
CMBLK5      STST        15              STORE FINAL STATUS
            RTWP                        RETURN TO CALLING PROGRAM
```

(B) COMPARE BLOCK SUBROUTINE

FIGURE 8-17. COMPARE BLOCKS OF BYTES SAMPLE SUBROUTINE.

8 13 USING MAIN AND AUXILIARY TMS 9902s FOR I/O

The EVMBUG XOP routines (XOP8 to 14) are written to accomplish input and output through a TMS 9902. When the EVMBUG monitor is entered, the address for all I/O is set to the main TMS 9902. Any time a user program branches back into EVMBUG at address >0080, or when the RESET function is activated, the CRU address is set to the main TMS 9902. However, a user program may use all of the above-mentioned XOP calls to program the auxiliary TMS 9902 in the system by first moving the desired R12 base address to location >EC28. Figure 8-18 is a sample program wherein two serial I/O ports are activated for conversation with each other. Two terminals are assumed to be connected, one to EIA Port 1 and one to Port 2, and the operators may type messages to each other. This principle can be expanded to support any of a number of TMS 9902s. (A variety of custom line interfaces may be used with a TMS 9902.)

The write-character XOP service routine first ensures that the Request-to-Send signal is active. This signal is not deactivated by EMVBUG, so that modem users will retain their data carrier. If a modem user wishes to drop the data carrier, the affected TMS 9902 must be addressed by the user program, and then the Request-to-Send signal deactivated through the CRU.

Only the main TMS 9902, at CRU R12 base address >0000 is initialized by EVMBUG; others in the system must be initialized by the user. Note the first portion of the example program shown in Figure 8-20. Part of EVMBUG's initialization is to sense the baud rate of the attached terminal. If the baud rate is 110, 300, or 1200 baud, then the XOP routine waits 200 milliseconds after transmitting a carriage return. In addition, 1200 baud causes every character transmitted to be followed by 25 milliseconds of delay time. Only at 2400 and 9600 baud are characters transmitted without delays.

For 110, 300, and 1200 baud, the monitor ASRFLAG is set to one to cause a 'wait state' following writing of a carriage return. If the EVMBUG I/O XOP routines are used for the other I/O port, the state of the monitor's ASRFLAG will also govern delay loops used by the Write-Character XOP. The user should then swap out the contents of the ASRFLAG, as listed in table 8-7.

TABLE 8-7. ASRFLAG VALUES.

ASRFLAG * VALUE	RECOMMENDED BAUD RATE	DESCRIPTION/RECOMMENDATION
Positive No.	2400, 9600	No delays. Use for CRTs, modems.
Zero	110, 300	Carriage Return Delay only. Use for hardcopy terminals.
Negative No.	1200	Carriage Return and Character padding delays. Use with "TNF" command if termiinal is not a TI ASR733.

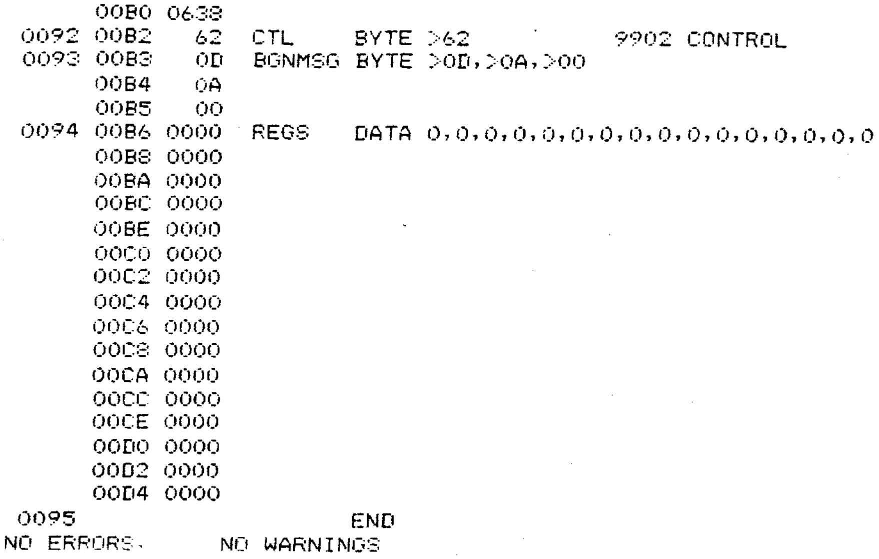
* ASRFLAG located in RAM at Memory Address >EC44.

```
0000
                  IDT  'TWOTRM'
           *-------------------------------------------------------
           *      TWO TERMINAL PROGRAM EXAMPLE
           *      THIS ROUTINE INITIALIZES THE AUXILIARY I/O PORT
           *      OF THE TMS9995 MICROCOMPUTER.  BOTH SERIAL
           *      PORTS ARE THEN USED IN THE CONVERSATIONAL MODE
           *      WITH EACH OTHER.  THE PROCEDURE IS TO INSPECT
           *      THE RECEIVE BUFFER BIT IN THE ADDRESSED TMS9902
           *      TO SEE IF A CHARACTER HAS BEEN ASSEMBLED
           *      IN THE UART.  IF SO, IT IS ECHOED TO THE
           *      ORIGINATING TERMINAL, AND THEN TRANSMITTED
           *      TO THE OTHER TERMINAL.  THEN THE OTHER
           *      TERMINALIS INSPECTED FOR A CHARACTER, ETC.
           *           1)  THE AUXILIARY TMS9902 MUST BE INITIALIZED.
           *           2)  THE OLD "ASR"-FLAG MUST BE SAVED.
           *               AND A NEW ONE DETERMINED FOR THE
           *               NEW TERMINAL (AUXILIARY PORT).
           *           3)  EVERY WRITE OPERATION CONSISTS OF
           *               MOVING THE DESIRED ADDRESS TO EVMBUG,
           *               AND MOVING THE DESIRED "ASR"-FLAG TO EVMBUG.
           *-------------------------------------------------------
0000 02E0         LWPI REGS          USE SPARE SPACE AT END OF PROG
0002 00B6'
0004 020C         LI   12,>0400      AUXILIARY PORT ADDRESS
0006 0400
           *      INITIALIZE AUXILIARY SERIAL PORT
0008 1D1F         SBO  31            RESET TIMING DELAY
000A 1000         NOP                RESET TIMING DELAY
000C 3220         LDCR @CTL,8        LOAD CONTROL CHARACTER
000E 00B2'
0010 1E0D         SBZ  13            BYPASS INTERVAL REGISTER
0012 04C0         CLR  0             BAUD RATE LOOP COUNTER
0014 04C2         CLR  2             ASR FLAG FOR THIS PORT
0016 1F0F  TSTSP  TB   15            LOOK AT RIN
0018 13FE         JEQ  TSTSP         WAIT FOR USER TO TYPE SOMETHING
001A 0580  SPLOOP INC  0             UP BAUD LOOP COUNTER
001C 1F0F         TB   15            RIN NOW HAS A SPACE:
001E 16FD         JNE  SPLOOP        DROP OUT ON A MARK
0020 0201         LI   1,TABLE       BAUD RATE TABLE
0022 00A2'
           *      NOW INSPECT BAUD RATE TABLE FOR A LOOP
           *      COUNT WHICH MATCHES, THE LOAD BAUD RATE.
0024 8C40  BDLOOP C    0,*1+         LOOK AT ATABEL LOOP COUNT
0026 1202         JLE  MATCH         IF < OR = WE HAVE A MATCH
0028 05C1         INCT 1             SKIP BAD BAUD RATE, NEXT LOOP
002A 10FC         JMP  BDLOOP        LOOK AT NEXT LOOP COUNT
002C 3311  MATCH  LDCR *1,12         LOAD BAUD RATE CONTROL VALUE
002E C051         MOV  *1,1          GET VALUE ITSELF
0030 0281         CI   1,>01A0       1200 BAUD ?
0032 01A0
0034 1103         JLT  HIRATE        NO, HIGHER BAUD RATE
0036 1603         JNE  BEGIN         NO, LOWER BAUD RATE
0038 0702         SETO 2             SET LOCAL ASR FLAG
003A 1001         JMP  BEGIN         AND PRINT BEGIN MESSAGE
003C 0582  HIRATE INC  2             MARK NO <CR> DELAY
           *      THE AUXILIARY PORT IS NOW UP. PRINT GREETING.
003E C820  BEGIN  MOV  @PRT2,@XOPCRU AUX. PORT ADR. TO EVMBUG
0040 00A0'
0042 EC2E
```

FIGURE 8-18. SAMPLE PROGRAM TO CONVERSE THROUGH MAIN AND AUXILIARY TMS 9902s. (1 of 3)

```
0054 0044 C0E0          MOV   @ASRFLG,3     SAVE MAIN PORT ASR-FLAG
     0046 EC44
0055 0048 C802          MOV   2,@ASRFLG     AUX. PORT ASR-FLAG
     004A EC44
0056 004C 2F40          XOP   0,13          READ BY OLD INIT. CHAR.
0057 004E 2FA0          XOP   @BGNMSG,14    PRINT BEGIN MESSAGE
     0050 00B3'
0058 0052 C820          MOV   @PRT1,@XOPCRU MAIN PORT ADR TO EVMBUG
     0054 009E'
     0056 EC2E
0059 0058 C803          MOV   3,@ASRFLG     MAIN PORT ASR-FLAG
     005A EC44
0060 005C 2FA0          XOP   @BGNMSG,14    PRINT BEGIN MESSAGE HERE, TOO
     005E 00B3'
0061                *       THIS IS THE MAIN LOOP.
0062                *       FIRST ADDRESS MAIN PORT, THEN THE AUXILIARY PORT
0063 0060 C320 LOOP   MOV   @PRT1,12      ADDRESS FOR MAIN PORT
     0062 009E'
0064 0064 1F15          TB    21            CHARACTER TYPED HERE ?
0065 0066 160B          JNE   NEXT          MO. TRY OTHER PORT
0066 0068 C80C          MOV   12,@XOPCRU    YES, GIVE ADDRESS TO EVMBUG
     006A EC2E
0067 006C C803          MOV   3,@ASRFLG     MOVE ASR-FLAG
     006E EC44
0068 0070 2EC0          XOP   0,11          READ/ECHO CHAR TO ORIGINATING
0069 0072 C820          MOV   @PRT2,@XOPCRU AUXILIARY PORT ADDRESS
     0074 00A0'
     0076 EC2E
0070 0078 C802          MOV   2,@ASRFLG     AUXILIARY PORT ASR-FLAG
     007A EC44
0071 007C 2F00          XOP   0,12          WRITE CHARACTER TO OTHER TERMINAL
0072 007E C320 NEXT     MOV   @PRT2,12      ADDRESS FOR AUXILIARY PORT
     0080 00A0'
0073 0082 1F15          TB    21            CHARACTER TYPED HERE ?
0074 0084 16ED          JNE   LOOP          NO, TRY MAIN PORT
0075 0086 C80C          MOV   12,@XOPCRU    YES, GIVE ADDRESS TO EVMBUG
     0088 EC2E
0076 008A C802          MOV   2,@ASRFLG     MOVE ASR-FLAG
     008C EC44
0077 008E 2EC0          XOP   0,11          READ/ECHO CHAR TO ORIGINATING
0078 0090 C820          MOV   @PRT1,@XOPCRU MAIN PORT ADDRESS
     0092 009E'
     0094 EC2E
0079 0096 C803          MOV   3,@ASRFLG     MAIN PORT ASR-FLAG
     0098 EC44
0080 009A 2F00          XOP   0,12          WRITE CHARACTER TO MAIN TERMINAL
0081 009C 10E1          JMP   LOOP
0082                *DATA AREA
0083                *
0084 009E 0000 PRT1     DATA  >0000         MAIN PORT R12 BASE ADDRESS
0085 00A0 0400 PRT2     DATA  >0400         AUXILIARY PORT R12 BASE ADDRESS
0086      EC44 ASRFLG   EQU   >EC44         EVMBUG ASR FLAG ADDRESS
0087      EC2E XOPCRU   EQU   >EC2E         EVMBUG XOP R12 ADDRESS
0088 00A2 0010 TABLE    DATA  >10,>34       9600 BAUD
     00A4 0034
0089 00A6 0040          DATA  >40,>D0       2400 BAUD
     00A8 00D0
0090 00AA 0200          DATA  >200,>4D0     300  BAUD
     00AC 04D0
0091 00AE 0400          DATA  >400,>638     110  BAUD
```

FIGURE 8-18. SAMPLE PROGRAM TO CONVERSE THROUGH MAIN AND AUXILIARY TMS 9902s. (2 of 3)



```
       00B0 0638
  0092 00B2   62  CTL    BYTE >62                 9902 CONTROL
  0093 00B3   0D  BGNMSG BYTE >0D,>0A,>00
       00B4   0A
       00B5   00
  0094 00B6 0000  REGS   DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       00B8 0000
       00BA 0000
       00BC 0000
       00BE 0000
       00C0 0000
       00C2 0000
       00C4 0000
       00C6 0000
       00C8 0000
       00CA 0000
       00CC 0000
       00CE 0000
       00D0 0000
       00D2 0000
       00D4 0000
  0095                       END
 NO ERRORS.    NO WARNINGS
```

FIGURE 8-18. SAMPLE PROGRAM TO CONVERSE THROUGH MAIN AND AUXILIARY TMS 9902s. (3 of 3)

APPENDIX A

990 OBJECT RECORD FORMAT

A.1. GENERAL

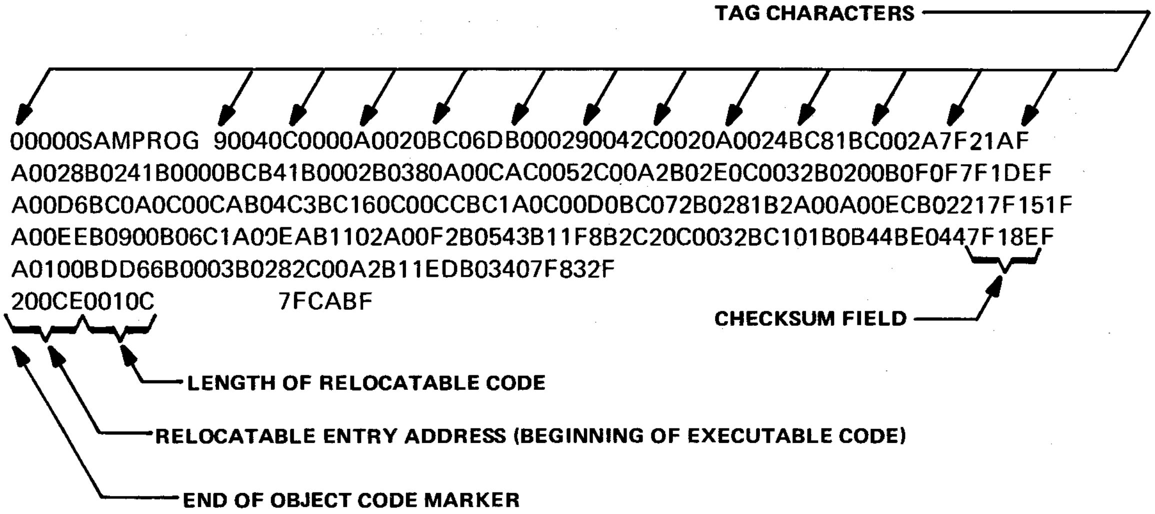
The TMS 9995 uses the standard 990 family object record format.

The required object Code can be produced during execution of the TMS 9995 EVM assembler or on any assembler present on a 9900 host system. This object format has a tag character for each 16-bit word of coding which flags the loader to perform one of several operations.These operations include:

- Load the code at a user-specified absolute address and resolve relative addresses. (Most assemblers assemble a program as if it were loaded at memory address >0000; thus, relative addresses have to be resolved.
- Load entire program at a specific address.
- Set the program counter to the entry address after loading.
- Check for checksum errors that would indicate a data error in an object record.

A.2. STANDARD 990 OBJECT CODE

Standard 990 object code consists of a string of hexadecimal digits, each representing four bits, as shown in Figure A-1, below:



```
00000SAMPROG 90040C0000A0020BC06DB000290042C0020A0024BC81BC002A7F21AF
A0028B0241B0000BCB41B0002B0380A00CAC0052C00A2B02E0C0032B0200B0F0F7F1DEF
A00D6BC0A0C00CAB04C3BC160C00CCBC1A0C00D0BC072B0281B2A00A00ECB02217F151F
A00EEB0900B06C1A00EAB1102A00F2B0543B11F8B2C20C0032BC101B0B44BE0447F18EF
A0100BDD66B0003B0282C00A2B11EDB03407F832F
200CE0010C            7FCABF
```

TAG CHARACTERS
CHECKSUM FIELD
LENGTH OF RELOCATABLE CODE
RELOCATABLE ENTRY ADDRESS (BEGINNING OF EXECUTABLE CODE)
END OF OBJECT CODE MARKER

FIGURE A-1. SAMPLE OBJECT CODE.

The object record consists of a number of tag characters, each followed by one or two fields, as defined in Table A-1. The first character of a record is the first tag character, which tells the loader which field or pair of fields follows the tag. The next tag character follows the end of the field or pair of fields associated with the preceeding tag character. When the assembler has no more data for the record, the assembler writes the tag character 7, followed by the checksum field, and the tag character F, which requires no fields. The assembler then fills the rest of the record with blanks, and begins a new record with the appropriate tag character.

TABLE A-1. OBJECT OUTPUT TAGS SUPPLIED BY ASSEMBLERS.

TAG CHARACTER	HEXADECIMAL FIELD (FOUR CHARACTERS)	SECOND FIELD	MEANING
0	Length of all relocatable code	8-char program identifier	Program start
1	Entry address	None	Absolute entry address
2	Entry Address	None	Relocatable entry address
3	Loc of last appearance of symbol	6-char symbol	External ref last used in relocatable code
4	Loc of last appearance of symbol	6-char symbol	External ref last used in abs code
5	Location	6-char symbol	Relocatable external definition
6	Location	6-char symbol	Absolute external definition
7	Checksum for current record	None	Checksum
8	Ignore checksum	None	Do not checksum for error
9	Load address	None	Absolute load add
A	Load address	None	Relocatable load add
B	Data	None	Absolute data
C	Data	None	Relocatable data

D	Load bias value*	None	Load point specified
F	None	None	End-of-record
G	Location	6-char symbol	Relocatable symbol definition
H	Locatio~	6-char symbol	Absolute symbol definition

Tag character 0 is followed by two fields. The first field contains the number of bytes of relocatable code, and the second field contains the program identifier assigned to the program by an IDT assembler directive. When no IDT directive is entered, the field contains blanks. The loader uses the program identifier to identify the program, and the number of bytes of relocatable code to determine the load bias for the next module or program. The PX9ASM assembler is unable to determine the value for the first field until the entire module has been assembled, so PX9ASM places a tag character 0, followed by a zero field, and the program identifier at the beginning of the object code file. At the end of the file, PX9ASM places another tag character zero followed by the number of bytes of relocatable code and eight blanks.

Tag characters 3 and 4 are used for external references. Tag character 3 is used when the last appearance of the symbol in the second field is a relocatable code. Tag character 4 is used when the last appearance of the symbol is absolute code. The hexadecimal field contains the location of the last appearance. The symbol in the second field is the external reference. Both fields are used by the linking loader to provide the desired linking to the external reference.

For each external reference in a program, there is a tag character in the object code, with a location or an absolute zero, and the symbol that is referenced. When the object code field contains absolute zero, no location in the program requires the address that corresponds to the reference (an IDT character string, for example). Otherwise, the address corresponding to the reference will be placed in the location specified in the object code by the linking loader. The location specified in the object code similarly contains absolute zero or another location. When it contains absolute zero, no further linking is required. When it contains a location, the address corresponding to the reference will be placed in that address by the linking loader. The location of each appearance of a reference in a program contains either an absolute zero or another location into which the linking loader will place the referenced address.

Tag characters 5 and 6 are used for external definitions. Tag character 5 is used when the location is relocatable. Tag character 6 is used when the location is absolute. Both fields are used by the linking loader to provide the desired linking to the external definition. The second field contains the symbol of the external definition.

Tag character 7 preceedes the checksum, which is an error detection word. The checksum is formed as the record is being written. It is the 2's complement of the sum of the 8-bit ASCII values of the characters of the record from the first tag of the record through the checksum tag 7. If the tag character 7 is replaced by an 8, the checksum will be ignored. The 8 tag can be used when object code is changed in editing and it is desired to ignore checksum.

Tag characters 9 and A are used with load addresses for data that follows. Tag character 9 is used when the load address is absolute. Tag character A is used when the load address is relocatable. The hexadecimal field contains the address at which the following data word is to be loaded. A load address is required for a data word that is to be placed in memory at some address other than the next address. The load address is used by the loader.

Tag characters B and C are used with data words. Tag character B is used when the data is absolute; an instruction word or a word that contains text characters or absolute constants, for example. Tag character C is used for a word that contains a relocatable address. The hexadecimal field contains the data word. The loader places the word in the memory location specified in the preceeding load address field, or in the memory location that follows the preceeding data word.

To have object code loaded at a specific memory address, preceed the object program with the D tag, followed by the desired memory address (e.g., DFD00).

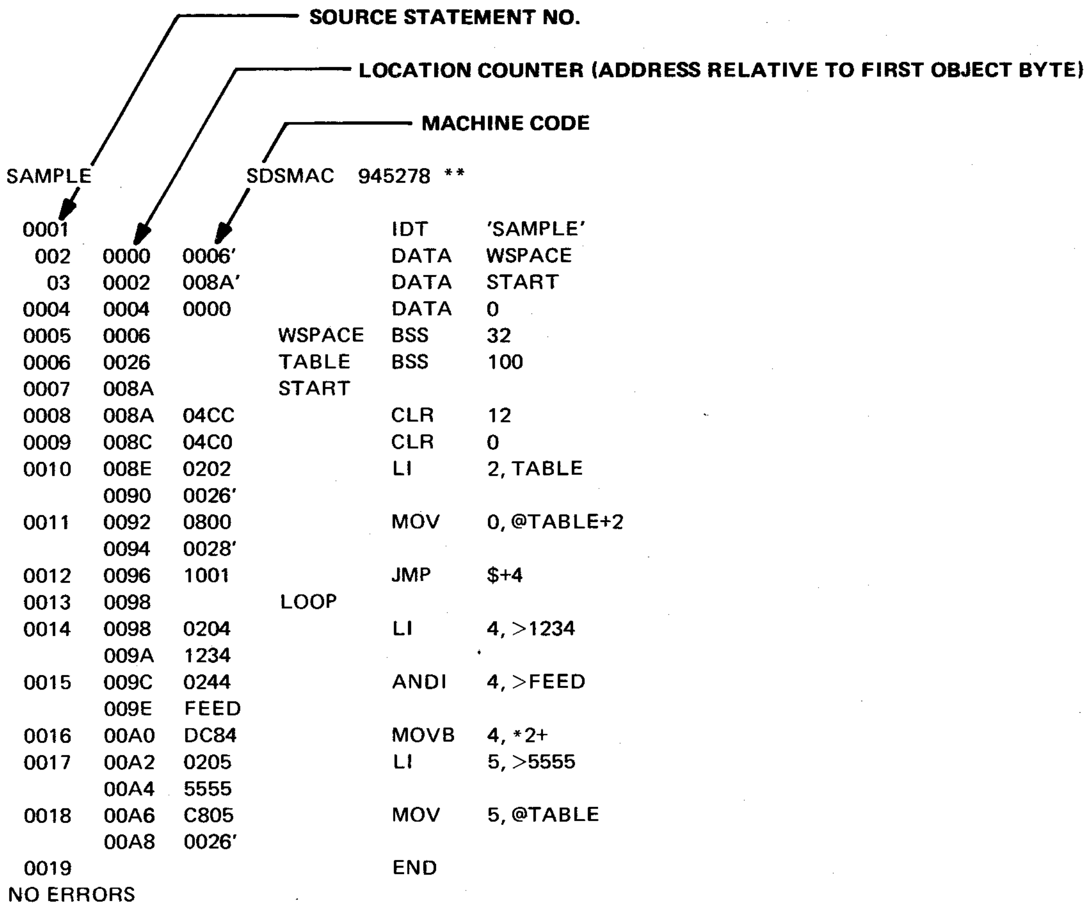
Tag character F indicates the end of record. It may be followed by blanks.

Tag characters G and H are used when the symbol table option is specified with other 990 assemblers. Tag character G is used when the location or value of the symbol is relocatable, and tag character H is used when the location or value of the symbol is absolute. The first field contains the location or value of the symbol, and the second field contains the symbol to which the location is assigned.

The last record of an object code file has a colon (:) in the first character position of the record, followed by blanks. This record is referred to as an end-of-module separator record.

EXAMPLE:

Figure 5-2, Section 5 is an example of an assembler source listing and corresponding object code. A comparison of the object tag characters and fields with the machine code in the source listing will show how object code is constructed for use by the loader.



```
SAMPLE                  SDSMAC  945278 **

0001                                IDT     'SAMPLE'
 002    0000    0006'               DATA    WSPACE
  03    0002    008A'               DATA    START
0004    0004    0000                DATA    0
0005    0006            WSPACE      BSS     32
0006    0026            TABLE       BSS     100
0007    008A            START
0008    008A    04CC                CLR     12
0009    008C    04C0                CLR     0
0010    008E    0202                LI      2, TABLE
        0090    0026'
0011    0092    0800                MOV     0, @TABLE+2
        0094    0028'
0012    0096    1001                JMP     $+4
0013    0098            LOOP
0014    0098    0204                LI      4, >1234
        009A    1234
0015    009C    0244                ANDI    4, >FEED
        009E    FEED
0016    00A0    DC84                MOVB    4, *2+
0017    00A2    0205                LI      5, >5555
        00A4    5555
0018    00A6    C805                MOV     5, @TABLE
        00A8    0026'
0019                                END
NO ERRORS
```

FIGURE A-2. SAMPLE ASSEMBLER SOURCE LISTING AND OBJECT CODE

APPENDIX B

ASCII CODE

TABLE B-1. ASCII CONTROL CODES

CONTROL	BINARY CODE	HEXADECIMAL CODE
NUL - Null	000 0000	00
SOH - Start of heading	000 0001	01
STX - Start of text	000 0010	02
ETX - End of text	000 0011	03
EOT - End of transmission	000 0100	04
ENQ - Enquiry	000 0101	05
ACK - Acknowledge	000 0110	06
BEL - Bell	000 0111	07
BS - Backspace	000 1000	08
HT - Horizontal tabulation	000 1001	09
LF - Line feed	000 1010	0A
VT - Vertical tab	000 1011	0B
FF - Form feed	000 1100	0C
CR - Carriage Return	000 1101	0D
SO - Shift out	000 1110	0E
SI - Shift in	000 1111	0F
DLE - Data link escape	001 0000	10
DC1 - Device control 1	001 0001	11
DC2 - Device control 2	001 0010	12
DC3 - Device control 3	001 0011	13
DC4 - Device control 4 (stop)	001 0100	14
NAK - Negative acknowledge	001 0101	15
SYN - Synchronous idle	001 0110	16
ETB - End of transmission bloc	001 0111	17
CAN - Cancel	001 1000	18
EM - End of medium	001 1001	19
SUB - Substitute	001 1010	1A
ESC - Escape	001 1011	1B
FS - File separator	001 1100	1C
GS - Group separator	001 1101	1D
RS - Record separator	001 1110	1E
US - Unit separator	001 1111	1F
DEL - Delete/rubout	111 1111	7F

NOTE

Hexadecimal codes 01-1F can be generated using most keyboard devices with the CONTROL (SHIFT) key pressed while pressing another keyboard key. For example, hexadecimal codes 01-19 can be generated on the TMS 9995 using the SHIFT key and keys A through Y respectively, with the exception of keys V and X, which have shift functions dedicated to display right and cancel respectively.

TABLE B-2. ASCII CHARACTER CODES

ASCII CHARACTER	BINARY CODE	HEX CODE	ASCII CHARACTER	BINARY CODE	HEX CODE
Space	010 0000	20	@	100 0000	40
!	010 0001	21	A	100 0001	41
"(dbl quote)	010 0010	22	B	100 0010	42
#	010 0011	23	C	100 0011	43
$	010 0100	24	D	100 0100	44
%	010 0101	25	E	100 0101	45
&	010 0110	26	F	100 0110	46
´(sgl quote)	010 0111	27	G	100 0111	47
(	010 1000	28	H	100 1000	48
)	010 1001	29	I	100 1001	49
*(asterisk)	010 1010	2A	J	100 1010	4A
+	010 1011	2B	K	100 1011	4B
,(comma)	010 1100	2C	L	100 1101	4C
-(minus)	010 1101	2D	M	100 1101	4D
.(period)	010 1110	2E	N	100 1110	4E
/	010 1111	2F	O	100 1111	4F
0	011 0000	30	P	101 0000	50
1	011 0001	31	Q	101 0001	51
2	011 0010	32	R	101 0010	52
3	011 0011	33	S	101 0011	53
4	011 0100	34	T	101 0100	54
5	011 0101	35	U	101 0101	55
6	011 0110	36	V	101 0110	56
7	011 0111	37	W	101 0111	57
8	011 1000	38	X	101 1000	58
9	011 1001	39	Y	101 1001	59
:	011 1010	3A	Z	101 1010	5A
;	011 1011	3B	[	101 1011	5B
<	011 1100	3C		101 1100	5C
	011 1101	3D	]	101 1101	5D
>	011 1110	3E		101 1110	5E
?	011 1111	3F	-(underln)	101 1111	5F

TABLE B-2. ASCII CHARACTER CODES (CONTINUED)

ASCII CHARACTER	BINARY CODE	HEX CODE	ASCII CHARACTER	BINARY CODE	HEX CODE
	110 0000	60	p	111 0000	70
a	110 0001	61	q	111 0001	71
b	110 0010	62	r	111 0010	72
c	110 0011	63	s	111 0011	73
d	110 0100	64	t	111 0100	74
e	110 0101	65	u	111 0101	75
f	110 0110	66	v	111 0110	76
g	110 0111	67	w	111 0111	77
h	110 1000	68	x	111 1000	78
i	110 1001	69	y	111 1001	79
j	110 1010	6A	z	111 1010	7A
k	110 1011	6B	{	111 1011	7B
l	110 1100	6C	|	111 1100	7C
m	110 1101	6D	}	111 1101	7D
n	110 1110	6E	~	111 1110	7E
o	110 1111	6F			

APPENDIX C

BINARY, DECIMAL AND HEXADECIMAL NUMBERING SYSTEMS

C.1. GENERAL

This appendix covers the numbering systems which are used throughout this manual:

- BINARY (Base 2)

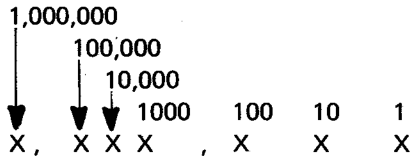
- DECIMAL (Base 10)

- HEXADECIMAL (Base 16)

C.2 POSITIVE NUMBERS

C.2.1 Decimal (Base 10)

When a numerical quantity is viewed from right to left, the right-most digit represents the base number to the exponent 0. The next digit represents the base number to the exponent 1, the next to the exponent 2, then exponent 3, and so on. For example, using the base 10 (decimal):



OR

10^6	10^5	10^4	10^3	10^2	10^1	10^0
X,	X	X	X,	X	X	X

For example, 75,265 can be broken down as follows:

$$\underline{75,\ 264}$$

$$\begin{aligned}
4 \times 10^0 &= 4 \times 1 &&= 4\\
6 \times 10^1 &= 6 \times 10 &&= 60\\
2 \times 10^2 &= 2 \times 100 &&= 200\\
5 \times 10^3 &= 5 \times 1000 &&= 5000\\
7 \times 10^4 &= 7 \times 10{,}000 &&= \underline{+70000}\\
& && \ 75264_{10}
\end{aligned}$$

C.2.2 Binary (Base 2)

Base 10 numbers use ten digits, base 2 numbers use only 0 and 1. When viewed from right to left, they each represent the number 2 to the powers 0, 1, 2, etc., respectively, as shown below:

2^{15}		2^6	2^5	2^4	2^3	2^2	2^1	2^0
(32,768)	●●●	(64)	(32)	(16)	(8)	(4)	(2)	(1)
X	●●●	X	X	X	X	X	X	X

For example, Binary 11011 can be translated into base 10 as follows:

$$\underline{1}\ \ \underline{1}\ \ \underline{0}\ \ \underline{1}\ \ \underline{1}$$

$$\begin{aligned}
1 \times 2^0 &= 1 \times 1 &&= 1\\
1 \times 2^1 &= 1 \times 2 &&= 2\\
0 \times 2^2 &= 0 \times 4 &&= 0\\
1 \times 2^3 &= 1 \times 8 &&= 8\\
1 \times 2^4 &= 1 \times 16 &&= \underline{+16}\\
& && \ 27_{10}
\end{aligned}$$

or, Binary 11011 equals 27.

Binary is the language of the digital computer. For example, to place the decimal quantity 23 into a 16-bit memory cell, set the bits to the following:

```
BIT 0                              15
    -------------------------------
    0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1
    -------------------------------
```

which is 1 + 2 + 4 + 16 = 23.

C.2.3 Hexadecimal (Base 16)

Whereas binary uses two digits and decimal uses ten digits, hexadecimal uses 16 (0 to 9, A, B, C, D, E, and F).

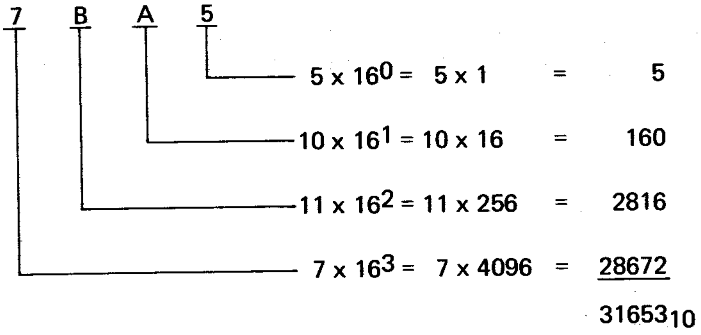
The letters A through F are used to represent the decimal numbers 10 through 15, as shown below:

N_{10}	N_{16}	N_{10}	N_{16}
0	0	8	8
1	1	9	9
2	2	10	A
3	3	11	B
4	4	12	C
5	5	13	D
6	6	14	E
7	7	15	F

When viewed from right to left, each digit in a hexadecimal number is a multiplier of 16 to the powers 0, 1, 2, 3, etc., as shown below:

16^3	16^2	16^1	16^0
(4096)	(256)	(16)	(1)
X	X	X	X

For example, >7BA5 can be translated into base 10 as follows:


7 B A 5
5 x 16^0 = 5 x 1 = 5
10 x 16^1 = 10 x 16 = 160
11 x 16^2 = 11 x 256 = 2816
7 x 16^3 = 7 x 4096 = 28672
31653$_{10}$

Or, >7BA5 equals 31,653.

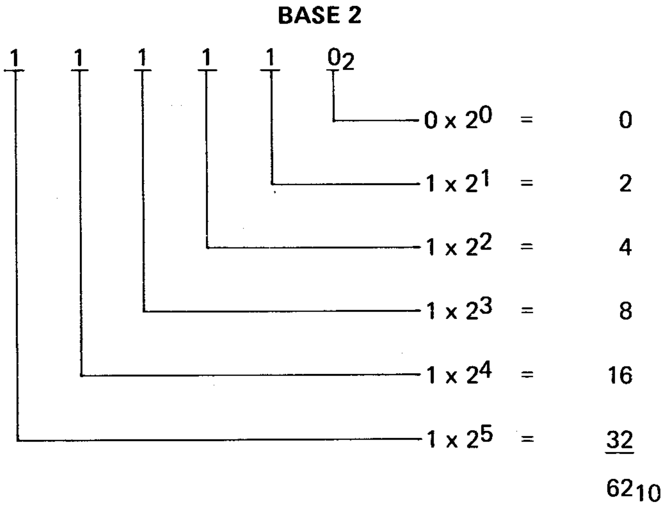
Because it would be awkward to write out 16-digit binary numbers to show the contents of a 16-bit memory word, hexadecimal is used instead. Thus:

>003E

is used instead of

0000 0000 0011 1110 (Binary)

to represent 62, as computed below:



(Note that separating the 16 binary bits into four-bit parts facilitates recognition and translation into hexadecimal.)

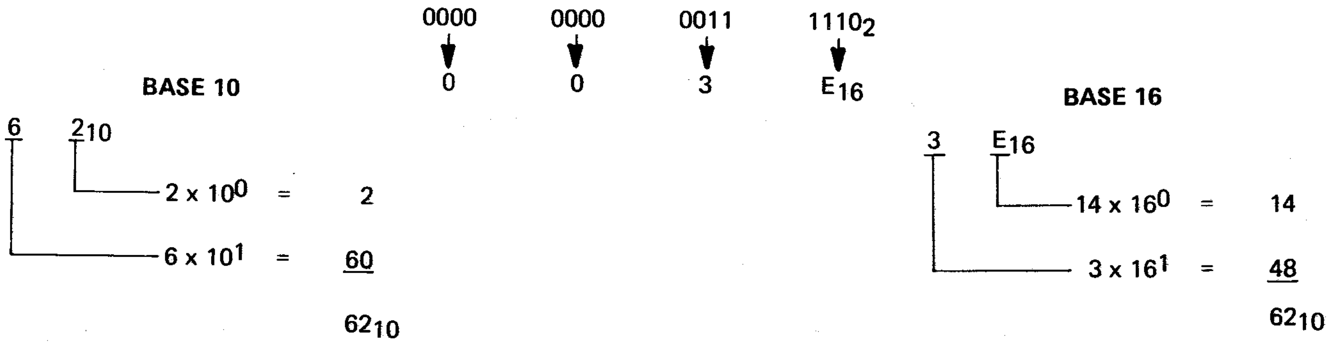


Table C-1 is a chart for converting decimal to hexadecimal and vice-versa. Table C-2 shows binary, decimal and hexdadecimal equivalents for numbers 0 to 15. Note that Table C-1 is divided into four parts, each part representing four of the 16 bits of a memory cell or word (bits 0 to 15), with bit 0 being the most significant bit (MSB) and bit 15 being the least significant bit (LSB). Note also that the MSB is on the left and and represents the highest poer of 2, and the LSB is on the right and represents the 0 power of 2, or 1. As explained later, the MSB can also be used to signify number polarity (+ or -).

To convert a binary number to decimal or hexadecimal, convert the positive binary value, as described in paragraph C-4.

TABLE C-1. HEXADECIMAL/DECIMAL CONVERSION CHART

	MSB							LSB
	16		16		16		16	
BITS	0 1 2 3		4 5 6 7		8 9 10 11		12 13 14 15	
	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
	0	0	0	0	0	0	0	0
	1	4 096	1	256	1	16	1	1
	2	8 192	1	512	1	32	2	2
	3	12 288	3	768	3	48	3	3
	4	16 384	4	6 024	4	64	4	4
	5	20 480	5	1 280	5	80	5	5
	6	24 576	6	1 536	6	96	6	6
	7	28 672	7	1 792	7	112	7	7
	8	32 768	8	2 048	8	128	8	8
	9	36 864	9	2 304	9	144	9	9
	A	40 960	A	2 560	A	160	A	10
	B	45 056	B	2 816	B	176	B	11
	C	49 152	C	3 072	C	192	C	12
	D	53 248	D	3 328	D	208	D	13
	E	57 344	E	3 584	E	224	E	14
	F	61 440	F	3 840	F	240	F	15

To convert a number from hexadecimal, add the decimal equivalents for each hex digit. For example, >7A82 would equal in decimal 28,672 + 2,560 + 128 + 2. To convert hexadecimal to decimal, find the nearest decimal number in the above table less than or equal to the number being converted. Set down the hexadecimal equivalent, then subtract this number from the nearest decimal number. Using the remainder(s), repeat this process. For example:

```
31,362 = >7000 + 2690          7000
 2,690 = >A00 + 130             A00
   130 = >80 + 2                 80
     2 = >2                       2
                              ______
                              >7A82
```

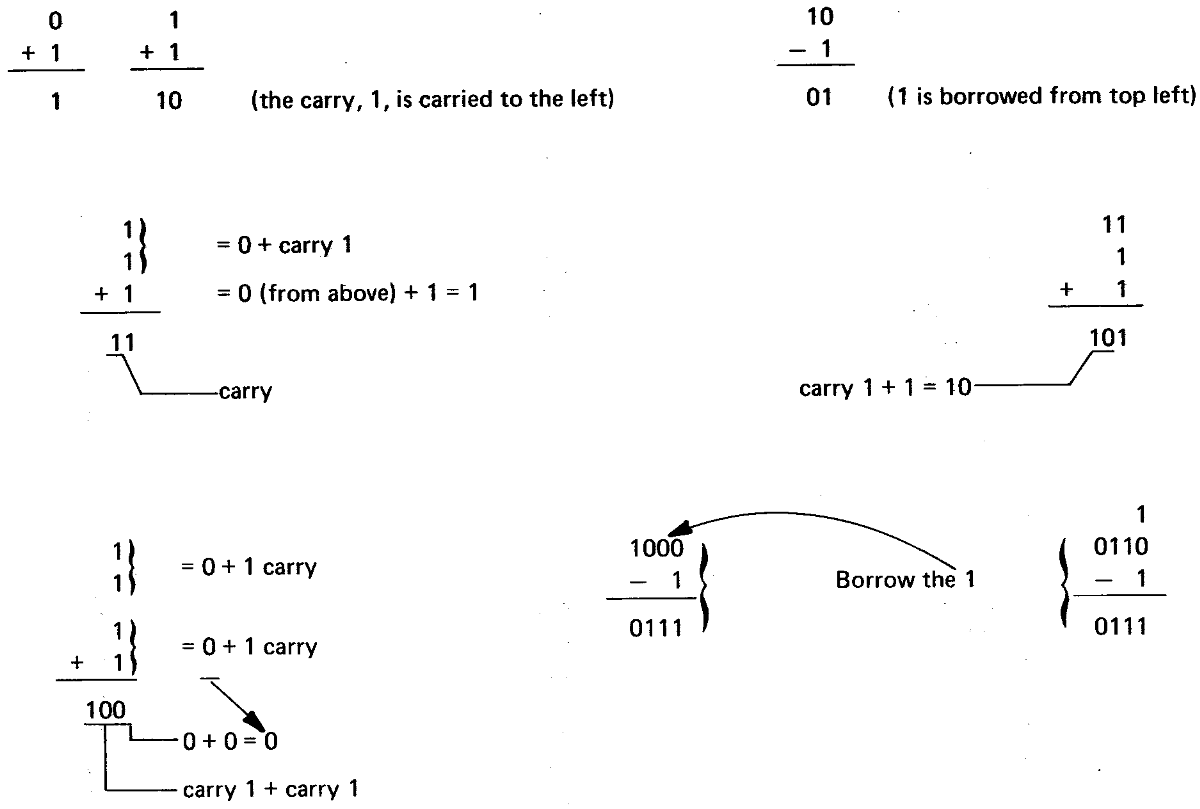
TABLE C-2. BINARY, DECIMAL, AND HEXADECIMAL EQUIVALENTS.

BINARY	DECIMAL	HEXADECIMAL (>)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F
1 0000	16	10
1 0001	17	11
1 0010	18	12
1 0011	19	13
1 0100	20	14
1 0101	21	15
1 0110	22	16
1 0111	23	17
1 1000	24	18
1 1001	25	19
1 1010	26	1A
1 1011	27	1B
1 1100	28	1C
1 1101	29	1D
1 1110	30	1E
1 1111	31	1F
10 0000	32	20

C.3. ADDING AND SUBTRACTING BINARY

Adding and subtracting in binary uses the same conventions as for decimal (i.e., carrying over in addition and borrowing in subtraction).

Basically:



C-4. POSITIVE/NEGATIVE CONVERSION (BINARY)

To compute the negative equivalent of a positive binary or hexadecimal number, or interpret a binary or hexadecimal negative number (to determine its positive equivalent), use the two´s complement of the binary number:

NOTE

To convert a binary number to decimal, convert the positive binary value, NOT the negative binary value, and add the sign.

Two´s complementing a binary number involves two simple steps:

1. Obtain the one´s complement of the number (1´s become 0´s; 0´s become 1´s [i.e., invert the bits]).

2. Add 1 to the one´s complement.

For example, with the MSB (left-most bit) being a sign bit:

010	$(+2_2)$	111	(-1_2)	110	(-2_2)	101	(-3_2)
101	Invert	000	Invert	001	Invert	010	Invert
+ 1	Add 1	+ 1	Add 1	+ 1	Add1	+ 1	
110	(-2_2)	001	$(+1_2)$	010	$(+2_2)$	011	$(+3_2)$

This can be expanded to 16-bit positive numbers:

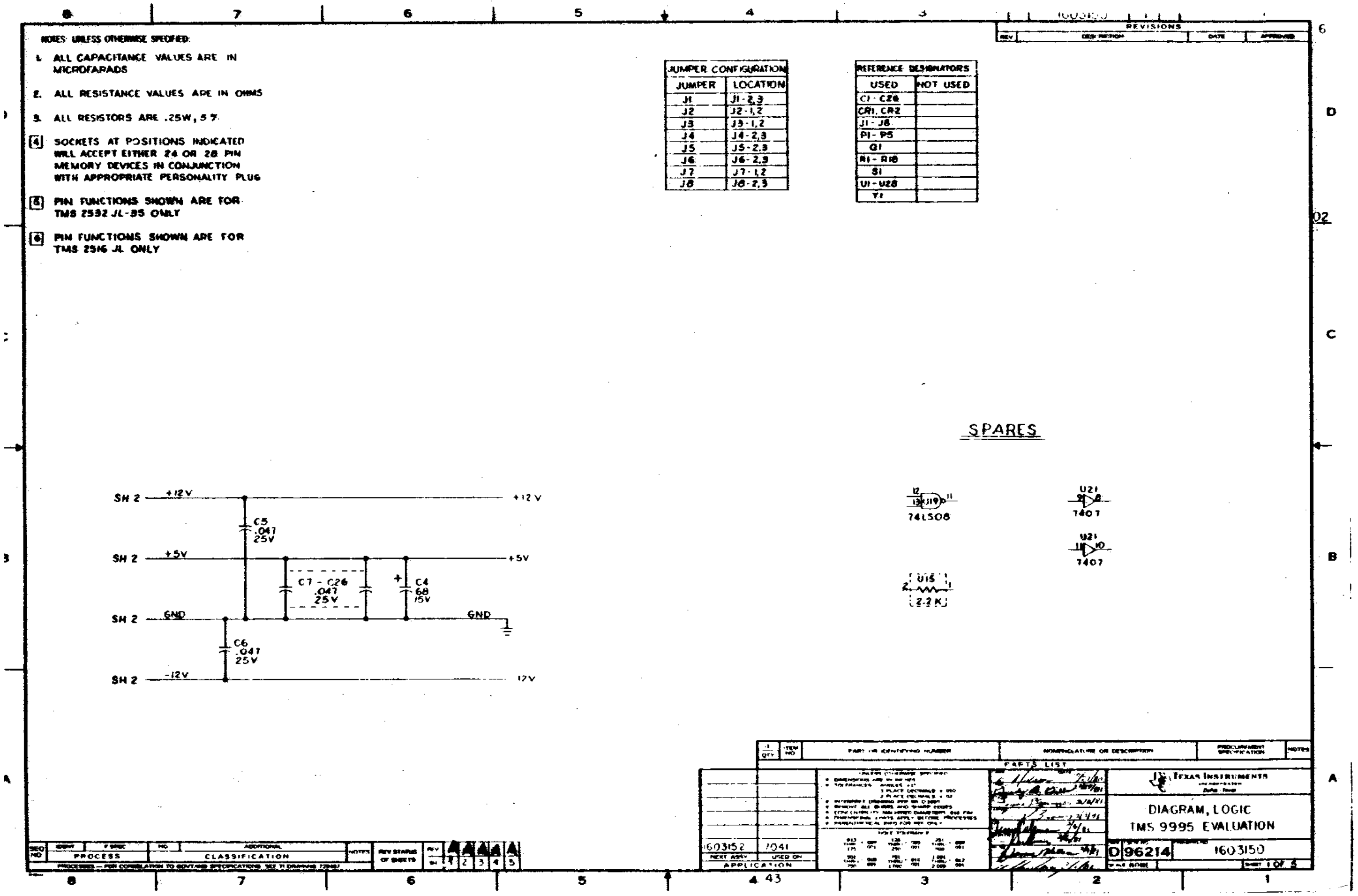
$=39F6_{16})$	0011	1001	1111	0110	$(39F6_{16} = +14{,}838_{10})$
	1100	0110	0000	1001	Invert
				+1	Add 1
$=C60A_{16})$	1100	0110	0000	1010	$(C60A_{16} = -14{,}838_{10})$ Two's Complement
	SIGN BIT (–)				

And to 16-bit negative numbers:

$(=C60A_{16})$	1100	0110	0000	1010	$(C60A_{16} = -14{,}838_{10})$
	0011	1001	1111	0101	Invert
				+1	Add 1
$(=39F6_{16})$	0011	1001	1111	0110	$(39F6_{16} = +14{,}838_{10})$ Two's Complement
	SIGN BIT (+)				

APPENDIX D

TMS9995 EVM SCHEMATICS



NOTES: UNLESS OTHERWISE SPECIFIED:

1. ALL CAPACITANCE VALUES ARE IN MICROFARADS
2. ALL RESISTANCE VALUES ARE IN OHMS
3. ALL RESISTORS ARE .25W, 5 %

[4] SOCKETS AT POSITIONS INDICATED WILL ACCEPT EITHER 24 OR 28 PIN MEMORY DEVICES IN CONJUNCTION WITH APPROPRIATE PERSONALITY PLUG

[5] PIN FUNCTIONS SHOWN ARE FOR TMS 2532 JL-35 ONLY

[6] PIN FUNCTIONS SHOWN ARE FOR TMS 2516 JL ONLY

JUMPER CONFIGURATION	
JUMPER	LOCATION
J1	J1 - 2,3
J2	J2 - 1,2
J3	J3 - 1,2
J4	J4 - 2,3
J5	J5 - 2,3
J6	J6 - 2,3
J7	J7 - 1,2
J8	J8 - 2,3

REFERENCE DESIGNATORS	
USED	NOT USED
C1 - C26	
CR1, CR2	
J1 - J8	
P1 - P5	
Q1	
R1 - R18	
S1	
U1 - U28	
Y1	

REVISIONS

PARTS LIST

TEXAS INSTRUMENTS

DIAGRAM, LOGIC
TMS 9995 EVALUATION

D 96214 1603150

SHEET 1 OF 5

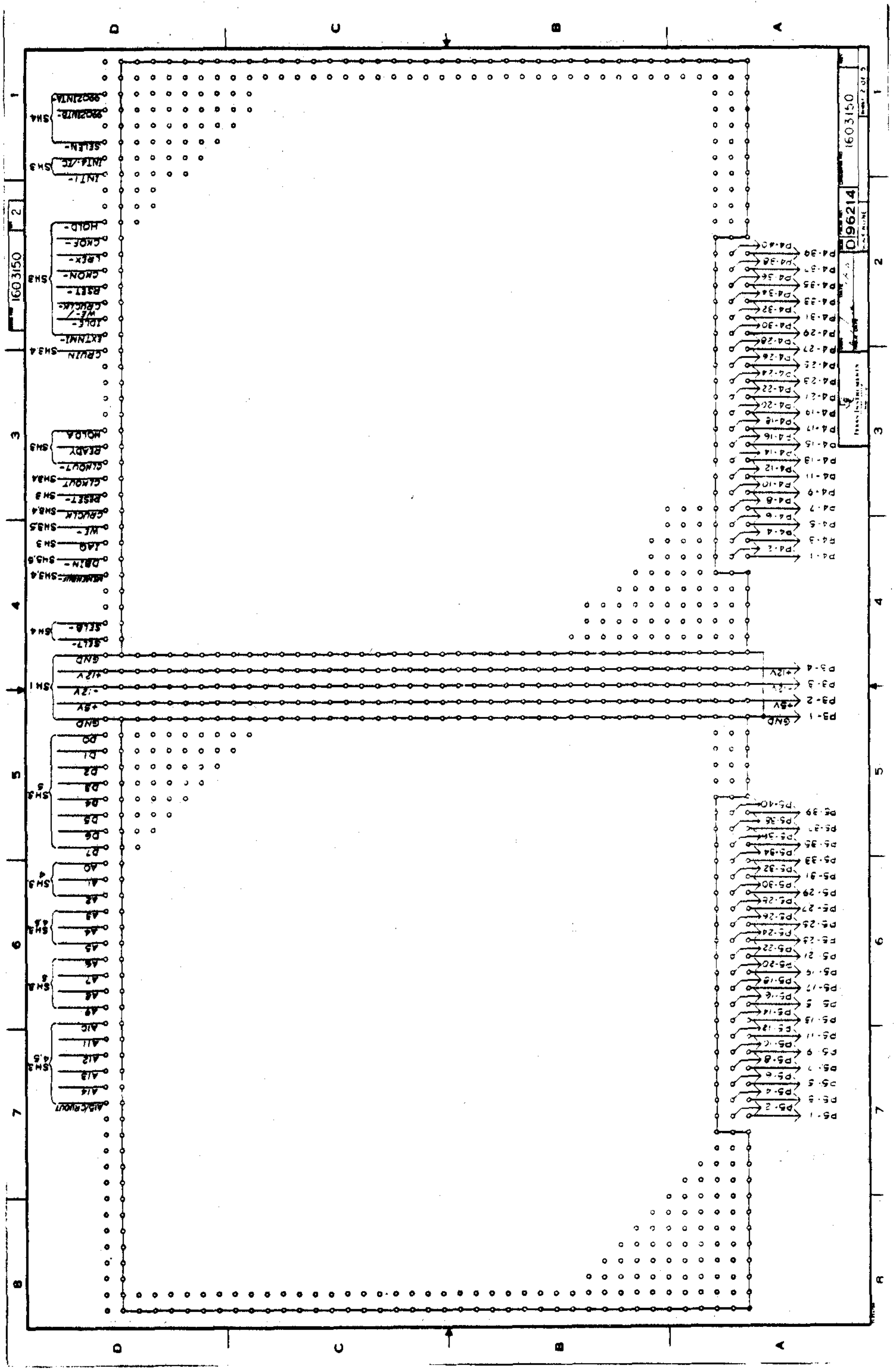
1603152 7041

NEXT ASSY USED ON

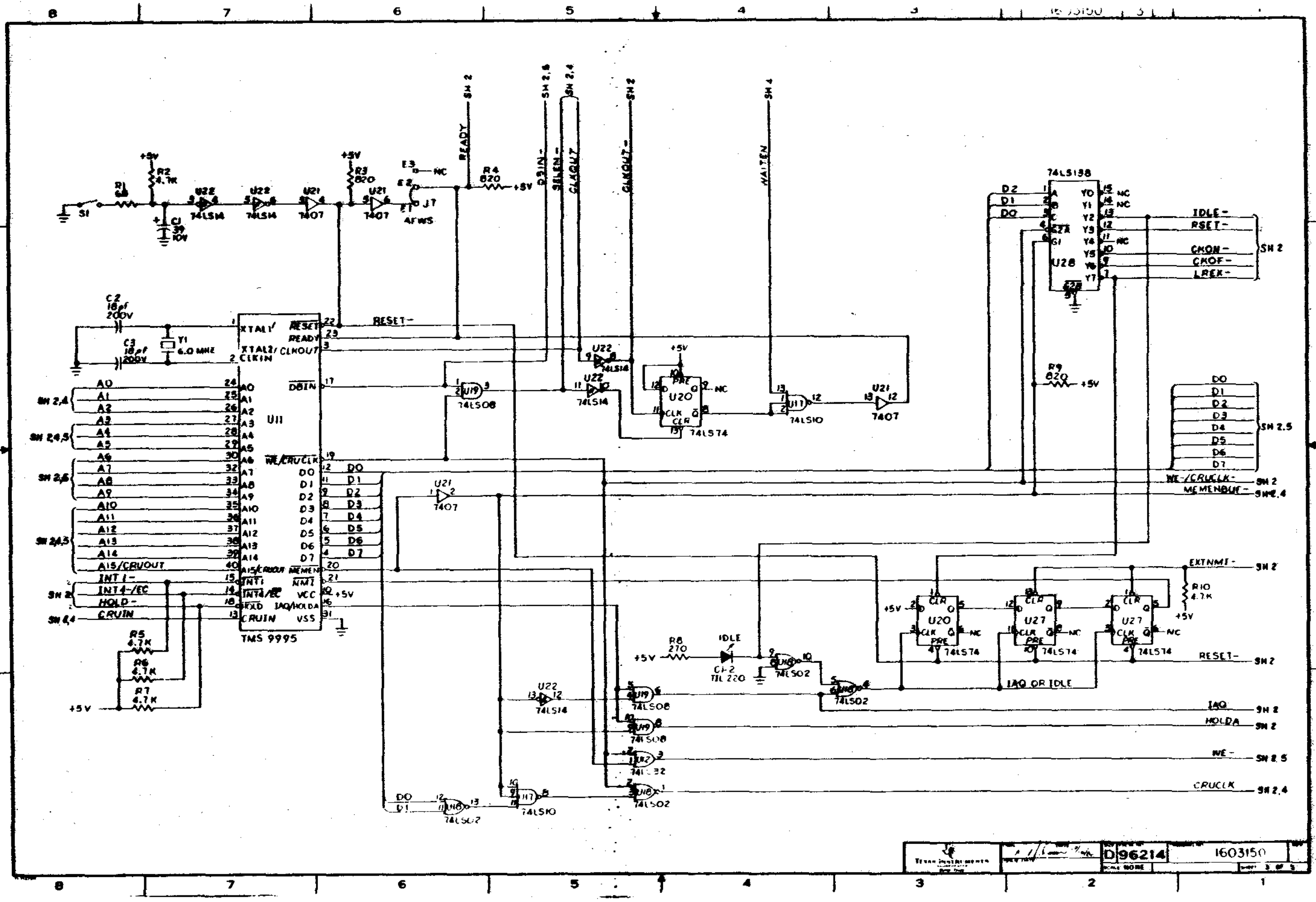
APPLICATION

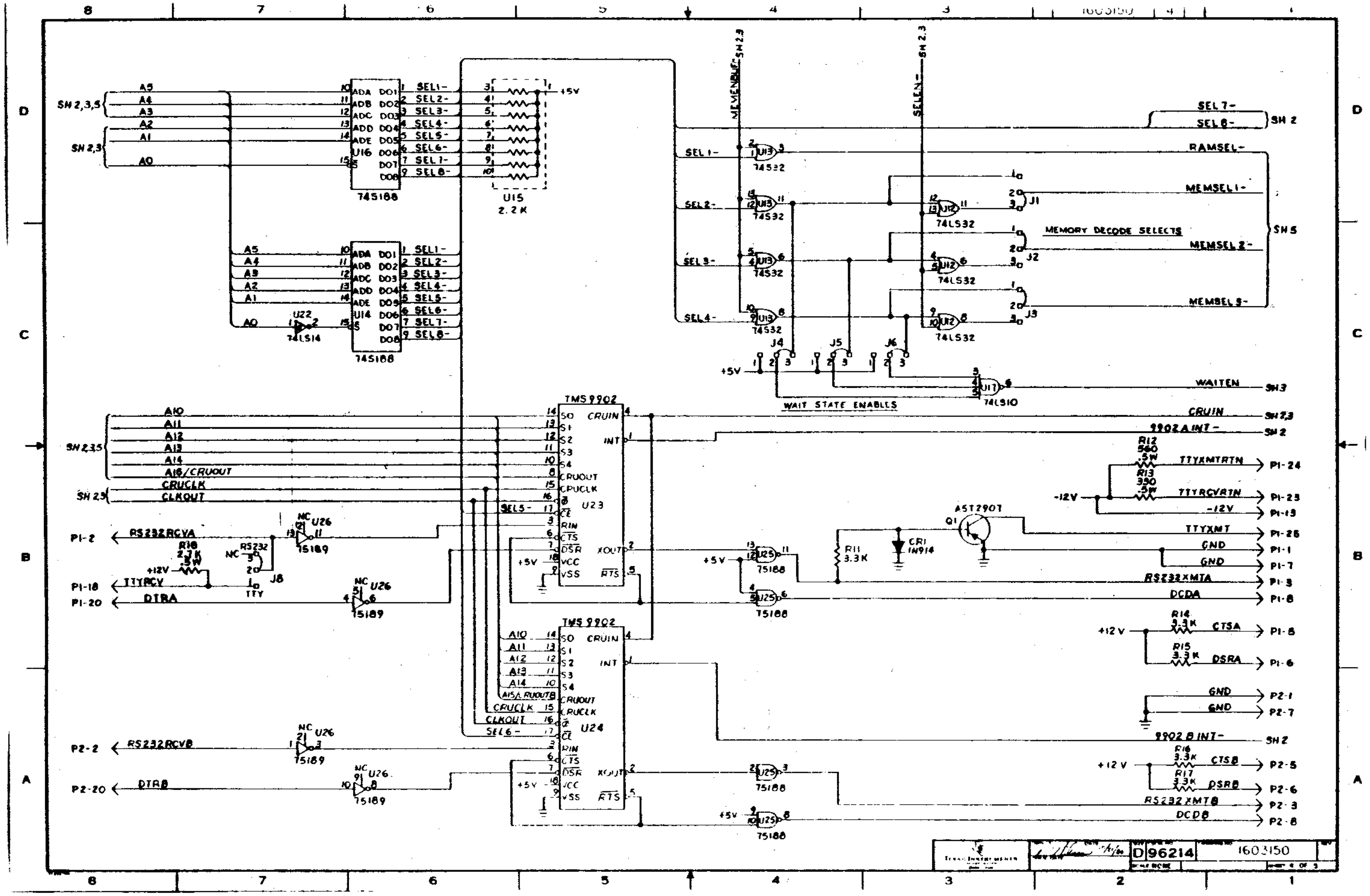
PROCESS CLASSIFICATION

43

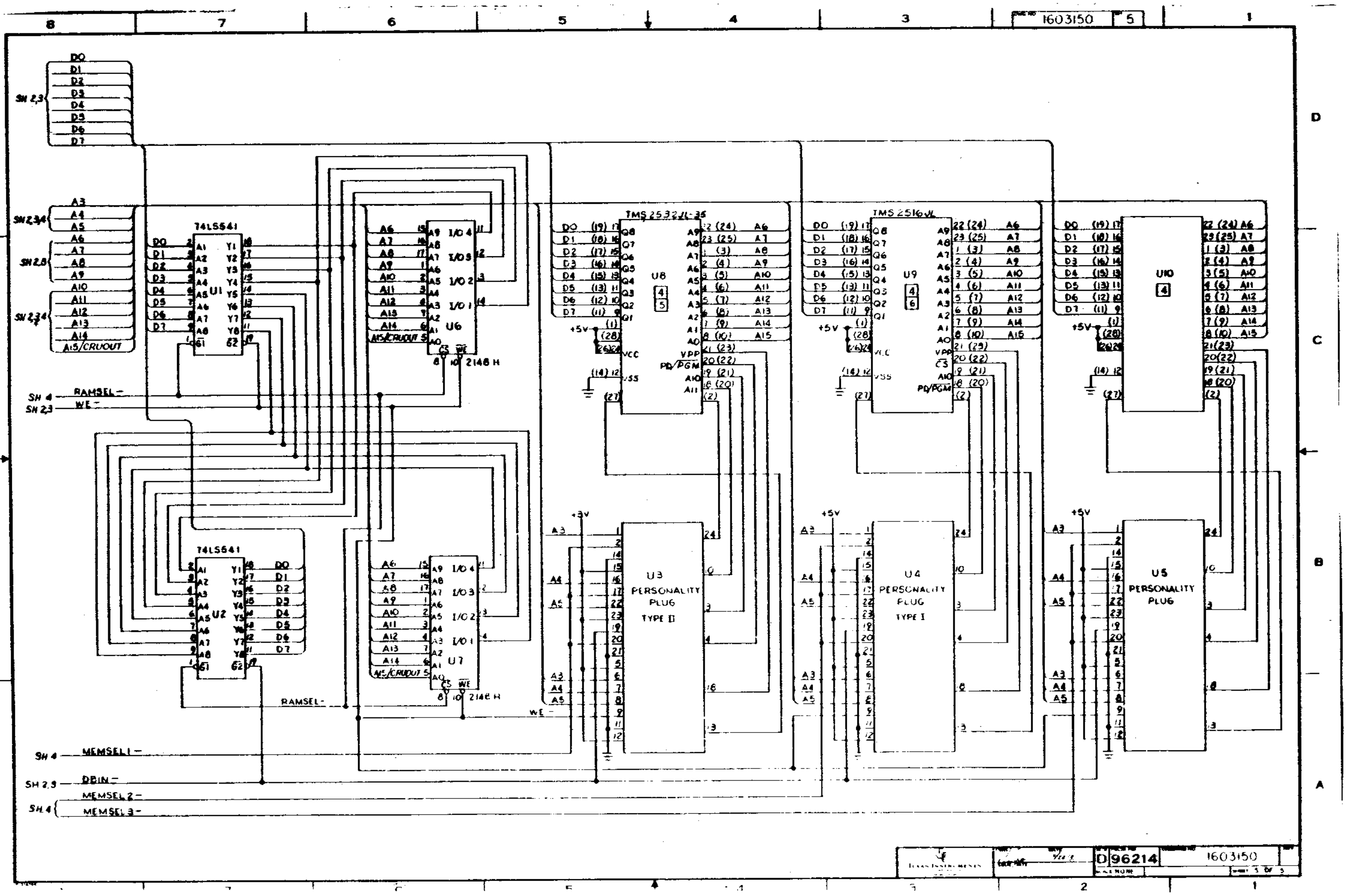


D
C
B
A
1
2
3
4
5
6
7
8
1603150
2
D96214
TEXAS INSTRUMENTS
SH4
SELEN-
9902INTR-
9902INTA
SH3
INT1-
INT4/EC
SH2
HOLD-
CKOE-
LREX-
CKON-
RSET-
CRUCLK-
WE-/
IDLE-
EXTINT-
SH3,4
CRUIN
SH3
HOLDA
READY
CLKOUT-
SH3,4
CLKOUT
SH3
RESET-
SH3,4
CRUCLK
SH3,5
WE-
SH3
IAQ
SH3,5
DBIN-
SH3,4
MEMEN-
SH4
SELB-
SELT-
SH1
GND
+12V
-12V
+5V
GND
SH3,5
D0
D1
D2
D3
D4
D5
D6
D7
SH3,4
A0
A1
A2
SH3,4,5
A3
A4
A5
SH3,5
A6
A7
A8
A9
SH3,4,5
A10
A11
A12
A13
A14
A15/CRUOUT
P4-1
P4-2
P4-40
P3-1
P3-2
P3-3
P3-4
P5-1
P5-2
P5-40


8
7
6
5
4
3
2
1
D
C
B
A
S1
R1
68
+5V
R2
4.7K
C1
39
10V
U22
74LS14
U21
7407
R3
820
E3
NC
E2
E1
J7
AFWS
READY
SH 2
R4
820
DBIN-
SH 2,6
SELEN-
SH 2,4
CLKOUT
CLKOUT-
WAITEN
SH 4
C2
18pf
200V
C3
18pf
200V
Y1
6.0 MHZ
XTAL1
XTAL2/CLKIN
RESET
READY
CLKOUT
RESET-
DBIN
U11
TMS 9995
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
A11
A12
A13
A14
A15/CRUOUT
SH 2,4
SH 2,4,5
SH 2,6
WE/CRUCLK
D0
D1
D2
D3
D4
D5
D6
D7
MEMEN
NMI
VCC
IAQ/HOLDA
VSS
INT1-
INT4-/EC
HOLD-
CRUIN
R5
4.7K
R6
4.7K
R7
4.7K
U19
74LS08
U20
PRE
CLR
CLK
74LS74
U17
74LS10
74LS138
U28
Y0
Y1
Y2
Y3
Y4
Y5
Y6
Y7
G1
G2A
G2B
IDLE-
RSET-
CKON-
CKOF-
LREX-
R9
820
SH 2,5
WE-/CRUCLK-
MEMENBUF-
SH 2,4
EXTNMI-
R10
4.7K
U27
IAQ OR IDLE
R8
270
IDLE
CR2
TIL 220
U18
74LS02
74LS32
IAQ
HOLDA
WE-
CRUCLK
TEXAS INSTRUMENTS
D 96214
1603150
SHEET 3 OF 5



74S188
U16
U14
74S188
U15
2.2K
+5V
U22
74LS14
SEL1-
SEL2-
SEL3-
SEL4-
SEL5-
SEL6-
SEL7-
SEL8-
SH 2,3,5
SH 2,3
A5
A4
A3
A2
A1
A0
MEMENABLE-SH 2,3
SELEN- SH 2,3
74S32
74LS32
J1
J2
J3
J4
J5
J6
U13
U12
U17
74LS10
WAIT STATE ENABLES
MEMORY DECODE SELECTS
RAMSEL-
MEMSEL1-
MEMSEL2-
MEMSEL3-
SH 2
SH 5
WAITEN
SH 3
TMS 9902
U23
U24
A10
A11
A12
A13
A14
A15/CRUOUT
CRUCLK
CLKOUT
SH 2,3,5
SH 2,3
CRUIN
9902 A INT-
9902 B INT-
RS232RCVA
RS232RCVB
DTRA
DTRB
TTYRCV
TTY
RS232
NC
R18
2.7K
.5W
+12V
J8
U26
75189
U25
75188
R11
3.3K
CR1
1N914
A5T2907
Q1
R12
560
.5W
R13
390
.5W
-12V
TTYXMTRTN
TTYRCVRTN
TTYXMT
GND
RS232XMTA
DCDA
R14
3.3K
CTSA
R15
3.3K
DSRA
R16
3.3K
CTSB
R17
3.3K
DSRB
RS232XMTB
DCDB
P1-2
P1-18
P1-20
P1-24
P1-23
P1-13
P1-25
P1-1
P1-7
P1-3
P1-8
P1-5
P1-6
P2-1
P2-7
P2-5
P2-6
P2-3
P2-8
P2-2
P2-20
TEXAS INSTRUMENTS
D 96214
1603150
8
7
6
5
4
3
2
1
D
C
B
A


8
7
6
5
4
3
2
1
D
C
B
A
1603150
5
SH 2,3
D0
D1
D2
D3
D4
D5
D6
D7
SH 2,3,4
A3
A4
A5
SH 2,8
A6
A7
A8
A9
A10
SH 2,3,4
A11
A12
A13
A14
A15/CRUOUT
74LS541
U1
74LS541
U2
U6
U7
2148 H
I/O 1
I/O 2
I/O 3
I/O 4
CS
WE
A15/CRUOUT 5
SH 4
RAMSEL-
SH 2,3
WE-
TMS2532JL-35
U8
TMS2516JL
U9
U10
+5V
VCC
VSS
VPP
PD/PGM
U3
PERSONALITY
PLUG
TYPE II
U4
PERSONALITY
PLUG
TYPE I
U5
PERSONALITY
PLUG
MEMSEL1-
DBIN-
MEMSEL2-
MEMSEL3-
SH 4
SH 2,3
TEXAS INSTRUMENTS
D
96214
1603150
SCALE NONE
SHEET 5 OF 5

APPENDIX E

TMS 9995 MICROCOMPUTER

ARCHITECTURE

1. INTRODUCTION

1.1 DESCRIPTION

The TMS 9995 microcomputer is a single-chip 16-bit central processing unit (CPU) with 256 bytes of on-chip random access memory (RAM). A member of the TMS 9900 family of microprocessor and peripheral circuits, the TMS 9995 is fabricated using N-channel silicon-gate MOS technology. The rich instruction set of the TMS 9995 is based upon a unique memory-to-memory architecture that features multiple register files resident in memory. Memory-resident register files allow faster response to interrupts and increased programming flexibility. The inclusion of RAM, timer function, clock generator, interrupt interface, and a flexible flag register on-chip facilitates support of small system implementations.

All members of the TMS 9900 family of peripheral circuits are compatible with the TMS 9995. Providing a performance upgrade to the TMS 9900 microprocessor, the TMS 9995 instruction set is an opcode-compatible superset of the TMS 9900 processor family.

1.2 KEY FEATURES

- 16-Bit instruction word
- Memory-to-Memory architecture
- 65,536 byte/32,768 word directly addressable memory address space
- Minicomputer instruction set including signed multiply and divide instructions
- Multiple 16-word register files (Workspaces) residing in memory
- 256 bytes of on-chip RAM
- Separate memory and interrupt bus structures
- 8-Bit memory data bus
- 7 prioritized hardware interrupts
- 16 software interrupts (XOPS)
- Programmed and DMA I/O capability
- Serial I/O via communication register unit (CRU)
- On-chip time/event counter
- On-chip programmable flags (16)
- Macro instruction detection (MID) feature
- Automatic first wait state generation feature
- Single 5-volt supply
- 40-pin package
- N-Channel silicon gate MOS technology
- On-chip clock generator

2. ARCHITECTURE

2.1 MEMORY ALLOCATION

The basic word of the TMS 9995 architecture is 16 bits in length. These 16 bits are divided into 8-bit bytes for external memory in the manner shown in Figure 1. A word is, therefore, defined as two consecutive 8-bit bytes in memory. All words (instruction opcodes, operand addresses, word-length data, etc.) are restricted to even address boundaries, i.e., the most significant half, or 8 bits, resides at an even address and the least significant half resides at the subsequent odd address. Any memory access involving a full word that is directed by software to utilize an odd address will result in the word starting with this odd address minus one to be accessed.

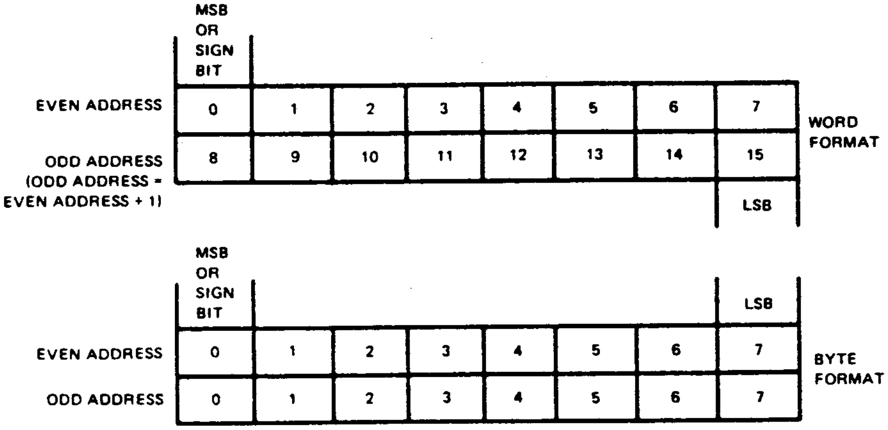
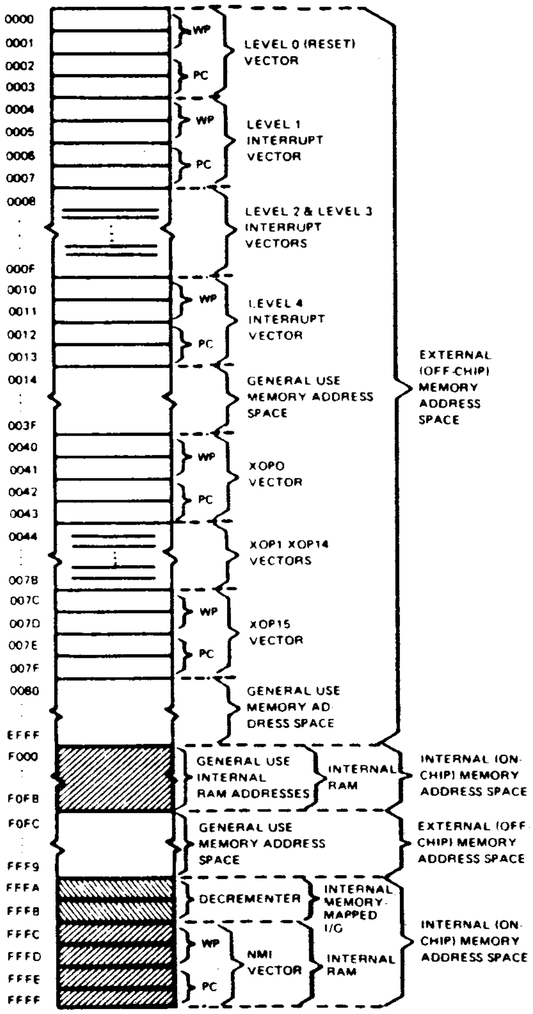


FIGURE 1 – WORD AND BYTE FORMATS

The instruction set of the TMS 9995 allows both word and byte operations. Byte instructions may address either byte as necessary. A byte access of this type will not affect the other byte of the word involved since the other byte will not be accessed during the execution of the byte instruction.

The TMS 9995 memory map is shown in Figure 2. Shown are the locations in the memory address space for the Reset, NMI, other interrupt and XOP trap vectors, and the dedicated address segments for the on-chip RAM and the on-chip memory-mapped I/O.


0000
0001
0002
0003
0004
0005
0006
0007
0008
000F
0010
0011
0012
0013
0014
003F
0040
0041
0042
0043
0044
007B
007C
007D
007E
007F
0080
EFFF
F000
F0FB
F0FC
FFF9
FFFA
FFFB
FFFC
FFFD
FFFE
FFFF
WP
PC
LEVEL 0 (RESET) VECTOR
LEVEL 1 INTERRUPT VECTOR
LEVEL 2 & LEVEL 3 INTERRUPT VECTORS
LEVEL 4 INTERRUPT VECTOR
GENERAL USE MEMORY ADDRESS SPACE
XOPO VECTOR
XOP1 XOP14 VECTORS
XOP15 VECTOR
GENERAL USE MEMORY AD DRESS SPACE
EXTERNAL (OFF-CHIP) MEMORY ADDRESS SPACE
GENERAL USE INTERNAL RAM ADDRESSES
INTERNAL RAM
INTERNAL (ON-CHIP) MEMORY ADDRESS SPACE
GENERAL USE MEMORY ADDRESS SPACE
EXTERNAL (OFF-CHIP) MEMORY ADDRESS SPACE
DECREMENTER
INTERNAL MEMORY-MAPPED I/O
NMI VECTOR
INTERNAL RAM
INTERNAL (ON-CHIP) MEMORY ADDRESS SPACE

NOTE: Addresses are byte addresses in hex

FIGURE 2 – TMS9995 MEMORY MAP

2.2 TMS 9995 ORGANIZATION

The block diagram of the TMS 9995 is shown in Figure 3. A flow chart, representative of the TMS 9995 functional operation, is shown in Figure 4.

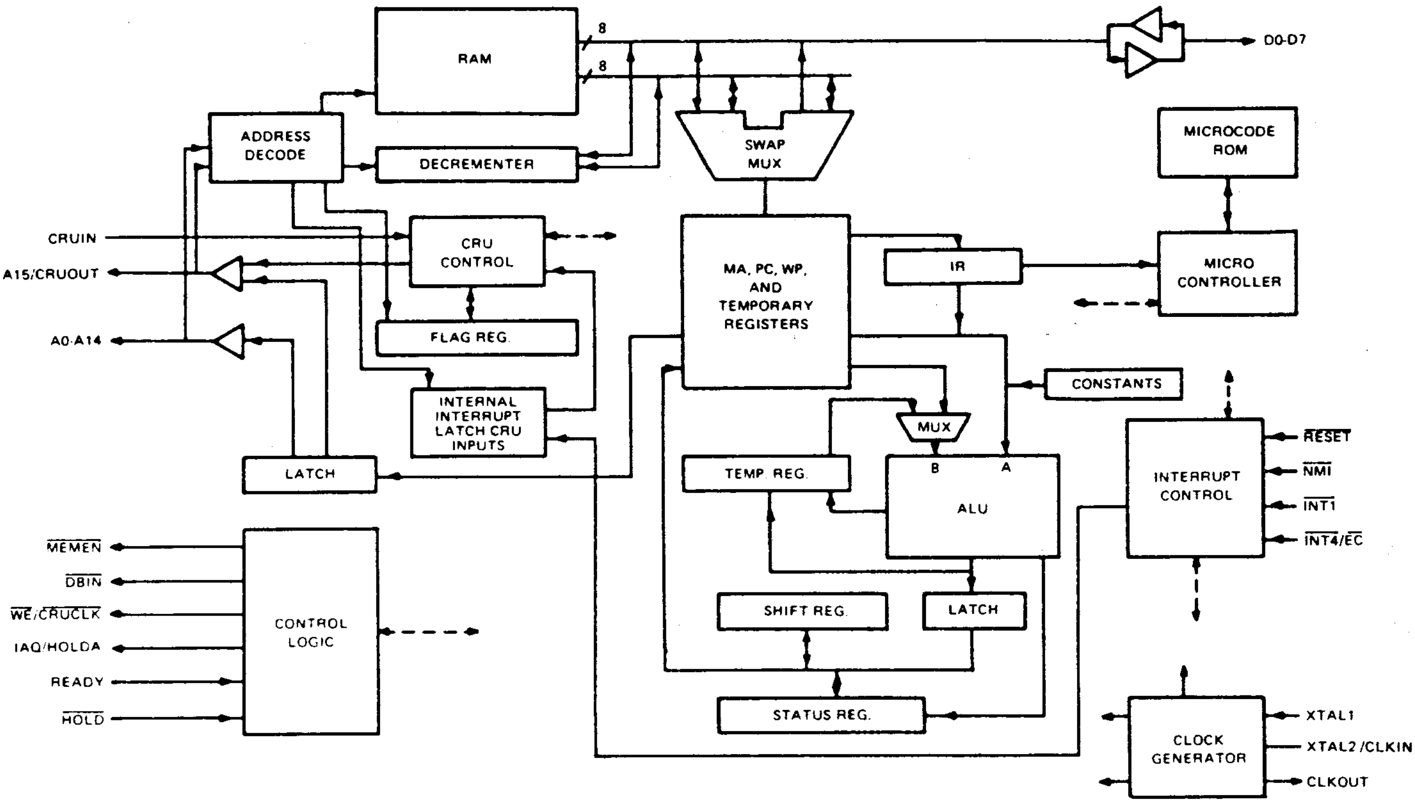


FIGURE 3 – TMS9995 BLOCK DIAGRAM

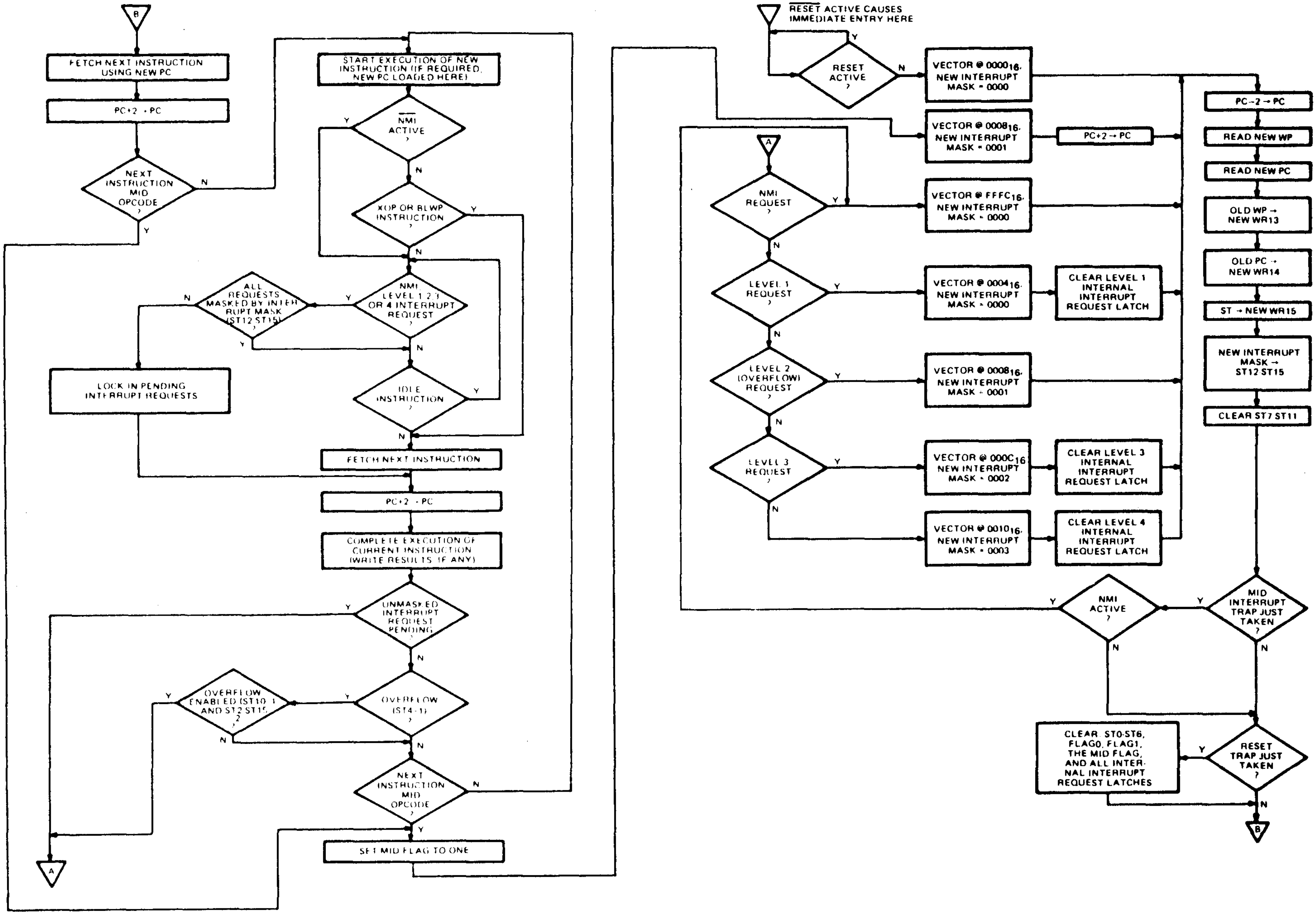

B
FETCH NEXT INSTRUCTION USING NEW PC
PC+2 → PC
NEXT INSTRUCTION MID OPCODE ?
N
Y
START EXECUTION OF NEW INSTRUCTION (IF REQUIRED, NEW PC LOADED HERE)
NMI ACTIVE ?
XOP OR BLWP INSTRUCTION ?
NMI LEVEL 1 2 3 OR 4 INTERRUPT REQUEST ?
ALL REQUESTS MASKED BY INTERRUPT MASK (ST12 ST15) ?
LOCK IN PENDING INTERRUPT REQUESTS
IDLE INSTRUCTION ?
FETCH NEXT INSTRUCTION
PC+2 → PC
COMPLETE EXECUTION OF CURRENT INSTRUCTION (WRITE RESULTS IF ANY)
UNMASKED INTERRUPT REQUEST PENDING ?
OVERFLOW (ST4=1) ?
OVERFLOW ENABLED (ST10=1 AND ST12-ST15 ≥ 2) ?
NEXT INSTRUCTION MID OPCODE ?
SET MID FLAG TO ONE
A
RESET ACTIVE CAUSES IMMEDIATE ENTRY HERE
RESET ACTIVE ?
VECTOR @ 0000₁₆, NEW INTERRUPT MASK = 0000
VECTOR @ 0008₁₆, NEW INTERRUPT MASK = 0001
PC+2 → PC
A
NMI REQUEST ?
VECTOR @ FFFC₁₆, NEW INTERRUPT MASK = 0000
LEVEL 1 REQUEST ?
VECTOR @ 0004₁₆, NEW INTERRUPT MASK = 0000
CLEAR LEVEL 1 INTERNAL INTERRUPT REQUEST LATCH
LEVEL 2 (OVERFLOW) REQUEST ?
VECTOR @ 0008₁₆, NEW INTERRUPT MASK = 0001
LEVEL 3 REQUEST ?
VECTOR @ 000C₁₆, NEW INTERRUPT MASK = 0002
CLEAR LEVEL 3 INTERNAL INTERRUPT REQUEST LATCH
VECTOR @ 0010₁₆, NEW INTERRUPT MASK = 0003
CLEAR LEVEL 4 INTERNAL INTERRUPT REQUEST LATCH
PC−2 → PC
READ NEW WP
READ NEW PC
OLD WP → NEW WR13
OLD PC → NEW WR14
ST → NEW WR15
NEW INTERRUPT MASK → ST12 ST15
CLEAR ST7 ST11
MID INTERRUPT TRAP JUST TAKEN ?
NMI ACTIVE ?
RESET TRAP JUST TAKEN ?
CLEAR ST0-ST6, FLAG0, FLAG1, THE MID FLAG, AND ALL INTERNAL INTERRUPT REQUEST LATCHES
B

FIGURE 4 – TMS9995 FLOW CHART

2.2.1 Arithmetic Logic Unit

The arithmetic logic unit (ALU) is the computational component of the TMS 9995. It performs all arithmetic and logic functions required to execute instructions. The functions include addition, subtraction, AND, OR, exclusive OR, and complement. A separate comparison circuit performs the logic and arithmetic comparisons to control bits 0 through 2 of the status register. The ALU is arranged in two 8-bit halves to accommodate byte operations. Each half of the ALU operates on one byte of the operand. During word operand operations, both halves of the ALU function in conjunction with each other. However, during byte operand processing, results from the least significant half of the ALU are ignored. The most-significant half of the ALU performs all operations on byte operands so that the status circuitry used in word operations is also used in byte operations.

2.2.2 Internal Registers

The following three (3) internal registers are accessible to the user (programmer):

- Program Counter (PC)
- Status Register (ST)
- Workspace Pointer (WP)

2.2.2.1 Program Counter

The program counter (PC) is a 15-bit counter that contains the word address of the next instruction following the instruction currently executing. The microprocessor references this address to fetch the next instruction from memory and increments the address in the PC when the new instruction is executing. If the current instruction in the microprocessor alters the contents of PC, then a program branch occurs to the location specified by the altered contents of PC. All context switching (see Section 2.2.2.3.2) operations plus simple branch and jump instructions affect the contents of PC.

2.2.2.2 Status Register

The status register (ST) is a fully implemented 16-bit register that reports the results of program comparisons, indicates program status conditions, and supplies the arithmetic overflow enable and interrupt mask level to the interrupt priority circuits. Each bit position in the register signifies a particular function or condition that exists in the microprocessor. Figure 5 illustrates the bit position assignments. Some instructions use the status register to check for a prerequisite condition; others affect the values of the bits in the register; and others load the entire status register with a new set of parameters. Interrupts also modify the status register. The description of the instruction set later in this document details the effect of each instruction on the status register (see Section 3).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ST0	ST1	ST2	ST3	ST4	ST5	ST6	ST7	ST8	ST9	ST10	ST11	ST12	ST13	ST14	ST15
L>	A>	EQ	C	OV	OP	X	*	*	*	OV EN	*	INTERRUPT MASK			

*NOTE: ST7, ST8, ST9, and ST11 are not used in the TMS9995, but still physically exist in the register. These bits could therefore be used as flag bits, but software transportability should be kept in mind when doing so as these bits are defined in other 9900 microprocessor family and 990 minicomputer family products.

L>	:	Logical Greater Than	C	:	Carry Out	X	:	XOP In Progress
A>	:	Arithmetic Greater Than	OV	:	Overflow	OV EN	:	Overflow Interrupt Enable
EQ	:	Equal/TB Indicator	OP	:	Parity (Odd No. of Bits)			

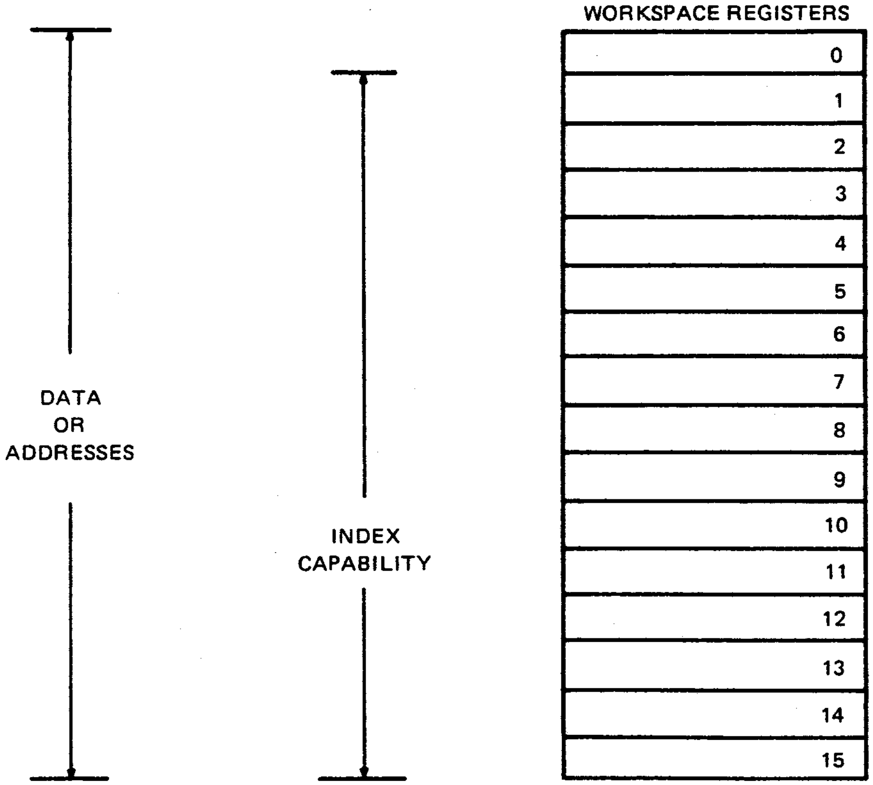
FIGURE 5 – STATUS REGISTER BIT ASSIGNMENTS

2.2.2.3 Workspace

The TMS 9995 uses blocks of memory words called workspaces for instruction operand manipulation. A workspace occupies 16 contiguous words in any part of memory that is not reserved for other use. The individual workspace registers may contain data or address, or function as operand registers, accumulators, address registers, or index registers. Some workspace registers take on special significance during execution of certain instructions. Table 1 lists each of these dedicated workspace registers and the instructions that use them. Figure 6 defines the workspace registers that are allowed to be used as index registers.

TABLE 1 – DEDICATED WORKSPACE REGISTERS

REGISTER NO.	CONTENTS	USED DURING
0	Shift count (optional) Multiplicand and MSW of result MSW of dividend and quotient	Shift instructions (SLA, SRA, SRC, and SLC) Signed Multiply Signed Divide
1	LSW of result LSW of dividend and remainder	Signed Multiply Signed Divide
11	Return Address Effective Address	Branch and Link Instruction (BL) Extended Operation (XOP)
12	CRU Base Address	CRU instructions (SBO, SBZ, TB, LDCR, and STCR)
13	Saved WP register	Context switching (BLWP, RTWP, XOP, interrupts)
14	Saved PC register	Context switching (BLWP, RTWP, XOP, interrupts)
15	Saved ST register	Context switching (BLWP, RTWP, XOP, interrupts)

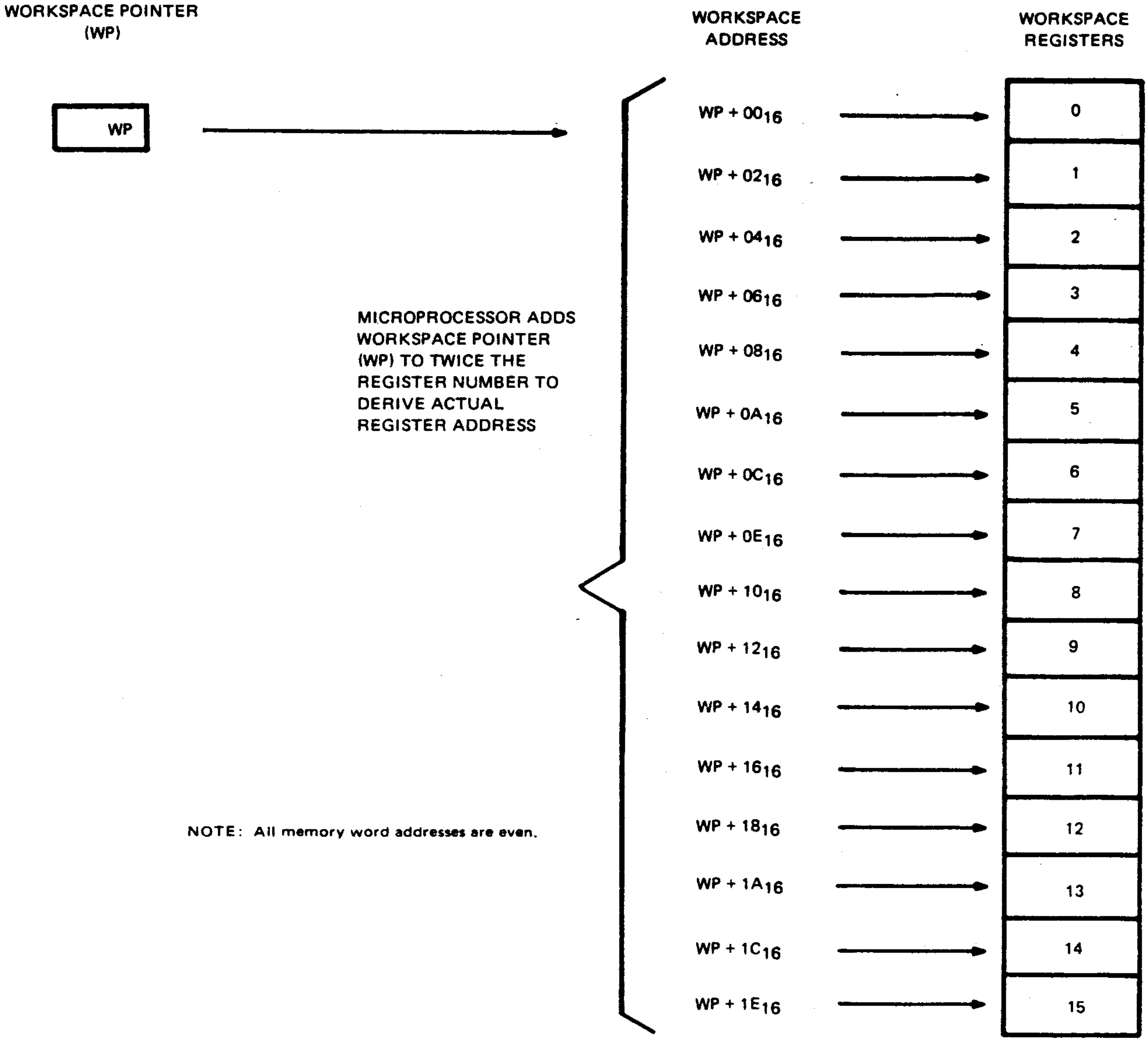


NOTE: The WP register contains the address of workspace register zero.

FIGURE 6 – WORKSPACE REGISTERS USABLE AS INDEX REGISTERS

2.2.2.3.1 Workspace Pointer

To locate the workspace in memory, a hardware register called the workspace pointer (WP) is used. The workspace pointer is a 16-bit register that contains the memory address of the first word in the workspace. The address is left-justified with the 16th bit (LSB) hardwired to logic zero. The TMS 9995 accesses each register in the workspace by adding twice the register number to the contents of the workspace pointer and initiating a memory request for that word. Figure 7 illustrates the relationship between the workspace pointer and its corresponding workspace in memory



NOTE: All memory word addresses are even.

FIGURE 7 – WORKSPACE POINTER AND REGISTERS

For instructions performing byte operations, use of the workspace register addressing mode (see Section 3.2) will result in the most significant byte of the workspace register involved to be used as the operand for the operation. Since the workspace is also addressable as a memory address, the least significant byte may be directly addressed using any one of the general memory addressing modes.

2.2.2.3.2 Context Switching

The workspace concept is particularly valuable during operations that require a context switch, which is a change from one program environment to another, as in the case of a subroutine or an interrupt service routine. Such an operation using a conventional multi-register arrangement requires that at least part of the contents of the register file be stored and reloaded using a memory cycle to store or fetch each word. The TMS 9995 accomplishes this operation by changing the workspace pointer. A context switch requires only three store cycles and two fetch cycles, exchanging the program counter, status register and workspace pointer. After the switch, the workspace pointer contains the starting address of a new 16-word workspace in memory for use in the new routine. A corresponding time saving occurs when the original context is restored. Instructions in the TMS 9995 that result in a context switch include: Call subroutine (BLWP), Return from Subroutine (RTWP) and the Extended Operation (XOP) instruction. All interrupts also cause a context switch by forcing the TMS 9995 to trap to a service subroutine.

2.3 TMS 9995 INTERFACES

Each TMS 9995 system interface uses one or more of the signals from one or more of the signal groupings given in the pin description list in Section 3. Each interface is described in detail in the following paragraphs.

2.3.1 TMS 9995 Memory Interface

The signals used in the TMS 9995 interface to system memory are shown in Figure 8.

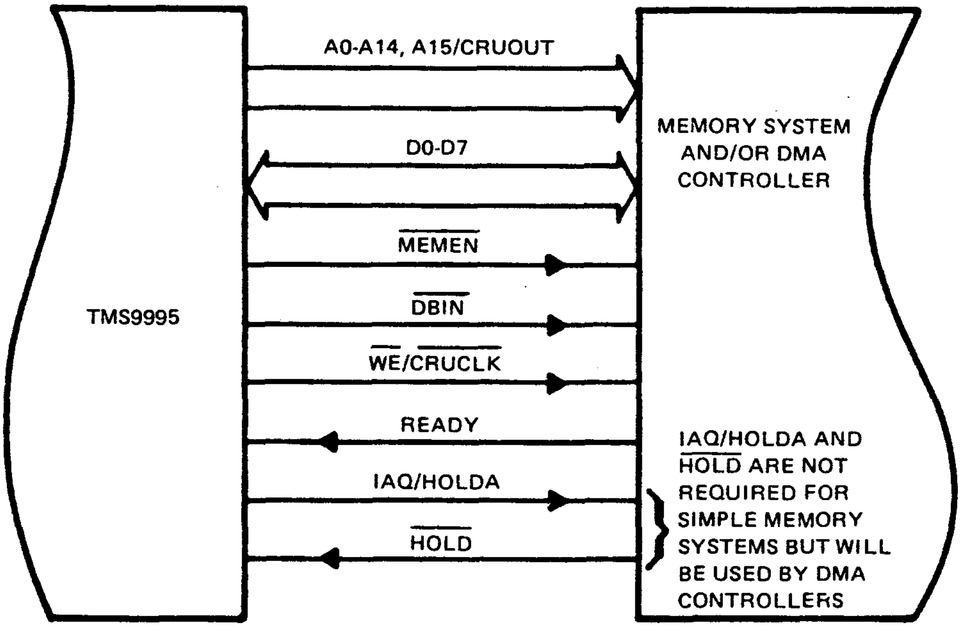

TMS9995
A0-A14, A15/CRUOUT
D0-D7
$\overline{\text{MEMEN}}$
$\overline{\text{DBIN}}$
$\overline{\text{WE}}/\overline{\text{CRUCLK}}$
READY
IAQ/HOLDA
$\overline{\text{HOLD}}$
MEMORY SYSTEM AND/OR DMA CONTROLLER
IAQ/HOLDA AND $\overline{\text{HOLD}}$ ARE NOT REQUIRED FOR SIMPLE MEMORY SYSTEMS BUT WILL BE USED BY DMA CONTROLLERS

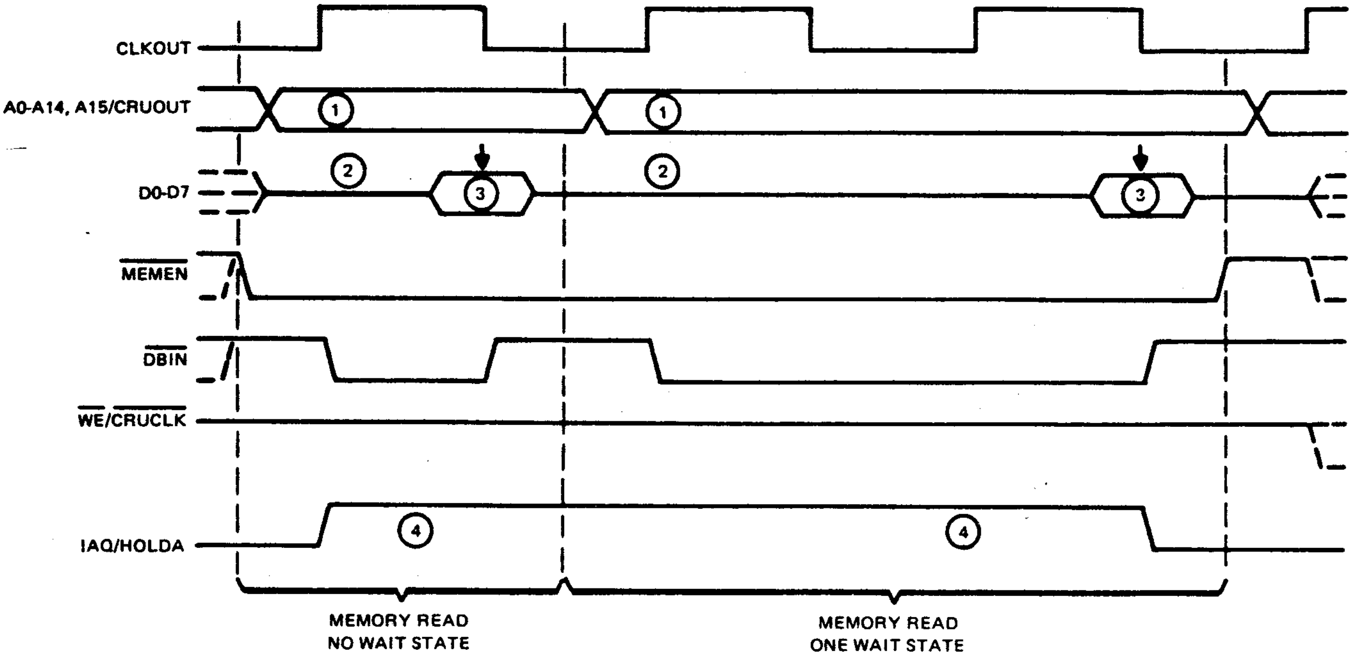
FIGURE 8 – TMS9995 MEMORY INTERFACE

2.3.1.1 *External Memory Address Space*

The details of memory accesses that are external to the TMS 9995 (off-chip accesses) are given in the following paragraphs. (See Figure 2 for the addresses that are in the external memory-address space.)

2.3.1.1.1 Memory Read Operations

To perform a memory read operation, the TMS 9995 first outputs the appropriate address on A0-A14 and A15/CRUOUT, and asserts $\overline{\text{MEMEN}}$. The TMS 9995 then places its data bus drivers in the high impedance state, asserts $\overline{\text{DBIN}}$, and then reads in the data byte. Completion of the memory read cycle and/or generation of Wait states is determined by the READY input as detailed in Section 2.3.1.3. Timing relationships of the memory read sequence are shown in Figure 9. Note that $\overline{\text{MEMEN}}$ remains active (low) between consecutive memory operations.



NOTES:

1. Valid address
2. In input mode (drivers @ High-Z)
3. Memory Read Data must be valid at CLKOUT edge indicated
4. IAQ/HOLDA will only be asserted during memory read cycles if an instruction opcode is being read (timing shown is for an instruction fetch from external memory —, i.e., two consecutive byte reads).

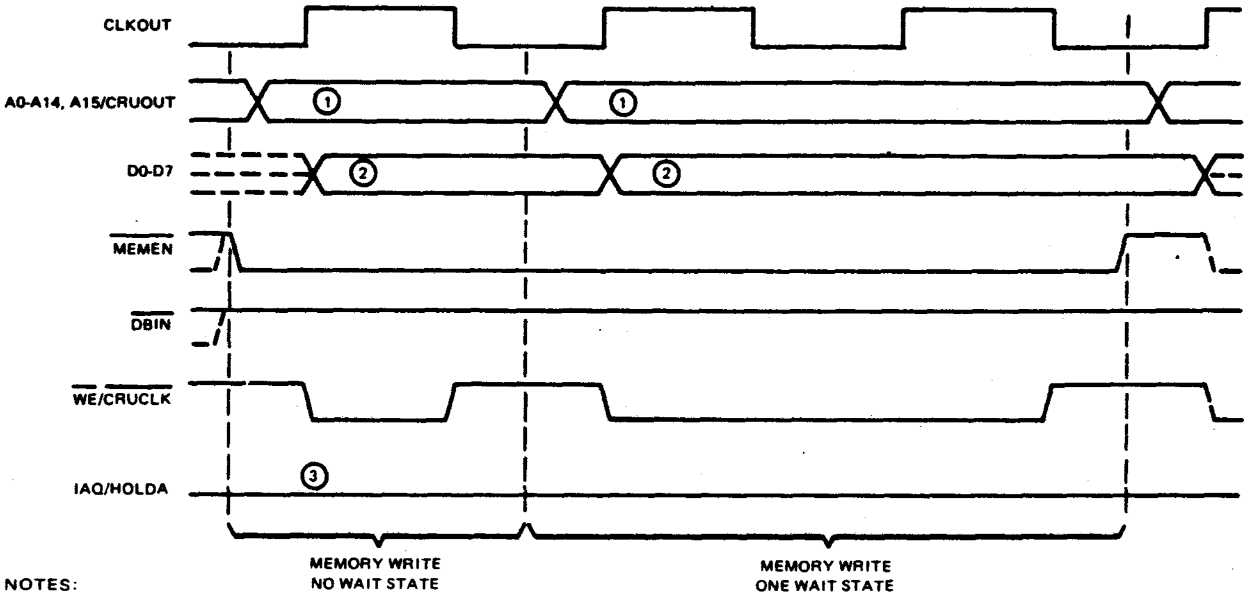
FIGURE 9 – TMS9995 MEMORY READ CYCLE

Although not explicitly shown in Figure 9, reading a word (two 8-bit bytes) frcm external memory requires two memory read cycles that occur back-to-back (a Hold state request will not be granted between cycles). If an instruction directs that a byte read from external memory is to be performed, only the byte specifically addressed will be read (one memory read cycle). External words are accessed most-significant (even) byte first, followed by the least-significant (odd) byte.

During memory read cycles in which an instruction opcode is being read, IAQ/HOLDA is asserted as shown in Figure 9. Note that since an instruction opcode is a word in length, IAQ/HOLDA remains asserted between the two byte read operations involved when an instruction opcode is read from the external memory address space.

2.3.1.1.2 Memory Write Operations

To perform a memory write operation, the TMS 9995 first outputs the appropriate address on A0-A14 and A15/CRUOUT, and asserts $\overline{\text{MEMEN}}$. The TMS 9995 then outputs the data byte being written to memory on pins D0 through D7, and then asserts $\overline{\text{WE}}/\overline{\text{CRUCLK}}$. Completion of the memory write cycle and/or generation of Wait states is determined by the Ready input as detailed in Section 2.3.1.3. Timing relationships of the memory write sequence are shown in Figure 10. Note that $\overline{\text{MEMEN}}$ remains active (low) between consecutive memory operations.



NOTES:

1 Valid address

2 Valid memory write data

3 IAQ/HOLDA will never be asserted during a memory write cycle

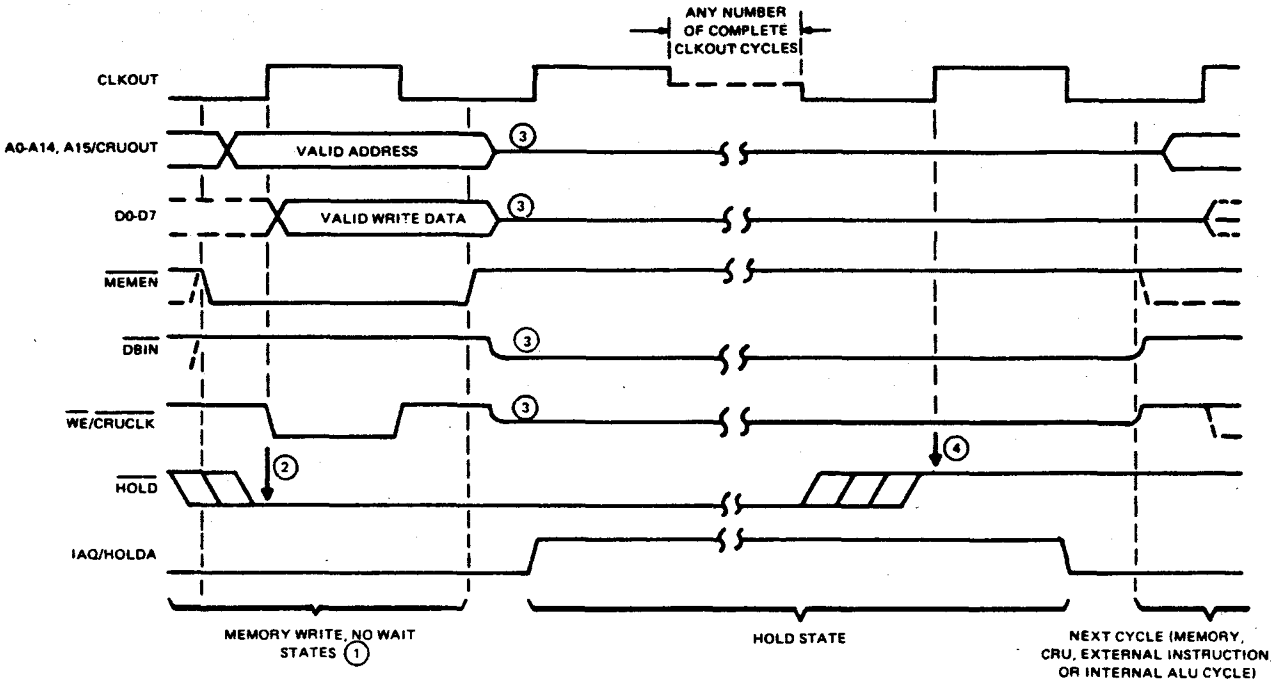
FIGURE 10 – TMS9995 MEMORY WRITE CYCLE

Writing a word (two 8-bit bytes) to external memory requires two memory write cycles that occur back-to-back. (A Hold stăte request will not be granted between cycles.) If an instruction directs that a byte write to external memory is to be performed, only the byte specifically addressed will be written (one memory write cycle). External words are accessed most-significant (even) byte first followed by the least-significant (odd) byte.

2.3.1.1.3 Direct Memory Access

The TMS 9995 Hold state allows both external devices and the TMS 9995 to share a common external memory. To gain direct memory access (DMA) to the common memory, the external device first requests the TMS 9995 to enter a Hold state by asserting (taking low) the $\overline{\text{HOLD}}$ input. The TMS 9995 will then enter a Hold state following completion of the cycle (either memory, CRU, external Instruction, or internal ALU cycles) that it is currently performing. Note, however, that a Hold state is not entered between the first and second byte accesses of a full word in the external memory address space, and a Hold state is not entered between the first and second clock cycles of a CRU cycle.

Upon entry of a Hold state, the TMS 9995 puts its address, data, $\overline{\text{DBIN}}$, and $\overline{\text{WE}}/\overline{\text{CRUCLK}}$ drivers in the high impedance mode, and asserts IAQ/HOLDA. The external device can then utilize these signal lines to communicate with the common memory. After the external device has completed its memory transactions, it releases HOLD, and the TMS 9995 continues instruction execution at the point where it had been suspended. Timing relationships for this sequence are shown in Figure 11.


ANY NUMBER OF COMPLETE CLKOUT CYCLES
CLKOUT
A0-A14, A15/CRUOUT
VALID ADDRESS
D0-D7
VALID WRITE DATA
MEMEN
DBIN
WE/CRUCLK
HOLD
IAQ/HOLDA
MEMORY WRITE, NO WAIT STATES ①
HOLD STATE
NEXT CYCLE (MEMORY, CRU, EXTERNAL INSTRUCTION OR INTERNAL ALU CYCLE)

NOTES:

① Cycle before the hold state could have been memory (with any number of wait states), CRU, external instruction, or internal ALU

② $\overline{\text{HOLD}}$ must be valid at last low-to-high CLKOUT transition of a cycle for next low-to-high CLKOUT transition to begin a hold state

③ In high-impedance mode (output drivers)

④ Next cycle will begin after first low-to-high CLKOUT transition at which $\overline{\text{HOLD}}$ is high

FIGURE 11 – TMS9995 HOLD STATE

To allow DMA loading of external memory on power-up, the TMS 9995 does not begin instruction execution after a Reset state until $\overline{\text{HOLD}}$ has been removed if $\overline{\text{HOLD}}$ was active (low) at the time $\overline{\text{RESET}}$ was taken from low to high $\overline{\text{RESET}}$ released).

External devices cannot access the internal (on-chip) memory address space of the TMS 9995 when it is in the Hold state.

Since IAQ (Instruction Opcode Acquisition) and HOLDA (Hold Acknowledge) are multiplexed on a single signal, IAQ/HOLDA, this signal must be gated with $\overline{\text{MEMEN}}$ using external logic to separate IAQ and HOLDA. When $\overline{\text{MEMEN}}$ = 0, IAQ/HOLDA can indicate IAQ, and when $\overline{\text{MEMEN}}$ = 1, IAQ/HOLDA can indicate HOLDA.

2.3.1.2 *Internal Memory Address Space*

Access of the internal (on-chip) memory address space is transparent to the TMS 9995 instruction set. That is, operands can be read from and written into locations in the internal memory space simply by using the appropriate addresses via any of the addressing modes in the TMS 9995 instruction set, and instructions can even be executed from the internal memory space by loading the appropriate address into the program counter of the TMS 9995.

The TMS 9995 indicates to the external world when these internal memory address space accesses are occurring by asserting the same signals used for accessing external memory (see Figure 8) in a manner very similar to an external memory address space access. There are a few differences in these cycles, however, and these differences are detailed in the following paragraphs.

When performing an internal memory address space access, the TMS 9995 outputs the same signals that it would for an external memory space access, with the same timing (see Figures 9 and 10) except for the following:

(1) A single cycle (read or write) is output as both internal bytes are accessed simultaneously. (Externally, it appears as though a single byte memory access cycle to an internal address is occurring.)

(2) The cycle always has no Wait states, and the READY input is ignored by the TMS 9995 (see Section 2.3.2.3).

(3) During read cycles, the data bus (D0-D7) output drivers are put in the high-impedance mode. During write cycles, the data bus outputs non-specific data.

During read cycles to the internal memory address space, the TMS 9995 does not make the read data available to the external world. If an instruction is executed from the internal memory address space, IAQ/HOLDA is still asserted, but only during the one read cycle shown externally while the full word is read internally.

When in a Hold state, external devices are not able to access the internal memory address space.

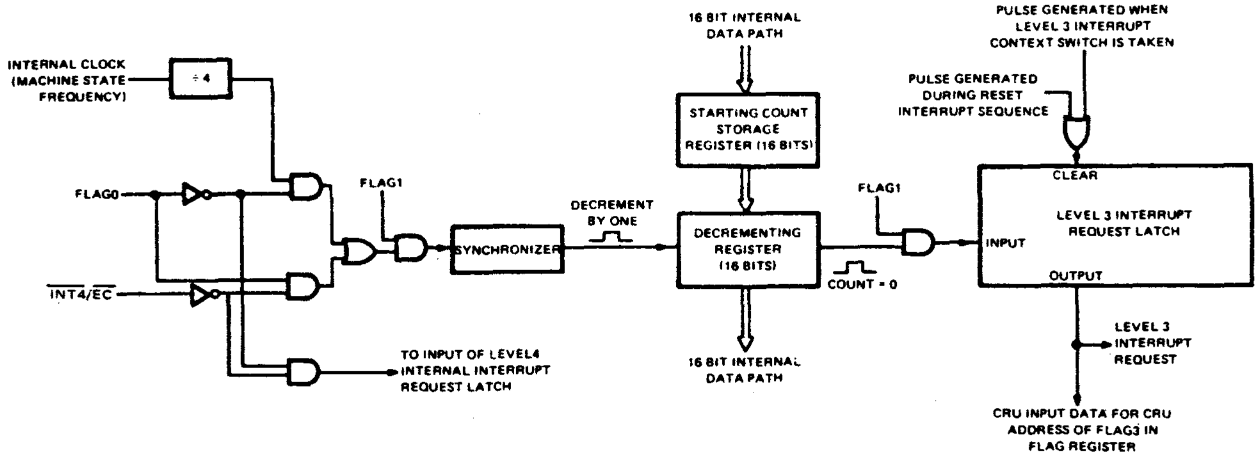
2.3.1.2.1 Internal RAM

The 256 bytes of internal random-access read/write memory (RAM), the memory addresses of which are shown in Figure 2, are organized internally as 128 16-bit words. Since the TMS 9995 has 16-bit internal data paths, two 8-bit bytes are accessed each time a memory access is made to the internal RAM.

Byte accesses are transparent to the internal RAM. That is, when an instruction addresses a byte in the internal RAM, the TMS 9995 will: (1) read the entire word but only use the byte specifically addressed for a read operation and, (2) only write to the specifically addressed byte and not alter the contents of the other byte in the word during a write operation.

2.3.1.2.2 Decrementer (Timer/Event Counter)

Accessible via one of the word addresses (see Figure 2) of the internal memory-mapped I/O address space is the decrementer. The on-chip decrementer logic can function as a programmable real-time clock, an event timer, or as an external event counter. A block diagram of the decrementer that is representative of its functional operation (but not necessarily representative of its specific logic implementation) is shown in Figure 12.


INTERNAL CLOCK (MACHINE STATE FREQUENCY)
÷4
FLAG0
$\overline{\text{INT4}}/\overline{\text{EC}}$
FLAG1
SYNCHRONIZER
DECREMENT BY ONE
TO INPUT OF LEVEL4 INTERNAL INTERRUPT REQUEST LATCH
16 BIT INTERNAL DATA PATH
STARTING COUNT STORAGE REGISTER (16 BITS)
DECREMENTING REGISTER (16 BITS)
16 BIT INTERNAL DATA PATH
COUNT = 0
FLAG1
PULSE GENERATED WHEN LEVEL 3 INTERRUPT CONTEXT SWITCH IS TAKEN
PULSE GENERATED DURING RESET INTERRUPT SEQUENCE
CLEAR
LEVEL 3 INTERRUPT REQUEST LATCH
INPUT
OUTPUT
LEVEL 3 INTERRUPT REQUEST
CRU INPUT DATA FOR CRU ADDRESS OF FLAG3 IN FLAG REGISTER

NOTE: FLAG0 and FLAG1 are bits in the Flag Register

FIGURE 12 – DECREMENTER FUNCTIONAL BLOCK DIAGRAM

The decrementer is configured as either a timer or an event counter using bit FLAG0 of the internal Flag register. The decrementer is enabled/disabled using bit FLAG1 of the internal Flag register. (See Section 2.3.3.2.1 for details of the Flag register and accessing the bits in it.) When FLAG0 is set to zero, the decrementer will function as a timer. When FLAG0 is set to one, the decrementer will function as an event counter. When FLAG1 is set to zero, the decrementer is disabled and will not be allowed to decrement and request level 3 interrupt traps. When FLAG1 is set to one, the decrementer is enabled and will decrement and request level 3 interrupt traps. It should be noted that when the decrementer is configured as a timer, $\overline{\text{INT4}}/\overline{\text{EC}}$ will be usable as an external interrupt level 4 trap request. When the decrementer is configured as an event counter, $\overline{\text{INT4}}/\overline{\text{EC}}$ is the input for the "event counter" pulses, and an interrupt level 4 trap request input is no longer available externally or internally.

The general operation of the decrementer is as follows. FLAG0 of the Flag register is first set to select the desired mode of operation. The desired start count is then loaded into the Starting Count Storage Register by performing a memory write of the count word to the dedicated internal memory mapped I/O address of the decrementer. (This also loads the Decrementing Register with the same count.) The decrementer is then enabled and allowed to start decrementing by setting FLAG1 of the Flag Register to one. (Both FLAG0 and FLAG1 are set to zero when the TMS 9995 is reset. (See Section 2.3.2.1.1.) When the count in the Decrementing Register reaches zero, the level 3 internal interrupt request latch is set (see Section 2.3.2.2.3), the Decrementing Register is reloaded from the Starting Count Storage Register, and decrementing continues. Note that writing a start count of 0000_{16} to the decrementer will disable it.

When configured as a timer, the decrementer functions as a programmable real-time clock by decreasing the count in the Decrementing Register by one for each fourth CLKOUT cycle. Loading the decrementer with the appropriate start count causes an interrupt to be requested every time the count in the Decrementing Register reaches zero. The decrementer can also be used as an event timer when configured as a timer by reading the decrementer (which is accomplished by performing a memory read from the dedicated internal memory mapped I/O address of the decrementer) at the start and stop points of the event of interest and comparing the two values. The difference will be a measurement of the elapsed time.

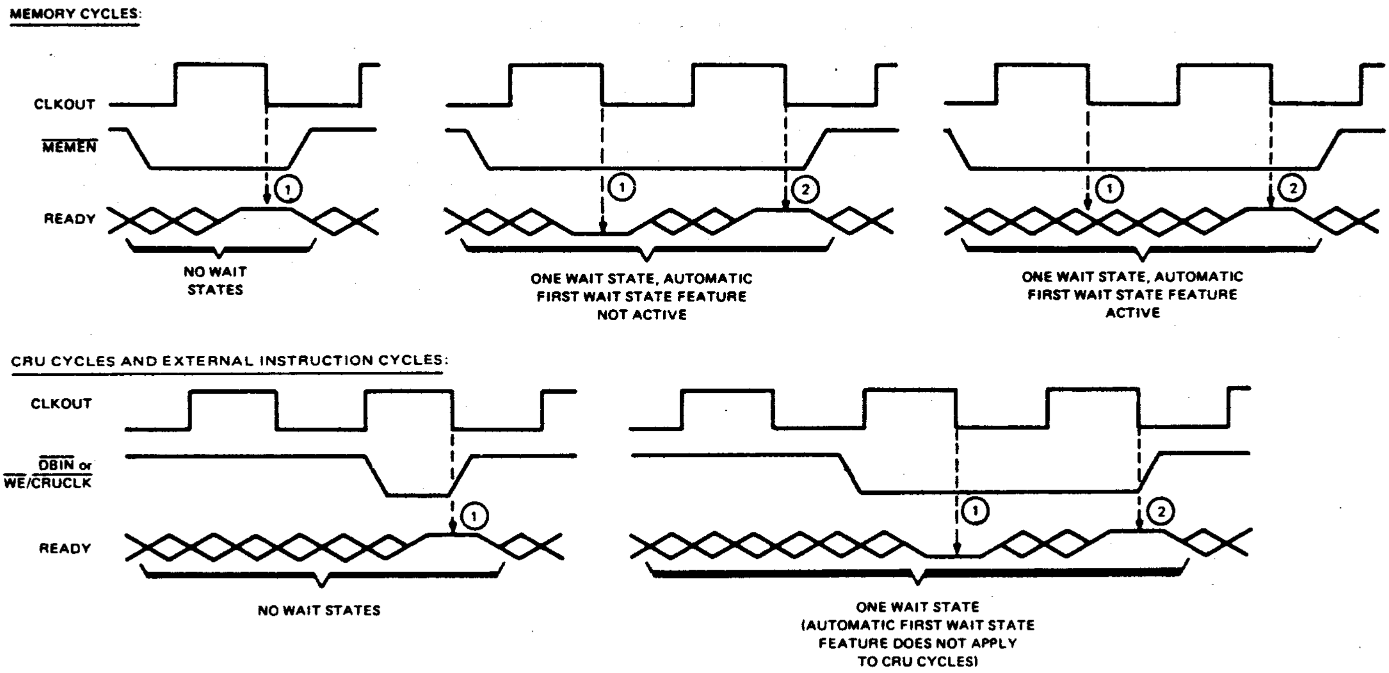
When configured as an event counter, operation is as previously discussed except that each high-to-low transition on $\overline{\text{INT4}}/\overline{\text{EC}}$ will cause the Decrementing Register to decrement. These $\overline{\text{INT4}}/\overline{\text{EC}}$ high-to-low transitions can be asynchronous with respect to CLKOUT. Note that INT4/EC can function as a negative edge-triggered interrupt by loading a start count of one.

The decrementer should always be accessed as a full word (two 8-bit bytes). Reading a byte from the decrementer does not present a problem since only the byte specifically addressed will be read. Writing a single byte to either of the bytes of the decrementer will result in the data byte being written into the byte specifically addressed and random bits being written into the other byte of the decrementer.

2.3.1.3 *Wait State Generation*

Wait states can be generated for external memory cycles, external CRU cycles and external instruction cycles for the TMS 9995 using the READY input. A Wait state is defined as extension of the present cycle by one CLKOUT cycle. The timing relationships of the READY input to the memory interface and the CRU interface signals are shown in Figure 13. Note that Wait states cannot be generated for memory cycles that access the internal memory address space or for CRU cycles that access the internal CRU address space, as the READY input will be ignored during these cycles.

The Automatic First Wait State Generation feature of the TMS 9995 allows a Wait state to be inserted in each external memory cycle, regardless of the READY input, as shown in Figure 13. The Automatic First Wait State Generation feature can be invoked when $\overline{\text{RESET}}$ is asserted. If READY is active (high) when $\overline{\text{RESET}}$ goes through a low-to-high transition, the first Wait state in each external memory cycle will be automatically generated. If READY is inactive (low) when $\overline{\text{RESET}}$ goes through a low-to-high transition, no Wait state will be inserted automatically in each external memory cycle. There is a one and one-half CLKOUT cycle time minimum setup time requirement on READY before the $\overline{\text{RESET}}$ low-to-high transition. The recommended external circuitry for invoking or inhibiting the Automatic First Wait State Generation feature is shown in Figure 14. Note that this feature does not apply to internal memory address space accesses, external instruction cycles, or any CRU cycles. Wait states cannot be generated during internal ALU/other operation cycles. The READY input is ignored during these cycles.


MEMORY CYCLES:
CLKOUT
MEMEN
READY
1
2
NO WAIT STATES
ONE WAIT STATE, AUTOMATIC FIRST WAIT STATE FEATURE NOT ACTIVE
ONE WAIT STATE, AUTOMATIC FIRST WAIT STATE FEATURE ACTIVE
CRU CYCLES AND EXTERNAL INSTRUCTION CYCLES:
CLKOUT
DBIN or WE/CRUCLK
READY
NO WAIT STATES
ONE WAIT STATE (AUTOMATIC FIRST WAIT STATE FEATURE DOES NOT APPLY TO CRU CYCLES)

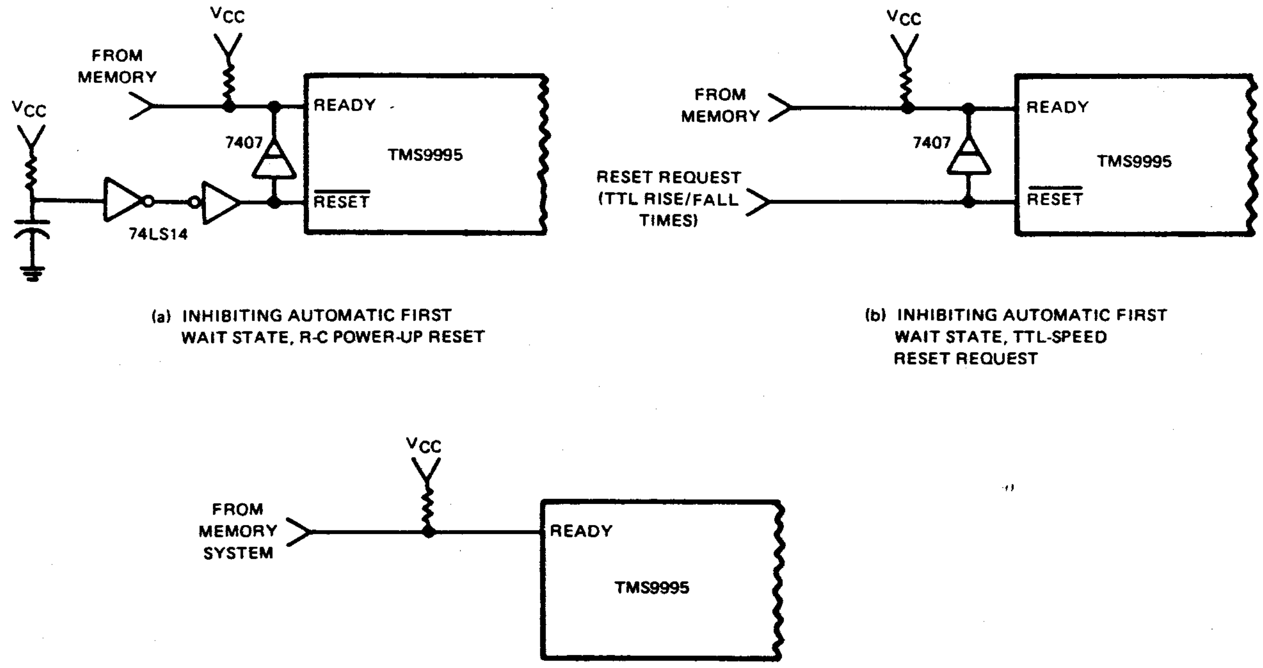
NOTES:

(1) First sample time of READY in cycle

(2) Second sample time of READY in cycle. Additional wait states can be generated by keeping READY low at this and subsequent sample times.

XXXX denotes "don't care"

FIGURE 13 – WAIT STATE GENERATION FOR EXTERNAL MEMORY, EXTERNAL CRU CYCLES, AND EXTERNAL INSTRUCTION CYCLES



(a) INHIBITING AUTOMATIC FIRST WAIT STATE, R-C POWER-UP RESET

(b) INHIBITING AUTOMATIC FIRST WAIT STATE, TTL-SPEED RESET REQUEST

(c) INVOKING AUTOMATIC FIRST WAIT STATE (RESET CAN BE R-C POWER-UP OR OTHERWISE)

FIGURE 14 – EXTERNAL CIRCUITRY FOR INVOKING/INHIBITING AUTOMATIC FIRST WAIT STATE GENERATION FEATURE

2.3.2 TMS 9995 Interrupts

The TMS 9995 implements seven prioritized, vectored interrupts, some of which are dedicated to predefined functions and the remaining are user-definable. Table 2 defines the source (internal or external), assignment, priority level, trap vector location in memory, and enabling/resulting status register interrupt mask values for each interrupt.

TABLE 2 – INTERRUPT LEVEL DATA

PRIORITY LEVELS (In Order of Priority)	VECTOR LOCATION (Memory Address, In Hex)	MASK VALUES TO ENABLE ACCEPTING THE INTERRUPT (ST12 THRU ST15)	MASK VALUE AFTER TAKING THE INTERRUPT (ST12 THRU ST15)	SOURCE AND ASSIGNMENT
0 (Highest Priority)	0000	0_{16} thru F_{16} (see Note 1)	0000	External: Reset ($\overline{\text{RESET}}$ Signal)
MID	0008 (see Note 2)	0_{16} thru F_{16} (see Note 1)	0001 (see Note 2)	Internal: MID
NMI	FFFC	0_{16} thru F_{16} (see Note 1)	0000	External: User-defined ($\overline{\text{NMI}}$ Signal)
1	0004	1_{16} thru F_{16}	0000	External: User-defined ($\overline{\text{INT1}}$ Signal)
2	0008 (see Note 2)	2_{16} thru F_{16} (see Note 3)	0001 (see Note 2)	Internal: Arithmetic Overflow
3	000C	3_{16} thru F_{16}	0002	Internal: Decrementer
4	0010	4_{16} thru F_{16}	0003	External: User-defined ($\overline{\text{INT4}}/\overline{\text{EC}}$ Signal; see Note 4).

NOTES:
1. Level 0, MID, and NMI cannot be disabled with the Interrupt Mask.
2. MID and Level 2 use the same trap vector and change the Interrupt Mask to the same value.
3. Generation of a Level 2 request by an Arithmetic Overflow condition (ST4 set to 1) is also enabled/disabled by bit ST10 of the Status Register.
4. $\overline{\text{INT4}}/\overline{\text{EC}}$ is not an input for Level 4 interrupt trap requests (Level 4 is not usable) when the Decrementer is configured as an Event Counter.

The TMS 9995 will grant interrupt requests only between instructions (except for Level 0 Reset), which will be granted whenever it is requested, i.e., in the middle of an instruction). The TMS 9995 performs additional functions for certain interrupts, and these functions will be detailed in subsequent sections. The basic sequence that the TMS 9995 performs to service all interrupt requests is as follows:

(1) Prioritize all pending requests and grant the request for the highest priority interrupt that is not masked by the current value of the interrupt mask in the status register or the instruction that has just been executed. (See Section 4.5 for these instructions.)

(2) Make a context switch using the trap vector specified for the interrupt being granted.

(3) Reset ST7 through ST11 in the status register to zero, and change the interrupt mask (ST12 through ST15) as appropriate for the level of the interrupt being granted.

(4) Resume execution with the instruction located at the new address contained in the PC, and using the new WP. All interrupts will be disabled until after this first instruction is executed, unless: (a) $\overline{\text{RESET}}$ is requested, in which case it will be granted, or (b) the interrupt being granted is the MID request and the $\overline{\text{NMI}}$ interrupt is requested simultaneously (in which case the NMI request will be granted before the first instruction indicated by the MID trap vector is executed.)

This sequence has several important characteristics. First of all, for those interrupts that are maskable with the interrupt mask in the status register, the mask will get changed to a value that will permit only interrupts of higher priority to interrupt their service routines. Secondly, status bit ST10 (overflow interrupt enable) is reset to zero by the servicing of any interrupt so that overflow interrupt requests cannot be generated by an unrelated program segment. Thirdly, the disabling of other interrupts until after the first instruction of the service routine is executed permits the routine to disable other interrupts by changing the interrupt mask with the first instruction. (The exception with MID and NMI is explained in Section 2.3.2.2.1.) Lastly, the vectoring and prioritizing scheme of the TMS 9995 permits interrupts to be automatically nested in most cases. If a higher priority interrupt occurs while in an interrupt service routine, a second context switch occurs to service the higher priority interrupt. When that routine is complete, a return instruction (RTWP) restores the saved context to complete processing of the lower priority interrupt. Interrupt routines should, therefore, terminate with the return instruction to restore original program parameters.

Additional details of the TMS 9995 interrupts are supplied in the following paragraphs.

2.3.2.1 *External Interrupt Requests*

Each of these interrupts is requested when the designated signal is supplied to the TMS 9995.

2.3.2.1.1 Interrupt Level 0 ($\overline{\text{RESET}}$)

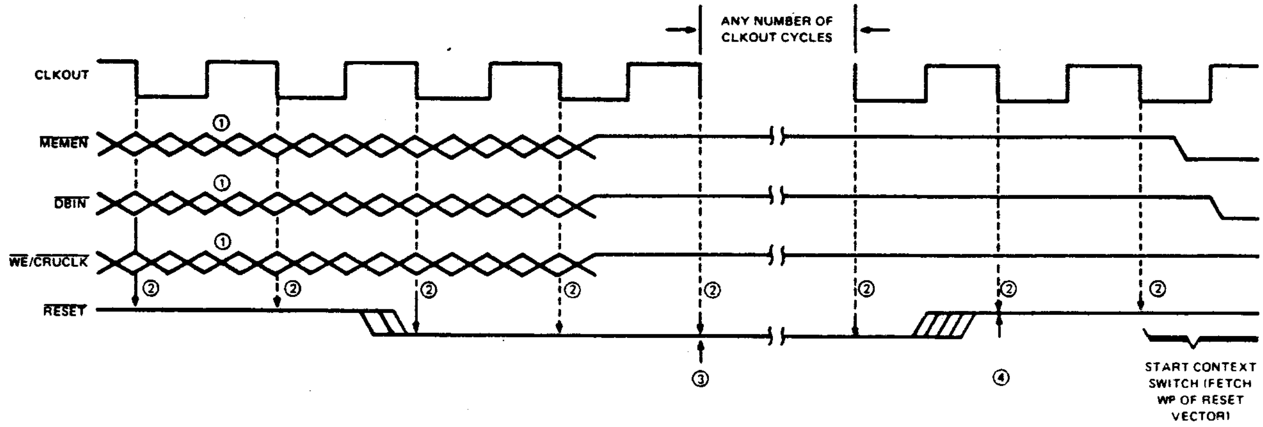
Interrupt Level 0 is dedicated to the $\overline{\text{RESET}}$ input of the TMS 9995. When active (low), $\overline{\text{RESET}}$ causes the TMS 9995 to stop instruction execution and to inhibit (take to logic level high) $\overline{\text{MEMEN}}$, $\overline{\text{DBIN}}$, and $\overline{\text{WE}}$/CRUCLK. The TMS 9995 will remain in this Reset state as long as $\overline{\text{RESET}}$ is active.

When $\overline{\text{RESET}}$ is released (low-to-high transition), the TMS 9995 performs a context switch with the Level 0 interrupt trap vector (WP and PC of trap vector are in memory word addresses 0000_{16} and 0002_{16}, respectively.) Note that the old WP, PC and ST are stored in registers 13, 14, and 15 of the new workspace. The TMS 9995 then resets all status register bits, the internal interrupt request latches (see Sections 2.3.2.1.3 and 2.3.2.2.3 for details of these latches), Flag Register bits FLAG0 and FLAG1 (see Section 2.3.3.2.1 for details of the Flag Register), and the MID Flag (see Section 2.3.3.2.2). After this, the TMS 9995 starts execution with the new PC.

If HOLDA is active (high) due to $\overline{\text{HOLD}}$ being active (low) when $\overline{\text{RESET}}$ becomes active, $\overline{\text{RESET}}$ will cause HOLDA to be released (taken low) at the same time as $\overline{\text{MEMEN}}$, $\overline{\text{DBIN}}$, and $\overline{\text{WE}}$/CRUCLK are taken inactive (high). $\overline{\text{HOLD}}$ can remain active as long as $\overline{\text{RESET}}$ is active and HOLDA will not be asserted. If $\overline{\text{HOLD}}$ is active when $\overline{\text{RESET}}$ is released (low-to-high transition), HOLDA will be asserted before the $\overline{\text{RESET}}$ context switch occurs and the TMS 9995 will remain in this hold state until $\overline{\text{HOLD}}$ is released. This RESET and $\overline{\text{HOLD}}$ priority scheme facilitates DMA loading of external RAM upon power-up.

Timing relationships of the $\overline{\text{RESET}}$ signal are shown in Figure 15.

Release of the $\overline{\text{RESET}}$ signal is also the time at which the Automatic First Wait State function of the TMS 9995 can be invoked (see Section 2.3.1.3).


ANY NUMBER OF CLKOUT CYCLES
CLKOUT
MEMEN
DBIN
WE/CRUCLK
RESET
①
②
③
④
START CONTEXT SWITCH (FETCH WP OF RESET VECTOR)

NOTES:

① Don't care XXX indicates that any type of TMS9995 cycle can be taking place

② RESET is sampled at every high-to-low CLKOUT transition

③ RESET is required to be active (low) for a minimum of two samples to initiate the sequence. The context switch would begin one CLKOUT cycle after ③ if RESET were inactive (high) at ③ .

④ The context switch using the Reset trap vector begins one CLKOUT cycle after RESET is sampled as having returned to the inactive (high) level.

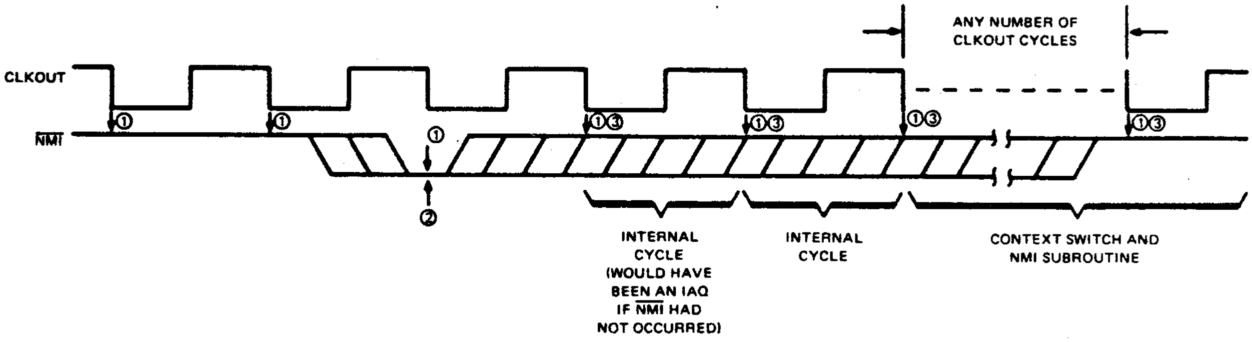
FIGURE 15 – TMS9995 RESET SIGNAL TIMING RELATIONSHIPS

2.3.2.1.2 Non-Maskable Interrupt (NMI)

The NMI signal is the request input for the NMI level interrupt and allows ROM loaders, single-step/breakpoint/maintenance panel functions, or other user-defined functions to be implemented for the TMS 9995. This signal and its associated interrupt level are named "LOAD" in previous 9900 family products.

NMI being active (low) according to the timing illustrated in Figure 16 constitutes a request for the NMI level interrupt. The TMS 9995 services this request exactly according to the basic sequence previously described, with the priority level, trap vector location, and enabling/resulting status register interrupt mask values as defined in Table 2. Note that the TMS 9995 will always grant a request for the NMI level interrupt immediately after execution of the currently executing instruction is completed since NMI is exempt from the interrupt-disabling-after-execution characteristic of certain instructions and also the current value of the interrupt mask.

It should also be noted that the TMS 9995 implements four bytes of its internal RAM at the memory address of the NMI vector. This allows usage of the NMI level in minimum-chip TMS 9995 systems. It also requires, however, that this vector must be initialized, upon power-up, before the NMI level interrupt can be requested.


ANY NUMBER OF CLKOUT CYCLES
CLKOUT
$\overline{\text{NMI}}$
①
①
①
②
①③
①③
①③
①③
INTERNAL CYCLE (WOULD HAVE BEEN AN IAQ IF $\overline{\text{NMI}}$ HAD NOT OCCURRED)
INTERNAL CYCLE
CONTEXT SWITCH AND NMI SUBROUTINE

NOTES:

① $\overline{\text{NMI}}$ is sampled at every high-to-low CLKOUT transition

② To be recognized, $\overline{\text{NMI}}$ must be active (low) at the end of an instruction. Since instructions are variable in length, the minimum active time for $\overline{\text{NMI}}$ is variable according to the instruction being executed. Shown by ② is the last possible time that $\overline{\text{NMI}}$ must be recognized at or by before execution of the next instruction will commence. The $\overline{\text{NMI}}$ context switch begins three CLKOUT cycles after execution of the current instruction is complete.

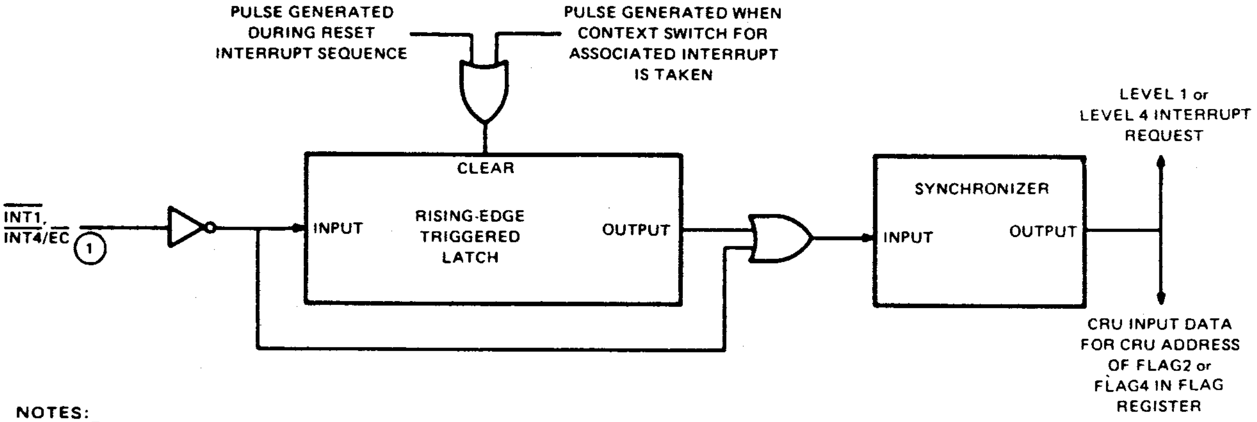
③ After an $\overline{\text{NMI}}$ context switch sequence has been initiated, $\overline{\text{NMI}}$ can remain active (low) indefinitely without causing consecutive $\overline{\text{NMI}}$ trap requests. To enable another $\overline{\text{NMI}}$ trap request, $\overline{\text{NMI}}$ must be taken inactive (high) and be sampled at least once at the inactive level.

FIGURE 16 – TMS9995 NMI SIGNAL TIMING RELATIONSHIPS

2.3.2.1.3 Interrupt Levels 1 and 4 ($\overline{\text{INT1}}$ and $\overline{\text{INT4}}/\overline{\text{EC}}$)

The $\overline{\text{INT1}}$ and $\overline{\text{INT4}}/\overline{\text{EC}}$ signals are the request inputs for the Level 1 and Level 4 interrupts, respectively. (Note that if the decrementer is configured as an event counter, $\overline{\text{INT4}}/\overline{\text{EC}}$ is no longer a Level 4 interrupt request input, however. See Section 2.3.1.2.2). Levels 1 and 4 are maskable, user-definable interrupts.

The $\overline{\text{INT1}}$ and $\overline{\text{INT4}}/\overline{\text{EC}}$ interrupt inputs can accept either asynchronous pluses or asynchronous levels as input signals. An internal interrupt request latch stores the occurrence of a pulse. A block diagram of the TMS 9995 internal logic for these request latches that is representative of their functional operation (but not necessarily representative of their specific logic implementation) is shown in Figure 17. Note that with this implementation only a single interrupt source is allowed if the input signal is a pulse, but multiple interrupt sources can be wired-ORed together provided that each source supplies a level as the input signal. (The levels are then removed one at a time by a hardware/software mechanism activated by the interrupt subroutine as each interrupting source is serviced by the subroutine.)


PULSE GENERATED DURING RESET INTERRUPT SEQUENCE
PULSE GENERATED WHEN CONTEXT SWITCH FOR ASSOCIATED INTERRUPT IS TAKEN
LEVEL 1 or LEVEL 4 INTERRUPT REQUEST
CLEAR
$\overline{\text{INT1}}$, $\overline{\text{INT4}}/\overline{\text{EC}}$ ①
INPUT
RISING-EDGE TRIGGERED LATCH
OUTPUT
SYNCHRONIZER
INPUT
OUTPUT
CRU INPUT DATA FOR CRU ADDRESS OF FLAG2 or FLAG4 IN FLAG REGISTER

NOTES:

① A separate latch and synchronizer is implemented for Level 1 ($\overline{\text{INT1}}$) and Level 4 ($\overline{\text{INT4}}/\overline{\text{EC}}$). For Level 1, the input shown here is directly from the $\overline{\text{INT1}}$ pin. For Level 4 the input shown here is from the gating shown in Figure 12.

FIGURE 17 – FUNCTIONAL BLOCK DIAGRAM OF INTERNAL INTERRUPT REQUEST LATCH

The TMS 9995 services each of these requests exactly according to the basic sequence previously described with the priority levels, trap vector locations, and enabling/resulting status register interrupt mask values as defined in Table 2. Each internal interrupt request latch will get reset when the context switch for its associated interrupt level occurs.

2.3.2.2 *Internally Generated Interrupts*

Each of these interrupts is requested when the designated condition has occurred in the TMS 9995.

2.3.2.2.1 Macro Instruction Detection (MID) Interrupt

The acquisition and attempted execution of an MID interrupt opcode will cause the MID level interrupt to be requested before execution of the next instruction begins (MID interrupt opcodes are defined in Section 4.5.15). In addition to requesting the MID level interrupt, the MID flag is set to one "1" (see Section 2.3.3.2.2). The TMS 9995 services this request exactly according to the basic sequence previously described, with the priority level, trap vector location, and enabling/resulting status register interrupt mask values as defined in Table 2. Note that the TMS 9995 will always grant a request for the MID level interrupt since MID is not affected by the interrupt mask and is higher in priority than any other interrupt except for Level 0, Reset. If the NMI interrupt is requested during an MID interrupt context switch, the MID interrupt context switch will be immediately followed by the NMI interrupt service sequence before the first instruction indicated by the MID interrupt is executed. This is done so that the NMI interrupt can be used for a single-step function with MID opcodes. Servicing the MID interrupt request is viewed as "execution" of an MID interrupt opcode. NMI allows the TMS 9995 to be halted immediately after encountering an MID opcode.

It should also be noted that the MID interrupt shares its trap vector with Level 2, the Arithmetic Overflow interrupt. (See Section 2.3.2.2.2.) The interrupt subroutine beginning with the PC of this vector should examine the MID Flag to determine the cause of the interrupt. If the MID Flag is set to "1", an MID interrupt has occurred, and if the MID Flag is set to "0", an Arithmetic Overflow interrupt has occurred. The portion of this interrupt subroutine that handles MID interrupts should always, before returning from the subroutine, reset the MID Flag to "0".

The MID interrupt has basically two applications. The MID opcodes can be considered to be illegal opcodes. The MID interrupt is then used to detect errors of this nature. The second, and primary application of the MID interrupt, is to allow the definition of additional instructions for the TMS 9995. MID opcodes are used as the opcodes for these macro instructions. Software in the MID interrupt service routine emulates the execution of these instructions. The benefit of this implementation of macros is that the macro instructions can be implemented in microcode in future processors and software will then be directly transportable to these future processors.

Note that the TMS 9995 interrupt request processing sequence does create some difficulties for re-entrant usage of MID interrupt macro instructions. In general, to avoid possible errors, MID interrupt macro instructions should not be used in the NMI and Level 1 interrupt subroutines, and should only be used in the Reset subroutine if Reset is a complete initialization of the system.

2.3.2.2.2 Arithmetic Overflow Interrupt

The occurrence of an arithmetic overflow condition, defined as status register bit 4 (ST4) getting set to one (see Table 7. for those conditions that set ST4 to one), can cause the Level 2 interrupt to be requested. Note that this request will be granted immediately after the instruction that caused the overflow condition. The TMS 9995 services this request exactly according to the basic sequence previously described with the priority level, trap vector location, and enabling/resulting status register interrupt mask values as defined in Table 2.

In addition to being maskable with the interrupt mask, the Level 2 overflow interrupt request is enabled/disabled by status register bit 10 (ST10), the Arithmetic Overflow Enable Bit (i.e., ST10 = 1 enables overflow interrupt request; ST10 = 0 disables overflow interrupt request). If servicing the overflow interrupt request is temporarily overridden by servicing of a higher priority interrupt, the occurrence of the overflow condition will be retained in the contents of the status register, i.e., ST4 = 1, which is saved by the higher priority context switch. Returning from the higher priority interrupt subroutine via an RTWP instruction causes the overflow condition to be reloaded into status register bit ST4 and the overflow interrupt to be requested again (upon completion of RTWP instruction). The arithmetic overflow interrupt subroutine must reset ST4 or ST10 to zero in the status word saved in register 15 before the routine is complete to prevent generating another overflow interrupt immediately after the return.

It should also be noted that the Level 2 arithmetic overflow interrupt shares its trap vector with the MID interrupt. Section 2.3.2.2.1 describes how the interrupt subroutine beginning with the PC of this vector can determine the cause of the interrupt.

2.3.2.2.3 Decrementer Interrupt

The occurrence of an interrupt request by the decrementer (see Section 2.3.1.2.2) will cause the Level 3 internal interrupt request latch to get set. This latch is similar to those for Levels 1 and 4 in that it is reset by servicing a Reset interrupt or when the context switch for its associated interrupt level occurs (Figure 17).

The Level 3 internal interrupt request latch being set constitutes a request for a Level 3 interrupt, and the TMS 9995 services this request exactly according to the basic sequence previously described with the priority level, trap vector location, and enabling/resulting status register interrupt mask values as defined in Table 2.

2.3.3 Communication Register Unit Interface

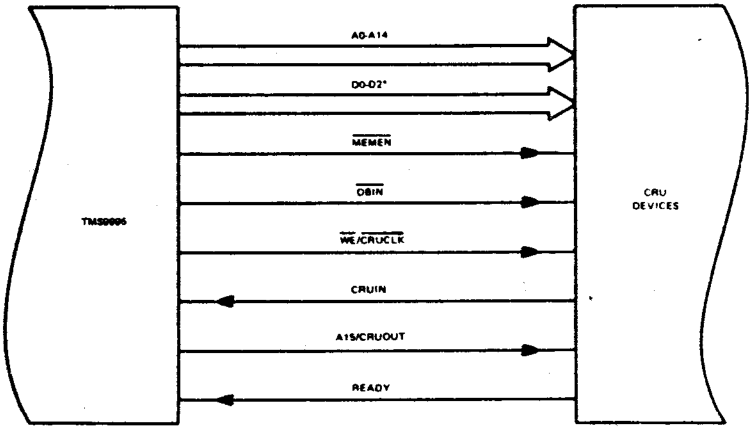
The TMS 9995 accomplishes bit I/O of varying field width through the use of the Communications Register Unit (CRU). In applications demanding a bit-oriented I/O interface, the CRU performs its most valuable act: transferring a specified number of bits to or from memory and a designated device. Thus, the CRU is simply a linking mechanism between memory and peripherals.

Acting as a shift register, the CRU is a separate hardware structure of the TMS 9995 microprocessor. This structure can serially transfer up to 16 bits of data between the CPU and a specified device in a single operation. The 32768-bit CRU address space may be divided into any combination of devices, each containing any number of input or output bits. When given the bit address of a device, the CRU can test or modify any bit in that unit. Several consecutive addresses can be occupied by a device. These CRU applications are controlled by single and multiple-bit 9995 instructions.

Single-bit instructions facilitate the testing or modification of a particular bit in a device. The device in which a bit is to be tested (TB), set to zero (SBZ), or set to one (SBO) is designated by the sum of the value in Register 12 and an 8-bit signed displacement value included as an operand of that instruction. Details of these instructions are given in Section 4.5.7.

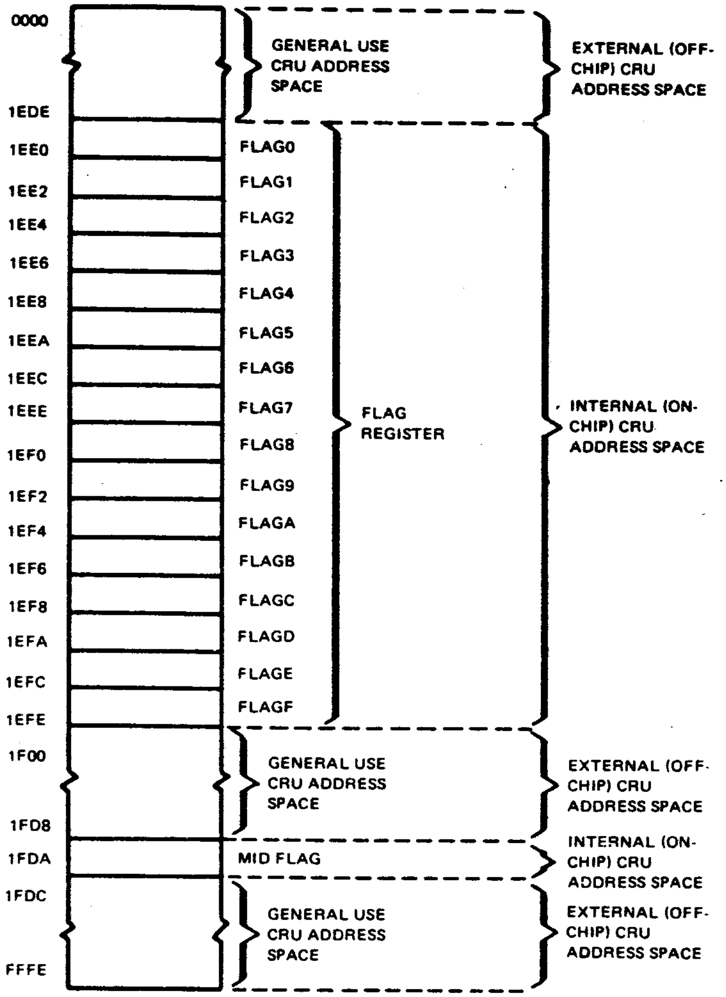
Multiple-bit instructions control the serial transfer of up to 16 bits between memory and peripherals. The device with which communication is to take place is addressed by Register 12. The memory address to or from which data is to be transferred, as well as the number of bits to be transferred are included as operands of the multiple-bit instruction. Details of these instructions are given in Section 4.5.6.

The signals used in the TMS 9995 interface to the CRU are shown in Figure 18. The CRU address map is shown in Figure 19.



NOTE:
D0-D2 are used to distinguish between CRU and external instruction cycles. If external instructions are not used in a system, D0 D2 are not necessary in the CRU interface.

FIGURE 18 – TMS9995 CRU INTERFACE


0000
1EDE
1EE0
1EE2
1EE4
1EE6
1EE8
1EEA
1EEC
1EEE
1EF0
1EF2
1EF4
1EF6
1EF8
1EFA
1EFC
1EFE
1F00
1FD8
1FDA
1FDC
FFFE
GENERAL USE CRU ADDRESS SPACE
EXTERNAL (OFF-CHIP) CRU ADDRESS SPACE
FLAG0
FLAG1
FLAG2
FLAG3
FLAG4
FLAG5
FLAG6
FLAG7
FLAG8
FLAG9
FLAGA
FLAGB
FLAGC
FLAGD
FLAGE
FLAGF
FLAG REGISTER
INTERNAL (ON-CHIP) CRU ADDRESS SPACE
GENERAL USE CRU ADDRESS SPACE
EXTERNAL (OFF-CHIP) CRU ADDRESS SPACE
MID FLAG
INTERNAL (ON-CHIP) CRU ADDRESS SPACE
GENERAL USE CRU ADDRESS SPACE
EXTERNAL (OFF-CHIP) CRU ADDRESS SPACE

NOTE: These hex addresses are the software base addresses and are obtained by placing the 15-bit Address Bus CRU bit address in a 16-bit field, left-justifying the 15 bits in the field, and setting the LSB of the field to zero.

FIGURE 19 – CRU ADDRESS MAP

The concept of "CRU space" is the key to CRU operations. An ideological area exists in which peripheral devices reside in the form of an address. The CRU space is this ideological area; it has monotonically increasing bit addresses. Each bit represents a bistable I/O point which can be read from or written to. CRU address space and memory address space are independent of each other. Memory space is byte-addressable, and CRU space is bit-addressable. Therefore, a desired device is accessed by placing its software base address in Register 12 and exercising the CRU commands.

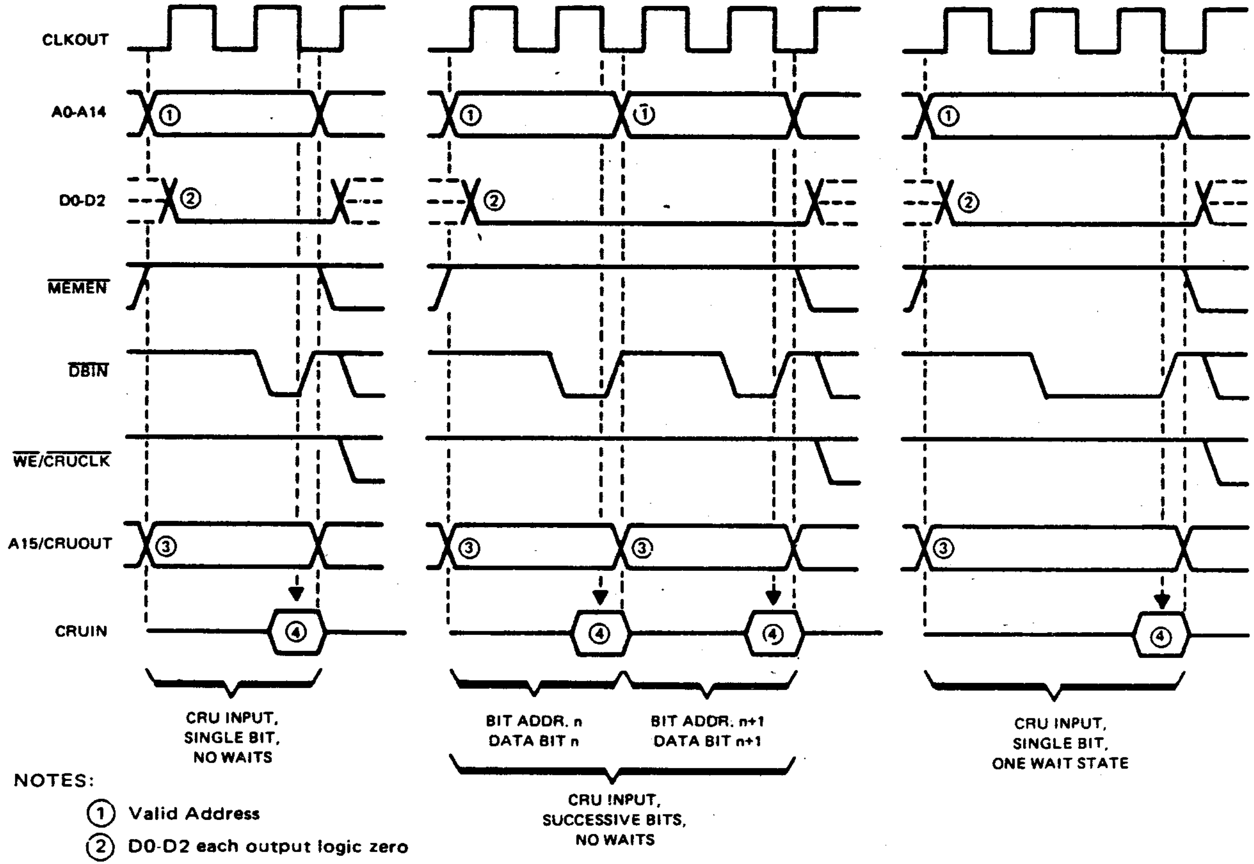
CRU nomenclature is built around the four address types involved in its operation. The software base address, hardware base address, address displacement, and CRU bit address interact to link memory to peripherals in bit-serial communication via the CRU.

The software base address consists of the entire 16 bits of R12. In R12, the programmer loads twice the value of the CRU hardware address of the device with which he wishes to communicate. Because only bits 0 through 14 of Register 12 are placed on the address bus, the programmer needs to shift the hardware base address left one position (equivalent to multiplying by two).

Bits 0 through 14 of Register 12 form the hardware base address. For the single-bit instructions, the hardware base address is added to the address displacement to obtain the CRU bit address. For multiple-bit instructions the hardware base address is the CRU bit address.

2.3.3.1 *External CRU Devices*

To input a data bit from an external (off-chip) CRU device, the TMS 9995 first outputs the appropriate address on A0-A14. The TMS 9995 leaves $\overline{\text{MEMEN}}$ high, outputs logic zeroes on D0-D2, strobes $\overline{\text{DBIN}}$, and reads in the data bit on CRUIN. Completion of each CRU input cycle and/or generation of Wait states is determined by the READY input as detailed in Section 2.3.1.3. Timing relationships of the CRU input cycle are shown in Figure 20.


CLKOUT
A0-A14
D0-D2
$\overline{\text{MEMEN}}$
$\overline{\text{DBIN}}$
$\overline{\text{WE}}$/CRUCLK
A15/CRUOUT
CRUIN
①
②
③
④
CRU INPUT, SINGLE BIT, NO WAITS
BIT ADDR. n DATA BIT n
BIT ADDR. n+1 DATA BIT n+1
CRU INPUT, SUCCESSIVE BITS, NO WAITS
CRU INPUT, SINGLE BIT, ONE WAIT STATE

NOTES:

① Valid Address

② D0-D2 each output logic zero

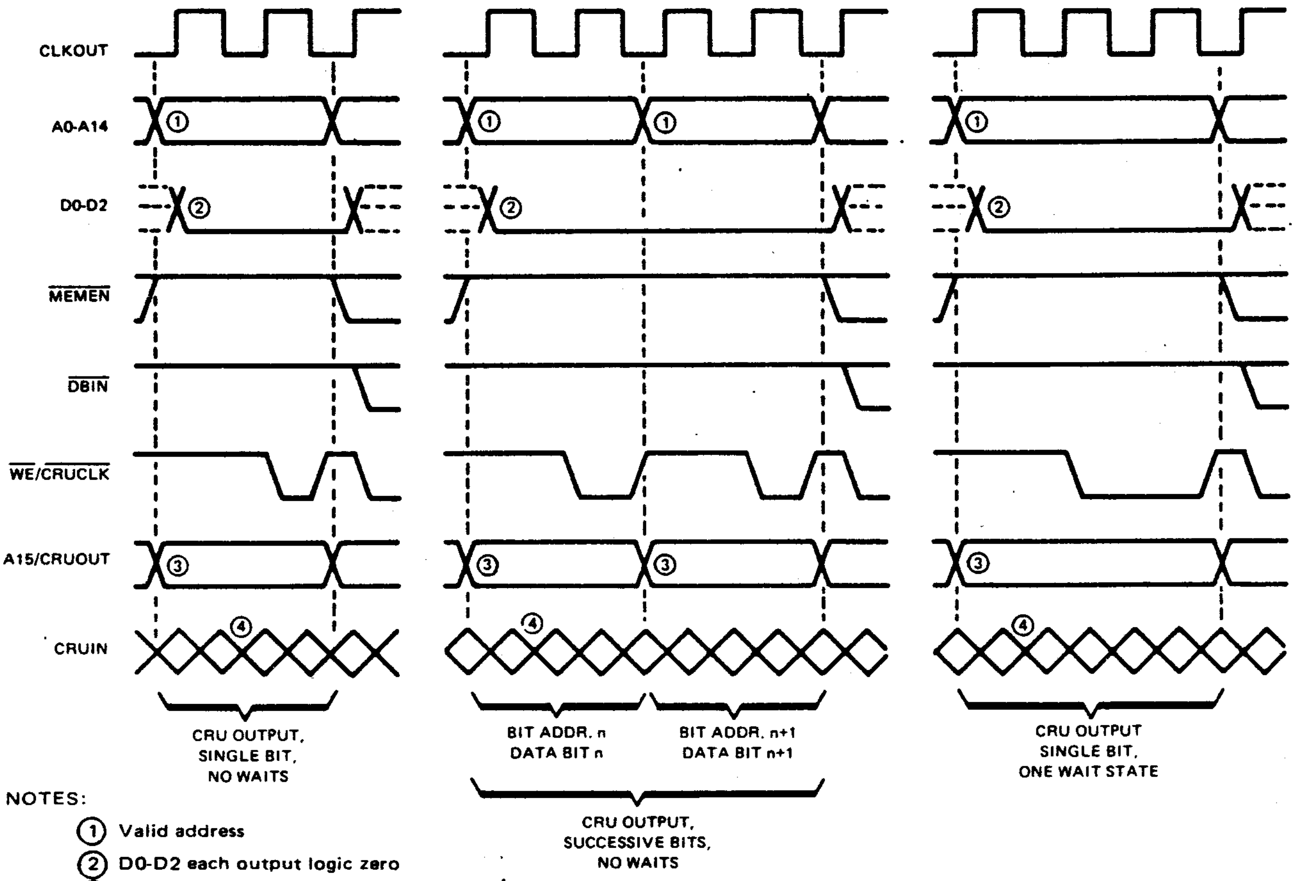
③ Non-specific output bit

④ CRU input bit must be valid on CRUIN at CLKOUT edge indicated

FIGURE 20 – TMS9995 CRU INPUT CYCLE

To output a data bit to an external (off-chip) CRU device, the TMS 9995 first outputs the appropriate address on A0-A14. The TMS 9995 leaves $\overline{\text{MEMEN}}$ high, outputs logic zeroes on D0-D2, outputs the data bit on A15/CRUOUT, and strobes $\overline{\text{WE}}$/CRUCLK. Completion of each CRU output cycle and/or generation of Wait states is determined by the READY input as detailed in Section 2.3.1.3. Timing relationships of the CRU output cycle are shown in Figure 21.

For multiple-bit transfers, these input and output cycles are repeated until transfer of the entire field of data bits specified by the CRU instruction being executed has been accomplished.


CLKOUT
A0-A14
D0-D2
MEMEN
DBIN
WE/CRUCLK
A15/CRUOUT
CRUIN
①
②
③
④
CRU OUTPUT, SINGLE BIT, NO WAITS
BIT ADDR. n DATA BIT n
BIT ADDR. n+1 DATA BIT n+1
CRU OUTPUT, SUCCESSIVE BITS, NO WAITS
CRU OUTPUT SINGLE BIT, ONE WAIT STATE

NOTES:

① Valid address

② D0-D2 each output logic zero

③ Valid CRU output bit for address being output

④ Don't care

FIGURE 21 – TMS9995 CRU OUTPUT CYCLE

2.3.3.1.1 Single-Bit CRU Operations

The TMS 9995 performs three single-bit CRU functions: Test Bit (TB), Set Bit to One (SBO), and Set Bit to Zero (SBZ). The SBO instruction performs a CRU output cycle with logic one for the data bit, and the SBZ instruction performs a CRU output cycle with logic zero for the data bit. A TB instruction transfers the addressed CRU bit from the CRUIN input line to bit 2 of the status register (bit ST2, the EQUAL bit).

The TMS 9995 develops a CRU bit address for the single-bit operations from the CRU base address contained in workspace register 12 and the signed displacement count contained in bits 8 through 15 of the instruction. The displacement allows two's complement addressing from base minus 128 bits through base plus 127 bits. The base address from WR12 is added to the signed displacement specified in the instruction and the result is placed onto the address bus. Figure 22 illustrates the development of a single-bit CRU address.

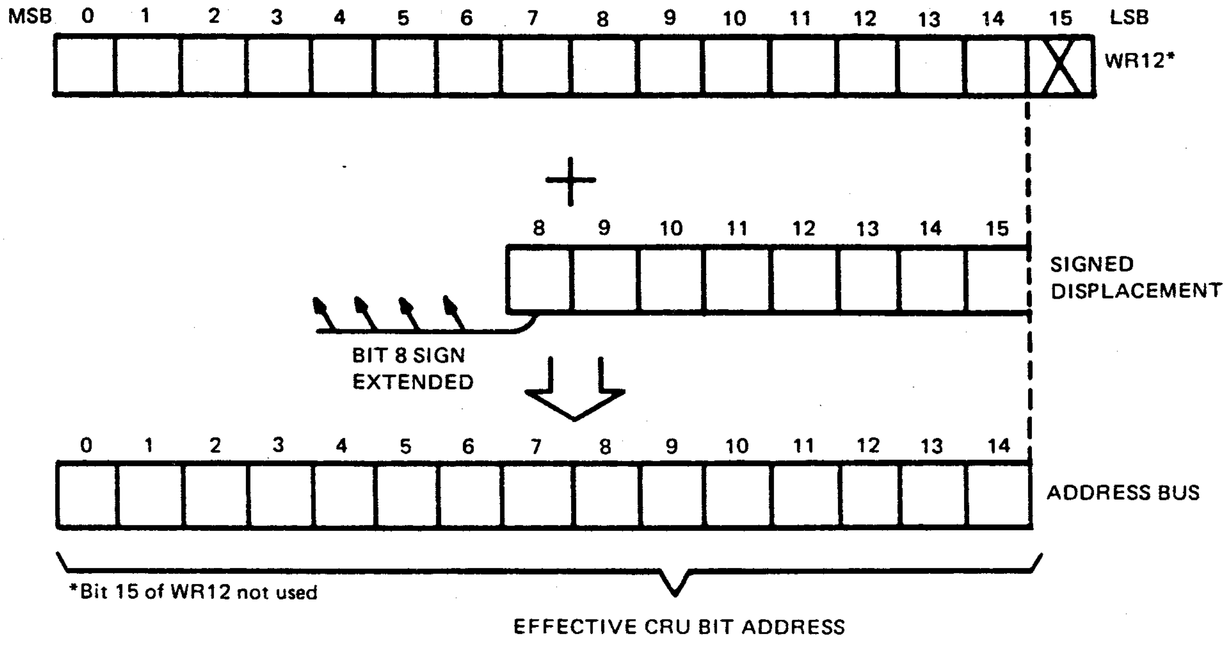


FIGURE 22 – SINGLE BIT CRU ADDRESS DEVELOPMENT

2.3.3.1.2 Multiple Bit CRU Operations

The TMS 9995 performs two multiple-bit CRU operations: store communications register (STCR) and load communications register (LDCR). Both operations perform a data transfer from the CRU-to-memory or from memory-to-CRU as illustrated in Figure 23. Although the figure illustrates a full 16-bit transfer operation, any number of bits from 1 through 16 may be involved.

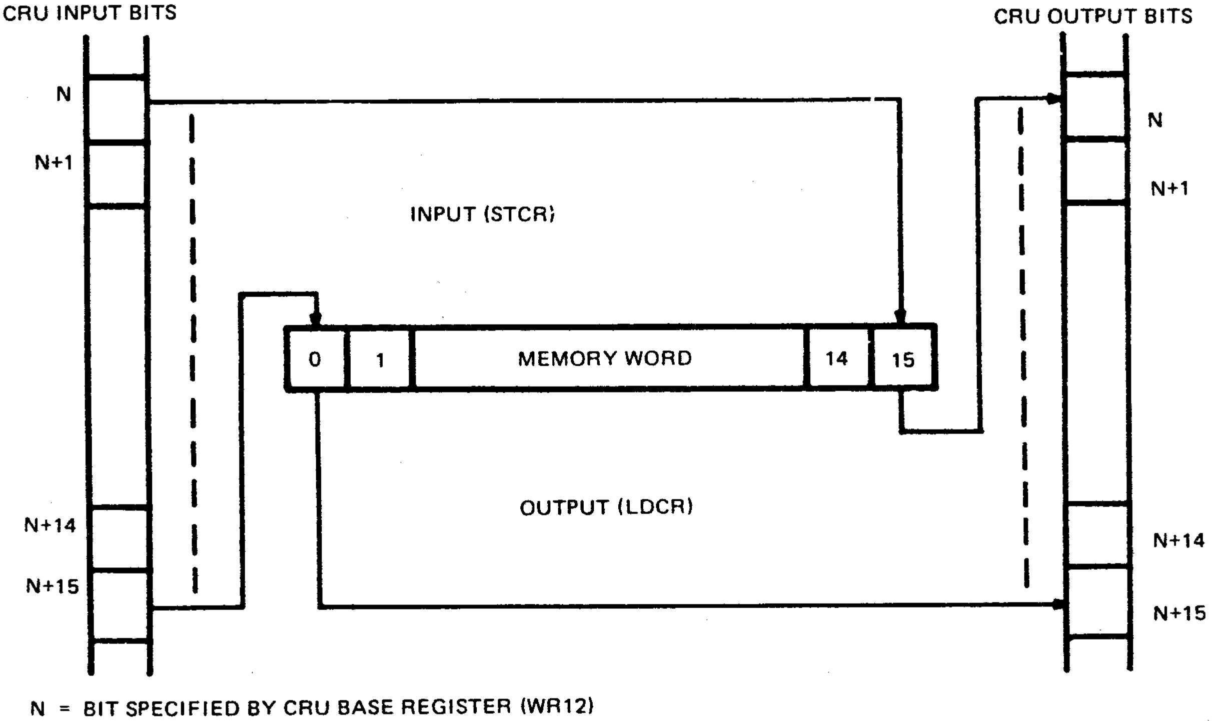


FIGURE 23 – LDCR/STCR DATA TRANSFERS

The LDCR instruction fetches a word from memory and right shifts it to serially transfer it to CRU output bits. If the LDCR involves eight or fewer bits, those bits come from the right-justified field within the addressed byte of the memory word. If the LDCR involves nine or more bits, those bits come from the right-justified field within the whole memory word. Register 12, bits 0 through 14, defines the starting bit address. When transferred to the CRU interface, each successive bit receives an address that is sequentially greater than the address for the previous bit. This addressing mechanism results in an order reversal of the bits; that is, bit 15 of the memory word (or bit 7) becomes the lowest addressed bit in the CRU and bit 0 becomes the highest bit in the CRU field.

A STCR instruction transfers data from the CRU to memory. If the operation involves a byte or less transfer, the transferred data will be stored right-justified in the memory byte with leading bits set to zero. If the operation involves from nine to 16 bits, the transferred data is stored right-justified in the memory word with leading bits set to zero. When the input from the CRU device is complete, the lowest addressed bit from the CRU is in the least-significant bit position in the memory word or byte.

2.3.3.2 *Internal CRU Devices*

Access of internal (on-chip) CRU devices is transparent to the TMS 9995 CRU instructions. Data can be input from and output to the bits of the internal CRU devices simply by using the appropriate CRU addresses to access these bits.

The TMS 9995 will indicate to the external world when these internal CRU bit accesses are occurring by asserting the same signals used for accessing external CRU devices (see Figure 18). The timing of these signals for internal CRU input and output cycles will be identical to the timing for external CRU input and output cycles (see Figure 20 and 21) except that during internal CRU cycles, the READY input is ignored, i.e., Wait states cannot be generated, and, during internal CRU input cycles, the TMS 9995 will ignore the CRUIN input signal. The internal bit being input will not be available to the external world on CRUIN.

The functional characteristics of the internal CRU devices are described in the following paragraphs.

2.3.3.2.1 Flag Register

Accessible via CRU input and output instructions that are executed to dedicated internal CRU bit addresses (see Figure 19) is the internal Flag Register. The 16-bit Flag Register contains both predefined TMS 9995 systems flags and user-definable flags as detailed in Table 3. The predefined system flags are the configuration bit for the Decrementer, the Decrementer enable bit, and the internal interrupt request latch CRU inputs. Note that CRU output operations to the internal interrupt request latch Flag addresses will not cause these latches to be either set or reset. These Flag bits are input only and allow the presence of these interrupt requests to be detected when the occurrence of the interrupts themselves is inhibited by the value of the interrupt mask in the status register.

2.3.3.2.2 MID Flag

Accessible via CRU input and output instructions that are executed to a dedicated internal CRU bit address (see Figure 19) is the MID Flag. The MID Flag is set to one by a MID interrupt, and reset to zero by the software of the MID interrupt routine (see Section 2.3.2.2.1). Note that setting the MID Flag to one with a CRU instruction will not cause the MID interrupt to be requested.

2.3.4 External Instructions

The TMS 9995 has five external instructions (see Table 4) that allow user-defined external functions to be initiated under program control. These instructions are CKON, CKOF, RSET, IDLE, and LREX. These mnemonics, except for IDLE, relate to functions implemented in the 990 minicomputer and do not restrict use of the instructions to initiate various user-defined functions. Execution of an IDLE instruction causes the TMS 9995 to enter the Idle state and remain in this state until a request occurs for an interrupt level that is not masked by the current value of the interrupt mask in the status register. (Note that the Reset and NMI interrupt levels are not masked by any interrupt mask value.) When any of these five instructions are executed by the TMS 9995, the TMS 9995 will use the CRU interface (see Figure 18) to perform a cycle that is identical to a single-bit CRU output cycle (see Figure 21) except for the following: (1) the address being output will be non-specific, (2) the data bit being output will be non-specific, (3) a code, specified in Table 4, will be output on D0-D2 to indicate the external instruction being executed. Note that completion of each external instruction and/or generation of Wait states is determined by the READY input as detailed in Section 2.3.1.3.

TABLE 3 — FLAG REGISTER BIT DEFINITIONS

BIT	CRU BIT ADDRESS[†]	DESCRIPTION
FLAG0	1EE0	Set to 0: Decrementer configured as Interval Timer. Set to 1: Decrementer configured as Event Counter.
FLAG1	1EE2	Set to 0: Decrementer not enabled Set to 1: Decrementer enabled (will decrement and can set internal latch that requests a level 3 interrupt).
FLAG2	1EE4	Level 1 Internal Interrupt Request Latch CRU Input (Input-only). 0: Level 1 request not present 1: Level 1 request present
FLAG3	1EE6	Level 3 Internal Interrupt Request Latch CRU Input (Input-only). 0: Level 3 request not present 1: Level 3 request present
FLAG4	1EE8	Level 4 Internal Interrupt Request Latch CRU Input (Input-only). 0: Level 4 request not present 1: Level 4 request present
FLAG5 FLAG6 FLAG7 FLAG8 FLAG9 FLAGA FLAGB FLAGC FLAGD FLAGE FLAGF	1EEA 1EEC 1EEE 1EF0 1EF2 1EF4 1EF6 1EF8 1EFA 1EFC 1EFE	User Defined

[†] These hex numbers are those obtained by placing the 15-bit Address Bus CRU address in a 16-bit field, left justifying the 15 bits in the field, and setting the LSB of the field to zero.

TABLE 4 — TMS 9995 EXTERNAL INSTRUCTION CODES

INSTRUCTION	CODE DURING CYCLE		
	D0	D1	D2
CRU: SBO, SBZ, TB, LDCR or STCR	0	0	0
IDLE	0	1	0
RSET	0	1	1
CKON	1	0	1
CKOF	1	1	0
LREX	1	1	1

When the TMS 9995 is in the Idle state, cycles with the Idle code will occur repeatedly until a request for an interrupt level that is not masked by the interrupt mask in the status register occurs.

A Hold state can occur during an Idle state, with entry to and return from the Hold state occurring at the Idle code cycle boundaries. (See Section 2.3.1.1.3 for details of entry to and return from the Hold state.)

2.3.5 TMS 9995 Internal ALU/Other Operation Cycles

When the TMS 9995 is performing an operation internally and is not using the memory, CRU, or external instruction interfaces[†] or is not in the Hold state, the TMS 9995 will, for as many CLKOUT cycles as needed, do the following with its interface signals:

(1) Output a non-specific address on A0-A14 and A15/CRUOUT

(2) Output non-specific data on D0-D7

(3) Output logic level high on $\overline{\text{MEMEN}}$, $\overline{\text{DBIN}}$, and $\overline{\text{WE}}/\overline{\text{CRUCLK}}$

(4) Output logic level low on IAQ/HOLDA, and

(5) Ignore the READY and CRUIN inputs.

The HOLD input is still active, however, as the TMS 9995 can enter a Hold state while performing an internal ALU/other operation. Also, all interrupt inputs are still active.

[†] Internal memory space and internal CRU device accesses are defined as using the memory and CRU interfaces.

APPENDIX F

TMS 9995 MICROCOMPUTER

INSTRUCTION SET

4. TMS 9995 INSTRUCTION SET

4.1 DEFINITION

Each TMS 9995 instruction performs one of the following operations:

- Arithmetic, logical, comparison, or manipulation operations on data
- Loading or storage of internal registers (program counter, workspace pointer, or status)
- Data transfer between memory and external devices via the CRU
- Control functions

4.2 ADDRESSING MODES

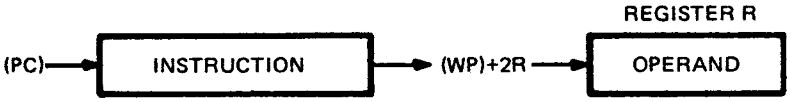
The TMS 9995 instructions contain a variety of available modes for addressing random memory data, e.g., program parameters and flags, or formatted memory data (character strings, data lists, etc.). These addressing modes are:

- Workspace Register Addressing
- Workspace Register Indirect Addressing
- Workspace Register Indirect Auto Increment Addressing
- Symbolic (Direct) Addressing
- Indexed Addressing
- Immediate Addressing
- Program Counter Relative Addressing
- CRU Relative Addressing

The following figures graphically describe the derivation of effective address for each addressing mode. The applicability of addressing modes to particular instructions is described in Section 4.5 along with the description of the operations performed by each instruction. The symbols following the names of the addressing modes (R, *R, *R+, @LABEL or @TABLE (R) are the general forms used by TMS 9995 assemblers to select the addressing modes for register R.

4.2.1 Workspace Register Addressing, R

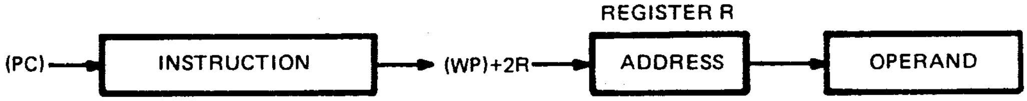
Workspace Register R contains the operand



The Workspace Register addressing mode is specified by setting the two-bit T-field (T_S or T_D) of the instruction word equal to 00.

4.2.2 Workspace Register Indirect Addressing, *R

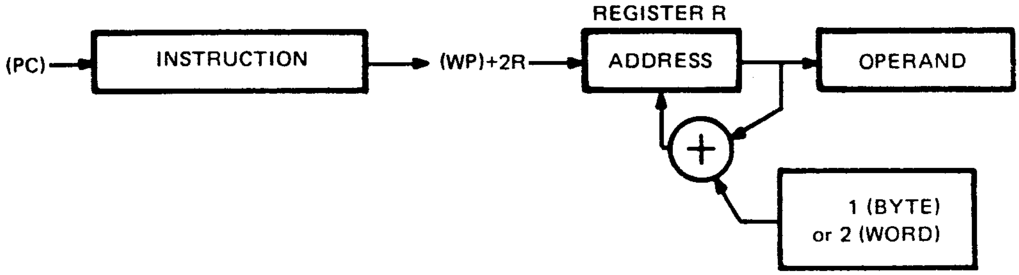
Workspace Register R contains the address of the operand.



The Workspace Register Indirect addressing mode is specified by setting the two-bit T-field (T_S or T_D) in the instruction word equal to 01.

4.2.3 Workspace Register Indirect Auto Increment Addressing, *R+

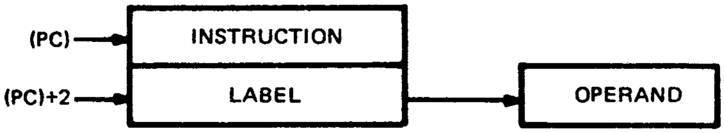
Workspace Register R contains the address of the operand. After acquiring the address of the operand, the contents of Workspace Register R are incremented.



The Workspace Register Indirect Auto Increment addressing mode is specified by setting the two-bit T-field (T_S or T_D) in the instruction word equal to 11.

4.2.4 Symbolic (Direct) Addressing, @LABEL

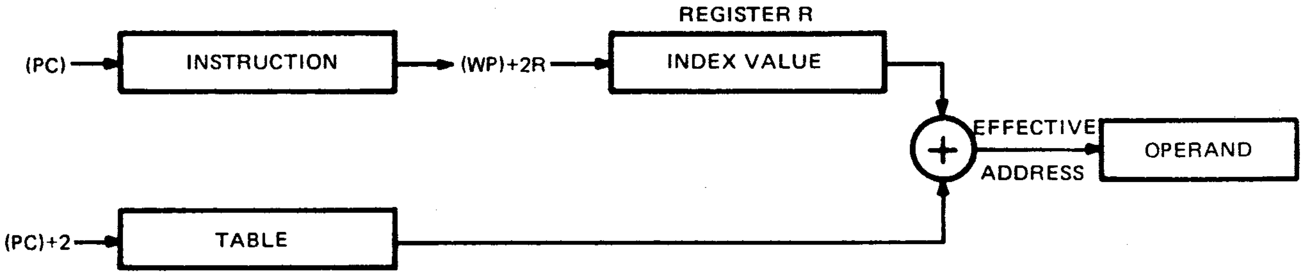
The word following the instruction contains the address of the operand.



The Symbolic addressing mode is specified by setting the two-bit T-field (T_S or T_D) in the instruction word equal to 10 and setting the corresponding S or D field equal to 0.

4.2.5 Indexed Addressing, @TABLE (R)

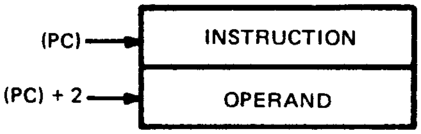
The word following the instruction contains the base address. Workspace Register R contains the index value. The sum of the base address and the index value results in the effective address of the operand.



The indexed addressing mode is specified by setting the two-bit T-field (T_S or T_D) of the instruction word equal to 10 and setting the corresponding S or D field not equal to 0. The value in the S or D field is the register which contains the index value.

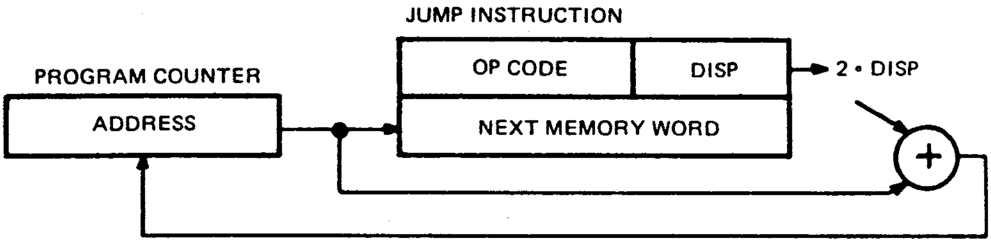
4.2.6 Immediate Addressing

The word following the instruction contains the operand.



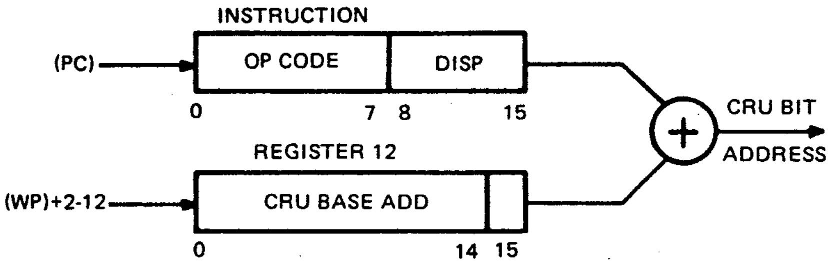
4.2.7 Program Counter Relative Addressing

The eight-bit signed displacement in the right byte (bits 8 through 15) of the instruction is multiplied by 2 and added to the updated contents of the program counter. The result is placed in the PC.



4.2.8 CRU Relative Addressing

The eight-bit signed displacement in the right byte of the instruction is added to the CRU base address (bits 0 through 14 of workspace register 12). The result is the CRU address of the selected CRU bit.



4.3 DEFINITION OF TERMINOLOGY

The terminology used in describing the instructions of the TMS 9995 is defined in Table 6.

4.4 STATUS REGISTER MANIPULATION

Various TMS 9995 machine instructions affect the status register. Figure 5 shows the status register bit assignments. Table 7 lists the instructions and their effect on the status register.

4.5 INSTRUCTIONS

4.5.1 Dual Operand Instructions with Multiple Addressing for Source and Destination Operand

General	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Format:	OP CODE			B	T_D		D				T_S		S			

If B = 1, the operands are bytes and the operand addresses are byte addresses. If B = 0, the operands are words and the LSB of the operand address is ignored.

The addressing mode for each operand is determined by the T-field of that operand.

T_S or T_D	S or D	ADDRESSING MODE	NOTES
00	0, 1 . . . 15	Workspace register	1
01	0, 1 . . . 15	Workspace register indirect	
10	0	Symbolic	4
10	1, 2 . . . 15	Indexed	2, 4
11	0, 1 . . . 15	Workspace register indirect auto increment	3

NOTES:
1. When a workspace register is the operand of a byte instruction (bit 3 = 1), the left byte (bits 0 through 7) is the operand and the right byte (bits 8 through 15) is unchanged.
2. Workspace register 0 may not be used for indexing.
3. The workspace register is incremented by 1 for byte instructions (bit 3 = 1) and is incremented by 2 for word instructions (bit 3 = 0).
4. When T_S = T_D = 10, two words are required in addition to the instruction word. The first word is the source operand base address and the second word is the destination operand base address.

TABLE 6 – DEFINITION OF TERMINOLOGY

TERM	DEFINITIONS
B	Byte Indicator (1 = byte; 0 = word)
C	Bit Count
D	Destination address register
DA	Destination address
IOP	Immediate operand
LSB (n)	Least-significant (right most) bit of (n)
MSB (n)	Most-significant (left most) bit of (n)
N	Don't care
PC	Program Counter
Result	Result of operation performed by instruction
S	Source address register
SA	Source address
ST	Status register
STn	Bit n of status register
TD	Destination address modifier
TS	Source address modifier
W	Workspace register
WRn	Workspace register n
(n)	Contents of n
a → b	a is transferred to b
|n|	Absolute value of n
+	Arithmetic addition
–	Arithmetic subtraction
AND	Logical AND
OR	Logical OR
⊕	Logical exclusive OR
$\bar{n}$	Logical complement of n
•	Arithmetic multiplication

TABLE 7 – STATUS REGISTER BIT DEFINITIONS†

BIT	NAME	INSTRUCTION AND/OR INTERRUPT	CONDITION TO SET BIT TO 1, OTHERWISE SET TO 0 FOR INSTRUCTION LISTED. ALSO, THE EFFECT OF OTHER INSTRUCTIONS AND INTERRUPTS
ST0	Logical Greater Than	C, CB	If MSB (SA) = 1 and MSB (DA) = 0, or If MSB (SA) = MSB (DA) and MSB of [(DA) – (SA)] = 1.
		CI	If MSB (W) = 1 and MSB of IOP = 0, or if MSB (W) = MSB of IOP and MSB of [IOP – (W)] = 1.
		ABS, LDCR	If (SA) ≠ 0
		RTWP	If bit (0) of WR15 is 1
		LST	If bit (0) of selected WR is 1
		A, AB, AI, ANDI, DEC, DECT, LI, MOV, MOVB, NEG, ORI, S, SB, DIVS, MPYS, INC, INCT, INV, SLA, SOC, SOCB, SRA, SRC, SRL, STCR, SZC, SZCB, XOR	If result ≠ 0
		Reset Interrupt	Unconditionally sets status bit to 0
		All other instructions and interrupts	Do not affect the status bit (see Note 1)
ST1	Arithmetic Greater Than	C, CB	If MSB (SA) = 0 and MSB (DA) = 1, or If MSB (SA) = MSB (DA) and MSB of [(DA) – (SA)] = 1.
		CI	If MSB (W) = 0 and MSB of IOP = 1, or if MSB (W) = MSB of IOP and MSB of [IOP – (W)] = 1.
		ABS, LDCR	If MSB (SA) = 0 and (SA) ≠ 0
		RTWP	If bit (1) of WR15 is 1
		LST	If bit (1) of selected WR is 1
		A, AB, AI, ANDI, DEC, DECT, LI, MOV, MOVB, NEG, ORI, S, SB, DIVS, MPYS, INC, INCT, INV, SLA, SOC, SOCB, SRA, SRC, SRL, STCR, SZC, SZCB, XOR	If MSB of result = 0 and result ≠ 0
		Reset Interrupt	Unconditionally sets status bit to 0
		All other instructions and interrupts	Do not affect the status bit (see Note 1)

See Table 6 for definitions of terminology used in this table.

TABLE 7 – STATUS REGISTER BIT DEFINITIONS (Continued)

BIT	NAME	INSTRUCTION AND/OR INTERRUPT	CONDITION TO SET BIT TO 1, OTHERWISE SET TO 0 FOR INSTRUCTION LISTED. ALSO, THE EFFECT OF OTHER INSTRUCTIONS AND INTERRUPTS
ST2	Equal	C, CB	If (SA) = (DA)
		CI	If (W) = IOP
		COC	If (SA) and $(\overline{DA})$ = 0
		CZC	If (SA) and (DA) = 0
		TB	If CRUIN = 1 for addressed CRU bit
		ABS, LDCR	If (SA) = 0
		RTWP	If bit (2) of WR15 is 1
		LST	If bit (2) of selected WR is 1
		A, AB, AI, ANDI, DEC, DECT, LI, MOV, MOVB, NEG, ORI, S, SB, DIVS, MPYS, INC, INCT, INV, SLA, SOC, SOCB, SRA, SRC, SRL, STCR, SZC, SZCB, XOR	If result = 0
		Reset Interrupt	Unconditionally sets status bit to 0
		All other instructions and interrupts	Do not affect the status bit (see Note 1)
ST3	Carry	A, AB, ABS, AI, DEC, DECT, INC, INCT, NEG, S, SB	If CARRY OUT = 1
		SLA, SRA, SRL, SRC	If last bit shifted out = 1
		RTWP	If bit (3) of WR15 is 1
		LST	If bit (3) of selected WR is 1
		Reset Interrupt	Unconditionally sets status bit to 0
		All other instructions and interrupts	Do not affect the status bit (see Note 1)
ST4	Overflow	A, AB	If MSB (SA) = MSB (DA) and MSB of result ≠ MSB (DA)
		AI	If MSB (W) = MSB of IOP and MSB of result ≠ MSB (W)
		S, SB	If MSB (SA) ≠ MSB (DA) and MSB of result ≠ MSB (DA)
		DEC, DECT	If MSB (SA) = 1 and MSB of result = 0
		INC, INCT	If MSB (SA) = 0 and MSB of result = 0
		SLA	If MSB changes during shift
		DIV	If MSB (SA) = 0 and MSB (DA) = 1, or if MSB (SA) = MSB (DA) and MSB of [(DA) – (SA)] = 0
		DIVS	If the quotient cannot be expressed as a signed 16 bit quantity (8000_{16} is a valid negative number)
		ABS, NEG	If (SA) = 8000_{16}
		RTWP	If bit (4) of WR15 is 1
		LST	If bit (4) of selected WR is 1
		Reset Interrupt	Unconditionally sets status bit to 0
		All other instructions and interrupts	Do not affect the status bit (see Note 1)

TABLE 7 – STATUS REGISTER BIT DEFINITIONS (Concluded)

BIT	NAME	INSTRUCTION AND/OR INTERRUPT	CONDITION TO SET BIT TO 1, OTHERWISE SET TO 0 FOR INSTRUCTION LISTED. ALSO, THE EFFECT OF OTHER INSTRUCTIONS AND INTERRUPTS
ST5	Odd Parity	CB, MOVB	If (SA) has odd number of 1's
		LDCR	If $1 \le C \le 8$ and (SA) has odd number of 1's. If C = 0 or $9 \le C \le 15$, does not affect the status bit.
		STCR	If $1 \le C \le 8$ and the stored bits have an odd number of 1's. If C = 0 or $9 \le C \le 15$, does not affect the status bit.
		AB, SB, SOCB, SZCB	If result has odd number of 1's.
		RTWP	If bit (5) of WR15 is 1
		LST	If bit (5) of selected WR is 1
		Reset Interrupt	Unconditionally sets status bit to 0
		All other instructions and Interrupts	Do not affect the status bit (see Note 1)
ST6	XOP	XOP	If XOP instruction is executed
		RTWP	If bit (6) of WR15 is 1
		LST	If bit (6) of selected WR is 1
		Reset Interrupt	Unconditionally sets status bit to 0
		All other instructions and interrupts	Do not affect the status bit (see Note 1)
ST7 ST8 ST9 and ST11	Unused Bits	RTWP	If corresponding bit of WR15 is 1
		LST	If corresponding bit of selected WR is 1.
		XOP, Any Interrupt	Unconditionally sets each of these status bits to 0
		All other instructions	Do not affect these status bits (see Note 1)
ST10	Arithmetic Overflow Enable	RTWP	If bit (10) of WR is 1
		LST	If bit (10) of selected WR is 1
		XOP, Any Interrupt	Unconditionally sets status bit to 0
		All other instructions	Do not affect the status bit (see Note 1)
ST12 ST13 ST14 and ST15	Interrupt Mask	LIMI	If corresponding bit of IOP is 1
		RTWP	If corresponding bit of WR15 is 1
		LST	If corresponding bit of selected WR is 1.
		RST, Reset and NMI Interrupts	Unconditionally sets each of these status bits to 0
		All other interrupts	If ST12 – ST15 = 0, no change If ST12 = ST15 ≠ 0, set to one Less than level of the interrupt trap taken
		All other instructions	Do not affect these status bits (see Note 1)

OTE 1: The X instruction itself does not affect any status bit; the instruction executed by the X instruction sets status bits as defined fo that instruction.

MNEMONIC	OP CODE			B	MEANING	RESULT COMPARED TO 0	STATUS BITS AFFECTED	DESCRIPTION
	0	1	2	3				
A	1	0	1	0	Add	Yes	0-4	(SA) + (DA) → (DA)
AB	1	0	1	1	Add bytes	Yes	0-5	(SA) + (DA) → (DA)
C	1	0	0	0	Compare	No	0-2	Compare (SA) to (DA) and set appropriate status bits
CB	1	0	0	1	Compare bytes	No	0-2,5	Compare (SA) to (DA) and set appropriate status bits
S	0	1	1	0	Subtract	Yes	0-4	(DA) – (SA) → (DA)
SB	0	1	1	1	Subtract bytes	Yes	0-5	(DA) – (SA) → (DA)
SOC	1	1	1	0	Set ones corresponding	Yes	0-2	(DA) OR (SA) → (DA)
SOCB	1	1	1	1	Set ones corresponding bytes	Yes	0-2,5	(DA) OR (SA) → (DA)
SZC	0	1	0	0	Set zeroes corresponding	Yes	0-2	(DA) AND $(\overline{SA})$ → (DA)
SZCB	0	1	0	1	Set zeroes corresponding bytes	Yes	0-2,5	(DA) AND $(\overline{SA})$ → (DA)
MOV	1	1	0	0	Move	Yes	0-2	(SA) → (DA)
MOVB	1	1	0	1	Move bytes	Yes	0-2,5	(SA) → (DA)

4.5.2 Dual Operand Instructions with Multiple Addressing Modes for the Source Operand and Workspace Register Addressing for the Destination

General	0 1 2 3 4 5	6 7 8 9	10 11	12 13 14 15
Format:	OP CODE	D	T_S	S

The addressing mode for the source operand is determined by the T_S field.

T_S	S	ADDRESSING MODE	NOTES
00	0, 1 . . . 15	Workspace register	
01	0, 1 . . . 15	Workspace register indirect	
10	0	Symbolic	
10	1, 2 . . . 15	Indexed	1
11	0, 1 . . . 15	Workspace register indirect auto increment	2

NOTES: 1. Workspace register 0 may not be used for indexing.
2. The workspace register is incremented by 2.

MNEMONIC	OP CODE	MEANING	RESULT COMPARED TO 0	STATUS BITS AFFECTED	DESCRIPTION
	0 1 2 3 4 5				
COC	0 0 1 0 0 0	Compare ones corresponding	No	2	Test (D) to determine if 1's are in each bit position where 1's are in (SA). If so, set ST2.
CZC	0 0 1 0 0 1	Compare zeroes corresponding	No	2	Test (D) to determine if 0's are in each bit position where 1's are in (SA). If so, set ST2.
XOR	0 0 1 0 1 0	Exclusive OR	Yes	0-2	(DA) ⊕ (SA) → (D)
MPY	0 0 1 1 1 0	Multiply	No	–	Multiply unsigned (D) by unsigned (SA) and place unsigned 32-bit product in D (most-significant) and D+1 (least-significant). If WR15 is D, the next word in memory after WR15 will be used for the least significant half of the product.
DIV	0 0 1 1 1 1	Divide	No	4	If unsigned (SA) is less than or equal to unsigned (D), perform no operation and set ST4. Otherwise, divide unsigned (D) and (D+1) by unsigned (SA). Quotient → (D), remainder → (D+1). If D = 15, the next word in memory after WR15 will be used for the remainder.

4.5.3 Signed Multiply and Divide Instructions

General	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Format:	OP CODE										T_S		S			

The addressing mode for the source operand is determined by the T_S field.

T_S	S	ADDRESSING MODE	NOTES
00	0, 1 . . . 15	Workspace register	1
01	0, 1 . . . 15	Workspace register indirect	1
10	0	Symbolic	1
10	1, 2 . . . 15	Indexed	1,2
11	0, 1 . . . 15	Workspace register indirect auto increment	1,3

NOTES: 1. Workspace registers 0 and 1 contain operands used in the signed multiply and divide operations.
2. Workspace register 0 may not be used for indexing.
3. The workspace register is incremented by 2.

MNEMONIC	OP CODE										MEANING	RESULT COMPARED TO 0	STATUS BITS AFFECTED	DESCRIPTION
	0	1	2	3	4	5	6	7	8	9				
MPYS	0	0	0	0	0	0	0	1	1	1	Signed Multiply	Yes	0-2	Multiply signed two's complement integer in WR0 by signed two's complement integer (SA) and place signed 32-bit product in WR0 (most-significant) and WR1 least-significant.
DIVS	0	0	0	0	0	0	0	1	1	0	Signed Divide	Yes	0-2,4	If the quotient cannot be expressed as a signed 16 bit quantity (8000 (hex) is a valid negative number), set ST4. Otherwise, divide signed, two's complement integer in WR0 and WR1 by the signed two's complement integer (SA) and place the signed quotient in WR0 and the signed remainder in WR1. The sign of the quotient is determined by algebraic rules. The sign of the remainder is the same as the sign of the dividend and $\lvert REMAINDER \rvert < \lvert DIVISOR \rvert$

4.5.4 Extended Operation (XOP) Instruction

General	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Format::	0	0	1	0	1	1	D				T_S		S			

The T_S and S fields provide multiple mode addressing capability for the source operand. When the XOP is executed, the following transfers occur:

$(40_{16} + 4D) \longrightarrow (WP)$

$(42_{16} + 4D) \longrightarrow (PC)$

$SA \longrightarrow (\text{new } WR11)$

$(\text{old } WP) \longrightarrow (\text{new } WR13)$

$(\text{old } PC) \longrightarrow (\text{new } WR14)$

$(\text{old } ST) \longrightarrow (\text{new } WR15)$

After these transfers have been made, ST6 is set to one, and ST7, ST8, ST9, ST10 (Overflow Interrupt Enable), and ST11 are all set to zero.

The TMS 9995 does not service interrupt trap requests (except for the Reset and NMI Requests) at the end of the XOP instruction.

4.5.5 Single Operand Instructions

General	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Format:	OP CODE										T_S		S			

The T_S and S fields provide multiple mode addressing capability for the source operand.

MNEMONIC	OP CODE										MEANING	RESULT COMPARED TO ZERO	STATUS BITS AFFECTED	DESCRIPTION
	0	1	2	3	4	5	6	7	8	9				
B	0	0	0	0	0	1	0	0	0	1	Branch	No	–	SA → (PC)
BL	0	0	0	0	0	1	1	0	1	0	Branch and link	No	–	(PC) → (WR11); SA → (PC)
BLWP	0	0	0	0	0	1	0	0	0	0	Branch and load workspace pointer	No	–	(SA) → (WP); (SA + 2) → (PC); (old WP) → (new WR13); (old PC) → (new WR14); (old ST) → (new WR15); The TMS 9995 does not service interrupt trap requests (except for the Reset and NMI Requests) at the end of the BLWP instruction.
CLR	0	0	0	0	0	1	0	0	1	1	Clear Operand	No	–	0 → (SA)
SETO	0	0	0	0	0	1	1	1	0	0	Set to ones	No	–	$FFFF_{16}$ → (SA)
INV	0	0	0	0	0	1	0	1	0	1	Invert	Yes	0-2	($\overline{\text{SA}}$) → (SA)
NEG	0	0	0	0	0	1	0	1	0	0	Negate	Yes	0-4	−(SA) → (SA)
ABS	0	0	0	0	0	1	1	1	0	1	Absolute value*	No	0-4	|(SA)| → (SA)
SWPB	0	0	0	0	0	1	1	0	1	1	Swap bytes	No	–	(SA), bits 0 thru 7 → (SA) bits 8 thru 15; (SA), bits 8 thru 15 → (SA), bits 0 thru 7.
INC	0	0	0	0	0	1	0	1	1	0	Increment	Yes	0-4	(SA) + 1 → (SA)
INCT	0	0	0	0	0	1	0	1	1	1	Increment by two	Yes	0-4	(SA) + 2 → (SA)
DEC	0	0	0	0	0	1	1	0	0	0	Decrement	Yes	0-4	(SA) − 1 → (SA)
DECT	0	0	0	0	0	1	1	0	0	1	Decrement by two	Yes	0-4	(SA) − 2 → (SA)
X**	0	0	0	0	0	1	0	0	1	0	Execute	No	–	Execute the instruction at SA.

* Operand is compared to zero for status bit.

** If additional memory words for the execute instruction are required to define the operands of the instruction located at SA, these words will be accessed from PC and the PC will be updated accordingly. The instruction acquisition signal (IAQ) will not be true when the TMS 9995 accesses the instruction at SA. Status bits are affected in the normal manner for the instruction executed.

4.5.6 CRU Multiple-Bit Instruction

General	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Format:	OP CODE						C				T_S		S			

The C field specifies the number of bits to be transferred. If C = 0, 16 bits will be transferred. The CRU base register (WR12, bits 0 through 14) defines the starting CRU bit address. The bits are transferred serially and the CRU address is incremented with each bit transfer, although the contents of WR12 are not affected. T_S and S provide multiple mode addressing capability for the source operand. If eight or fewer bits are transferred (C = 1 through 8), the source address is a byte address. If nine or more bits are transferred (C = 0, 9 through 15), the source address is a word address. If the source is addressed in the workspace register indirect auto increment mode, the workspace register is incremented by one if C = 1 through 8, and is incremented by two otherwise. If the source is addressed in the register mode, and if the transfer is eight bits or less, bits 8 - 15 are unchanged.

MNEMONIC	OP CODE						MEANING	RESULT COMPARED TO 0	STATUS BITS AFFECTED	DESCRIPTION
	0	1	2	3	4	5				
LDCR	0	0	1	1	0	0	Load communication register	Yes	0-2,5*	Beginning with LSB of (SA), transfer the specified number of bits from (SA) to the CRU.
STCR	0	0	1	1	0	1	Store communication register	Yes	0-2,5*	Beginning with LSB of (SA), transfer the specified number of bits from the CRU to (SA). Load unfilled bit positions with 0.

*ST5 is affected only if $1 \leqslant C \leqslant 8$.

4.5.7 CRU Single-Bit Instructions

General	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Format:	OP CODE								SIGNED DISPLACEMENT							

The signed displacement is added to the contents of WR12 (bits 0-14) to form the address of the CRU bit to be selected.

MNEMONIC	OP CODE								MEANING	STATUS BITS AFFECTED	DESCRIPTION
	0	1	2	3	4	5	6	7			
SBO	0	0	0	1	1	1	0	1	Set bit to one	–	Set the selected output bit to 1.
SBZ	0	0	0	1	1	1	1	0	Set bit to zero	–	Set the selected output bit to 0.
TB	0	0	0	1	1	1	1	1	Test bit	2	If the selected CRU input bit = 1, set ST2; if the selected CRU input = 0, set ST2 = 0.

4.5.8 Jump Instructions

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
General Format:	OP CODE								SIGNED DISPLACEMENT							

Jump instructions cause the PC to be loaded with the value selected by PC relative addressing if the bits of ST are at specified values. Otherwise, no operation occurs and the next instruction is executed since the PC points to the next instruction. The signed displacement field is a word count to be added to PC. Thus, the jump instruction has a range of −128 to 127 words from memory-word address following the jump instruction.

No ST bits are affected by jump instructions.

MNEMONIC	OP CODE								MEANING	ST CONDITION TO LOAD PC
	0	1	2	3	4	5	6	7		
JEQ	0	0	0	1	0	0	1	1	Jump equal	ST2 = 1
JGT	0	0	0	1	0	1	0	1	Jump greater than	ST1 = 1
JH	0	0	0	1	1	0	1	1	Jump high	ST0 = 1 and ST2 = 0
JHE	0	0	0	1	0	1	0	0	Jump high or equal	ST0 = 1 or ST2 = 1
JL	0	0	0	1	1	0	1	0	Jump low	ST0 = 0 and ST2 = 0
JLE	0	0	0	1	0	0	1	0	Jump low or equal	ST0 = 0 or ST2 = 1
JLT	0	0	0	1	0	0	0	1	Jump less than	ST1 = 0 and ST2 = 0
JMP	0	0	0	1	0	0	0	0	Jump unconditional	Unconditional
JNC	0	0	0	1	0	1	1	1	Jump no carry	ST3 = 0
JNE	0	0	0	1	0	1	1	0	Jump not equal	ST2 = 0
JNO	0	0	0	1	1	0	0	1	Jump no overflow	ST4 = 0
JOC	0	0	0	1	1	0	0	0	Jump on carry	ST3 = 1
JOP	0	0	0	1	1	1	0	0	Jump odd parity	ST5 = 1

4.5.9 Shift Instructions

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
General Format:	OP CODE								C				W			

If C = 0, bits 12 through 15 of WR0 contain the shift count. If C = 0 and bits 12 through 15 of WR0 = 0, the shift count is 16.

MNEMONIC	OP CODE								MEANING	RESULT COMPARED TO 0	STATUS BITS AFFECTED	DESCRIPTION
	0	1	2	3	4	5	6	7				
SLA	0	0	0	0	1	0	1	0	Shift left arithmetic	Yes	0-4	Shift (W) left. Fill vacated bit positions with 0.
SRA	0	0	0	0	1	0	0	0	Shift right arithmetic	Yes	0-3	Shift (W) right. Fill vacated bit positions with original MSB of (W).
SRC	0	0	0	0	1	0	1	1	Shift right circular	Yes	0-3	Shift (W) right. Shift previous LSB into MSB.
SRL	0	0	0	0	1	0	0	1	Shift right logical	Yes	0-3	Shift (W) right. Fill vacated bit positions with 0's.

4.5.10 Immediate Register Instructions

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
General	OP CODE											O	W			
Format:	IOP															

MNEMONIC	OP CODE											MEANING	RESULT COMPARED TO 0	STATUS BITS AFFECTED	DESCRIPTION
	0	1	2	3	4	5	6	7	8	9	10				
AI	0	0	0	0	0	0	1	0	0	0	1	Add immediate	Yes	0-4	(W) + IOP → (W)
ANDI	0	0	0	0	0	0	1	0	0	1	0	AND immediate	Yes	0-2	(W) AND IOP → (W)
CI	0	0	0	0	0	0	1	0	1	0	0	Compare immediate	Yes	0-2	Compare (W) to IOP and set appropriate status bits.
LI	0	0	0	0	0	0	1	0	0	0	0	Load immediate	Yes	0-2	IOP → (W)
ORI	0	0	0	0	0	0	1	0	0	1	1	OR immediate	Yes	0-2	(W) OR IOP → (W)

4.5.11 Internal Register Load Immediate Instructions

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
General	OP CODE											0	0	0	0	0
Format:	IOP															

MNEMONIC	OP CODE											MEANING	DESCRIPTION
	0	1	2	3	4	5	6	7	8	9	10		
LWPI	0	0	0	0	0	0	1	0	1	1	1	Load workspace pointer immediate	IOP → (WP), no ST bits affected.
LIMI	0	0	0	0	0	0	1	1	0	0	0	Load interrupt mask	IOP, bits 12 thru 15 → ST12 thru ST15.

4.5.12 Internal Register Load and Store Instructions

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
General Format:	OP CODE												W			

MNEMONIC	OP CODE												MEANING	STATUS BITS AFFECTED	DESCRIPTION
	0	1	2	3	4	5	6	7	8	9	10	11			
STST	0	0	0	0	0	0	1	0	1	1	0	0	Store status Register	–	(ST) → (W)
LST	0	0	0	0	0	0	0	0	1	0	0	0	Load status Register	0-15	(W) → (ST)
STWP	0	0	0	0	0	0	1	0	1	0	1	0	Store workspace pointer	–	(WP) → (W)
LWP	0	0	0	0	0	0	0	0	1	0	0	1	Load workspace pointer	–	(W) → (WP)

4.5.13 Return Workspace Pointer (RTWP) Instruction

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

The RTWP instruction causes the following transfers to occur:

(WR15) → (ST)

(WR14) → (PC)

(WR13) → (WP)

4.5.14 External Instructions

General	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Format:	OP CODE											0	0	0	0	0

External instructions cause three data lines (D0 through D2) to be set to the levels described below, and the $\overline{\text{WE}}$/$\overline{\text{CRUCLK}}$ line to be pulsed, allowing external control functions to be initiated.

MNEMONIC	OP CODE											MEANING	STATUS BITS AFFECTED	DESCRIPTION	DATA BUS		
	0	1	2	3	4	5	6	7	8	9	10				D0	D1	D2
IDLE	0	0	0	0	0	0	1	1	0	1	0	Idle	–	Suspend TMS 9995 instruction execution until an unmasked interrupt level request occurs.	L	H	L
RSET	0	0	0	0	0	0	1	1	0	1	1	Reset	12-15	Set ST12-ST15 to zero.	L	H	H
CKOF	0	0	0	0	0	0	1	1	1	1	0	User defined	–	–	H	H	L
CKON	0	0	0	0	0	0	1	1	1	0	1	User defined	–	–	H	L	H
LREX	0	0	0	0	0	0	1	1	1	1	1	User defined	–	–	H	H	H

4.5.15 MID Interrupt Opcodes

The instruction opcodes that will cause an MID interrupt request (see Section 2.3.2.2) are (hex numbers):

0000-007F
00A0-017F
0210-021F
0230-023F
0250-025F
0270-027F
0290-029F
02B0-02BF
02D0-02DF
02E1-02FF
0301-033F
0341-035F
0361-037F
0381-039F
03A1-03BF
03C1-03DF
03E1-03FF
0780-07FF
0C00-0FFF

4.6 INSTRUCTION EXECUTION

4.6.1 Microinstruction Cycle

Each TMS 9995 instruction is executed by a sequence of machine states (microinstructions) with the length of each sequence depending upon the specific instruction being executed. Each microinstruction is completed in one CLKOUT cycle unless Wait states are added to a memory or CRU cycle. (Also, each external memory space access of a word and each external CRU cycle requires at least two CLKOUT cycles but will be accomplished with a single microinstruction).

4.6.2 Execution Sequence

The TMS 9995 incorporates an instruction prefetch scheme which minimizes, and in some cases eliminates, the time required to fetch the instruction from memory. Without the prefetch, a typical instruction execution sequence is as follows:

(1) Fetch instruction

(2) Decode instruction

(3) Fetch source operand, if needed

(4) Fetch destination operand, if needed

(5) Process the operands

(6) Store the results, if required

The TMS 9995 makes use of the fact that during Step 5 the memory interface is not required; therefore, the fetch of the next instruction can be accomplished in this time. This instruction is then decoded during the state(s) that is(are) required to store the results of the previous instruction, which creates even more execution overlap. Table 8 illustrates the case of maximum efficiency for an Add instruction (instruction opcodes and operands are located in the internal RAM). Note that it effectively takes only four machine states to perform all six steps.

TABLE 8 – EXECUTION SEQUENCE EXAMPLE

STEP	STATE COUNT	MEMORY CYCLE	INTERNAL FUNCTION
1		Fetch Instruction	Process Previous Operands
2	1	Write Results	Decode Instruction
3	2	Fetch Source	
4	3	Fetch Destination	
5	4	Fetch Next Instruction	Add
6		Write Results	Decode Instruction

It should be noted that the instruction prefetch scheme employed by the TMS 9995 can cause self-modifying software to execute incorrectly. Incorrect execution will result when an instruction is supposed to generate the opcode of the very next instruction to be executed. (The TMS 9995 will begin the fetch of the opcode of the next instruction before the currently executing instruction stores the results of its execution.)

4.6.3 TMS 9995 Instruction Execution Times

Instruction execution times for the TMS 9995 are a function of:

(1) Machine state time, t_{c2}.

(2) The location of the instruction opcode (internal or external memory).

(3) The location of the workspace and the operand(s) (internal or external memory).

(4) Addressing mode used where operands can be fetched via multiple addressing modes.

(5) Number of Wait states introduced, as appropriate.

Table 9 lists the number of clock cycles required to execute each TMS 9995 instruction for various combinations of on-chip/off-chip location of instruction opcodes, operands, and workspace. (Other combinations can be extropolated from the ones listed.) For instructions with multiple addressing modes for either or both operands, Table 9 lists CLKOUT cycles and associated off-chip memory accesses with all operands addressed in the workspace register mode. To determine the total number of CLKOUT cycles and associated off-chip memory accesses required for other addressing modes, the appropriate values from Table "A" (Table 10) are added to the base amounts for that instruction.

The total execution time for an instruction is:

$$T = t_{c2}\,[C1 + C2 + W\,(XM1 + XM2)]$$

where

T = total instruction execution time
t_{c2} = CLKOUT cycle time
$C1$ = base CLKOUT cycles
$C2$ = additional CLKOUT cycles for operand address derivation (values in Table"A" are for one operand only)
W = number of Wait states per off-chip (byte length) memory cycle
$XM1$ = base off-chip (byte length) memory cycles
$XM2$ = additional off-chip (byte length) memory cycles for operand address derivation (values in Table "A" are for one operand only)

Several examples are listed in Table 11.

TABLE 9 – INSTRUCTION EXECUTION TIMES

INSTRUCTION	Opcodes & All Operands On Chip		Opcodes & Immediate Operands Off Chip; All Other Operands On Chip		Opcodes & Immediate Operands Off Chip; Source Operand Off Chip; Destination Operand On Chip ⑥		Opcodes & All Operands Off Chip		Operand Address Derivation ①	
	C1	XM1	C1	XM1	C1	XM1	C1	XM1	Source	Dest
A	4	0	5	2	6	4	8	8	A	A
AB	4	0	5	2	5	3	5	5	A	A
ABS	3	0	4	2	6	6	6	6	A	–
AI	4	0	6	4	6	4	8	8	–	–
ANDI	4	0	6	4	6	4	8	8	–	–
B	3	0	4	2	4	2	4	2	A	–
BL	5	0	6	2	7	4	7	4	A	–
BLWP	11	0	12	2	14⑤	6⑤	17	12	A	–
C	4	0	5	2	6	4	7	6	A	A
CB	4	0	5	2	5	3	5	4	A	A
CI	4	0	6	4	6	4	7	6	–	–
CKOF	7	0	8	2	8	2	8	2	–	–
CKON	7	0	8	2	8	2	8	2	–	–
CLR	3	0	4	2	5	4	5	4	A	–
COC	4	0	5	2	6	4	7	6	A	–
CZC	4	0	5	2	6	4	7	6	A	–
DEC	3	0	4	2	6	6	6	6	A	–
DECT	3	0	4	2	6	6	6	6	A	–
DIV (ST4 is set)	6	0	7	2	8	4	10	8	A	–
DIV (ST4 is reset)②	28	0	29	2	30	4	34	12	A	–
DIVS (ST4 is set)	10	0	11	2	12	4	36	8	A	–
DIVS (ST4 is reset)②	33	0	34	2	35	4	39	12	A	–
IDLE③	7+2I	0	8+2I	2	8+2I	2	8+2I	2	–	–
INC	3	0	4	2	6	6	6	6	A	–
INCT	3	0	4	2	6	6	6	6	A	–
INV	3	0	4	2	6	6	6	6	A	–
JUMP (All Jump Instructions)	3	0	4	2	4	2	4	2	–	–
LDCR (C=0)	41	0	42	2	43	4	44	6	A	–
LDCR (1≤C≤15)	9+2C	0	10+2C	2	11+2C	4	12+2C	6	A	–
LI	3	0	5	4	5	4	6	6	–	–
LIMI	5	0	7	4	7	4	7	4	–	–
LREX	7	0	8	2	8	2	8	2	–	–
LST	5	0	6	2	6	2	7	4	–	–
LWP	4	0	5	2	6	2	6	4	–	–
LWPI	4	0	6	4	6	4	6	4	–	–
MOV	3	0	4	2	5	4	6	6	A	A
MOVB	3	0	4	2	4	3	4	4	A	A
MPY	23	0	24	2	25	4	28	10	A	–
MPYS	25	0	26	2	27	4	30	10	A	–
NEG	3	0	4	2	6	6	6	6	A	–
ORI	4	0	6	4	6	4	8	8	–	–
RSET	7	0	8	2	8	2	8	2	–	–
RTWP	6	0	7	2	7⑦	2⑦	10	8	–	–
S	4	0	5	2	6	4	8	8	A	A
SB	4	0	5	2	5	3	5	5	A	A
SBO	8	0	9	2	9	2	10	4	–	–

TABLE 9 – INSTRUCTION EXECUTION TIMES (Concluded)

INSTRUCTION	Opcodes & All Operands On Chip		Opcodes & Immediate Operands Off Chip; All Other Operands On Chip		Opcodes & Immediate Operands Off Chip; Source Operand Off Chip; Destination Operand On Chip ⑥		Opcodes & All Operands Off Chip		Operand Address Derivation ①	
	C1	XM1	C1	XM1	C1	XM1	C1	XM1	Source	Dest
SBZ	8	0	9	2	9	2	10	4	–	–
SETO	3	0	4	2	5	4	5	4	–	–
SHIFT (C≠0)	5+C	0	6+C	2	6+C	2	8+C	6	–	–
SHIFT (C=0, Bits 12-15 of WRO=0)	23	0	24	2	24	2	27	8	–	–
SHIFT (C=0, Bits 12-15 of WRO=N≠0)	7+N	0	8+N	2	8+N	2	11+N	8	–	–
SOC	4	0	5	2	6	4	8	8	A	A
SOCB	4	0	5	2	5	3	5	5	A	A
STCR (C=0)	43	0	44	2	46	6	47	8	A	–
STCR (1<C<8)	19+C	0	20+C	2	22+C	6	23+C	8	A	–
STCR (9<C<15)	27+C	0	28+C	2	30+C	6	31+C	8	A	–
STST	3	0	4	2	4	2	5	4	–	–
STWP	3	0	4	2	4	2	5	4	–	–
SWPB	13	0	14	2	16	6	16	6	A	–
SZC	4	0	5	2	6	4	8	8	A	A
SZCB	4	0	5	2	5	3	5	5	A	A
TB	8	0	9	2	9	2	10	4	–	–
X④	2	0	3	2	4	4	4	4	A	–
XOP	15	0	16	2	18⑤	6⑤	22	14	A	–
XOR	4	0	5	2	6	4	8	8	A	–
Interrupt Context Switch (For any interrupt, including Reset, NMI, MID, and overflow)	14⑧	0⑧	17⑤	6⑤	17⑤	6⑤	20⑨	12⑨	–	–

NOTES:

① Additional cycles to be added, if appropriate, are listed in Table "A" (Table 11).

② Execution time is dependent upon the partial quotient after each clock cycle during execution. Clock cycles shown are for worst-case operands.

③ Will remain in Idle state until an unmasked interrupt request occurs (I = number of CLKOUT cycles until request occurs).

④ Execution time shown does not include execution time of instruction at source address.

⑤ Trap vector off chip; New workspace on chip.

⑥ Registers for register-only instructions are on chip (Shift instructions, STST, LST, STWP, LWP) and registers for instructions where an additional register is required are on-chip (AI, ANDI, BL, CI, LDCR, LI, ORI, SBO, SBZ, STCR, TB, Shift instructions).

⑦ Workspace on chip

⑧ Trap vector on chip; New workspace on chip (NMI only)

⑨ Trap vector and New workspace on chip

TABLE 10 – OPERAND ADDRESS DERIVATION (TABLE "A")

ADDRESSING MODE	Workspace Registers, Base Address For Index-Addressed Operands, And Symbolic (Direct) Addresses On Chip		Workspace Registers On Chip; Base Address For Index-Addressed Operands And Symbolic (Direct) Addresses Off Chip		Workspace Registers Off Chip; Base Address For Index-Addressed Operands And Symbolic (Direct) Addresses On Chip		Workspace Registers, Base Address For Index-Addressed Operands, And Symbolic (Direct) Addresses Off Chip	
	C2	XM2	C2	XM2	C2	XM2	C2	XM2
WR (T_S or T_D = 00)	0	0	0	0	0	0	0	0
WR Indirect (T_S or T_D = 01)	1	0	1	0	2	2	2	2
WR Indirect Auto Increment (T_S or T_D = 11)	3	0	3	0	5	4	5	4
Symbolic (T_S or T_D = 10, S or D = 0)	1	0	2	2	1	0	2	2
Indexed (T_S or T_D = 10, S or D $\neq$ 0)	3	0	4	2	4	2	5	4

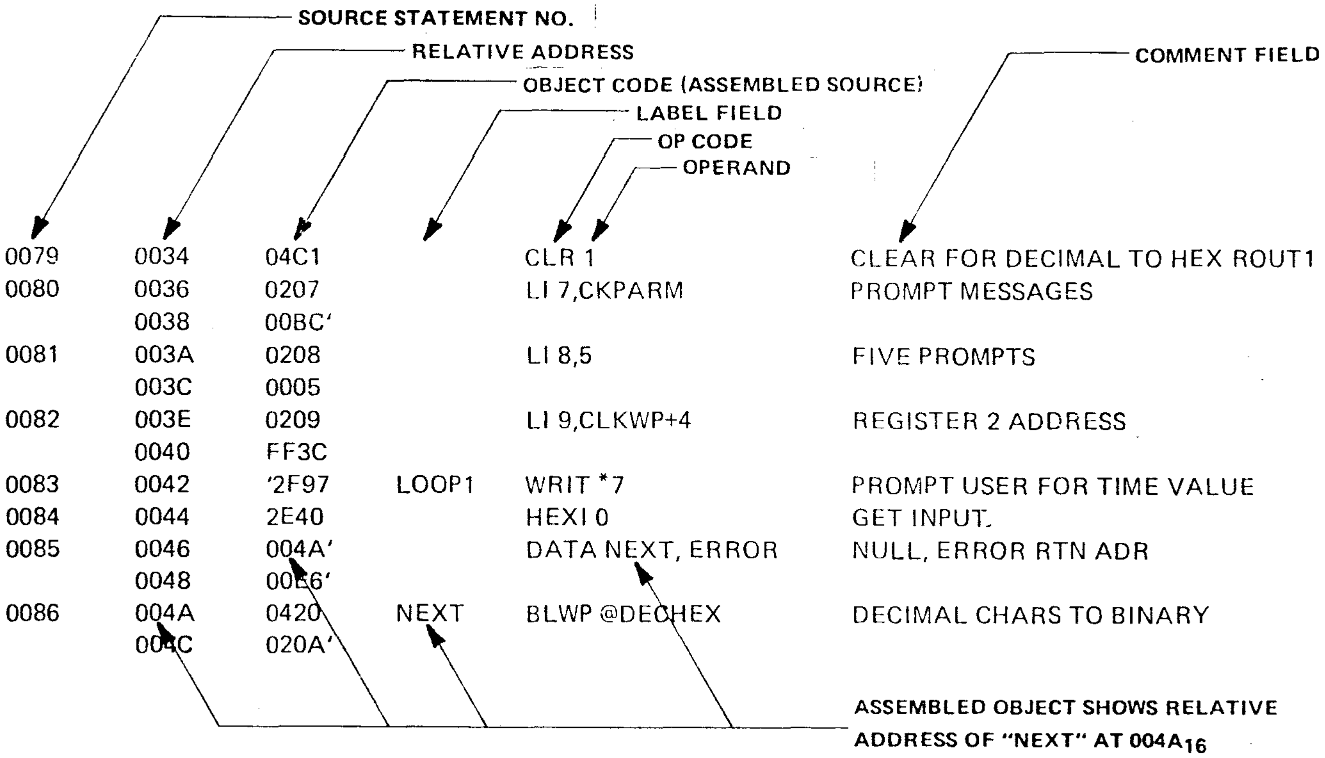
TABLE 11 – INSTRUCTION EXECUTION TIME EXAMPLES

EXAMPLE	Opcodes, base addresses for index-addressed operands, symbolic (direct) addresses, workspace registers, symbolic (direct) operands, and index-addressed operands all on chip.						Opcodes, base addresses for index-addressed operands, and symbolic (direct) addresses off chip; workspace registers, symbolic (direct) operands, and index-addressed operands on chip.						Opcodes, base addresses for index-addressed operands, symbolic (direct) addresses, workspace registers, symbolic (direct) operands, and index-addressed operands all off chip.					
	C1	XM1	C2	XM2	Total Clock Cycles		C1	XM1	C2	XM2	Total Clock Cycles		C1	XM1	C2	XM2	Total Clock Cycles	
					0 Wait States Off Chip	1 Wait State Off Chip					0 Wait States Off Chip	1 Wait State Off Chip					0 Wait States Off Chip	1 Wait State Off Chip
MOV R1, R2	3	0	0	0	3	3	4	2	0	0	4	6	6	6	0	0	6	12
MOV R1, *R2	3	0	1	0	4	4	4	2	1	0	5	7	6	6	2	2	8	16
MOV R1, *R2+	3	0	3	0	6	6	4	2	3	0	7	9	6	6	5	4	11	21
MOV R1, @LABEL	3	0	1	0	4	4	4	2	2	2	6	10	6	6	2	2	8	16
MOV R1, @TABLE (R2)	3	0	3	0	6	6	4	2	4	2	8	12	6	6	5	4	11	21
MOV *R2+, @LABEL	3	0	4	0	7	7	4	2	5	2	9	13	6	6	7	6	13	25
MOV @LABEL1, @LABEL2	3	0	2	0	5	5	4	2	4	4	8	14	6	6	4	4	10	20

APPENDIX G

SAMPLE PROGRAMS

This appendix contains listing of programs that can be loaded into memory or reassembled into memory for demonstration or entertainment purposes. These listings are commented to provide ancillary data and explain the individual programming techniques. Assembly listing format is as follows:


SOURCE STATEMENT NO.
RELATIVE ADDRESS
OBJECT CODE (ASSEMBLED SOURCE)
LABEL FIELD
OP CODE
OPERAND
COMMENT FIELD
ASSEMBLED OBJECT SHOWS RELATIVE ADDRESS OF "NEXT" AT 004A16

```
0079   0034   04C1               CLR 1              CLEAR FOR DECIMAL TO HEX ROUT1
0080   0036   0207               LI 7,CKPARM        PROMPT MESSAGES
       0038   00BC'
0081   003A   0208               LI 8,5             FIVE PROMPTS
       003C   0005
0082   003E   0209               LI 9,CLKWP+4       REGISTER 2 ADDRESS
       0040   FF3C
0083   0042   '2F97   LOOP1      WRIT *7            PROMPT USER FOR TIME VALUE
0084   0044   2E40               HEXI 0             GET INPUT.
0085   0046   004A'              DATA NEXT, ERROR   NULL, ERROR RTN ADR
       0048   00E6'
0086   004A   0420    NEXT       BLWP @DECHEX       DECIMAL CHARS TO BINARY
       004C   020A'
```

The code can be reassembled and loaded with the LMC EVMBUG command, or the change memory command (IM) can be used to insert assembled object code at the memory addresses shown in the listing.(beginning at >ED00, program start). The assembled object code is listed in column 3 of the listing, opposite the corresponding memory address in column 2. It is important that the programs be entered at the addresses noted, or that proper consideration be given to the labelled addresses which have been assembled into absolute addresses relative to the beginning of the program (address >ED00)> This consideration is important when entering the code using the enter memory (IM) command with program start not at address >ED)).

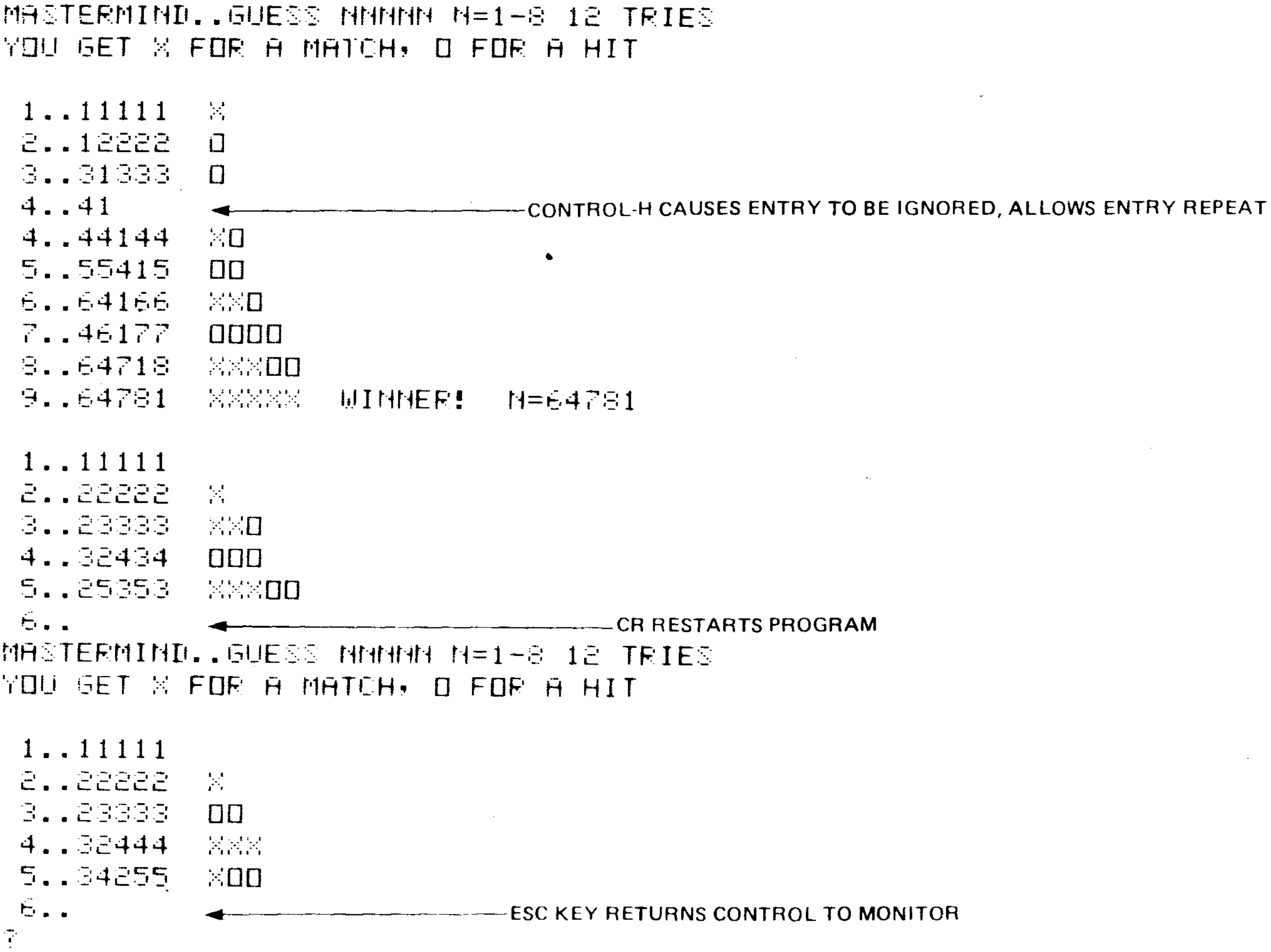
If the code is to be loaded beginning at an address other than >ED00 as a program start address, it must be refigured to the new program bias. For example, if the program was to be loaded beginning at >EC00, labelled addresses must be decreased by >100 (>ED00 - >EC00 = >100). Note that jump instructions create a displacement value and not a memory address; thus; jump instructions using labels are not affected by a new program start address.

At the back of each listing is a cross-reference of labels and number of the source statement in which they are used (column one of the listing contains source statement numbers.)

If the Line-By-Line Assembler (LBLA is used, an absolute address must be substituted for labelled addresses. These hexadecimal values are in the first column of the cross-reference table of labels.

G.2 MASTERMIND GAME

The printout generated during program execuion of the Mastermind game is presented below to illustrate how the game is played. The object of the game is to identify in proper sequence the digits making up a five digit number. Only the numbers 1 through 8 may be selected for each digit. The program returns an "O" for each digit entered that is part of the five digit number. The program returns an "X" when the required digit is placed in its proper position. The user must identify the number within 12 attempts to win the game.



```
MASTERMIND..GUESS NNNNN N=1-8 12 TRIES
YOU GET X FOR A MATCH, O FOR A HIT

 1..11111   X
 2..12222   O
 3..31333   O
 4..41      <-------------CONTROL-H CAUSES ENTRY TO BE IGNORED, ALLOWS ENTRY REPEAT
 4..44144   XO
 5..55415   OO
 6..64166   XXO
 7..46177   OOOO
 8..64718   XXXOO
 9..64781   XXXXX   WINNER!   N=64781

 1..11111
 2..22222   X
 3..23333   XXO
 4..32434   OOO
 5..25353   XXXOO
 6..        <-------------CR RESTARTS PROGRAM
MASTERMIND..GUESS NNNNN N=1-8 12 TRIES
YOU GET X FOR A MATCH, O FOR A HIT

 1..11111
 2..22222   X
 3..23333   OO
 4..32444   XXX
 5..34255   XOO
 6..        <-------------ESC KEY RETURNS CONTROL TO MONITOR
?
```

```
0000
0000
                 IDT 'MMIND'
          *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
          *  THIS PROGRAM PLAYS MASTERMIND ON THE TMS99995 MICRO-
          *  COMPUTER.  THE OBJECT OF THE GAME IS TO GUESS, BY
          *  LOGICAL DEDUCTION, A 5-DIGIT NUMBER GENERATED BY THE
          *  COMPUTER.  THE COMPUTER USES ONLY THE DIGITS 1 TO 8.  YOU
          *  HAVE 12 GUESSES TO ACCOMPLISH THIS.  THE COMPUTER WILL
          *  INDICATE A CORRECT DIGIT GUESSED BY A LETTER O AND
          *  INDICATE THE DIGIT IS CORRECTLY PLACED WITH THE
          *  5-DIGIT NUMBER WITH THE LETTER X.  OTHER RULES THAT APPLY
          *      - A CARRIAGE RETURN RESTARTS THE GAME
          *      - AN ESCAPE KEY INPUT RETURNS YOU TO THE MONITOR
          *      - CONTROL H KEY ALLOWS YOU TO SCRAP PRESENT LINE OF
          *            ENTRIES AND REENTER NEW LINE
          *  THIS GAME IS ASSEMBLED TO BE LOADED AT M.A. >ED00 BY
          *  USE BO THE AORG ASSEMBLER DIRECTIVE.  THIS PROGRAM CAN BE
          *  ASSEMBLED BY THE LBLA AT THE ADDRESSES SHOWN IN COLUMN
          *  TWO OF THE LISTING.  CORRESPONDING OBJECT CODE FOR THOSE
          *  ADDRESSES IS SHOWN IN COLUMN THREE.  GOOD LUCK!
          *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
     0000 R0     EQU  0            NO. OF GUESSES
     0001 R1     EQU  1            RANDOM NO. ARRAY ADDRESS
     0002 R2     EQU  2            RANDOM NO. COMPUTATION USE
     0003 R3     EQU  3            RANDOM NO. COMPUTATION USE
     0004 R4     EQU  4            10 CONSTANT FOR DECIMAL COMPUTATION
     0005 R5     EQU  5            CONTAINS ASII 'X'
     0006 R6     EQU  6            CONTAINS ASII 'O'
     0007 R7     EQU  7            ADDRESS OF X'S & O'S BUFFER
     0008 R8     EQU  8
     0009 R9     EQU  9            RANDOM NO. ARRAY ADDRESS
     000A R10    EQU  10           RANDOM NO. ARRAY ADDRESS+5
     000B R11    EQU  11           RANDOM NO. SEED
     000C R12    EQU  12           ASCII '1' (>3100)
     000D R13    EQU  13           CAST OUT CHARACTER MAP
ED00             AORG >ED00        LOAD AT M.A. >ED00
          *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
          *
          *  PROCEDURE AREA OF EXECUTABLE CODE
          *
          *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
ED00      START
ED00 02E0        LWPI WS           SET WORKSPACE POINTER
ED02 EDD6
ED04 2FA0        XOP  @RULES,14    PRINT RULES
ED06 EE0C
ED08      M005
ED08 2FA0        XOP  @CRLF,14     PRINT CR-LF
ED0A EE72
ED0C 04C0        CLR  R0           COUNTS 12 GUESSES
ED0E C049        MOV  R9,R1        R1 POINTS TO RANDOM ARRAY
          *  COMPUTE RANDOM NUMBER, MOVE TO LOCATION NN
ED10      M010
ED10 0202        LI   R2,509       COMPUTE RANDOM NUMBER
ED12 01FD
ED14 388B        MPY  R11,R2
ED16 0223        AI   R3,291
ED18 0123
ED1A C2C3        MOV  R3,R11
```

```
 0056                * CAUSE RANDOM DIGITS TO BE IN RANGE 1-8
 0057 ED1C 0953             SRL  R3,5
 0058 ED1E B0CC             AB   R12,R3         MAKE ASCII, RANGE 1-8
 0059 ED20 DC43             MOVB R3,*R1+        PUT IN RANDOM ARRAY
 0060 ED22 8281             C    R1,R10         TEST FOR END OF LOOP
 0061 ED24 1AF5             JL   M010           DO UNTIL R1=R10
 0062                *
 0063                * DETERMINE NUMBER OF UPCOMING GUESS
 0064                * PRINT UPCOMING GUESS NUMBER TO PROMPT USER
 0065                *
 0066 ED26           M015
 0067 ED26 0580             INC  R0             GUESS=GUESS+1
 0068                * CLEAR ARRAY THAT HOLDS ASCII X'S AND O'S
 0069                * IF CONTROL H PRESSED, START HERE
 0070 ED28 C087      RESTRT MOV  R7,R2          XOB ADDR TO R2
 0071 ED2A 04F2             CLR  *R2+           *
 0072 ED2C 04F2             CLR  *R2+           *
 0073 ED2E 04D2             CLR  *R2            *
 0074                * CONVERT GUESS NUMBER FOR OUTPUT
 0075 ED30 C080             MOV  R0,R2          GUESS NO. TO R2
 0076 ED32 04C1             CLR  R1
 0077 ED34 3C44             DIV  R4,R1          DIVIDE R1R2 BY 10
 0078 ED36 06C1             SWPB R1             QUOTIENT IN LEFT BYTE
 0079 ED38 F081             SOCB R1,R2          MERGE QUOTIENT & REMAINDER
 0080 ED3A 1302             JEQ  M020           PUT IN SPACE IF FIRST DIGIT=0
 0081 ED3C 0262             ORI  R2,>3030       MAKE ASCII DIGITS
      ED3E 3030
 0082 ED40           M020
 0083 ED40 0262             ORI  R2,>2030       MAKE ASCII SPACE & DIGIT
      ED42 2030
 0084 ED44 C802             MOV  R2,@GCD        PUT IN PRINT BUFFER
      ED46 EDF4
 0085 ED48 2FA0             XOP  @GUESNO,14     PRINT GUESS NUMBER
      ED4A EDF2
 0086                *
 0087                *
 0088                * INPUT CHARACTER & TEST FOR COLUMN MATCH
 0089                *
 0090 ED4C C209             MOV  R9,R8          RANDOM NUMBER ADDR IN R8
 0091 ED4E C047             MOV  R7,R1          X & O BUFF ADDR IN R1
 0092 ED50 0202             LI   R2,INPUT       INPUT BUFER ADDR IN R2
      ED52 EE5A
 0093 ED54 04CD             CLR  R13            CLEAR BIT MAP OF CAST OUT CHARACTER
 0094 ED56           M030
 0095 ED56 2F43             XOP  R3,13          READ DIGIT
 0096                * WAS CR, ESCHAPE, OR CONTROL-H KEY PRESSED?
 0097 ED58 0283             CI   R3,>0D00       CAR. RET. ENTERED?
      ED5A 0D00
 0098 ED5C 13D1             JEQ  START          YES, RESTART GAME
 0099 ED5E 0283             CI   R3,>1B00       ESCAPE KEY ENTERED?
      ED60 1B00
 0100 ED62 131C             JEQ  MONITR         YES, RETURN TO MONITOR
 0101 ED64 0283             CI   R3,>0800       CONTROL-H PRESSED?
      ED66 0800
 0102 ED68 13DF             JEQ  RESTRT         YES, RESTART THIS ENTRY
 0103 ED6A 9303             CB   R3,R12         IS NO. LESS THAN 1?
 0104 ED6C 1AF4             JL   M030           YES, READ ANOTHER
 0105 ED6E 0283             CI   R3,>3800       IS NO. GREATER THAN 8?
      ED70 3800
 0106 ED72 1BF1             JH   M030           YES, READ ANOTHER
```

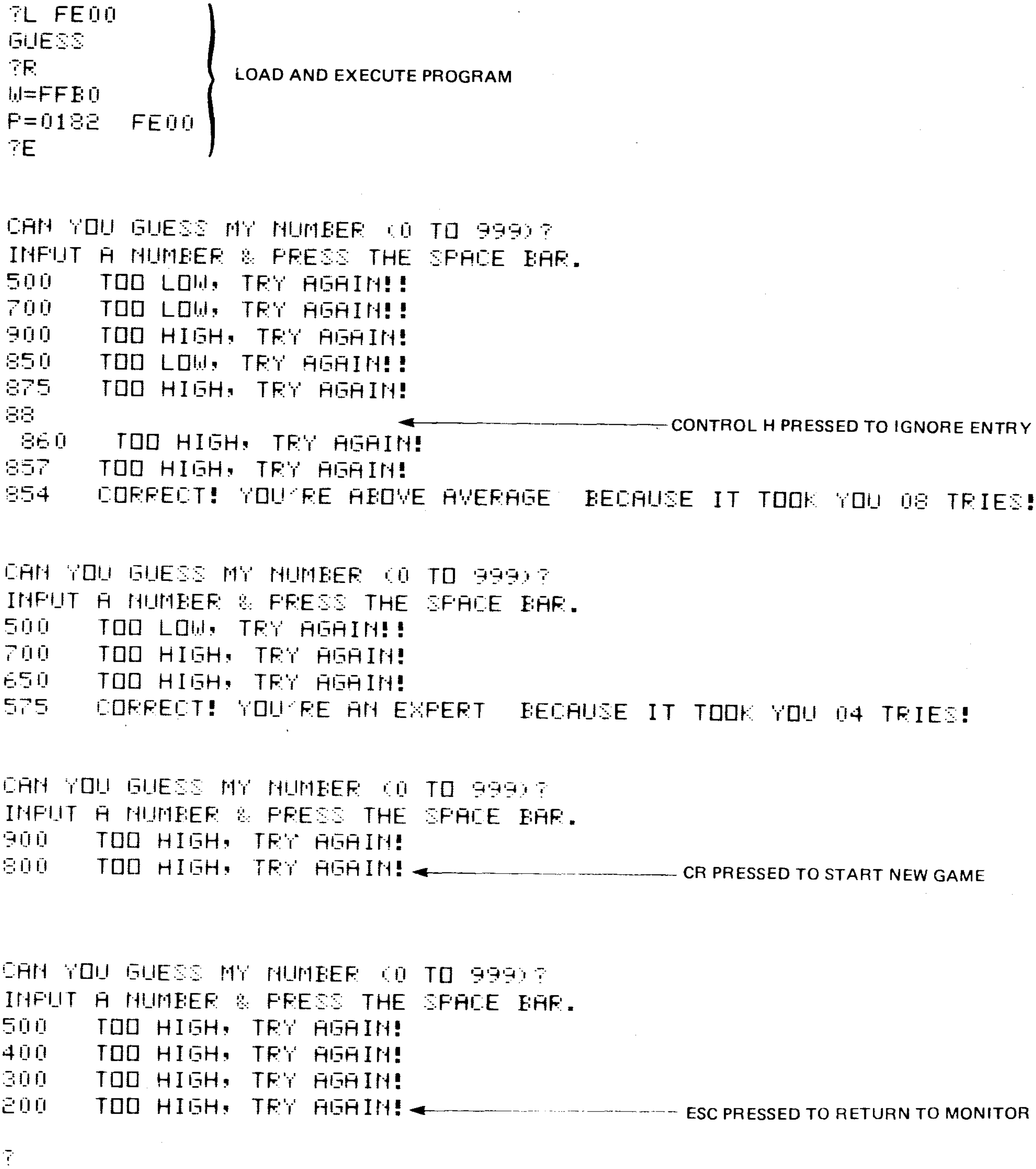
```
0107 ED74 2F03          XOP  R3,12          NO, IN RANGE, ECHO
0108               *  IS DIGIT A MATCH AND IN RIGHT COLUMN?
0109 ED76 9E03          CB   R3,*R8+        DIGIT IN RITH COLUMN
0110 ED78 1603          JNE  M040           NO, PUT CHAR IN CHAR BUFFER
0111 ED7A 06C3          SWPB R3             YES, PUT BINARY O IN MSB OF R3
0112 ED7C DC45          MOVB R5,*R1+        PUT AN X IN THE XO BUFFER
0113 ED7E 058D          INC  R13            MAP CAST OUT CHAR
0114 ED80          M040
0115 ED80 DC83          MOVB R3,*R2+        ZERO OR CHAR TO INPUT BUFFER
0116 ED82 0B1D          SRC  R13,1          PUT BIT IN MAP
0117 ED84 8288          C    R8,R10         FIFTH NUMBER INPUT?
0118 ED86 1AE7          JL   M030           NO, READ ANOTHER GUESS
0119 ED88 0281          CI   R1,XOB+5       YES, IS XO BUFFER FULL?
     ED8A EE0B
0120 ED8C 1A09          JL   M050           NO, NO WINNER YET
0121 ED8E 2FA0          XOP  @XOBP,14       YES, PRINT XO BUFF (ASS X'S)
     ED90 EE04
0122               *
0123 ED92 2FA0          XOP  @WINNER,14     PRINT WINNER
     ED94 EE60
0124               *
0125 ED96          M045
0126 ED96 2FA0          XOP  @NUMBER,14     PRINT NUMBER
     ED98 EDFA
0127               *
0128 ED9A 10B6          JMP  M005           PLAY ANOTHER GAME
0129 ED9C 0460     MONITR B    @>0080         RETURN TO MONITOR
     ED9E 0080
0130               *
0131               *
0132               *
0133               * TEST FOR O'S...
0134               *
0135 EDA0          M050
0136 EDA0 0202          LI   R2,INPUT       INPUT BUFFER START IN R2
     EDA2 EE5A
0137 EDA4          M052
0138 EDA4 D0F2          MOVB *R2+,R3        TEST BYTE FROM INPUT BUFFER
0139 EDA6 130C          JEQ  M060           BYTE CAST OUT IF EQUAL TO ZERO
0140 EDA8 C209          MOV  R9,R8          R8 POINTS TO WORK ARRAY
0141 EDAA 09BD          SRL  R13,11         POSITION CAST OUT CH MAP
0142 EDAC          M055
0143 EDAC 0B1D          SRC  R13,1          TEST FOR CAST OUT CHAR
0144 EDAE 9E03          CB   R3,*R8+        DOES BYTE MATCH WORK ARRAY ?
0145 EDB0 1805          JOC  M057           IF CAST OUT, M057
0146 EDB2 1604          JNE  M057           IF NOT EQUAL, M057
0147 EDB4 DC46          MOVB R6,*R1+        ON HIT, PUT O TIN XO BUFFER
0148 EDB6 026D          ORI  R13,>8000      MAP CAST OUT CHAR
     EDB8 8000
0149 EDBA B0C3          AB   R3,R3          SPOIL COMPARISON, FINISH LOOP
0150 EDBC          M057
0151 EDBC 8288          C    R8,R10         TEST FOR LAST DIGIT
0152 EDBE 1AF6          JL   M055           IF LOW, DO ANOTHER DIGIT
0153 EDC0          M060
0154 EDC0 0282          CI   R2,INPUT+5     LAST DIGIT IN INPUT BUFFER?
     EDC2 EE5F
0155 EDC4 1AEF          JL   M052           NO, DO NEXT DIGIT
0156 EDC6 2FA0          XOP  @XOBP,14       YES, PRINT XO BUFF
     EDC8 EE04
0157 EDCA 0280          CI   R0,12          TWELVE GUESSES MADE?
```

```
     EDCC 000C
0158 EDCE 1AAB        JL   M015          NO, MORE GUESSES REMAIN
0159 EDD0 2FA0        XOP  @SORRY,14     YES, PRINT SORRY
     EDD2 EE6A
0160 EDD4 10E0        JMP  M045          PRINT NUMBER FOR PLAYER
0161           *   *   *   *   *   *   *   *   *   *   *   *   *   *   *
0162           *
0163           *  DATA SECTION
0164           *
0165           *   *   *   *   *   *   *   *   *   *   *   *   *   *   *
0166           *  WORKSPACE
0167 EDD6 0000 WS     DATA 0,0,0,0      R0-R3
     EDD8 0000
     EDDA 0000
     EDDC 0000
0168 EDDE 000A        DATA 10           R4  CONVERSION CONSTANT
0169 EDE0   58        TEXT 'X '         R5
'    EDE1   20
0170 EDE2   4F        TEXT 'O '         R6
     EDE3   20
0171 EDE4 EE06        DATA XOB          R7
0172 EDE6 0000        DATA 0            R8
0173 EDE8 EDFE        DATA NN           R9
0174 EDEA EE03        DATA NN+5         R10
0175 EDEC 5555        DATA >5555        R11-RANDOM NUMBER SEED
0176 EDEE 3100        DATA >3100        R12
0177 EDF0 0000        DATA 0            R13-CAST OUT CHAR MAP
0178           *
0179           *  TEXT STATEMENTS
0180           *
0181           *  LINE NUMBER OF THIS GUESS
0182 EDF2 0D0A GUESNO DATA >0D0A        CR, LINE FEED
0183 EDF4 0000 GCD    DATA $-$          CONVERTED GUESS NUMBER
0184 EDF6   2E        TEXT '..'
     EDF7   2E
0185 EDF8   07        BYTE 7,0          BELL/STOP
     EDF9   00
0186           *  RANDOM NUMBER OF COMPUTER IN ASCII
0187 EDFA   20 NUMBER TEXT '  N='
     EDFB   20
     EDFC   4E
     EDFD   3D
0188 EDFE 0000 NN     DATA 0,0,0
     EE00 0000
     EE02 0000
0189           *  X'S AND O'S BUFFER SHOWING HITS & MISSES
0190 EE04   20 XOBP   TEXT '  '         SPACES FOR PRINTING
     EE05   20
0191 EE06 0000 XOB    DATA 0,0,0
     EE08 0000
     EE0A 0000
0192           *  RULES OUTPUT AT BEGINNING OF GAME
0193 EE0C      RULES
0194 EE0C 0D0A        DATA >0D0A
0195 EE0E   4D        TEXT 'MASTERMIND'
     EE0F   41
     EE10   53
     EE11   54
     EE12   45
     EE13   52
```

```
      EE14   4D
      EE15   49
      EE16   4E
      EE17   44
 0196 EE18   2E           TEXT '..GUESS NNNNN N=1-8 12 TRIES'
      EE19   2E
      EE1A   47
      EE1B   55
      EE1C   45
      EE1D   53
      EE1E   53
      EE1F   20
      EE20   4E
      EE21   4E
      EE22   4E
      EE23   4E
      EE24   4E
      EE25   20
      EE26   4E
      EE27   3D
      EE28   31
      EE29   2D
      EE2A   38
      EE2B   20
      EE2C   31
      EE2D   32
      EE2E   20
      EE2F   54
      EE30   52
      EE31   49
      EE32   45
      EE33   53
 0197 EE34 0D0A           DATA >0D0A
 0198 EE36   59           TEXT 'YOU GET X FOR A MATCH, O FOR A HIT'
      EE37   4F
      EE38   55
      EE39   20
      EE3A   47
      EE3B   45
      EE3C   54
      EE3D   20
      EE3E   58
      EE3F   20
      EE40   46
      EE41   4F
      EE42   52
      EE43   20
      EE44   41
      EE45   20
      EE46   4D
      EE47   41
      EE48   54
      EE49   43
      EE4A   48
      EE4B   2C
      EE4C   20
      EE4D   4F
      EE4E   20
      EE4F   46
      EE50   4F
      EE51   52
      EE52   20
      EE53   41
      EE54   20
      EE55   48
      EE56   49
      EE57   54
 0199 EE58   00            BYTE 0
 0200               *  BUFFER OF NUMBERS INPUT
 0201 EE5A 0000     INPUT  DATA 0,0,0
      EE5C 0000
      EE5E 0000
 0202               *
 0203 EE60   20     WINNER TEXT '  WINNER'
      EE61   20
      EE62   57
      EE63   49
      EE64   4E
      EE65   4E
      EE66   45
      EE67   52
 0204 EE68   21            BYTE >21,0
      EE69   00
 0205 EE6A   20     SORRY  TEXT ' SORRY'
      EE6B   53
      EE6C   4F
      EE6D   52
      EE6E   52
      EE6F   59
 0206 EE70   00            BYTE 0,0
      EE71   00
 0207 EE72   0D     CRLF   BYTE >D,>A,0,0
      EE73   0A
      EE74   00
      EE75   00
 0208               *
 0209      ED00            END  START
 0 ERRORS,     NO WARNINGS
```

G.3 HIGH-LO GAME

the printout of this game in execution (below) illustrates game rules and objectives. The program generates a number between 0 and 999. You have unlimited guesses to find the number, but you can be an expert, above average, average, or a turkey, depending upon how many guesses are needed to solve the problem.



```
?L FE00
GUESS
?R                  LOAD AND EXECUTE PROGRAM
W=FFB0
P=0182  FE00
?E


CAN YOU GUESS MY NUMBER (0 TO 999)?
INPUT A NUMBER & PRESS THE SPACE BAR.
500   TOO LOW, TRY AGAIN!!
700   TOO LOW, TRY AGAIN!!
900   TOO HIGH, TRY AGAIN!
850   TOO LOW, TRY AGAIN!!
875   TOO HIGH, TRY AGAIN!
88                                  <---------- CONTROL H PRESSED TO IGNORE ENTRY
 860   TOO HIGH, TRY AGAIN!
857   TOO HIGH, TRY AGAIN!
854   CORRECT! YOU'RE ABOVE AVERAGE  BECAUSE IT TOOK YOU 08 TRIES!

CAN YOU GUESS MY NUMBER (0 TO 999)?
INPUT A NUMBER & PRESS THE SPACE BAR.
500   TOO LOW, TRY AGAIN!!
700   TOO HIGH, TRY AGAIN!
650   TOO HIGH, TRY AGAIN!
575   CORRECT! YOU'RE AN EXPERT  BECAUSE IT TOOK YOU 04 TRIES!

CAN YOU GUESS MY NUMBER (0 TO 999)?
INPUT A NUMBER & PRESS THE SPACE BAR.
900   TOO HIGH, TRY AGAIN!
800   TOO HIGH, TRY AGAIN! <---------- CR PRESSED TO START NEW GAME


CAN YOU GUESS MY NUMBER (0 TO 999)?
INPUT A NUMBER & PRESS THE SPACE BAR.
500   TOO HIGH, TRY AGAIN!
400   TOO HIGH, TRY AGAIN!
300   TOO HIGH, TRY AGAIN!
200   TOO HIGH, TRY AGAIN! <---------- ESC PRESSED TO RETURN TO MONITOR

?
```

```
0001              *  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *
0002              *  THIS GUESSING GAME CAN BE RUN ON A TM 990/1XX MICRO-
0003              *  COMPUTER WITH 432 (>1B0) WORDS OF USER AVAILABLE
0004              *  RAM MEMORY.  IT IS WRITTEN TO BE LOADED AT M.A. >ED00
0005              *  AND CAN BE ASSEMBLED AT THAT ADDRESS USING THE LBLA
0006              *  OR BY LOADING THE OBJECT (COLUMN 3) AT THE MEMORY
0007              *  ADDRESSES (COLUMN 2).  THE OBJECT OF THIS PROGRAM IS TO
0008              *  GUESS WHICH NUMBER THE COMPUTER HAS GENERATED, AND TO
0009              *  DO THIS WITHOUT BECOMING A TURKEY.  FOLLOWING RULES APPLY
0010              *      - CARRIAGE RETURN BRINGS YOU TO MONITOR
0011              *      - ESCAPE KEY BRINGS YOU TO MONITOR
0012              *      - CONTROL-H KEY IGNORES THIS ENTRY
0013              *      - SPACE KEY CONTINUES GAME
0014              *      GOOD LUCK.
0015              *  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *
0016                     IDT  'GUESS'
0017              *  REGISTER EQUATES
0018        0000  R0     EQU  0            TENS MULTIPLIER
0019        0001  R1     EQU  1            GUESS NO. ACCUMULATOR
0020        0002  R2     EQU  2            MULTIPLY ANSWER
0021        0003  R3     EQU  3            ENTERED DIGIT
0022        0008  R8     EQU  8            CONTAINS COMPUTER'S NUMBER
0023        0009  R9     EQU  9            NO. TRIES/10
0024        000A  R10    EQU  10           NO. TRIES
0025        000C  R12    EQU  12           CRU ADDRESS (TMS9902)
0026              *  OBJECT CODE AT ABSOLUTE ADDRESS BEGINNING WITH >ED00
0027 ED00                AORG >ED00
0028              *  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *
0029              *  PROCEDURE AREA:  EXECUTABLE CODE
0030              *  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *
0031              *  INITIALIZE REGISTERS
0032 ED00 02E0  START  LWPI WSP          SET WORKSPACE POINTER
     ED02 EEA4
0033 ED04 0200         LI   R0,10        R0 = TENS MULTIPLIER
     ED06 000A
0034 ED08 04C9         CLR  R9           R9 = NO. OF TRIES
0035 ED0A 04CA         CLR  R10          R10 = NO. TO TRIES
0036 ED0C 020C         LI   R12,>0       TMS9902 CRU ADDR.
     ED0E 0000
0037              *  OUTPUT OPENING MESSAGE
0038 ED10 2FA0         XOP  @MESS1,14    OPENING MESSAGE
     ED12 EDB0
0039              *  THIS ROUTINE IS A NUMBER GENERATOR THAT GENERATES
0040              *  A NUMBER FORM 0 TO 999 BASED ON THE TIME TO RESPOND TO TH
0041              *  OPENING MESSAGE.  IT CHECKS A BIT AT THE TMS 9902 SERIAL
0042              *  INTERFACE THAT SIGNIFIES THAT A DIGIT HAS BEEN RECEIVED F
0043              *  THE TERMINAL IN RESPONSE TO THE OPENING MESSAGE.  RECEIPT
0044              *  THIS DIGIT MEANS A NUMBER IS BEING GUESSED.  WHILE WAITIN
0045              *  FOR THIS FIRST NUMBER, R8 IS CONTINUOUSLY INCREMENTED FRO
0046              *  0 TO 999.
0047 ED14 04C8  NEWNO  CLR  R8           R8 TO CONTAIN COMJPUTER'S NO.
0048 ED16 1F15  INCNO  TB   21           DIGIT RECEIVED?
0049 ED18 1307         JEQ  ECHO2        YES, ECHO CHARACTER
0050 ED1A 0288         CI   R8,999       NO. INCREMENTED TO 999?
     ED1C 03E7
0051 ED1E 13FA         JEQ  NEWNO        JES, CLEAR TO 0, RESTART
0052 ED20 0588         INC  R8           NO, INCREMENT NO. IN R8
0053 ED22 10F9         JMP  INCNO        LOOP, RECHECK FOR DIGIT INPUT
0054              *  AFTER FIRST DIGIT IS ENTERED.  COMJPUTER'S NO. IS IN R8.
0055              *  READ IN GUESSES AND CONVERT THESE TO THEXADECIMAL. SUM
```

```
GUESS      SDSMAC 3.2.0 78.274   20:22:07 TUESDAY, MAR 17, 1981.
                                                          PAGE 0003

 0056                * FOR COMPARISON TO COMPUTER'S NO. IN R8.  AS NEW NUMBER
 0057                * IS READ, OLD VALUE IS MULTIPLIED BY 10 AND NEW VALUE
 0058                * ADDED TO PRODUCT TO KEEP COMULATIVE TOTAL OF DIGITS
 0059                * ENTERED.
 0060 ED24 2F20 ECHO0  XOP  @LFCR,12      DO LINE-FEED, CR
      ED26 EE38
 0061 ED28 04C1 ECHO2  CLR  R1            CLEAR ACCUMMULATOR
 0062 ED2A 2EC3 ECHO1  XOP  R3,11         ECHO CHAR., PLACE IT IN R3
 0063 ED2C 06C3        SWPB R3            PLACE VALUE IN RIGHT BYTE
 0064                * WAS SPACE, CR, ESCAPE OR CONTROL-H PRESSED?
 0065 ED2E 0283        CI   R3,>0020      SPACE BAR PRESSED?
      ED30 0020
 0066 ED32 1311        JEQ  COMPRE        YES, COMPARE VALUES
 0067 ED34 0283        CI   R3,>000D      CARRIAGE RET. PRESSED?
      ED36 000D
 0068 ED38 13E3        JEQ  START         YES, RESTART PROGRAM
 0069 ED3A 0283        CI   R3,>001B      ESCAPE PRESSED?
      ED3C 001B
 0070 ED3E 1309        JEQ  MONITR        YES, RETURN TO MONITOR
 0071 ED40 0283        CI   R3,>0008      WAS CONTROL-H PRESSED?
      ED42 0008
 0072 ED44 13EF        JEQ  ECHO0         DO LFCR, RESTART GUESS
 0073 ED46 0243        ANDI R3,>000F      NO, SAVE 0-9 DIGIT ONLY
      ED48 000F
 0074 ED4A 3840        MPY  R0,R1         PREVIOUS NO. X10
 0075 ED4C A0C2        A    R2,R3         NEW NO. + ABOVE PRODUCT
 0076 ED4E C043        MOV  R3,R1         ANSWR TO ACCUMMULATOR
 0077 ED50 10EC        JMP  ECHO1         GET NEXT DIGIT
 0078 ED52 0460 MONITR B    @>0080        GO TO MONITOR
      ED54 0080
 0079                * COMPARE NUMBERS INPUT TO COMPUTER'S NUMBER
 0080 ED56 058A COMPRE INC  R10           INCREMENT NOS. GUESSED
 0081 ED58 8201        C    R1,R8         COMPARE TO COMPUTER'S NO.
 0082 ED5A 1102        JLT  LOW           NO. IS LESS THAT COMPUTER'S
 0083 ED5C 1504        JGT  HIGH          NO. IS MORE THAT COMPUTER'S
 0084 ED5E 1306        JEQ  EQUAL         NO. IS CORRECT VALUE
 0085                * MESAGES FOR TOO HIGH, TOO LOW
 0086 ED60 2FA0 LOW    XOP  @LOWM,14      TOO-LOW MESSAGE
      ED62 EE04
 0087 ED64 10E1        JMP  ECHO2         GET NEXT NUMBER
 0088 ED66 2FA0 HIGH   XOP  @HIGHM,14     TOO-HIGH MESSAGE
      ED68 EE1E
 0089 ED6A 10DE        JMP  ECHO2         GET NEXT NUMBER
 0090                * CORRECT NUMBER WAS GUESSED
 0091                * FIND OUT HOW MANY TRIES WAS USED AND OUTPUT MESSAGE
 0092 ED6C 2FA0 EQUAL  XOP  @CORECT,14    CORRECT GUESS MESSAGE
      ED6E EE3C
 0093 ED70 028A        CI   R10,7         TRY COUNT GREATER THAN 7?
      ED72 0007
 0094 ED74 1503        JGT  $+8           YES, CHECK AGAIN
 0095 ED76 2FA0        XOP  @SEVEN,14     NO, DO 0-7 TRIES MESSAGE
      ED78 EE55
 0096 ED7A 100E        JMP  COUNT         GO GET COUNT
 0097 ED7C 028A        CI   R10,9         TRY-COUNT GREATER THAN 9?
      ED7E 0009
 0098 ED80 1503        JGT  $+8           YES, CHECK AGAIN
 0099 ED82 2FA0        XOP  @NINE,14      NO, DO 8-9 TRIES MESSAGE
      ED84 EE5F
 0100 ED86 1008        JMP  COUNT         GO GET COUNT
 0101 ED88 028A        CI   R10,13        TRY-COUNTER GREATER THAT 13?
```

```
     ED8A 000D
0102 ED8C 1503         JGT  $+8           YES, OUTPUT TURKEY MESSAGE
0103 ED8E 2FA0         XOP  @THIRTN,14    NO, DO 10-13 TRIES MESSAGE
     ED90 EE6E
0104 ED92 1002         JMP  COUNT         GO GET COUNT
0105 ED94 2FA0         XOP  @TURKEY,14    OUTPUT >13 (TURKEY) MESSAGE
     ED96 EE77
0106                * IF CORRECT NUMBER FOUND, OUTPUT NO. OF TRIES
0107 ED98 3E40  COUNT  DIV  R0,R9         DIVIDE TRY-NO. BY 10
0108 ED9A 0269         ORI  R9,>0030      OR IN >30 FOR ASCII NO.
     ED9C 0030
0109 ED9E 026A         ORI  R10,>0030     OR IN >30 FOR ASCII NO.
     EDA0 0030
0110 EDA2 06C9         SWPB R9            REMAINDER IN LEFT BYTE
0111 EDA4 A289         A    R9,R10        2-DIGIT DECIMAL IN R10
0112 EDA6 C80A         MOV  R10,@NUMBER   MOVE QTY TO MESSAGE
     EDA8 EE96
0113 EDAA 2FA0         XOP  @CNT,14       OUTPUT NO. OF TRIES
     EDAC EE81
0114 EDAE 10A8         JMP START          GO TO BEGINNING OF PROGRAM
0115                *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
0116                * DATA AREA: DATA STATEMENTS, TEXT STATEMENTS, ETC.
0117                *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
0118                * MESSAGES
0119 EDB0 0A0D  MESS1  DATA >0A0D,>0A0A
     EDB2 0A0A
0120 EDB4   43         TEXT 'CAN YOU GUESS MY NUMBER (0 TO 999)? '
     EDB5   41
     EDB6   4E
     EDB7   20
     EDB8   59
     EDB9   4F
     EDBA   55
     EDBB   20
     EDBC   47
     EDBD   55
     EDBE   45
     EDBF   53
     EDC0   53
     EDC1   20
     EDC2   4D
     EDC3   59
     EDC4   20
     EDC5   4E
     EDC6   55
     EDC7   4D
     EDC8   42
     EDC9   45
     EDCA   52
     EDCB   20
     EDCC   28
     EDCD   30
     EDCE   20
     EDCF   54
     EDD0   4F
     EDD1   20
     EDD2   39
     EDD3   39
     EDD4   39
     EDD5   29
     EDD6   3F
     EDD7   20
0121 EDD8 0A0D          DATA >0A0D         LINE FEED, CR
0122 EDDA   49          TEXT 'INPUT A NUMBER & PRESS THE SPACE BAR. '
     EDDB   4E
     EDDC   50
     EDDD   55
     EDDE   54
     EDDF   20
     EDE0   41
     EDE1   20
     EDE2   4E
     EDE3   55
     EDE4   4D
     EDE5   42
     EDE6   45
     EDE7   52
     EDE8   20
     EDE9   26
     EDEA   20
     EDEB   50
     EDEC   52
     EDED   45
     EDEE   53
     EDEF   53
     EDF0   20
     EDF1   54
     EDF2   48
     EDF3   45
     EDF4   20
     EDF5   53
     EDF6   50
     EDF7   41
     EDF8   43
     EDF9   45
     EDFA   20
     EDFB   42
     EDFC   41
     EDFD   52
     EDFE   2E
     EDFF   20
0123 EE00 0A0D          DATA >0A0D,0       LINE FEED, CR, END MSG
     EE02 0000
0124 EE04 2020  LOWM    DATA >2020         DOUBLE SPACE
0125 EE06   54          TEXT 'TOO LOW, TRY AGAIN!!'
     EE07   4F
     EE08   4F
     EE09   20
     EE0A   4C
     EE0B   4F
     EE0C   57
     EE0D   2C
     EE0E   20
     EE0F   54
     EE10   52
     EE11   59
     EE12   20
     EE13   41
     EE14   47
     EE15   41
```

```
      EE16   49
      EE17   4E
      EE18   21
      EE19   21
 0126 EE1A 0A0D         DATA >0A0D,0      LINE FEED, CR, END MSG
      EE1C 0000
 0127 EE1E 2020  HIGHM  DATA >2020        TWO SPACES
 0128 EE20   54         TEXT 'TOO HIGH, TRY AGAIN!'
      EE21   4F
      EE22   4F
      EE23   20
      EE24   48
      EE25   49
      EE26   47
      EE27   48
      EE28   2C
      EE29   20
      EE2A   54
      EE2B   52
      EE2C   59
      EE2D   20
      EE2E   41
      EE2F   47
      EE30   41
      EE31   49
      EE32   4E
      EE33   21
 0129 EE34 0A0D         DATA >0A0D,0      LINE FEED, CR, END MSG
      EE36 0000
 0130 EE38 0A0D  LFCR   DATA >0A0D        LINE FEED, CR
 0131 EE3A   00         BYTE 0            END OF MESSAGE
 0132 EE3C 0707  CORECT DATA >0707,>0707  BELLS
      EE3E 0707
 0133 EE40 2020         DATA >2020        SPACES
 0134 EE42   43         TEXT 'CORRECT ! YOUR''RE '
      EE43   4F
      EE44   52
      EE45   52
      EE46   45
      EE47   43
      EE48   54
      EE49   20
      EE4A   21
      EE4B   20
      EE4C   59
      EE4D   4F
      EE4E   55
      EE4F   52
      EE50   27
      EE51   52
      EE52   45
      EE53   20
 0135 EE54   00         BYTE 0            END OF MESSAGE
 0136 EE55   41  SEVEN  TEXT 'AN EXPERT'
      EE56   4E
      EE57   20
      EE58   45
      EE59   58
      EE5A   50
      EE5B   45
```

```
     EE5C   52
     EE5D   54
0137 EE5E   00              BYTE 0
0138 EE5F   41   NINE       TEXT 'ABOVE AVERAGE '
     EE60   42
     EE61   4F
     EE62   56
     EE63   45
     EE64   20
     EE65   41
     EE66   56
     EE67   45
     EE68   52
     EE69   41
     EE6A   47
     EE6B   45
     EE6C   20
0139 EE6D   00              BYTE 0
0140 EE6E   41   THIRTN TEXT 'AVERAGE '
     EE6F   56
     EE70   45
     EE71   52
     EE72   41
     EE73   47
     EE74   45
     EE75   20
0141 EE76   00              BYTE 0
0142 EE77   41   TURKEY TEXT 'A TURKEY '
     EE78   20
     EE79   54
     EE7A   55
     EE7B   52
     EE7C   4B
     EE7D   45
     EE7E   59
     EE7F   20
0143 EE80   00              BYTE 0
0144 EE81   20   CNT        TEXT ' BECAUSE IT TOOK YOU
     EE82   42
     EE83   45
     EE84   43
     EE85   41
     EE86   55
     EE87   53
     EE88   45
     EE89   20
     EE8A   49
     EE8B   54
     EE8C   20
     EE8D   54
     EE8E   4F
     EE8F   4F
     EE90   4B
     EE91   20
     EE92   59
     EE93   4F
     EE94   55
     EE95   20
0145 EE96 0000   NUMBER DATA 0
0146 EE98   20              BYTE $20          PLACE ASCII NO. HERE
```

```
 0147 EE99    54          TEXT 'TRIES!'
      EE9A    52
      EE9B    49
      EE9C    45
      EE9D    53
      EE9E    21
 0148 EE9F    07          BYTE 7,7,7,0       BELLS (ASCII 07)
      EEA0    07
      EEA1    07
      EEA2    00
 0149 EEA4         WSP    EVEN               WORKSPACE START (R0 LOC)
 0150                     END
NO ERRORS,     NO WARNINGS
```

INDEX