TEXAS INSTRUMENTS

Improving Man's Effectiveness Through Electronics

Model 990 Computer
TMS 9900 Microprocessor
Assembly Language Programmer's Guide

MANUAL NO. 943441-9701
ORIGINAL ISSUE 1 JUNE 1974
REVISED 15 OCTOBER 1978

Digital Systems Division

LIST OF EFFECTIVE PAGES

INSERT LATEST CHANGED PAGES DESTROY SUPERSEDED PAGES

Note: The portion of the text affected by the changes is indicated by a vertical bar in the outer margins of the page.

Model 990 Computer TMS9900 Microprocessor Assembly Language Programmer's Guide (943441-9701)

Original Issue . . . . . 1 June 1974
Revised . . . . . .15 October 1978 (ECN 446281)

Total number of pages in this publication is 366 consisting of the following:

PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.
Cover	0	Appendix B Div	0	Appendix I Div	0
Effective Pages	0	B-1 - B-14	0	I-1 - I-2	0
iii - xii	0	Appendix C Div	0	Appendix J Div	0
1-1 - 1-2	0	C-1 - C-4	0	J-1 - J-8	0
2-1 - 2-18	0	Appendix D Div	0	Appendix K Div	0
3-1 - 3-120	0	D-1 - D-4	0	K-1 - K-16	0
4-1 - 4-20	0	Appendix E Div	0	Appendix L Div	0
5-1 - 5-2	0	E-1 - E-4	0	L-1 - L-8	0
6-1 - 6-14	0	Appendix F Div	0	Alphabetical Index Div	0
7-1 - 7-30	0	F-1 - F-4	0	Index-1 - Index-8	0
8-1 - 8-4	0	Appendix G Div	0	User's Response	0
9-1 - 9-8	0	G-1 - G-2	0	Business Reply	0
10-1 - 10-24	0	Appendix H Div	0	Cover Blank	0
Appendix A Div	0	H-1 - H-6	0	Cover	0
A-1 - A-4	0				

PREFACE

This manual describes the assembly language for the Model 990 Computer and the TMS 9900 microprocessor as implemented by PX9ASM, a one-pass assembler that executes under the Prototyping System PX990; by TXMIRA, a two-pass assembler that executes under TX990; by SDSMAC, a two-pass assembler that executes under Disc Executive DX10; and by the Cross Assembler, a two-pass assembler that is part of the Cross Support System. Except for a few differences that are expressed in Appendix L, the TMS 9940 microcomputer uses the same assembly language as the TMS 9900 microprocessor. However, the assembly language for the TMS 9940 microcomputer can be implemented only by the TXMIRA and the SDSMAC assemblers.

This manual describes:

- Source statement formats and elements
- Addressing modes
- Assembler directives and pseudo-instructions
- Assembly instructions
- Macro language, supported by SDSMAC
- Assembler output

Appendixes contain:

- The character set
- Instruction tables
- Directive tables
- A macro language summary
- CRU and TILINE examples
- TMS 9940 programming considerations.

This manual assumes that the reader is familiar with the computer architecture and I/O capabilities as described in the *990 Computer Family Systems Handbook.*

The following documents contain additional information related to the assembly language:

Title	Part Number
990 Computer Family Systems Handbook	*945250-9701*
Model 990 Computer Prototyping System Operation Guide	*945255-9701*

Title	Part Number
Model 990 Computer DX10 Operating System Documentation, Volume 4 – Development Operation	946250-9704
Model 990 Computer DX10 Operating System Documentation, Volume 3 – Application Programming Guide	946250-9703
Model 990 Computer TX990 Operating System Programmer's Guide (Release 2)	946259-9701
Model 990 Computer Cross Support System User's Guide	945252-9701
Model 990 Computer TMS 9900 Microprocessor Cross Support System Installation and Operation	945420-9701
Model 990 Computer Terminal Executive Development System (TXDS) Programmer's Guide	946258-9701
DX10 Operating System Production Operation Guide	946250-9702
TMS 9940 16-Bit Microcomputer Data Manual	*

* Available from:
Texas Instruments Incorporated
Microprocessor Marketing
Mail Station 653
P. O. Box 1443
Houston, Texas 77001

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

APPENDIXES

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

LIST OF TABLES

SECTION I

INTRODUCTION

1.1 ASSEMBLY LANGUAGE DEFINITION

An assembly language is a computer-oriented language for writing programs. It consists of mnemonic instructions and assembler directives. In assembly instructions, the user assigns symbolic addresses to memory locations and specifies instructions by means of symbolic operation codes called mnemonic operation codes. The user specifies instruction operands by means of symbolic addresses, numbers, and expressions consisting of symbolic addresses and numbers. Assembler directives control the process of making a machine language program from the assembly language program, place data in the program, and assign symbols to values to be used in the program. Assembler directives that place data in memory locations allow the user to assign symbolic addresses to those locations.

An assembly language is computer-oriented in that the mnemonic operation codes correspond directly with machine instructions. The chief advantage an assembly language offers over machine language is that the symbols of assembly language are easier to use and easier to remember than the zeros and ones of machine language. Other advantages are the use of expressions as operands and the use of decimal numbers in expressions and as operands.

1.2 ASSEMBLY LANGUAGE APPLICATION

An assembly language program, called a source program, must be processed by an assembler to obtain a machine language program that can be executed by the computer. Processing of a source program is called assembling, because it consists of assembling the binary values that correspond to the mnemonic operation code with the binary address information to form the machine language instruction.

To illustrate the place of assembly language in the development of programs, consider the following steps in program development:

1. Define the problem.

2. Flowchart the solution to the problem.

3. Code the solution by writing assembly language statements (machine instructions and assembler directives) that correspond to the steps of the flowchart.

4. Prepare the source program by writing the statements on the medium appropriate to the installation; i.e., keypunch the statements if a card reader is to be used as input to the assembler, etc.

5. Execute the assembler to assemble the machine language object code corresponding to the source program.

6. Debug the resulting object code by loading and executing the object code and by making corrections indicated by the results of executing the object code.

7. Repeat steps 5 and 6 until no further correction is required.

The use of assembly language in program development relieves the programmer of the tedious task of writing machine language instructions and keeping track of binary machine addresses within the program.

SECTION II

GENERAL PROGRAMMING INFORMATION

2.1 BYTE ORGANIZATION

Memory for the Model 990 Computer/TMS 9900 Microprocessor is addressed using byte addresses. A byte consists of eight bits of memory, as shown in figure 2-1. The bits may represent the states of eight independent two-valued quantities, or the configuration of a character in a code used for input, output, or data transmission. The bits also may represent a number which is interpreted either as a signed number in the range of -128 through +127 or as an unsigned number in the range of 0 through 255. The 990 computers and TMS 9900 microprocessor implements signed numbers in 2's complement form.

The most significant bit (MSB) is designated bit 0, and the least significant bit (LSB) is designated bit 7. A byte instruction may address any byte in memory.

2.2 WORD ORGANIZATION

A word in the memory for the Model 990 Computer/TMS 9900 Microprocessor consists of 16 bits, a byte at an even address and the following byte at an odd address. As shown in figure 2-2, the most significant bit of a memory word is designated bit 0, and the least significant bit is designated bit 15. A word may contain a computer instruction in machine language, a memory address, the bit configurations of two characters, or a number. When a word contains a number, the number may be interpreted as a signed number in the range of -32,768 through +32,767, or as an unsigned number in the range of 0 through 65,535. (Signed numbers are implemented in 2's complement form.)

Word boundaries are assigned to even-numbered addresses in memory. The even address byte contains bits 0 through 7 of the word, and the odd address byte contains bits 8 through 15. When word instructions address an odd byte, the word operand is the memory word consisting of the addressed byte and the preceding even-numbered byte. This is the memory word that would be accessed by the odd address minus one. For example, a memory address of 1023_{16} used as a word address would access the same word as memory address 1022_{16}.

NOTE

All instructions must begin on word boundaries. Instructions are 1, 2, or 3 words long.

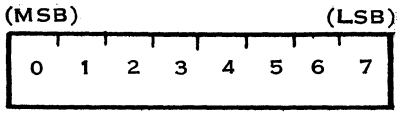


Figure 2-1. Memory Byte

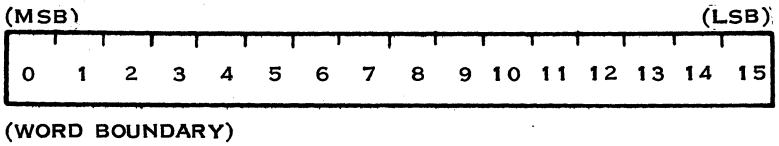


Figure 2-2. Memory Word

2.3 TRANSFER VECTORS

A transfer vector is a pair of memory addresses in two consecutive words of memory. The first word contains the address of a 16-word area of memory, called a workspace. The second word contains the address of a subroutine entry point. The Model 990 Computer/TMS 9900 Microprocessor uses a transfer vector in a type of transfer of control called a context switch. A context switch places the contents of the first word of a transfer vector in the Workspace Pointer (WP) register, making the workspace addressed by that word the active workspace. The 16 words of the active workspace become workspace registers 0 through 15, which are available for use as general purpose registers, address registers, or index registers. A context switch places the contents of the second word of a transfer vector in the Program Counter (PC), causing the instruction at that address to be executed next.

A context switch transfers control to an interrupt subroutine whenever an interrupt occurs. The transfer vectors for interrupt levels 0 through 15 are located in memory locations 0000_{16} through $003E_{16}$, as shown in figure 2-3. The address of the first byte of the vector for an interrupt level is the product of the level number times four.

The Model 990 Computer/TMS 9900 Microprocessor supports extended operations implemented by subroutines. These extended operations are effectively additional instructions that may perform user-defined functions. Up to 16 extended operations may be implemented. An extended operation machine instruction results in a context switch to the specified extended operation subroutine. The transfer vectors for extended operations 0 through 15 are located in memory locations 0040_{16} through $007E_{16}$ as shown in figure 2-3. The address of the first byte of the vector for an extended operation is the hexadecimal sum of the product of the extended operation number times four, plus 40_{16}.

In the Model 990/10 Computer, an extended operation may be implemented with user-supplied hardware. When a hardware module is connected for an extended operation, no context switch occurs for that operation, and the hardware performs the operation. Program execution continues when the operation has completed.

A context switch using the transfer vector at memory location $FFFC_{16}$ transfers control to a subroutine to load or restart the computer. Execution of an LREX instruction or activation of a switch on the control panel initiates the context switch.

A context switch to a user subroutine is performed by the BLWP instruction. The transfer vector is placed at a user defined location in memory.

2.4 STATUS REGISTER

The configuration of the Status Register of the Model 990 Computer and the TMS 9900 Microprocessor is shown in figure 2-4. The configuration of the Status Register of the Model 990/10 Computer with map option is shown in figure 2-5. Bits 0 through 6 and 12 through 15 are identical, and are the bits that are set and reset by the machine instructions. These bits have the following meanings:

- L>, bit 0 - Logical greater than
- A>, bit 1 - Arithmetic greater than
- EQ, bit 2 - Equal
- C, bit 3 - Carry

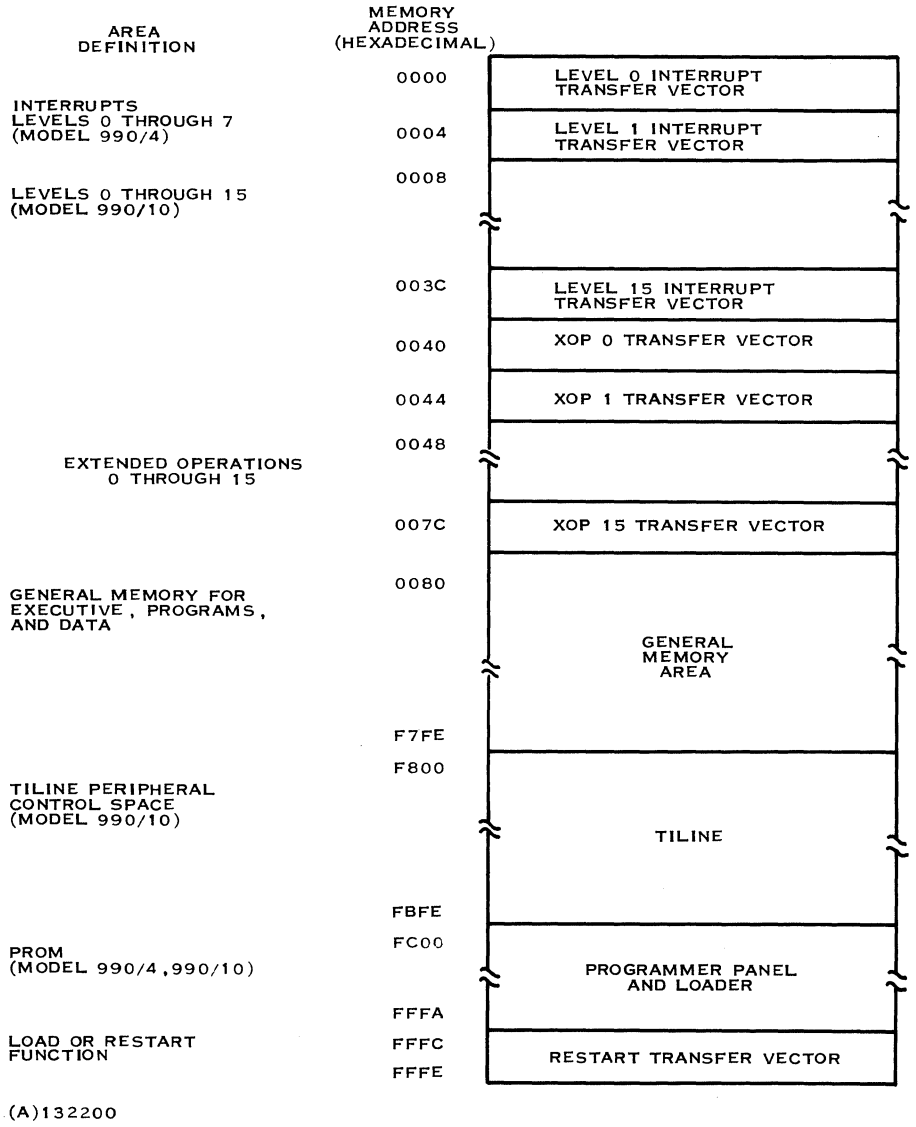


Figure 2-3. Typical Memory Map for Model 990 Computer/TMS 9900 Microprocessor

- OV, bit 4 - Overflow
- OP, bit 5 - Odd parity
- X, bit 6 - Extended operation
- Bits 12-15 - Interrupt mask

Two of the reserved bits in the Model 990/4 Status Register are defined for the Status Register of the Model 990/10. Bit 7, the PR bit, is set to one to inhibit execution of the privileged instructions. When execution of a privileged instruction is attempted with the PR bit set to one, an illegal instruction error occurs. Bit 7 must be set to zero to execute these instructions. An additional bit, bit 8, the Map File (MF) bit, specifies the memory map file for the memory mapping option. The memory mapping option provides access to memory addresses outside of the range

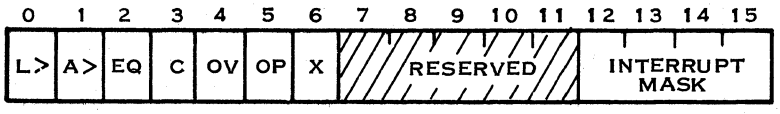


Figure 2-4. Status Register, Model 990 Computer TMS 9900

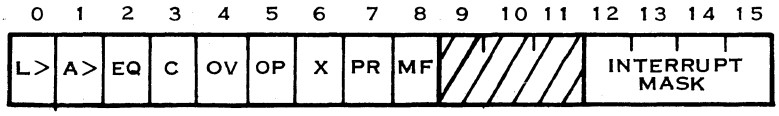


Figure 2-5. Status Register, Model 990/10 With Map Option

of addresses (32K words) of the address portions of instructions. When bit 8 is set to 0, the six mapping registers for map 0 are active. When bit 8 is set to 1, the six mapping registers for map 1 are active.

2.4.1 **LOGICAL GREATER THAN.** The logical greater than bit of the Status Register contains the result of a comparison of words or bytes as unsigned binary numbers. In this comparison, the most significant bits of words being compared represent 2^{15}, and the most significant bits of bytes being compared represent 2^{7}.

2.4.2 **ARITHMETIC GREATER THAN.** The arithmetic greater than bit of the Status Register contains the result of a comparison of words or bytes as two's complement numbers. In this comparison, the most significant bits of words or bytes being compared represent the sign of the number, zero for positive, or one for negative. For positive numbers, the remaining bits represent the binary value. For negative numbers, the remaining bits represent the two's complement of the binary value.

2.4.3 **EQUAL.** The equal bit of the Status Register is set when the words or bytes being compared are equal. Whether the comparison is that of unsigned binary numbers or two's complement numbers the significance of equality is the same.

2.4.4 **CARRY.** The carry bit of the Status Register is set by a carry out of the most significant bit of a word or byte (sign bit) during arithmetic operations. The carry bit is used by the shift operations to store the last bit shifted out of the workspace register being shifted.

2.4.5 **OVERFLOW.** The overflow bit of the Status Register is set when the result of an arithmetic operation is too large or too small to be correctly represented in two's complement representation. In addition operations, the overflow bit is set when the most significant bits of the operands are equal and the most significant bit of the result is not equal to the most significant bit of the destination operand. In subtraction operations, the overflow bit is set when the most significant bits of the operands are not equal, and the most significant bit of the result is not equal to the most significant bit of the destination operand. For a divide operation, the

overflow bit is set when the most significant sixteen bits of the dividend are greater than or equal to the divisor. For an arithmetic left shift, the overflow bit is set if the most significant bit of the workspace register being shifted changes value. For the absolute value and negate instructions, the overflow bit is set when the source operand is the maximum negative value, 8000_{16}.

2.4.6 **ODD PARITY.** The odd parity bit of the Status Register is set in byte operations when the parity of the result is odd, and is reset when the parity is even. The parity of a byte is odd when the number of bits having values of one is odd; when the number of bits having values of one is even, the parity of the byte is even. The odd parity bit is equal to the least significant bit of the sum of the bits in the byte.

2.4.7 **EXTENDED OPERATION.** The extended operation bit of the Status Register is set to one when a software implemented extended operation is initiated. An extended operation is initiated by a context switch using the transfer vector for the specified extended operation. After the WP and PC have been set to the values in the transfer vector, the extended operation bit is set.

2.4.8 **STATUS BIT SUMMARY.** Table 2-1 lists the instructions of the Model 990 Computer/ TMS 9900 Microprocessor instruction set and the status bits affected by each instruction. The effectivity column contains A to indicate applicability to all Model 990 Computers and the TMS 9900 Microprocessor. The column contains C to indicate applicability to all Model 990 Computers but not to the TMS 9900 Microprocessor. The column contains M to indicate applicability only to Model 990/10 Computers with mapping option. The interrupt mask is explained in a subsequent paragraph.

2.5 MEMORY ORGANIZATION

Figure 2-3 shows a generalized memory map applicable to Model 990 Computer/TMS 9900 Microprocessor memories. The area of low-order memory from address 0 through $7F_{16}$ is used for interrupt and extended operation transfer vectors as previously described. Addresses reserved for transfer vectors that are not used (interrupt levels 8 through 15 in Model 990/4 computers) may be used for instructions and/or data. Since many memory configurations are available as options, the programmer should ascertain the memory configuration for his system.

The area of memory from address 80_{16} through address $F7FE_{16}$ is available for workspaces, instructions, and data. Many users of Model 990 Computers will place an executive (PX990, TX990 or DX10) in a portion of this area. The remainder of this area (as supplied) is available for workspaces, instructions, and data for user programs. TMS 9900 users, and Model 990 Computer users who do not use PX990, or TX990 or DX10 may use the entire area (as supplied).

Various types and sizes of memory are available for the TMS 9900 Microprocessor and the Model 990/4 Computer. Addressing is not necessarily continuous. Addresses may be assigned according to the needs of an application, omitting addresses as appropriate.

In the Model 990/10 Computer, addresses $F800_{16}$ through $FBFE_{16}$ are reserved for TILINE communication with peripheral devices. These addresses may be assigned to registers in controllers for direct memory access devices. Input/Output from or to these devices is performed using any instruction that may be used to access memory. For I/O, the address in the instruction must be the TILINE address assigned to the appropriate register. An example of TILINE interface is shown in Appendix I.

Table 2-1. Status Bits Affected by Instructions

Mnemonic	Eff.	L>	A>	EQ	C	OV	OP	X
A	A	X	X	X	X	X	–	–
AB	A	X	X	X	X	X	X	–
ABS	A	X	X	X	X	X	–	–
AI	A	X	X	X	X	X	–	–
ANDI	A	X	X	X	–	–	–	–
B	A	–	–	–	–	–	–	–
BL	A	–	–	–	–	–	–	–
BLWP	A	–	–	–	–	–	–	–
C	A	X	X	X	–	–	–	–
CB	A	X	X	X	–	–	X	–
CI	A	X	X	X	–	–	–	–
CKOF	C	–	–	–	–	–	–	–
CKON	C	–	–	–	–	–	–	–
CLR	A	–	–	–	–	–	–	–
COC	A	–	–	X	–	–	–	–
CZC	A	–	–	X	–	–	–	–
DEC	A	X	X	X	X	X	–	–
DECT	A	X	X	X	X	X	–	–
DIV	A	–	–	–	–	X	–	–
IDLE	C	–	–	–	–	–	–	–
INC	A	X	X	X	X	X	–	–
INCT	A	X	X	X	X	X	–	–
INV	A	X	X	X	–	–	–	–
JEQ	A	–	–	–	–	–	–	–
JGT	A	–	–	–	–	–	–	–
JH	A	–	–	–	–	–	–	–
JHE	A	–	–	–	–	–	–	–
JL	A	–	–	–	–	–	–	–
JLE	A	–	–	–	–	–	–	–
JLT	A	–	–	–	–	–	–	–
JMP	A	–	–	–	–	–	–	–
JNC	A	–	–	–	–	–	–	–
JNE	A	–	–	–	–	–	–	–
JNO	A	–	–	–	–	–	–	–
JOC	A	–	–	–	–	–	–	–
JOP	A	–	–	–	–	–	–	–

Table 2-1. Status Bits Affected by Instructions (Continued)

Mnemonic	Eff.	L>	A>	EQ	C	OV	OP	X	Mnemonic	Eff.	L>	A>	EQ	C	OV	OP	X
LDCR	A	X	X	X	–	–	1	–	SBZ	A	–	–	–	–	–	–	–
LDD	M	–	–	–	–	–	–	–	SETO	A	–	–	–	–	–	–	–
LDS	M	–	–	–	–	–	–	–	SLA	A	X	X	X	X	X	–	–
LI	A	X	X	X	–	–	–	–	SOC	A	X	X	X	–	–	–	–
LIMI	A	–	–	–	–	–	–	–	SOCB	A	X	X	X	–	–	X	–
LMF	M	–	–	–	–	–	–	–	SRA	A	X	X	X	X	–	–	–
LREX	C	–	–	–	–	–	–	–	SRC	A	X	X	X	X	–	–	–
LWPI	A	–	–	–	–	–	–	–	SRL	A	X	X	X	X	–	–	–
MOV	A	X	X	X	–	–	–	–	STCR	A	X	X	X	–	–	1	–
MOVB	A	X	X	X	–	–	X	–	STST	A	–	–	–	–	–	–	–
MPY	A	–	–	–	–	–	–	–	STWP	A	–	–	–	–	–	–	–
NEG	A	X	X	X	X	X	–	–	SWPB	A	–	–	–	–	–	–	–
ORI	A	X	X	X	–	–	–	–	SZC	A	X	X	X	–	–	–	–
RSET	C	–	–	–	–	–	–	–	SZCB	A	X	X	X	–	–	X	–
RTWP	A	X	X	X	X	X	X	X	TB	A	–	–	X	–	–	–	–
S	A	X	X	X	X	X	–	–	X	A	2	2	2	2	2	2	2
SB	A	X	X	X	X	X	X	–	XOP	A	2	2	2	2	2	2	2
SBO	A	–	–	–	–	–	–	–	XOR	A	X	X	X	–	–	–	–

Notes: 1. When an LDCR or STCR instruction transfers eight bits or less, the OP bit is set or reset as in byte instructions. Otherwise these instructions do not affect the OP bit.

2. The X instruction does not affect any status bit; the instruction executed by the X instruction sets status bits normally for that instruction. When an XOP instruction is implemented by software, the XOP bit is set, and the subroutine sets status bits normally.

In the Model 990 Computers supplied with the optional front panel/loader ROM, addresses $FC00_{16}$ through $FFFB_{16}$ are reserved for the Programmed Read Only Memory (PROM) which contains the programmer panel program and a loader program. When the programmer panel is not connected, the program transfers control to the loader program. Control passes to the programmer panel program by a context switch using the transfer vector at address $FFFC_{16}$.

Any 16-word area of memory may be assigned as a workspace, and becomes the active workspace when the address of the first word of the area is placed in the WP register. Figure 2-6 shows a workspace, with those registers that have assigned functions identified in the figure.

Memory for the Model 990/10 Computer may contain more than 32K words, but the address format addresses only 32K words directly. The mapping option is used to address memory locations outside of the 32K word addressing capability. The mapping hardware has three 11-bit limit registers and three 16-bit bias registers for each of the three map files. The mapped address is a 20-bit address, the sum of the 16-bit processor address and the contents of a bias register extended to the right with five zeros. The least significant bit (which selects bytes) is ignored. The limit registers contain the one's complement of the limits, and determine which bias register is used. When the 11 most significant bits of the 16-bit address are less than or equal to limit 1, bias register 1 is used. When the same value is greater than limit 1 and less than or equal to limit 2, bias register 2 is used. When the same value is greater than limit 2 and less than or equal to limit 3, bias register 3 is used. When the same value is greater than limit 3, a mapping error interrupt occurs and memory is not accessed.

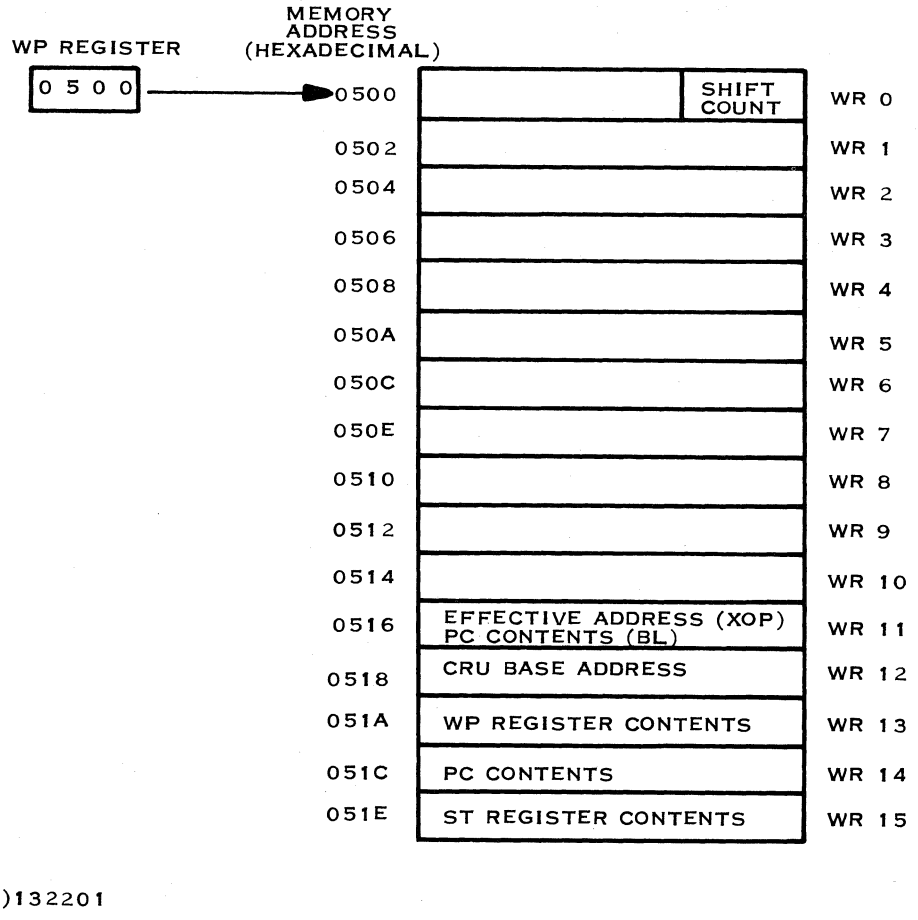


Figure 2-6. Model 990 Computer Workspace

When power is applied, the status register clears, selecting map file 0 and the limit and bias registers are set to zero. The limits (one's complement of limit register contents) are $FFFF_{16}$. This results in all addresses using bias register 1, which contains zero. The result is that all addresses are mapped into the same addresses. Map file 1 consists of three limit registers and three bias registers, and is intended for application programs. Map file 2 similarly consists of three limit registers and three bias registers, and is used to map one specified address outside of the current map. The LMF instruction loads map files 0 and 1.

For example, figure 2-7 shows a map file and the comparison of processor addresses to limits. Figure 2-7 also shows the addition of a bias register to a processor address. The contents of the map file are chosen in this example so that processor addresses 0000_{16} through $10FF_{16}$ map to addresses 000000_{16} through $0010FF_{16}$, processor addresses 1100_{16} through $A0FF_{16}$ map to addresses $0322E0_{16}$ through $03B2DF_{16}$, and processor addresses $A100_{16}$ through $F7FF_{16}$ map to addresses $04A100_{16}$ through $04F7FF_{16}$. Processor addresses greater than $F7FF_{16}$ result in error interrupts. This requires that limit register L1 contains 11101111000_2, the one's complement of the 11 most significant bits of $10FF_{16}$. Similarly, limit register L2 contains 01011111000_2 (one's complement of 11 most significant bits of $A0FF_{16}$) and limit register L3 contains 00001000000_2 (one's complement of the 11 most significant bits of $F7FF_{16}$). Bias register B1 contains 0000_{16}, bias register B2 contains $188F_{16}$, and bias register B3 contains 2000_{16}.

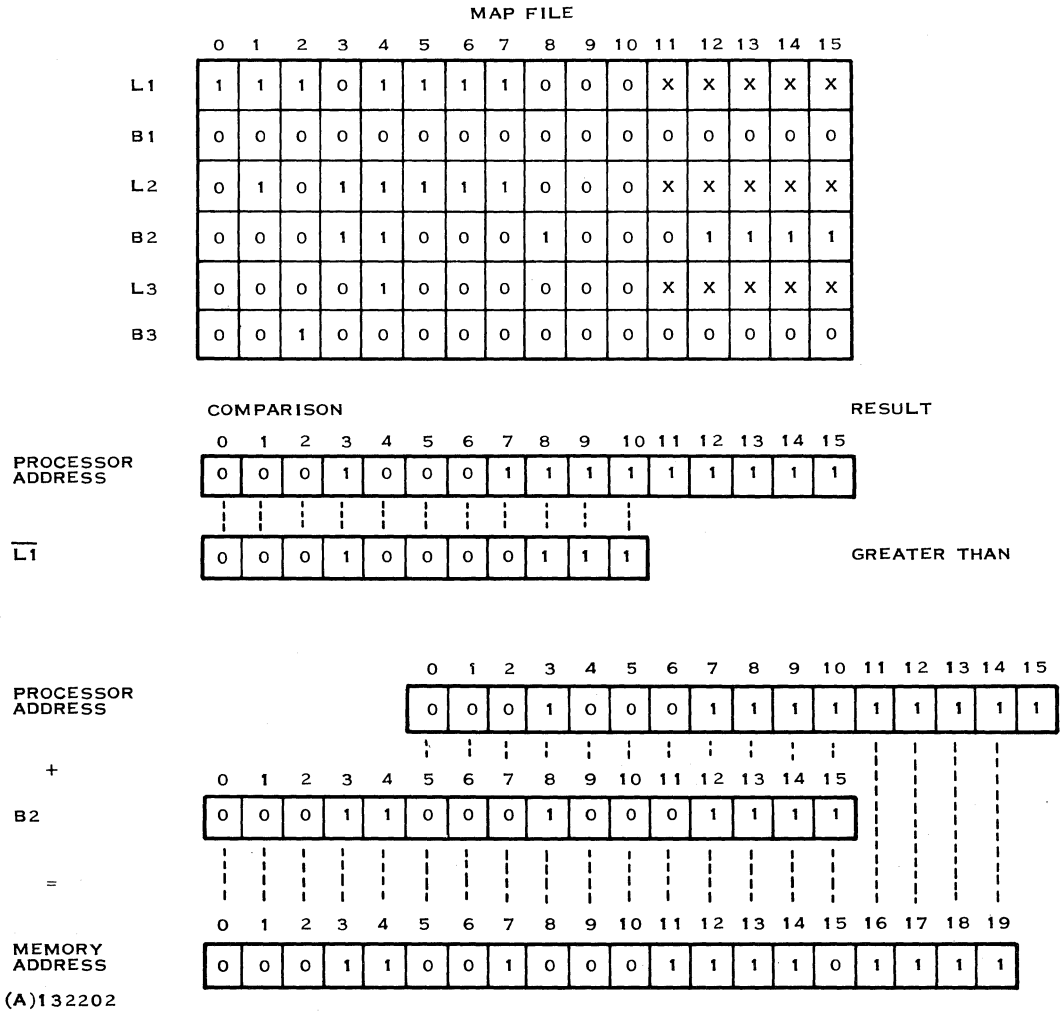


Figure 2-7. Address Development Model 990/10 Map Option

2.6 PRIVILEGED MODE

The Model 990/10 Computer has a privileged mode in which any instruction of the instruction set may be executed. When the computer is not in the privileged mode, and execution of a privileged instruction is attempted, the instruction is not executed and an error interrupt occurs. The privileged instructions perform operating system functions not appropriate in user programs. The specific instructions are identified in a subsequent section. The computer is placed in the privileged mode and the map file set to map file 0 when power is applied, when an interrupt occurs, and when an XOP instruction is executed.

2.7 SOURCE STATEMENT FORMAT

An assembly language source program consists of source statements which may contain assembler directives, machine instructions, pseudo-instructions, or comments. Each source statement is a source record as defined for the source medium; i.e., an 80-column card for punched card input, or a line of characters terminated by a carriage return for input from the keyboard of a terminal, such as the Model 733 ASR Data Terminal or a CRT Display Terminal.

The following conventions apply in the syntax definitions for machine instructions and assembler directives:

- Items in capital letters, and special characters, must be entered as shown.
- Items within angle brackets (< >) are defined by the user.
- Items in lower case letters are classes (generic names) of items.
- Items within brackets ([ ]) are optional.
- Items within braces ({ }) are alternative items; one must be entered.
- All ellipsis (. . .) indicates that the preceding item may be repeated.
- The symbol ƀ represents a blank or space.

The syntax for source statements other than comment statements is defined as follows:

[<label>]ƀ . . . opcodeƀ . . . [<operand>] [,<operand>] . . . ƀ . . . [<comment>]

This syntax definition means that a source statement may have a label, which is defined by the user. One or more blanks separate the label from the opcode. Mnemonic operation codes, assembler directives codes, and user-defined operation codes are all included in the generic term opcode, and any of these may be entered. One or more blanks separate the opcode from the operand, when an operand is required. Additional operands, when required, are separated by commas. One or more blanks separate the operand or operands from the comment field.

Comment statements consist of a single field starting with an asterisk (*) in the first character position followed by any ASCII character including a blank in each succeeding character position. Comment statements are listed in the source portion of the assembly listing and have no other effect on the assembly.

The maximum length of source records is 60 characters. However, only the first 52 characters will be printed on the Model 733 ASR Data terminal. The last source statement of a source program is followed by the end-of-record statement for the source medium, i.e., for punched cards, a card having a slash (/) punched in column 1 and an asterisk (*) punched in column 2.

Figure 2-8 shows source statements written on a coding form illustrating alternative methods of entering statements. The first four statements illustrate the alignment of the label, opcode, operands, and comments to begin in the same column in each statement. This method promotes readability, but may be time-consuming on some input devices, particularly data terminals. The last four statements show the use of horizontal tab characters represented by $\frac{H}{T}$ to separate the fields. On the Model 733 ASR Data Terminal, the tab character is entered by holding the CTRL key while pressing the I key. PX9ASM does not implement this use of $\frac{H}{T}$.

2.7.1 **CHARACTER SET.** The assemblers for the Model 990 Computers and the TMS 9900 Microprocessor recognize ASCII characters as follows:

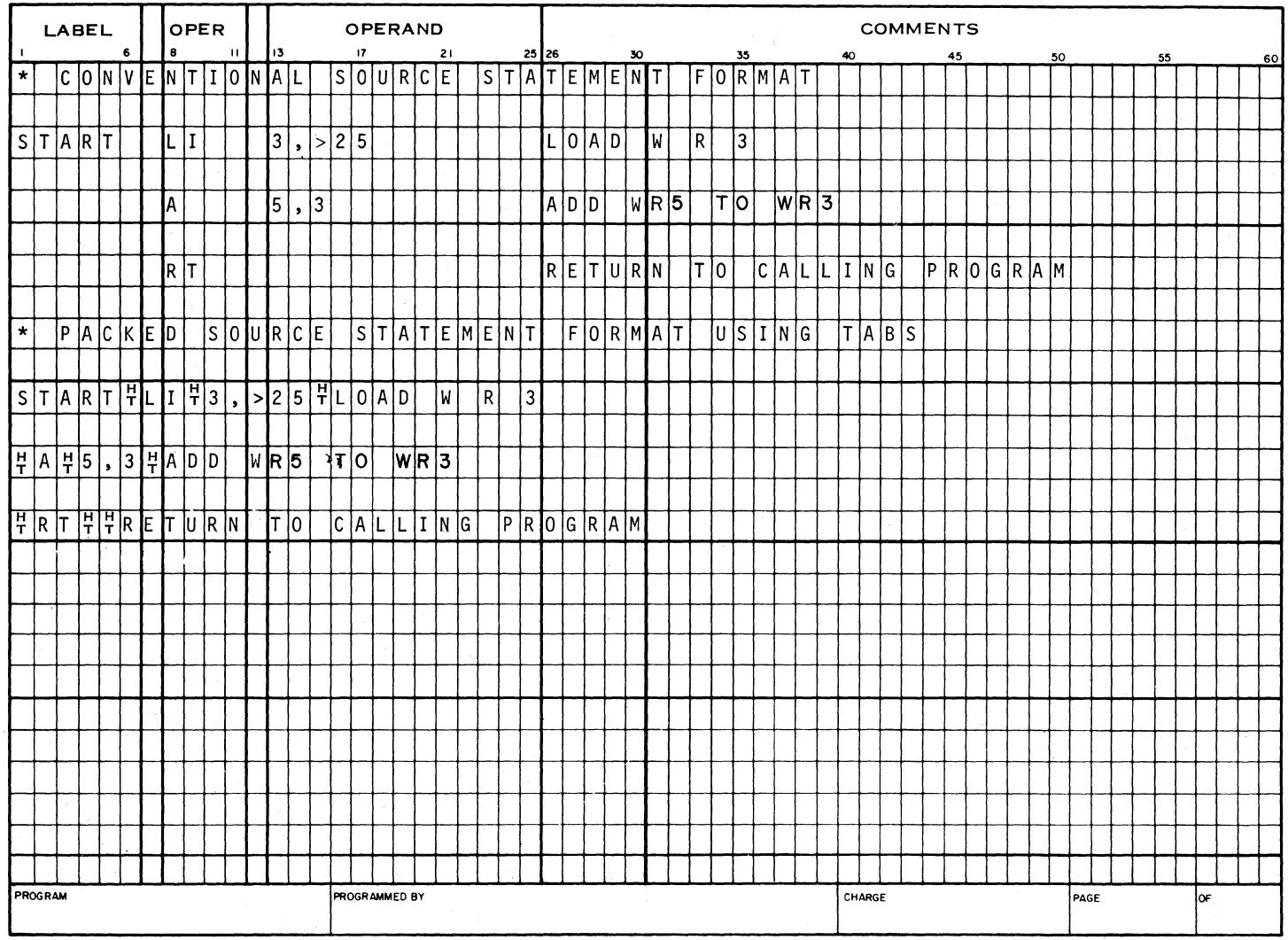
- The alphabet (capital letters only) and space character
- The numerals
- Twenty-two special characters
- Five characters defined for this language, that are undefined as ASCII characters
- The null character
- The tab character

Appendix A contains tables that list all 66 characters and show the ASCII and Hollerith codes for each.

2.7.2 **LABEL FIELD.** The label field begins in character position one of the source record and extends to the first blank. The label field contains a symbol containing up to six characters the first of which must be alphabetic. Additional characters may be any alphanumeric characters. A label is optional for machine instructions and for many assembler directives. When the label is omitted, the first character position must contain a blank. A source statement consisting of only a label field is a valid statement; it has the effect of assigning the current location to the label. This is usually equivalent to placing the label in the label field of the following machine instruction or assembler directive. However, when a statement consisting of a label only follows a TEXT or BYTE directive and is followed by a DATA directive or a machine instruction, the label will not have the value of a label in the following statement unless the TEXT or BYTE directive left the location counter on an even (word) location. An EVEN directive following the TEXT or BYTE directive prevents this problem.

2.7.3 **OPERATION FIELD.** The operation (opcode) field begins following the blank that terminates the label field, or in the first non-blank character position after the first character position when the label is omitted. The operation field is terminated by one or more blanks, and may not extend past character position 60 of the source record. The operation field contains an opcode, one of the following:

- Mnemonic operation code of a machine instruction
- Assembler directive operation code
- Symbol assigned to an extended operation by a DXOP directive
- Pseudo-instruction operation code



(A)132203 A

Figure 2-8. Source Statement Formats

2.7.4 **OPERAND FIELD.** The operand field begins following the blank that terminates the operation field, and may not extend past character position 60 of the source record. The operand field may contain one or more expressions, terms, or constants, according to the requirements of the opcode. The operand field is terminated by one or more blanks.

2.7.5 **COMMENT FIELD.** The comment field begins following the blank that terminates the operand field, and may extend to the end of the source record if required. The comment field may contain any ASCII character, including blank. The contents of the comment field are listed in the source portion of the assembly listing and have no other effect on the assembly.

2.8 EXPRESSIONS

Expressions are used in the operand fields of assembler directives and machine instructions. An expression is a constant or symbol, or a series of constants, a series of symbols, or a series of constants and symbols separated by arithmetic operators. Each constant or symbol may be preceded by a minus sign (unary minus). An expression may contain no embeded blanks, or symbols that are defined as extended operations. Symbols that are defined as external references may not be operands of arithmetic operations. For PX9ASM, only one symbol in an expression may be subsequently defined in the program, and that symbol must not be part of an operand in a multiplication or division operation within the expression. For the Cross Assembler, TXMIRA, and SDSMAC, an expression may contain more than one symbol that is not previously defined. When these symbols are absolute, they may also be operands of multiplication or division operations within an expression. In all assemblers, an expression that contains a relocatable symbol or relocatable constant immediately following a multiplication or division operator is an illegal expression. Also, when the result of evaluating an expression up to a multiplication or division operator is relocatable, the expression is illegal. An expression in which the number of relocatable symbols or constants added to the expression exceeds the number of relocatable symbols or constants subtracted from the expression by more than one is an illegal expression.

If NA = Number of relocatable values added and
NS = Number of relocatable values subtracted and

Then if

$$NA - NS = \begin{cases} 0 & \text{The expression is absolute} \\ 1 & \text{The expression is relocatable} \\ \text{Other than 0 or 1,} & \text{the expression is illegal} \end{cases}$$

An expression containing relocatable symbols or constants of several different relocation types (see Section VIII) is absolute if it is absolute with respect to all relocation types. If it is relocatable with respect to one relocation type and absolute with respect to all other relocation types, then the expression is relocatable. For example, the expression

RED + BLUE - GREEN + 2

is program-relocatable if BLUE is a program-relocatable symbol and the symbols RED and GREEN are both data-relocatable. If the symbols RED, BLUE, and GREEN were program-relocatable, data-relocatable, and common-relocatable, respectively, the expression would be invalid. TXMIRA and PX9ASM only support program-relocatable symbol.

In TXMIRA, if the current value of an expression is relocatable with respect to one relocation type, a symbol of another relocation type may not be included until the value of the expression becomes absolute. For example, the expression

BLUE - GREEN - RED

would be valid if BLUE and GREEN are of the same relocation type but would be invalid otherwise.

The following are examples of valid expressions:

BLUE+1	The sum of the value of symbol BLUE plus 1.
GREEN-4	The result of subtracting 4 from the value of symbol GREEN.
2*16+RED	The sum of the value of symbol RED plus the product of 2 times 16.
440/2-RED	The result of dividing 440 by 2 and subtracting the value of symbol RED from the quotient. RED must be absolute.

2.8.1 WELL-DEFINED EXPRESSIONS. Some assembler directives require well-defined expressions in the operand fields. For an expression to be well-defined, any symbols or assembly-time constants in the expression must have been previously defined. Also, the evaluation of a well-defined expression must be absolute, and a well-defined expression may not contain a character constant.

2.8.2 ARITHMETIC OPERATORS. The arithmetic operators in expressions are as follows:

- + for addition
- - for subtraction
- * for multiplication
- / for signed division
- // for logical right shift (SDSMAC only)

In evaluating an expression, the assembler first negates any constant or symbol preceded by a unary minus, then performs the arithmetic operations from left to right. The assembler does not assign precedence to any operation other than unary minus. All operations are integer operations. The assembler truncates the fraction in division.

For example, the expression 4+5*2 would be evaluated 18, not 14, and the expression 7+1/2 would be evaluated 4, not 7.

The logical right shift operator (//) allows a logical division by a power of two.

Examples:

```
>8000//1 = >4000      >AAAB//1 = >5555
>FFFF//0 = >FFFF      >FFFF//16 = >0000
```

SDSMAC checks for overflow conditions when arithmetic operations are performed at assembly time and gives a warning message whenever an overflow occurs, or when the sign of the result is not as expected in respect to the operands and the operation performed. Examples where a VALUE TRUNCATED message is given are:

```
>4000*2      >7FFF+1      -1*>8000
>8000*2      >8000-1      -2*>8001
```

2.9 CONSTANTS

Constants are used in expressions. The assemblers recognize four types of constants: decimal integer constants, hexadecimal integer constants, character constants, and assembly-time constants.

2.9.1 DECIMAL INTEGER CONSTANTS. A decimal integer constant is written as a string of numerals. The range of values of decimal integers is -32,768 to +65,535. Positive decimal integer constants greater than 32,767 are considered negative when interpreted as two's complement values. Operands of arithmetic instructions other than multiply and divide are interpreted as two's complement numbers, and all comparisons compare numbers both as signed and unsigned values.

The following are valid decimal constants:

1000	Constant, equal to 1000 or $3E8_{16}$.
-32768	Constant, equal to -32768 or 8000_{16}.
25	Constant, equal to 25, or 19_{16}.

2.9.2 HEXADECIMAL INTEGER CONSTANTS. A hexadecimal integer constant is written as a string of up to four hexadecimal numerals preceded by a greater than (>) sign. Hexadecimal numerals include the decimal values 0 through 9 and the letters A through F.

The following are valid hexadecimal constants:

>78	Constant, equal to 120, or 78_{16}.
>F	Constant, equal to 15, or F_{16}.
>37AC	Constant, equal to 14252 or $37AC_{16}$.

2.9.3 CHARACTER CONSTANTS. A character constant is written as a string of one or two characters enclosed in single quotes. For each single quote required within a character constant, two consecutive single quotes are required to represent the quote. The characters are represented internally as eight-bit ASCII characters, with the leading bit set to zero. A character constant consisting only of two single quotes (no character) is valid, and is assigned the value 0000_{16}.

The following are valid character constants:

'AB'	Represented internally as 4142_{16}.
'C'	Represented internally as 0043_{16}.
'N'	Represented internally as $004E_{16}$.
'''D'	Represented internally as 2744_{16}.

2.9.4 ASSEMBLY-TIME CONSTANTS. An assembly-time constant is written as an expression in the operand field of an EQU directive, described in a subsequent paragraph. When using TXMIRA or PX9ASM, any symbol in the expression must have been previously defined. The value of the label is determined at assembly time, and is considered to be absolute or relocatable according to the relocatability of the expression, not according to the relocatability of the location counter value.

2.10 SYMBOLS

Symbols are used in the label field, the operator field, and the operand field. A symbol is a string of alphanumeric characters, (A through Z and 0 through 9), the first of which must be an alphabetic character (A through Z), and none of which may be a blank. When more than six characters are used in a symbol, the assembler prints all the characters, but accepts only the first six characters for processing. User-defined symbols are valid only during the assembly in which they are defined.

Symbols used in the label field become symbolic addresses. They are associated with locations in the program, and must not be used in the label field of other statements. Mnemonic operation codes and assembler directive names are valid user-defined symbols when placed in the label field.

NOTE

When using SDSMAC, the ';' and '$' characters are considered alphabetic.

The DXOP directive defines a symbol to be used in the operator field. Any symbol that is used in the operand field must be placed in the label field of a statement, or in the operand field of a REF directive except for a symbol in the operand field of a DXOP directive or a predefined symbol.

2.11 PREDEFINED SYMBOLS

The predefined symbols are the dollar sign character ($) and the workspace register symbols. The dollar sign character is used to represent the current location within the program. The workspace register symbols are as follows:

Symbol	Value	Symbol	Value	Symbol	Value	Symbol	Value
R0	0	R4	4	R8	8	R12	12
R1	1	R5	5	R9	9	R13	13
R2	2	R6	6	R10	10	R14	14
R3	3	R7	7	R11	11	R15	15

NOTE

The workspace register symbols (R0, R1 . . .) are normally undefined in PX9ASM and TXMIRA. However, they can be optionally defined.

The following are examples of valid symbols:

START	Assigned the value of the location at which it appears in the label field.
A1	Assigned the value of the location at which it appears in the label field.
OPERATION	OPERAT is assigned the value of the location at which it appears in the label field.
$	Represents the current location.

2.12 TERMS

Terms are used in the operand fields of machine instructions and an assembler directive. A term is a decimal or hexadecimal constant, an absolute assembly-time constant, or label having an absolute value.

The following are examples of valid terms:

12	The value is 12, or C_{16}.
>C	The value is 12, or C_{16}.
WR2	Valid if WR2 is defined having an absolute value.
R3	Predefined as a value of 3.

If START were a relocatable symbol, the following statement would not be valid as a term:

WR2	EQU	START+4	WR2 would be a relocatable value 4 greater than the value of START. Not valid as a term, but valid as a symbol.

2.13 CHARACTER STRINGS

Several assembler directives require character strings in the operand field. A character string is written as a string of characters enclosed in single quotes. For each single quote in a character string, two consecutive single quotes are required to represent the required single quote. The maximum length of the string is defined for each directive that requires a character string. The characters are represented internally as eight-bit ASCII characters, with the leading bits set to zeros. Appendix A gives a complete list of valid characters within character strings.

The following are valid character strings:

'SAMPLE PROGRAM'	Defines a 14-character string consisting of: S A M P L E b P R O G R A M.
'PLAN "C"'	Defines an 8-character string consisting of: P L A N b ' C '.
'OPERATOR MESSAGE * PRESS START SWITCH'	Defines a 37-character string consisting of the expression enclosed in single quotes.

SECTION III

ASSEMBLY INSTRUCTIONS

3.1 GENERAL

This section describes the mnemonic instructions of the assembly language for the PX9ASM, TXMIRA and SDSMAC assemblers, and for the Cross Assember. Detailed assembly instruction descriptions follow descriptions of the addressing modes used in the assembly language and the addressing formats of the assembly instructions. The section also includes examples of programming the various instructions.

3.2 ADDRESSING MODES

One of five addressing modes may be used in the instructions that specify a general address for the source or destination operand. Table 3-1 lists these modes and shows how each is used in the assembly language. Each of the modes is described in a subsequent paragraph.

Table 3-1. Addressing Modes

Addressing Mode	T field value (Note 1)	Example	Note
Workspace Register	0	5	
Workspace Register Indirect	1	*7	
Symbolic Memory	2	@LABEL	2,3
Indexed Memory	2	@LABEL(5)	2,4
Workspace Register Indirect Autoincrement	3	*7+	

Notes:

1. The T field is described in the addressing format descriptions.
2. The instruction requires an additional word for each T field value of 2. This word contains a memory address.
3. The S or D field is set to zero by the assembler.
4. Workspace register 0 cannot be used for indexing.

3.2.1 **WORKSPACE REGISTER ADDRESSING.** Workspace register addressing specifies a workspace register that contains the operand. A workspace register address is written as a term having a value of 0 through 15.

The following examples show the coding of instructions that have two workspace register addresses each:

MOV R4,R8	Copy the contents of workspace register 4 into workspace register 8.
COC R15,R10	Compare the bits of workspace register 10 that correspond to the one bits in workspace register 15 to one.

3.2.2 **WORKSPACE REGISTER INDIRECT ADDRESSING.** Workspace register indirect addressing specifies a workspace register that contains the address of the operand. An indirect workspace register address is written as a term preceded by an asterisk (*). The following example shows coding of instructions having workspace register indirect addresses.

A *R7,*R2	Add the contents of the word at the address in workspace register 7 to the contents of the word at the address in workspace register 2, and place the sum in the word at the address in workspace register 2.
MOV *R7,R0	Copy the contents of the address in workspace register 7 into workspace register 0.

3.2.3 **SYMBOLIC MEMORY ADDRESSING.** Symbolic memory addressing specifies the memory address that contains the operand. A symbolic memory address is written as an expression preceded by an at sign (@). The following are coding examples of instructions having symbolic memory addresses:

S @TABLE1,@LIST4	Subtract the contents of the word at location TABLE1 from the contents of the word at location LIST4, and place the remainder in the word at location LIST4.
C R0,@STORE	Compare the contents of workspace register 0 with the contents of the word at location STORE.
MOV @12,@>7C	Copy the word at address $000C_{16}$ into location $007C_{16}$.

NOTE

When using SDSMAC, symbols previously defined as having relocatable values or values greater than 15 need not have the '@'.

3.2.4 **INDEXED MEMORY ADDRESSING.** Indexed memory addressing specifies the memory address that contains the operand. The address is the sum of the contents of a workspace register and a symbolic address. An indexed memory address is written as an expression preceded by an at

sign and followed by a term enclosed in parentheses. The workspace register specified by the term within the parentheses is the index register. Workspace register 0 may not be specified as an index register. the following are examples of coding of instructions having indexed memory addresses:

A @2(R7),R6	Add the contents of the word at the address computed by adding the contents of workspace register 7 and 2 to the contents of workspace register 6, and place the sum in workspace register 6.
MOV R7,@LIST4-6(R5)	Copy the contents of workspace register 7 into a word of memory. The address of the word of memory is the sum of the contents of workspace register 5 and the value of symbol LIST4 minus 6.

3.2.5 WORKSPACE REGISTER INDIRECT AUTO-INCREMENT ADDRESSING. Workspace register indirect auto-increment addressing specifies a workspace register that contains the address of the operand. After the address is obtained from the workspace register, the workspace register is incremented by 1 for a byte instruction or by 2 for a word instruction. The workspace register increment is one for byte operations and two for word operations. A workspace register auto-increment address is written as a term preceded by an asterisk and followed by a plus sign (+). The following are coding examples of instructions having workspace register indirect auto-increment addresses:

S *R3+,R2	Subtract the contents of the word at the address in workspace register 3 from the contents of workspace register 2, place the result in workspace register 2, and increment the address in workspace register 3 by two.
C R5,*R6+	Compare the contents of workspace register 5 with the contents of the word at the address in workspace register 6, and increment the address in workspace register 6 by two.

3.3 PROGRAM COUNTER RELATIVE ADDRESSING

Program counter relative addressing is used by the jump instructions. A program counter relative address is written as an expression that corresponds to an address at a word boundary. The assembler evaluates the expression and subtracts the sum of the current location plus two. One-half of the difference is the value that is placed in the object code. This value must be in the range of -128 through +127. When the instruction is in relocatable code (that is, when the location counter is relocatable), the relocation type of the evaluated expression must match the relocation type of the current location counter. When the instruction is in absolute code, the expression must be absolute. The following example shows a program counter relative address:

JMP THERE	Jumps unconditionally to location THERE.

3.4 CRU BIT ADDRESSING

The CRU bit instructions use a well-defined expression that represents a displacement from the CRU base address (bits 3 through 14 of workspace register 12). The displacement, in the range of -128 through +127, is added algebraically to the base address in workspace register 12. The following are examples of CRU bit instructions having CRU bit addresses:

SBO 8	Sets CRU bit to one at the CRU address 8 greater than the CRU base address. If workspace register 12 contained 0020_{16}, CRU bit 24 would be set by this instruction. ($24 = (20_{16}/2) + 8$)

SBZ DTR	Sets CRU bit to zero. Assuming that DTR has the value 10, and workspace register 12 contains 0040_{16}, the instruction sets bit 42 to zero. ($42 = (40_{16}/2) + 10$)

3.5 IMMEDIATE ADDRESSING

Immediate instructions use the contents of the word following the instruction word as an operand of the instruction. The immediate value is an expression, and the value of the expression is placed in the word following the instruction by the assembler. Those immediate instructions that require two operands have a workspace register address preceding the immediate value. The following are examples of coding immediate instructions:

LIMI 5	Places 5 in the interrupt mask, enabling interrupt levels 0 through 5.
LI R5,>1000	Places 1000_{16} into workspace register 5.

NOTE

When using SDSMAC, an @ sign may proceed an immediate operand.

3.6 ADDRESSING SUMMARY

Table 3-2 shows the addressing required for each instruction of the Model 990/TMS 9900 instruction set. The first column lists the instruction mnemonics, and the second column lists the effectivity of the instruction. This column contains A for those instructions that apply to the Model 990/TMS 9900, and C for those instructions that apply to the Model 990 but not to the TMS 9900. The column contains M for those instructions that apply only to the Model 990 Computers with mapping option. The third and fourth columns specify the required address, as follows:

- G - General address:

  Workspace register address

  Indirect workspace register address

  Symbolic memory address

  Indexed memory address

  Indirect workspace register auto-increment address

- WR - Workspace register address
- PC - Program counter relative address
- CRU - CRU bit address
- I - Immediate value
- * - The address into which the result is placed, when two operands are required.

Table 3-2. Instruction Addressing

Mnemonic	Eff.	First Operand	Second Operand	Mnemonic	Eff.	First Operand	Second Operand
A	A	G	G*	LDCR	A	G	Note 1
AB	A	G	G*	LDD	M	G	–
ABS	A	G	–	LDS	M	G	–
AI	A	WR*	I	LI	A	WR*	I
ANDI	A	WR*	I	LIMI	A	I	–
B	A	G	–	LMF	M	WR*	Note 2
BL	A	G	–	LREX	C	–	–
BLWP	A	G	–	LWPI	A	I	–
C	A	G	G	MOV	A	G	G*
CB	A	G	G	MOVB	A	G	G*
CI	A	WR	I	MPY	A	G	WR*
CKOF	C	–	–	NEG	A	G	–
CKON	C	–	–	ORI	A	WR*	I
CLR	A	G	–	RSET	C	–	–
COC	A	G	WR	RTWP	A	–	–
CZC	A	G	WR	S	A	G	G*
DEC	A	G	–	SB	A	G	G*
DECT	A	G	–	SBO	A	CRU	–
DIV	A	G	WR*	SBZ	A	CRU	–
IDLE	C	–	–	SETO	A	G	–
INC	A	G	–	SLA	A	WR*	Note 3
INCT	A	G	–	SOC	A	G	G*
INV	A	G	–	SOCB	A	G	G*
JEQ	A	PC	–	SRA	A	WR*	Note 3
JGT	A	PC	–	SRC	A	WR*	Note 3
JH	A	PC	–	SRL	A	WR*	Note 3
JHE	A	PC	–	STCR	A	G*	Note 1
JL	A	PC	–	STST	A	WR	–
JLE	A	PC	–	STWP	A	WR	–
JLT	A	PC	–	SWPB	A	G	–
JMP	A	PC	–	SZC	A	G	G*
JNC	A	PC	–	SZCB	A	G	G*
JNE	A	PC	–	TB	A	CRU	–
JNO	A	PC	–	X	A	G	–
JOC	A	PC	–	XOP	A	G	Note 4
JOP	A	PC	–	XOR	A	G	WR*

Notes:

1. The second operand is the number of bits to be transferred, 0-15, 0 = 16 bits.
2. The second operand specifies a memory map file, 0 or 1.
3. The second operand is the shift count, 0 - 15. 0 means count is in bits 12 - 15 of workspace register 0. When count = 0 and bits 12 - 15 of workspace register 0 = 0, count is 16.
4. Second operand specifies the extended operation, 0 - 15. Disposition of result may or may not be in the first operand address, determined by the user.

3.7 ADDRESSING FORMATS

The required addressing previously described relates to the ten addressing formats of the Model 990 Computer/TMS 9900 Microprocessor. These formats are shown and described in the following paragraphs.

3.7.1 FORMAT I - TWO ADDRESS INSTRUCTIONS. The operand field of Format I instructions contains two general addresses separated by a comma. The first address is the source address; the second is the destination address. The following mnemonic operation codes use Format I.

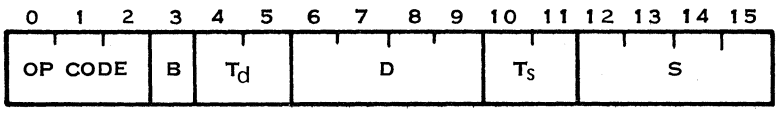
A	MOV	SOC
AB	MOVB	SOCB
C	S	SZC
CB	SB	SZCB

The following example shows a source statement for a Format I instruction:

```
SUM   A   @LABEL1,*R7
```

Adds the contents of the word at location LABEL1 to the contents of the word at the address in workspace register 7, and places the sum in the word at the address in workspace register 7. SUM is the location in which the instruction is placed.

The assembler assembles Format I instructions as follows:



The bit fields are:

- Op Code - Three bits that define the machine operation.
- B - Byte indicator, 1 for byte instructions, 0 for word instructions.
- T_d - Addressing mode (table 3-1) for destination.
- D - Destination workspace register.
- T_s - Addressing mode (table 3-1) for source.
- S - Source workspace register.

When T_s or T_d is equal to 10_2, the instruction occupies two words of memory, and the second word contains a memory address used with S or D, respectively, in developing the effective address. When both T_s and T_d are equal to 10_2, the instruction occupies three words of memory. The second word contains the memory address for the source operand, and the third word contains the memory address for the destination operand.

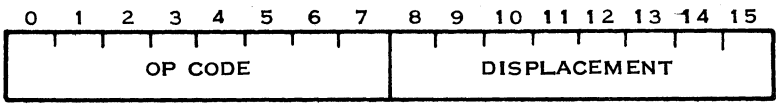
3.7.2 **FORMAT II - JUMP INSTRUCTIONS.** Format II instructions use program counter relative addresses which are coded as expressions that correspond to instruction locations on word boundaries. The following mnemonic operation codes are Format II jump instructions:

JEQ	JLE	JNE
JGT	JLT	JNO
JH	JMP	JOC
JHE	JNC	JOP
JL		

The following is an example of a source statement for a Format II jump instruction:

NOW JMP @BEGIN	Jumps unconditionally to the instruction at location BEGIN. The address of location BEGIN must not be greater than the address of location NOW by more than 127 words, nor less than the address of location NOW by more than 128 words.

The assemblers assemble Format II instructions as follows:



The bit fields are:

- Op Code - Eight bits that define the machine operation.
- Displacement - Signed displacement value.

The signed displacement value is shifted one bit position to the left and added to the contents of the PC after the PC has been incremented to the address of the following instruction. In other words, it is a displacement in words from the sum of the instruction address plus two.

3.7.3 **FORMAT II - BIT I/O INSTRUCTIONS.** The operand field of Format II CRU bit I/O instructions contains a well-defined expression. It is a CRU bit address, relative to the contents of workspace register 12. The following mnemonic operation codes are Format II CRU bit I/O instructions:

SBO	SBZ	TB

The following example shows a source statement for a Format II CRU bit I/O instruction:

SBO 5	Sets a CRU bit to one. If workspace register 12 contains 10_{16}, CRU bit 13 is set by this instruction.

The format assembled for Format II instructions is shown and described in the preceding paragraph. For CRU bit instructions the signed displacement is shifted one bit position to the left and added to the contents of workspace register 12. In other words, it is a displacement in bits from the contents of bits 3 through 14 of workspace register 12.

3.7.4 FORMAT III - LOGICAL INSTRUCTIONS. The operand field of Format III instructions contains a general address followed by a comma and a workspace register address. The general address is the source address, and the workspace register address is the destination address. The following mnemonic operation codes use Format III:

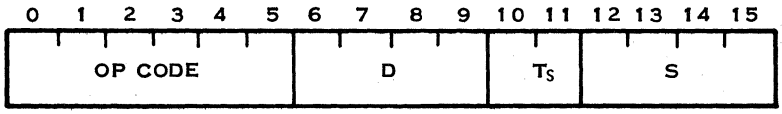
COC CZC XOR

The following example shows a source statement for a Format III instruction:

```
COMP XOR @LABEL8(R3),R5
```

Perform an exclusive OR operation of the contents of a memory word and the contents of workspace register 5, and place the result in workspace register 5. The address of the memory word is the sum of the contents of workspace register 3 and the value of symbol LABEL8.

The assemblers assemble Format III instructions as follows:



The bit fields are:

- Op Code - Six bits that define the machine operation.
- D - Destination workspace register.
- T_s - Addressing mode (table 3-1) for source.
- S - Source workspace register.

When T_s is equal to 10_2, the instruction occupies two words of memory. The second word contains the memory address for the source operand.

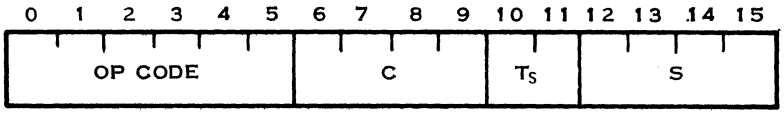
3.7.5 FORMAT IV - CRU INSTRUCTIONS. The operand field of Format IV instructions contains a general address followed by a comma and a well defined expression. The general address is the memory address from which or into which bits will be transferred. The CRU address for the transfer is the contents of bits 3 through 14 of workspace register 12. The term is the number of bits to be transferred, a value of 0 through 15 (a 0 value transfers 16 bits). For 8 or fewer bits the effective address is a byte address. For 9 or more bits the effective address is a word address. The following mnemonic operation codes use Format IV:

LDCR STCR

The following example shows a source statement for a Format IV instruction:

```
LDCR  *R6+,8      Place 8 bits from the byte of memory at the address
                  in workspace register 6 into eight consecutive CRU
                  lines at the CRU base address in workspace register
                  12.
```

The assemblers assemble Format IV instructions as follows:



The bit fields are:

- Op Code - Six bits that define the machine operation.
- C - Four bits that contain the bit count.
- T_s - Addressing mode (table 3-1) for source.
- S - Source workspace register.

When T_s is equal to 10_2, the instruction occupies two words of memory. The second word contains the memory address for the source operand.

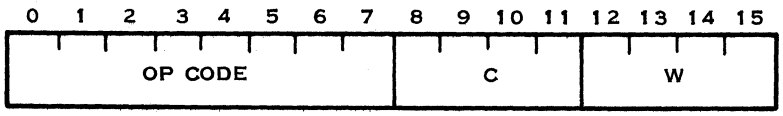
3.7.6 FORMAT V - REGISTER SHIFT INSTRUCTIONS. The operand field of Format V instructions contains a workspace register address followed by a comma and a well defined expression. The contents of the workspace register are shifted a number of bit positions specified by the term. When the term equals zero, the shift count must be placed in bits 12-15 of workspace register 0. The following mnemonic operation codes use Format V:

SLA SRC SRL SRA

The following example shows a source statement for a Format V instruction:

```
SLA   R6,4        Shift contents of workspace register 6 to the
                  left 4 bit positions, replacing the vacated bits
                  with zero.
```

The assemblers assemble Format V instructions as follows:



The bit fields are:

- Op Code - Eight bits that define the machine operation.
- C - Four bits that contain the shift count.
- W - Workspace register to be shifted.

3.7.7 FORMAT VI - SINGLE ADDRESS INSTRUCTIONS. The operand field of Format VI instructions contains a general address. The following mnemonic operation codes use Format VI:

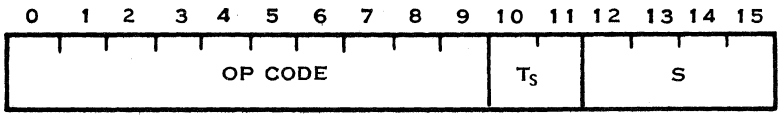
ABS	CLR	INCT	NEG
B	DEC	INV	SETO
BL	DECT	LDD	SWPB
BLWP	INC	LDS	X

The following example shows a source statement for a Format VI instruction:

```
CNT   INC   R7
```

Adds one to the contents of workspace register 7, and places the sum in workspace register 7. CNT is the location into which the instruction is placed.

The assemblers assemble Format VI instructions as follows:



The bit fields are:

- Op Code - Ten bits that define the machine operation.
- T_s - Addressing mode (table 3-1) for source.
- S - Source workspace register.

When T_s is equal to 10_2, the instruction occupies two words of memory. The second word contains the memory address for the source operand.

3.7.8 FORMAT VII - CONTROL INSTRUCTIONS. Format VII instructions require no operand field. The following operation codes use Format VII:

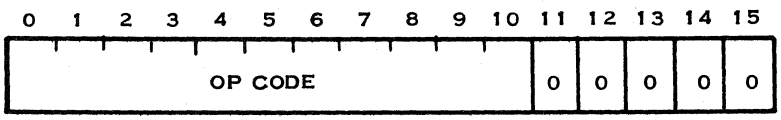
CKOF	IDLE	RSET
CKON	LREX	RTWP

The following example shows a source statement for a Format VII instruction:

```
RTWP
```

Returns control to the calling program, and restores the context of the calling program by placing the contents of workspace registers 13, 14, and 15 into the WP register, the PC, and the ST register.

The assemblers assemble Format VII instructions as follows:



The Op Code field contains eleven bits that define the machine operation. The five least significant bits are zeros.

3.7.9 FORMAT VIII - IMMEDIATE INSTRUCTIONS. The operand field of Format VIII instructions contains a workspace register address followed by a comma and an expression. The workspace register is the destination address, and the expression is the immediate operand. The following mnemonic operation codes use Format VIII:

AI	CI	ORI
ANDI	LI	

There are two additional Format VIII instructions that require only an expression in the operand field. The expression is the immediate operand. The destination is implied in the name of the instruction. The following mnemonic operation codes use this modified Format VIII:

LIMI	LWPI

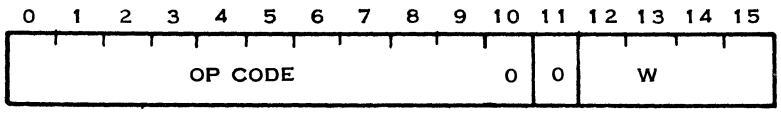
Another modification of Format VIII requires only a workspace register address in the operand field. The workspace register address is the destination. The source is implied in the name of the instruction. The following mnemonic operation codes use this modified Format VIII:

STST	STWP

The following are examples of source statement for Format VIII instructions:

ANDI	4,>000F	Perform an AND operation on the contents of workspace register 4 and immediate operand $000F_{16}$.
LWPI	WRK1	Place the address defined for the symbol WRK1 into the WP register.
STWP	R4	Place the contents of the WP register into workspace register 4.

The assemblers assemble Format VIII instructions as follows:



The bit fields are:

- Op Code - Eleven bits that define the machine operation.
- W - Workspace register operand.

A zero bit separates the two fields. The instructions that have no workspace register operand place zeros in the W field. The instructions that have immediate operands place the operands in the word following the word that contains the Op Code; i.e., these instructions occupy two words each.

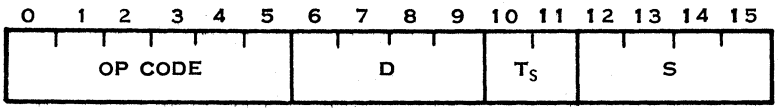
3.7.10 FORMAT IX - EXTENDED OPERATION INSTRUCTION. The operand field of a Format IX Extended Operation instruction contains a general address and a well defined expression. The general address is the address of the operand for the extended operation. The term specifies the extended operation to be performed and must be in the range of 0 to 15. The mnemonic operation code is XOP.

The following example shows a source statement for a Format IX Extended Operation instruction:

```
XOP  @LABEL(R4),12
```

Perform extended operation 12 using the address computed by adding the value of symbol LABEL to the contents of workspace register 4.

The assemblers assemble Format IX instructions as follows:



The bit fields are:

- Op Code - Six bits that define the machine operation.
- D - Four bits that define the extended operation.
- T_s - Addressing mode (table 3-1) for source.
- S - Source workspace register.

When T_s is equal to 10_2, the instruction occupies two words of memory. The second word contains the memory address for the source operand.

3.7.11 FORMAT IX - MULTIPLY AND DIVIDE INSTRUCTIONS. The operand field of Format IX Multiply and Divide instructions contains a general address followed by a comma and a workspace register address. The general address is the address of the multiplier or divisor, and the workspace register address is the address of the workspace register that contains the multiplicand or dividend. The workspace register address is also the address of the first of two workspace registers to contain the result. The mnemonic operation codes are MPY and DIV.

The following example shows a source statement for a Format IX Multiply instruction:

```
MPY  @ACC,R9
```

Multiply the contents of workspace register 9 by the contents of the word at location ACC, and place the product in workspace registers 9 and 10, with the 16 least significant bits of the product in workspace register 10.

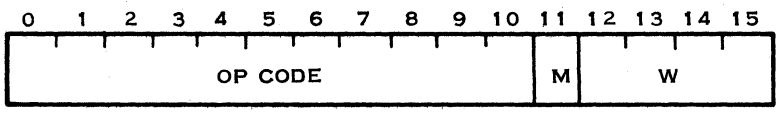
The assembler assembles Multiply and Divide instructions similarly to the format shown in the preceding paragraph, except that the D field contains the workspace register operand.

3.7.12 **FORMAT X - MEMORY MAP FILE INSTRUCTION.** This format applies only to the Model 990 Computer with map option. The operand field of a Format X Memory Map File instruction contains a workspace register address followed by a comma and a well defined expression which evaluates to either a 0 or a 1. The workspace register address specifies a workspace register that contains the address of a six-word area of memory that contains the map file data. The term specifies the map file into which the data is to be loaded. The mnemonic operation code is LMF.

The following example shows a source statement for a Format X Memory Map File instruction:

```
LMF     R4,0        Load memory map file 0 with the six-word
                    area of memory at the address in workspace
                    register 4.
```

The assembler assembles a Format X instruction as follows:



The bit fields are:

- Op Code - Eleven bits that define the machine operation.
- M - A single bit that specifies a memory map file, 0 or 1.
- W - Workspace register operand.

3.8 INSTRUCTION DESCRIPTIONS

The instruction descriptions in the following paragraphs are divided into the following functional categories:

- Arithmetic Instructions
- Branch Instructions
- Compare Instructions
- Control and CRU Instructions
- Load and Move Instructions
- Logical Instructions
- Shift Instructions
- Extended Operation Instruction
- Long Distance Addressing Instructions

The syntax definition for each instruction is shown, using the conventions described in a previous paragraph. The generic names used in these definitions are:

- ga_s - General address of source operand
- ga_d - General address of destination operand
- wa - Workspace register address
- iop - Immediate operand
- wa_d - Destination workspace register address
- disp - Displacement of CRU lines from the CRU base register
- exp - Expression that represents an instruction location.
- cnt - Count of bits for CRU transfer
- m - Memory map file
- scnt - Shift count
- op - Number (0-15) of extended operation

Source statements that contain machine instructions use the label field, the operation field, the operand field, and the comment field. Use of the label field is optional for machine instructions. When the label field is used, the label is assigned the address of the machine instruction. The assembler advances the location to a word boundary (even address) before assembling a machine instruction. The operation (opcode) field contains the mnemonic operation code of the instruction. The contents of the operand field is defined for each instruction. The use of the comment field is optional. When the comment field is used, it may contain any ASCII character, including blank, and has no effect on the assembly process other than to be printed in the listing.

A description of the operation of the instruction follows the syntax definition. The status bits affected by the instruction are listed. In the execution results, the following conventions are used:

- ( ) Indicates "the contents of"
- → Indicates "replaces"
- || Indicates the absolute value

The generic names used in the syntax definitions are also used in the execution results.

Application notes are included, referring to a fuller explanation in the programming examples paragraphs as appropriate.

The Op Code given for each instruction is a four hexadecimal digit number corresponding to an instruction word in which the address fields contain zeros. Next is the addressing mode. The instruction formats show the machine language form of the instruction, and use the terminology previously defined for the addressing formats.

3.9 ARITHMETIC INSTRUCTIONS

The arithmetic instructions are described in the following paragraphs. The instructions are:

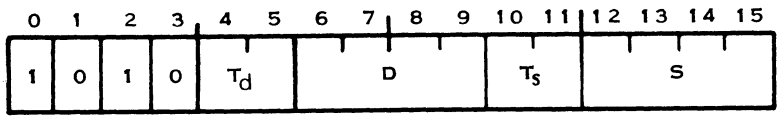
Instruction	Mnemonic	Paragraph
Add Words	A	3.10
Add Bytes	AB	3.11
Absolute Value	ABS	3.21
Add Immediate	AI	3.12
Decrement	DEC	3.19
Decrement by Two	DECT	3.20
Divide	DIV	3.16
Increment	INC	3.17
Increment by Two	INCT	3.18
Multiply	MPY	3.15
Negate	NEG	3.22
Subtract Words	S	3.13
Subtract Bytes	SB	3.14

3.10 ADD WORDS A

Op Code: A000

Addressing mode: Format I

Format:



Syntax definition:

[<label>]b . . . Ab . . . <ga_s>,<ga_d>b . . . [<comment>]

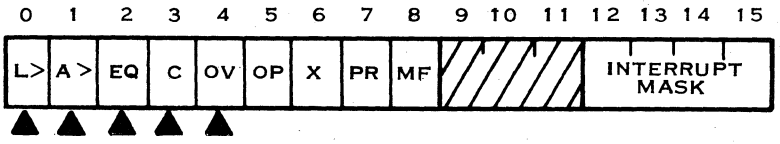
Example:

```
LABEL  A  @ADDR1(R2),@ADDR2(R3)
```

Definition: Add a copy of the source operand (word) to the destination operand (word) and replace the destination operand with the sum. The AU compares the sum to zero and sets/resets

the status bits to indicate the result of the comparison. When there is a carry out of bit zero, the carry status bit sets. When there is an overflow (the sum cannot be represented as a 16-bit, two's complement value), the overflow status bit sets.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results: $(ga_s) + (ga_d) \rightarrow (ga_d)$

Application notes: A is used to add signed integer words. For example, if the address labeled TABLE contains 3124_{16} and workspace register 5 contains 8_{16}, then the instruction

```
A   5,@TABLE
```

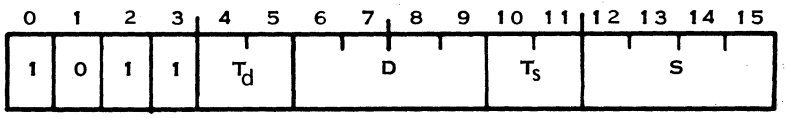
results in the contents of TABLE changing to $312C_{16}$ and the contents of workspace register 5 not changing. The logical and arithmetic greater than status bits set and the equal, carry, and overflow status bits reset.

3.11 ADD BYTES AB

Op Code: B000

Addressing mode: Format I

Format:



Syntax definition:

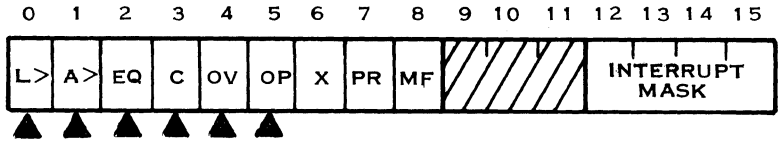
[<label>]b . . . ABb . . . <ga_s>,<ga_d>b . . . [<comment>]

Example:

```
LABEL   AB   3, 2
```

Definition: Add a copy of the source operand (byte) to the destination operand (byte), and replace the destination operand with the sum. When the destination operand is addressed in the workspace register mode, only the leftmost byte (bits 0-7) of the addressed workspace register is used. The AU compares the sum to zero and sets/resets the status bits to indicate the results of the comparison. When there is a carry out of the most significant bit of the byte, the carry status bit sets. When there is an overflow (the sum cannot be represented within a byte as an 8-bit two's complement value), the overflow status bit sets. The odd parity bit sets when the bits in the sum (destination operand) establish odd parity and resets when the bits in the sum establish even parity.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, overflow and odd parity.



Execution results: $(ga_s) + (ga_d) \rightarrow (ga_d)$

Application notes: AB is used to add signed integer bytes. For example, if the contents of workspace register 3 is 7400_{16}, the contents of memory location 2122_{16} is $F318_{16}$, and the contents of workspace 2 is 2123_{16}, then the instruction

```
AB      3,*2+
```

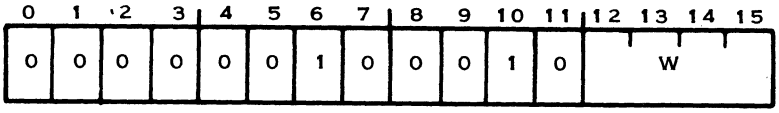
changes the contents of memory location 2122_{16} to $F38C_{16}$ and the contents of workspace register 2 to 2124_{16}, while the contents of workspace register 3 remain unchanged. The logical greater than, overflow, and odd parity status bits set, while the arithmetic greater than, equal, and carry status bits reset.

3.12 ADD IMMEDIATE AI

Op Code: 0220

Addressing mode: Format VIII

Format:



Syntax definition:

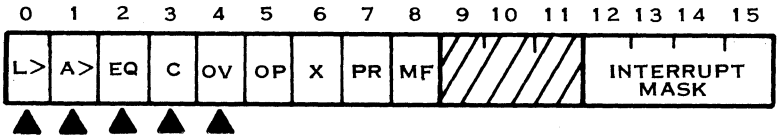
[<label>]b . . . AIb . . . <wa>,<iop>b . . . [<comment>]

Example:

```
LABEL   AI   2, 7   ADD 7 TO THE CONTENTS OF WSR2
```

Definition: Add a copy of the immediate operand, the contents of the word following the instruction word in memory, to the contents of the workspace register specified in the W field and replace the contents of the workspace register with the results. The AU compares the sum to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry out of bit zero, the carry status bit sets. When there is an overflow (the result cannot be represented within a word as a two's complement value), the overflow status bit sets.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results: (wa) + iop → (wa)

Application notes: Use the AI instruction to add an immediate value to the contents of a workspace register. For example, if workspace register 6 contains a zero, then the instruction

```
AI   6,>C
```

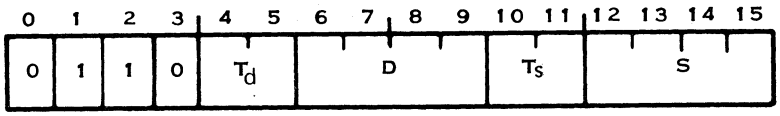
changes the contents of workspace register 6 to $000C_{16}$. The logical greater than and arithmetic greater than status bits set while the equal, carry, and overflow status bits reset.

3.13 SUBTRACT WORDS S

Op Code: 6000

Addressing mode: Format I

Format:



Syntax definition:

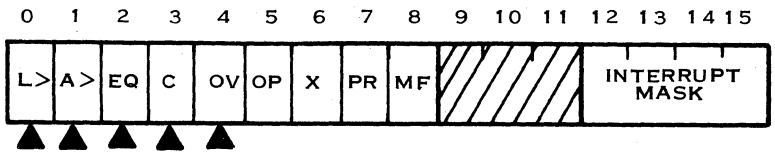
[<label>]b . . . Sb . . . <ga_s>,<ga_d>b . . . [<comment>]

Example:

```
LABEL    S    2, 3    SUBTRACT THE CONTENTS OF WR2 FROM THE CONTENTS
                      OF WR3
```

Definition: Subtract a copy of the source operand from the destination operand and place the difference in the destination operand. The AU compares the difference to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry out of bit zero, the carry status bit sets. When there is an overflow (the difference cannot be represented within a word as a two's complement value), the overflow status bit sets. The source operand remains unchanged.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results: $(ga_d) - (ga_s) \rightarrow (ga_d)$

Application notes: Use the S instruction to subtract signed integer values. For example, if memory location OLDVAL contains a value of 1225_{16} and memory location NEWVAL contains a value of 8223_{16}, then the instruction

```
S    @OLDVAL,@NEWVAL
```

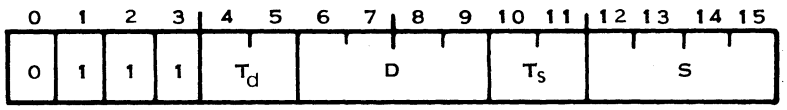
results in the contents of NEWVAL changing to $6FFE_{16}$. The logical greater than, arithmetic greater than, carry, and overflow status bits set while the equal status bit resets.

3.14 SUBTRACT BYTES SB

Op Code: 7000

Addressing mode: Format I

Format:



Syntax definitions:

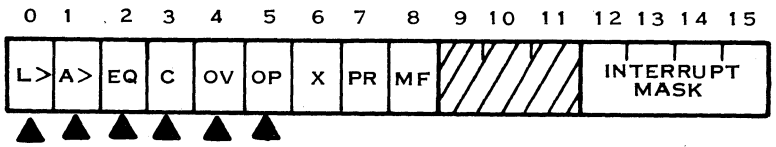
[<label>]b . . . SBb . . . <ga_s>,<ga_d>b . . . [<comment>]

Example:

```
LABEL   SB   2, 3   SUBTRACT THE LEFTMOST BYTE OF WSR2 FROM THE
                    LEFTMOST BYTE OF WSR3
```

Definition: Subtract a copy of the source operand (byte) from the destination operand (byte) and replace the destination operand byte with the difference. When the destination operand byte is addressed in the workspace register mode, only the leftmost byte (bits 0-7) in the workspace register is used. The AU compares the result byte to zero and sets/resets the status bits accordingly. When there is a carry out of the most significant bit of the byte, the carry status bit sets. When there is an overflow (the difference cannot be represented as an 8-bit, two's complement value in a byte), the overflow status bit sets. If the result byte establishes odd parity (an odd number of logic one bits in the byte), the odd parity status bit sets.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, overflow, and odd parity.



Execution results: $(ga_d) - (ga_s) \rightarrow (ga_d)$

Application notes: Use the SB instruction to subtract signed integer bytes. For example, if workspace register 6 contains the value $121C_{16}$, memory location $121C_{16}$ contains the value 2331_{16}, and workspace register 1 contains the value 1344_{16}, then the instruction

```
SB      *6+,1
```

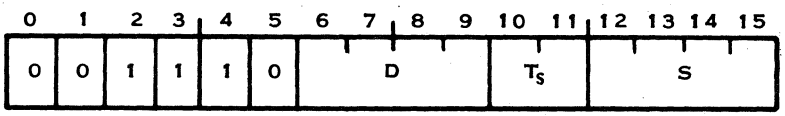
results in the contents of workspace register 6 changing to $121D_{16}$ and the contents of workspace register 1 changing to $F044_{16}$. The logical greater than status bit sets while the other status bits affected by this instruction reset.

3.15 MULTIPLY MPY

Op Code: 3800

Addressing mode: Format IX

Format:



Syntax definition:

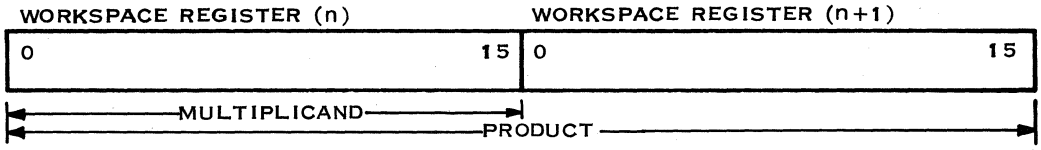
[<label>]b . . . MPYb . . . <ga_s>,<wa_d>b . . . [<comment>]

Example:

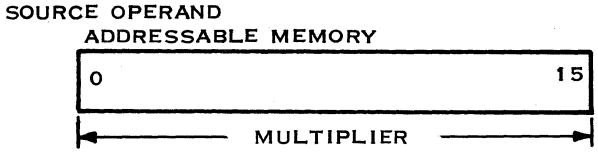
```
LABEL   MPY   @ADDR, 3      MULTIPLY (WSR3) BY (ADDR). THE RESULT IS
                            RIGHT JUSTIFIED IN THE 32-BITS OF WSR3, WSR4.
```

Definition: Multiply the first word in the destination operand (a consecutive 2-word area in workspace) by a copy of the source operand and replace the 2-word destination operand with the the result. The multiplication operation may be graphically represented as follows:

Destination operand workspace registers



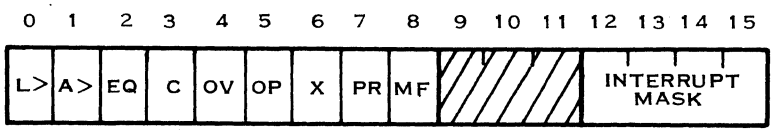
Source operand



The first word of the destination operand shown above is addressed by the contents of the D field. This word contains the multiplicand (unsigned magnitude value of 16 bits) right-justified in the workspace register (represented by workspace n above). The 16-bit, unsigned multiplier is located in the source operand. When the multiplication operation is complete, the product appears, right-justified in the entire 2-word area addressed by the D field as a 32-bit unsigned magnitude value. The maximum value of either input operand is $FFFF_{16}$ and the maximum value of the unsigned product is $(16^8 - 2(16^4) + 1)$ or $FFFE0001_{16}$.

If the destination operand is specified as workspace register 15, the first word of the destination operand is workspace register 15 and the second word of the destination operand is the memory word immediately following the workspace memory area.

Status bits affected: None



Execution results: $(ga_s) \cdot (wa_d)$. The product (32-bit magnitude) is placed in wa_d and $wa_d + 1$, with the most significant half in wa_d.

Application notes: Use the MPY instruction to perform a magnitude multiplication. For example, if workspace register 5 contains 0012_{16}, workspace register 6 contains $1B31_{16}$, and memory location NEW contains 0005_{16}, then the instruction

```
MPY     @NEW,5
```

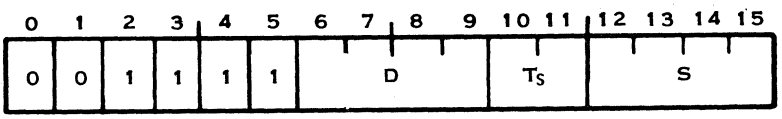
changes the contents of workspace register 5 to 0000_{16} and workspace register 6 to $005A_{16}$. The source operand is unchanged. The status register is not affected by this instruction.

3.16 DIVIDE DIV

Op Code: 3C00

Addressing mode: Format IX

Format:



Syntax definition:

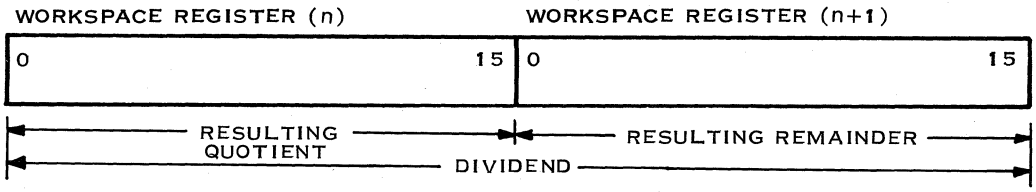
[<label>]b . . . DIVb . . . <ga_s>,<wa_d>b . . . [<comment>]

Example:

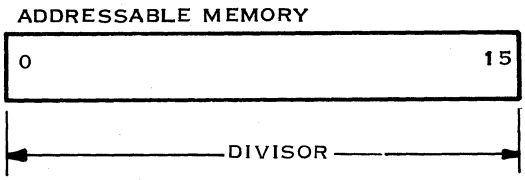
```
LABEL   DIV   @ADDR(2), 3    DIVIDE (WSR3, WSR4) BY (ADDR+(WSR2)) AND
                             STORE THE INTEGER RESULT IN WSR3 WITH THE
                             REMAINDER IN WSR4.
```

Definition: Divide the destination operand (a consecutive 2-word area of workspace) by a copy of the source operand (one word), using integer rules, and place the quotient in the first of the 2-word destination operand area and place the remainder in the second word of that same area. This division is graphically represented as follows:

Destination operand workspace registers



Source operand

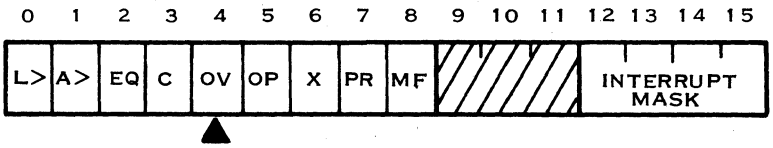


The first of the destination operand workspace registers, shown above, is addressed by the contents of the D field. The dividend is located right-justified in this 2-word area. When the division is complete, the quotient (result) is placed in the first workspace register of the destination operand (represented by n above) and the remainder is placed in the second word of the destination operand (represented by n+1 above).

When the source operand is greater than the first word of the destination operand, normal division occurs. If the source operand is less than or equal to the first word of the destination operand, normal division will result in a quotient that cannot be represented in a 16-bit word. In this case, the AU sets the overflow status bit, leaves the destination operand unchanged, and aborts the division operation.

If the destination operand is specified as workspace register 15, the first word of the destination operand is workspace register 15 and the second word of the destination operand is the word in memory immediately following the workspace area.

Status bits affected: Overflow



Execution results: The contents of wa_d and $wa_d + 1$ (32-bit magnitude) are divided by the contents of ga_s and the quotient is placed in wa_d. The remainder is placed in $wa_d + 1$.

Application notes: Use the DIV instruction to perform a magnitude division. For example, if workspace register 2 contains a zero and workspace register 3 contains $000C_{16}$, and the contents of LOC is 0005_{16}, then the instruction

```
DIV     @LOC,2
```

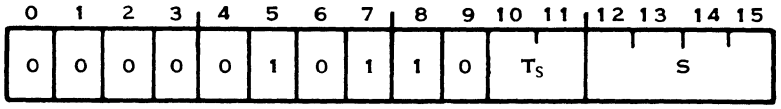
results in a 0002_{16} in workspace register 2 and a 0002_{16} in workspace register 3. The overflow status bit resets. If workspace register 2 contained the value 0005_{16}, the magnitude contained in the destination operand would equal 327,692 and division by the value 5 would result in a quotient of 65,538, which cannot be represented in a 16-bit word. This attempted division would set the overflow status bit and the AU would abort the operation.

3.17 INCREMENT INC

Op Code: 0580

Addressing mode: Format VI

Format:



Syntax definition:

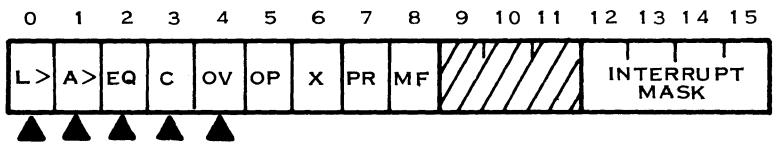
[<label>]b . . . INCb . . . <ga_s>b . . . [<comment>]

Example:

```
LABEL   INC   @ADDR(2)+     INCREMENT THE CONTENTS OF THE EFFECTIVE
                            LOCATION.
```

Definition: Add one to the source operand and replace the source operand with the result. The AU compares the sum to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry out of bit zero, the carry status bit sets. When there is an overflow (the sum cannot be represented in a 16-bit, two's complement value), the overflow status bit sets.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results: $(ga_s) + 1 \rightarrow (ga_s)$

Application notes: Use the INC instruction to count and index *byte* arrays, add a value of one to an addressable memory location, or set flags. For example, if COUNT contains a zero, the instruction

```
INC     @COUNT
```

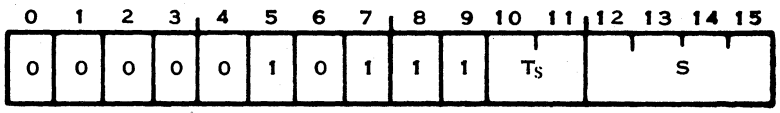
places a 0001_{16} in COUNT and sets the logical greater than and arithmetic greater than status bits, while the equal, carry, and overflow status bits reset. Refer to a subsequent paragraph for additional application notes.

3.18 INCREMENT BY TWO INCT

Op Code: 05C0

Addressing mode: Format VI

Format:



Syntax definition:

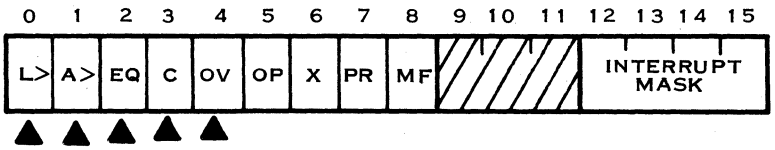
[<label>]b . . . INCTb . . . <ga_s>b . . . [<comment>]

Example:

```
LABEL   INCT   3              ADD 2 TO THE CONTENTS OF WSR3
```

Definition: Add a value of two to the source operand and replace the source operand with the sum. The AU compares the sum to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry out of bit zero, the carry status bit sets. When there is an overflow, (the sum cannot be represented in a 16-bit word as a two's complement value), the overflow status bit sets.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results: $(ga_s) + 2 \rightarrow (ga_s)$

Use the INCT instruction to count and index *word* arrays, and add the value of two to an addressable memory location. For example, if workspace register 5 contains the address (2100_{16}) of the fifteenth word of an array, the instruction

```
INCT   5
```

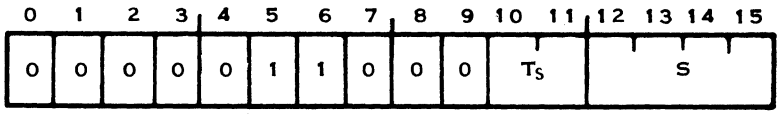
changes workspace register 5 to 2102_{16}, which points to the sixteenth word of the array. The logical greater than and arithmetic greater than status bits are set while the equal, carry, and overflow status bits are reset. Refer to a subsequent paragraph for additional application notes.

3.19 DECREMENT DEC

Op Code: 0600

Addressing mode: Format VI

Format:



Syntax definition:

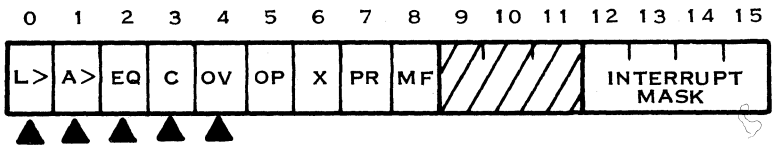
[<label>]b . . . DECb . . . <ga_s>b . . . [<comments>]

Example:

```
LABEL   DEC   2          SUBTRACT 1 FROM THE CONTENTS OF WSR2
```

Definition: Subtract a value of one from the source operand and replace the source operand with the result. The AU compares the result to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry out of bit zero, the carry status bit sets. When there is an overflow (the difference cannot be represented in a word as a two's complement value), the overflow status bit sets.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results: $(ga_s) - 1 \rightarrow (ga_s)$

Application notes: Use the DEC instruction to subtract a value of one from any addressable operand. The DEC instruction is also useful in counting and indexing *byte* arrays. For example, if COUNT contains a value of 1_{16}, then

```
DEC     @COUNT
```

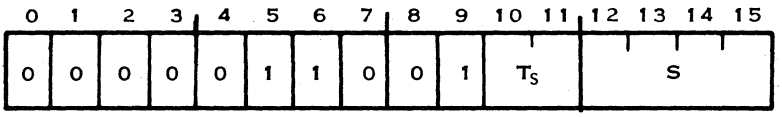
results in a value of zero in location COUNT and sets the equal and carry status bits while resetting the logical greater than, arithmetic greater than, and overflow status bits. The carry bit is always set except on transition from zero to minus one. Refer to a subsequent paragraph for additional application notes.

3.20 DECREMENT BY TWO DECT

Op Code: 0640

Addressing mode: Format VI

Format:



Syntax definition:

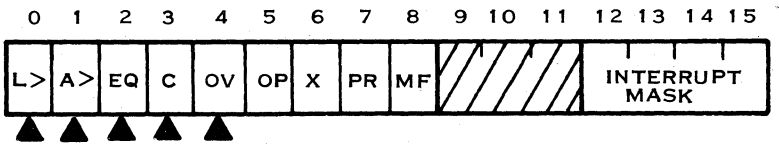
[<label>]b . . . DECTb . . . <ga_s>b . . . [<comment>]

Example:

```
LABEL   DECT   @ADDR        SUBTRACT 2 FROM THE CONTENTS OF ADDR
```

Definition: Subtract two from the source operand and replace the source operand with the result. The AU compares the result to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry out of bit zero, the carry status bit sets. When there is an overflow (the result cannot be represented in a word as a two's complement value), the overflow status bit sets.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results: $(ga_s) - 2 \rightarrow (ga_s)$

Application notes: The DECT instruction is useful in counting and indexing *word* arrays. Also, use the DECT instruction to subtract a value of two from any addressable operand. For example, if workspace register PRT (PRT equals 3) contains a value of $2C10_{16}$, then the instruction

```
DECT    PRT
```

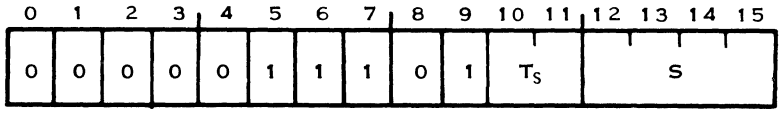
changes the contents of workspace register 3 to $2C0E_{16}$. The logical greater than, arithmetic greater than and carry status bits set while the equal and overflow status bits reset. Refer to a subsequent paragraph for additional application notes.

3.21 ABSOLUTE VALUE ABS

Op Code: 0740

Addressing mode: Format VI

Format:



Syntax definition:

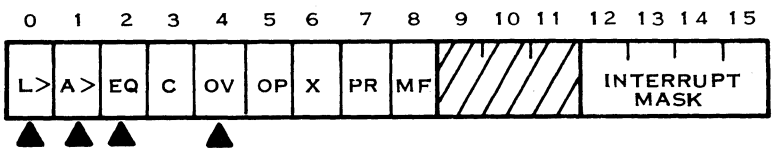
[<label>]b . . . ABSb . . . <ga_s>b . . . [<comment>]

Example:

```
LABEL   ABS   *2          REPLACE THE CONTENTS OF THE INDIRECT
                          ADDRESS OF WSR2 WITH ITS ABSOLUTE VALUE
```

Definition: Compute the absolute value of the source operand and replace the source operand with the result. The absolute value is the two's complement of the source operand when the sign bit (bit zero) is equal to one. When the sign bit is equal to zero, the source operand is unchanged. The AU compares the original source operand to zero and sets/resets the status bits to indicate the results of the comparison.

Status bits affected: Logical greater than, arithmetic greater than, equal, and overflow.



Execution results: $|(ga_s)| \rightarrow (ga_s)$

Application notes: Use the ABS instruction to take the absolute value of an operand. For example, if the third word in array LIST contains the value $FF3C_{16}$ and workspace register seven contains the value 4_{16}, then the instruction

```
ABS     @LIST(7)
```

changes the contents of the third word in array LIST to $00C4_{16}$. The logical greater than status bit sets while the arithmetic greater than and equal status bits reset. The overflow bit is set if the operand is 8000_{16}; otherwise, it is reset. Refer to a subsequent paragraph for additional application notes.

Multiple CPU Systems: Several 990/10 CPUs can be connected together to create a multiple CPU systems. In these systems, the CPUs must share a common memory. Simultaneous access attempts to memory by more than one CPU can result in a loss of data. To prevent this conflict, software "memory busy" flags in memory can be used. When a program desires access to memory, it must first check the flag to determine if any other program is actively using memory. If memory is not busy, the program sets the busy flag to lock out other programs and begins its memory transfers. When the program is finished with memory, it clears the busy flag to allow access to other programs.

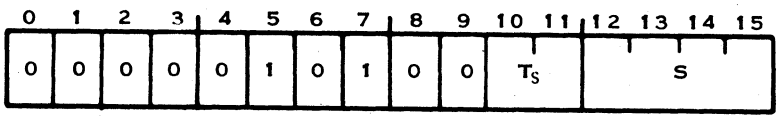
However, the busy flag system is not fool proof. If two CPUs check the status of the busy flag in successive memory cycles, each CPU proceeds as if it has exclusive access to memory. This conflict occurs because the first CPU does not set the flag until after the second CPU reads it. All instructions in the 990 instruction set, except one, allow this problem to occur since they release memory which executing the instruction (i.e., while checking the state of the busy flag). However, the ABS instruction maintains control over memory even during execution of the instruction after the flag has been fetched from memory. This feature prevents other programs from accessing memory until the first program has evaluated the flag and has had a chance to change it. Therefore, use the ABS instruction to examine memory busy flags in all memory-sharing applications.

3.22 NEGATE NEG

Op Code: 0500

Addressing mode: Format VI

Format:



Syntax definition:

[<label>]b . . . NEGb . . . <ga_s>b . . . [<comment>]

Example:

```
LABEL   NEG  2          REPLACE CONTENTS OF WSR2 WITH ITS
                        ADDITIVE INVERSE
```

Definition: Replace the source operand with the two's complement of the source operand. The AU determines the two's complement value by inverting all bits of the source operand and adding one to the resulting word. The AU then compares the result to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected: Logical greater than, arithmetic greater than, equal, and overflow.

Execution results: - $(ga_s) \rightarrow (ga_s)$

Application notes: Use the NEG instruction to make the contents of an addressable memory location its additive inverse. For example, if workspace register 5 contains the value $A342_{16}$, then the instruction

```
NEG    5
```

changes the contents of workspace register 5 to $5CBE_{16}$. The logical greater than and arithmetic greater than status bits set while the equal status bit resets. The overflow bit is set if the operand is 8000_{16}; otherwise, it resets.

3.23 JUMP AND BRANCH INSTRUCTIONS

Branch instructions transfer control either unconditionally, or conditionally according to the state of one or more status bits of the status register. Table 3-3 lists the conditional branch (jump) instructions and shows the status bit or bits tested.

Table 3-3. Status Bits Tested by Instructions

Mnemonic	L>	A>	EQ	C	OV	OP	Jump if:
JH	X	–	X	–	–	–	L> = 1 and EQ = 0
JL	X	–	X	–	–	–	L> = 0 and EQ = 0
JHE	X	–	X	–	–	–	L> = 1 or EQ = 1
JLE*	X	–	X	–	–	–	L> = 0 or EQ = 1
JGT	–	X	–	–	–	–	A> = 1
JLT	–	X	X	–	–	–	A> = 0 and EQ = 0
JEQ	–	–	X	–	–	–	EQ = 1
JNE	–	–	X	–	–	–	EQ = 0
JOC	–	–	–	X	–	–	C = 1
JNC	–	–	–	X	–	–	C = 0
JNO	–	–	–	–	X	–	OV = 0
JOP	–	–	–	–	–	X	OP = 1

* JLE is a logical comparison of jump if low or equal, not the arithmetic comparison.

For all jump instructions, a displacement of zero results in execution of the next instruction in sequence. A displacement of -1 results in execution of the same instruction (a single-instruction loop).

The instructions are:

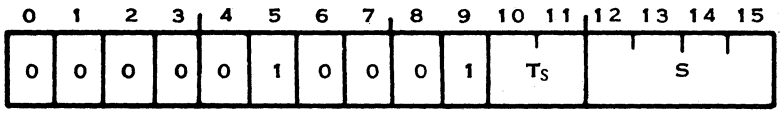
Instruction	Mnemonic	Paragraph
Branch	B	3.24
Branch and Link	BL	3.25
Branch and Load WP	BLWP	3.26
Jump if Equal	JEQ	3.35
Jump if Greater Than	JGT	3.33
Jump if High or Equal	JHE	3.31
Jump if Logical High	JH	3.29
Jump if Logical Low	JL	3.30
Jump if Low or Equal	JLE	3.32
Jump if Less Than	JLT	3.34
Unconditional Jump	JMP	3.28
Jump if No Carry	JNC	3.38
Jump if Not Equal	JNE	3.36
Jump if No Overflow	JNO	3.39
Jump if Odd Parity	JOP	3.40
Jump On Carry	JOC	3.37
Return WP	RTWP	3.27
Execute	X	3.41

3.24 BRANCH B

Op Code: 0440

Addressing mode: Format VI

Format:



Syntax definition:

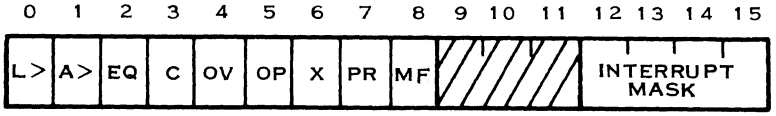
[<label>]b . . . Bb . . . <ga_s>b . . . [<comment>]

Example:

```
LABEL   B   @THERE          TRANSFER CONTROL TO LOCATION THERE
```

Definition: Replace the PC contents with the source address and transfer control to the instruction at that location.

Status bits affected: None



Execution results: $ga_s \rightarrow (PC)$

Application notes: Use the B instruction to transfer control to another section of code to change the linear flow of the program. For example, if the contents of workspace register 3 is $21CC_{16}$ then the instruction

```
B       *3
```

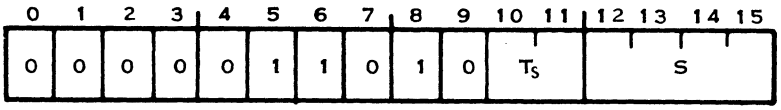
causes the word at location $21CC_{16}$ to be used as the next instruction, because this value replaces the contents of the PC when this instruction is executed.

3.25 BRANCH AND LINK BL

Op Code: 0680

Addressing mode: Format VI

Format:



Syntax definition:

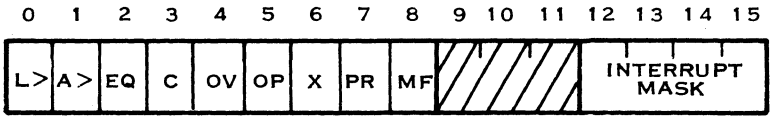
[<label>]b . . . BLb . . . <ga_s>b . . . [<comment>]

Example:

```
LABEL   BL   @SUBR          CALL SUBR AS A COMMON WS SUBROUTINE
```

Definition: Place the source address in the program counter, place the address of the instruction following the BL instruction (in memory) in workspace register 11, and transfer control to the new PC contents.

Status bits affected: None



Execution results: $ga_s \rightarrow$ (PC);

(old PC) $\rightarrow$ (Workspace register 11)

Application notes: Use the BL instruction when return linkage is required. For example, if the instruction

```
BL      @TRAN
```

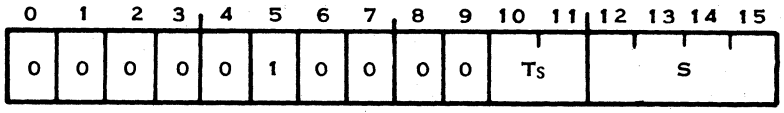
occurs at memory location (PC count) $04BC_{16}$, then this instruction has the effect of placing memory location TRAN in the PC and placing the value $04C0_{16}$ in workspace register 11. Refer to a subsequent paragraph for additional application notes.

3.26 BRANCH AND LOAD WORKSPACE POINTER BLWP

Op Code: 0400

Addressing mode: Format VI

Format:



Syntax definition:

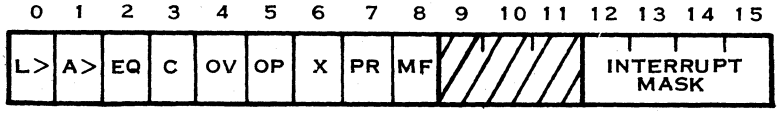
[<label>]b . . . BLWPb . . . <ga_s>b . . . [<comment>]

Example:

```
LABEL   BLWP   @VECT        BRANCH TO SUBROUTINE AT ADDRESS
                            (@VECT+2) AND EXECUTE CONTEXT SWITCH
```

Definition: Place the source operand in the WP and the word immediately following the source operand in the PC. Place the previous contents of the WP in the new workspace register 13, place the previous contents of the PC (address of the instruction following BLWP) in the new workspace register 14, and place the contents of the ST register in the new workspace register 15. When all store operations are complete, the AU transfers control to the new PC.

Status bits affected: None



Execution results: (ga_s) → (WP)
(ga_s + 2) → (PC)
(old WP) → (Workspace register 13)
(old PC) → (Workspace register 14)
(ST) → (Workspace register 15)

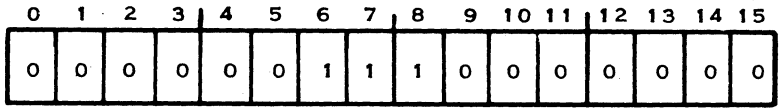
Application notes: Use the BLWP instruction for linkage to subroutines, program modules, or other programs that do not necessarily share the calling program workspace. Refer to a subsequent paragraph for a detailed explanation and example.

3.27 RETURN WITH WORKSPACE POINTER RTWP

Op Code: 0380

Addressing mode: Format VII

Format:



Syntax definition:

[<label>]b . . . RTWPb . . . [<comment>]

Example:

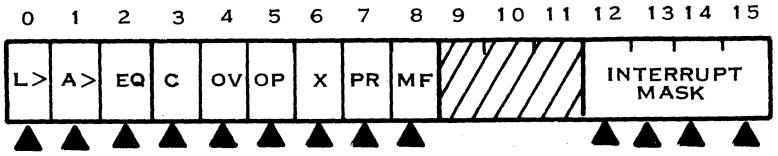
```
LABEL   RTWP                    RETURN FROM SUBROUTINE CALLED BY BLWP
```

Definition: Replace the contents of the WP register with the contents of the current workspace register 13. Replace the contents of the PC with the contents of the current workspace register 14. Replace the contents of the ST register with the contents of the current workspace register 15. The effect of this instruction is to restore the execution environment that existed prior to an interrupt, a BLWP instruction, or an XOP instruction.

Model 990/10 Computer: In the Model 990/10 Computer with the Privileged Mode bit (bit 7) of the ST register set to 1, only bits 0 through 5 of workspace register 15 are placed in bits 0 through 5 of the ST register. When bit 7 of the ST register is set to 0, the instruction places bits 0-8 and 12-15 of workspace register 15 into bits 0-8 and 12-15 of the ST register.

Model 990/4 Computer: In the Model 990/4 Computer, bits 0-7 and 12-15 of workspace register 15 are placed in bits 0-7 and 12-15 of the ST register.

Status bits affected: Restores all status bits to the value contained in workspace register 15.



Execution results: (Workspace register 13) → (WP)
(Workspace register 14) → (PC)
(Workspace register 15) → (ST)

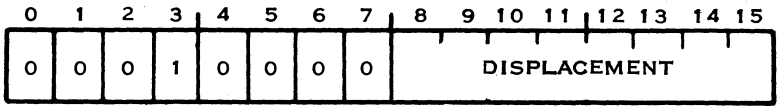
Application notes: Use the RTWP instruction to restore the execution environment after the completion of execution of an interrupt, a BLWP instruction, or an XOP instruction. Refer to a subsequent paragraph for additional information.

3.28 UNCONDITIONAL JUMP JMP

Op Code: 1000

Addressing mode: Format II

Format:



Syntax definition:

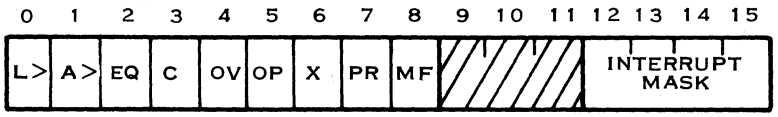
[<label>]b . . . JMPb . . . <exp>b . . . [<comment>]

Example:

```
LABEL   JMP   NXTLBL          JUMP TO NXTLBL
```

Definition: Add the signed displacement in the instruction word to the PC and replace the PC with the sum.

Status bits affected: None



Execution results: (PC) + Displacement → (PC)

The PC is incremented to the address of the next instruction prior to execution of an instruction. The execution results of jump instructions refer to the PC contents after the contents have been incremented to address the next instruction in sequence. The displacement (in words) is shifted to the left one bit position to orient the word displacement to the word address, and added to the PC contents.

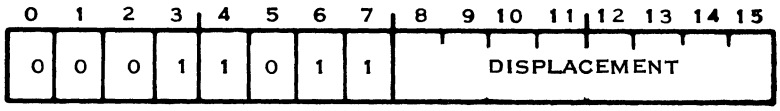
Application notes: Use the JMP instruction to transfer control to another section of the program module.

3.29 JUMP IF LOGICAL HIGH JH

Op Code: 1B00

Addressing mode: Format II

Format:



Syntax definition:

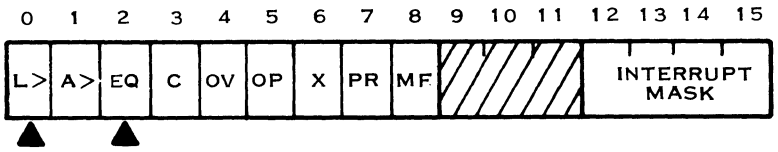
[<label>]b . . . JHb . . . <exp>b . . . [<comment>]

Example:

```
LABEL   JH   CONT          IF L> AND NOT EQ SKIP TO CONT
```

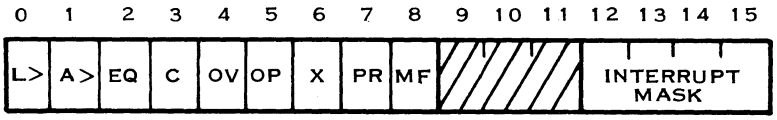
Definition: When the equal status bit is reset and the logical greater than status bit is set, add the signed displacement in the instruction word to the contents of the PC and replace the PC with the sum.

Status bits tested:



Jump if: L> = 1 and EQ = 0

Status bits affected: None



Execution results: If logical greater than bit is equal to 1 and equal bit is equal to 0: (PC) + Displacement → (PC).

If logical greater than bit is equal to 0 or equal bit is equal to 1: (PC) → (PC).

Refer to explanation of execution in paragraph 3.28.

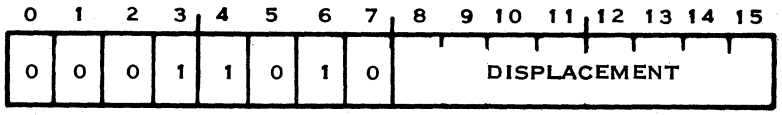
Application notes: Use the JH instruction to transfer control when the equal status bit is reset and the logical status bit is set.

3.30 JUMP IF LOGICAL LOW JL

Op Code: 1A00

Addressing mode: Format II

Format:



Syntax definition:

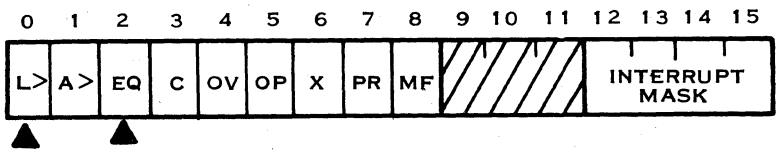
[<label>]b . . . JLb . . . <exp>to . . . [<comment>]

Example:

```
LABEL   JL   PREVLB          IF L> AND EQ ARE LOW, JUMP TO PREVLB
```

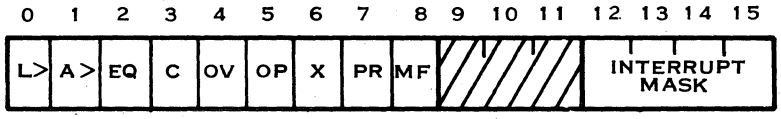
Definition: When the equal and logical greater than status bits are reset, add the signed displacement in the instruction word to the PC contents and replace the PC with the sum.

Status bits tested:



Jump if: L> = 0 and EQ = 0

Status bits affected: None



Execution results: If logical greater than bit and equal bit are equal to 0: (PC) + Displacement → (PC).

If logical greater than bit is equal to 1 or equal bit is equal to 1: (PC) → (PC).

Refer to explanation of execution in paragraph 3.28.

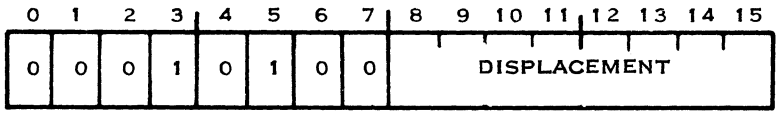
Application notes: Use the JL instruction to transfer control when the equal and logical greater than status bits are reset.

3.31 JUMP IF HIGH OR EQUAL JHE

Op Code: 1400

Addressing mode: Format II

Format:



Syntax definition:

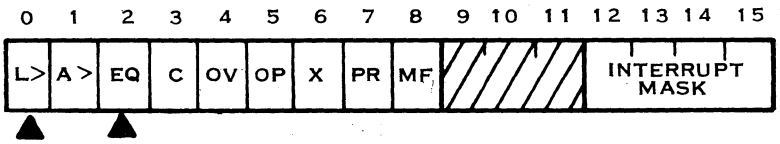
[<label>]b . . . JHEb . . . <exp>b . . . [<comment>]

Example:

```
LABEL   JHE   LABEL          LOOP HERE UNTIL EQ AND L> ARE RESET
```

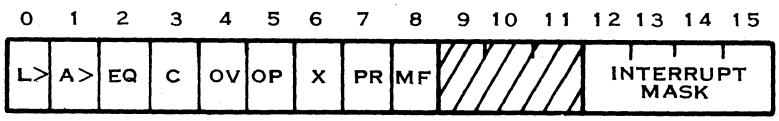
Definition: When the equal status bit or the logical greater than status bit is set, add the signed displacement in the instruction word to the PC and replace the contents of the PC with the sum.

Status bits tested:



Jump if: L> = 1 or EQ = 1

Status bits affected: None



Execution results: If logical greater than bit is equal to 1 or equal bit is equal to 1: (PC) + Displacement → (PC).

If logical greater than bit and equal bit are equal to 0: (PC) → (PC).

Refer to explanation of execution in paragraph 3.28.

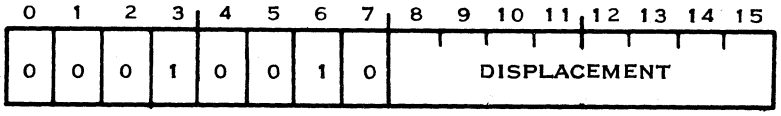
Application notes: Use the JHE instruction to transfer control when either the logical greater than or equal status bit is set.

3.32 JUMP IF LOW OR EQUAL JLE

Op Code: 1200

Addressing mode: Format II

Format:



Syntax definition:

[<label>]b . . . JLEb . . . <exp>b . . . [<comment>]

Example:

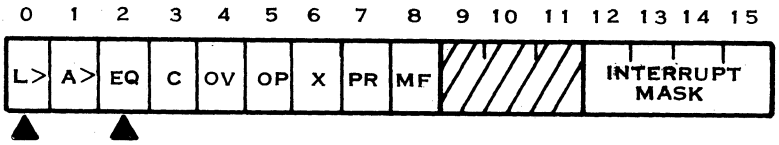
```
LABEL JLE THERE          JUMP TO THERE WHEN EQ=1 or L>=0
```

Definition: When the equal status bit is set or the logical greater than status bit is reset, add the signed displacement in the instruction word to the contents of the PC and replace the PC with the sum.

NOTE

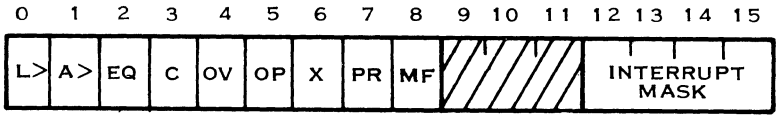
JLE is not jump if less than or equal.

Status bits tested:



Jump if: L> = 0 or EQ = 1

Status bits affected: None



Execution results: If logical greater than bit is equal to 0 or equal bit is equal to 1: (PC) + Displacement → (PC).

If logical greater than bit is equal to 1 and equal bit is equal to 0: (PC) → (PC).

Refer to explanation of execution in paragraph 3.28.

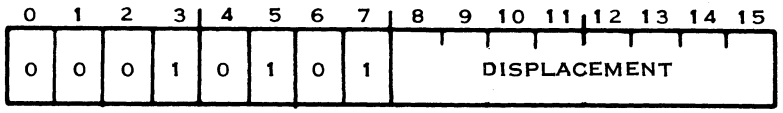
Application notes: Use the JLE instruction to transfer control when the equal status bit is set or the logical greater than status bit is reset.

3.33 JUMP IF GREATER THAN JGT

Op Code: 1500

Addressing mode: Format II

Format:



Syntax definition:

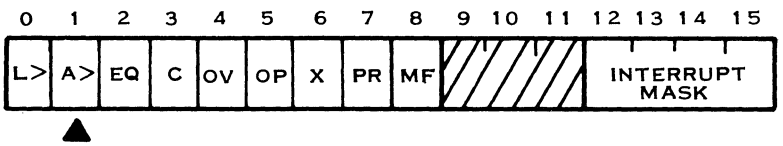
[<label>]b . . . JGTb . . . <exp>b . . . [<comment>]

Example:

```
LABEL   JGT   THERE          JUMP TO THERE IF A>=1
```

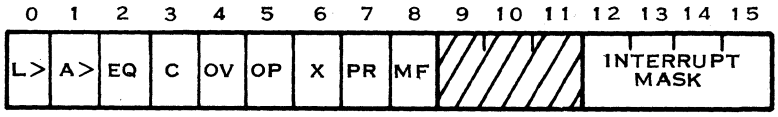
Definition: When the arithmetic greater than status bit is set, add the signed displacement in the instruction word to the PC and place the sum in the PC. Transfer control to the new PC location.

Status bits tested:



Jump if: A> = 1

Status bit affected: None



Execution results: If arithmetic greater than bit is equal to 1: (PC) + Displacement → (PC).

If arithmetic greater than bit is equal to 0: (PC) → (PC).

Refer to explanation of execution in paragraph 3.28.

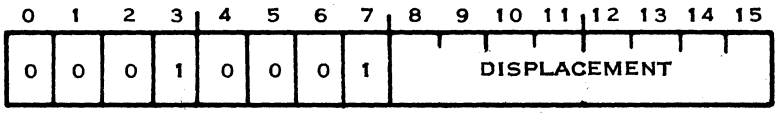
Application notes: Transfers control if the arithmetic greater than status bit is set.

3.34 JUMP IF LESS THAN JLT

Op Code: 1100

Addressing mode: Format II

Format:



Syntax definition:

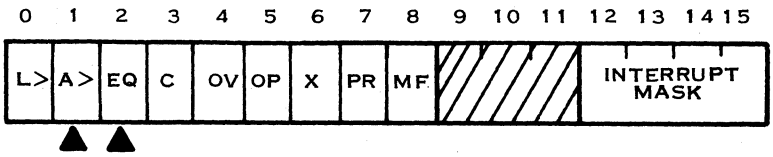
[<label>]b . . . JLTb . . . <exp>b . . . [<comment>]

Example:

```
LABEL   JLT   THERE          JUMP TO THERE IF A>=0 AND EQ=0
```

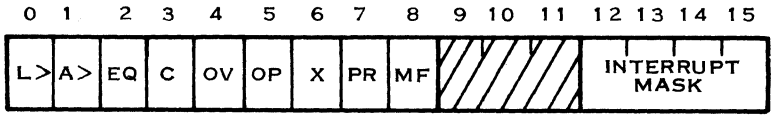
Definition: When the equal and arithmetic greater than status bits are reset, add the signed displacement in the instruction word to the PC and replace the PC contents with the sum.

Status bits tested:



Jump if: A> = 0 and EQ = 0

Status bits affected: None



Execution results: If arithmetic greater than bit and equal bit are equal to 0: (PC) + Displacement → (PC).

If arithmetic greater than bit is equal to 1 or equal bit is equal to 1: (PC) → (PC).

Refer to explanation of execution in paragraph 3.28.

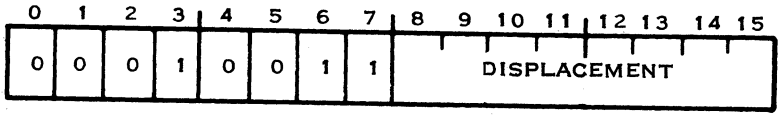
Application notes: Use the JLT instruction to transfer control when the equal and arithmetic greater than status bits are reset.

3.35 JUMP IF EQUAL JEQ

Op Code: 1300

Addressing mode: Format II

Format:



Syntax definition:

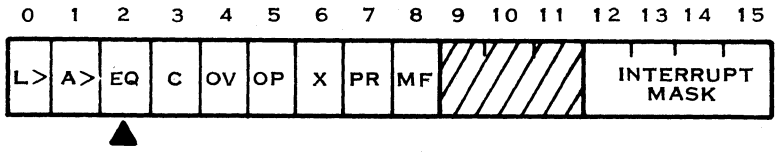
[<label>]b . . . JEQb . . . <exp>b . . . [<comment>]

Example:

```
LABEL   JEQ   LOC             JUMP TO LOC IF EQ=1
```

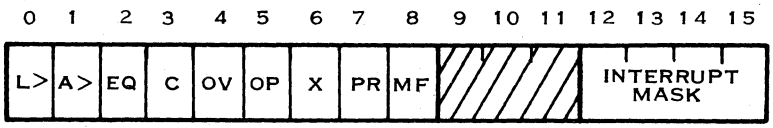
Definition: When the equal status bit is set, transfer control by adding the signed displacement in the instruction word to the program counter and then place the sum in the PC to transfer control.

Status bits tested:



Jump if: EQ = 1

Status bits affected: None



Execution results: If equal bit is equal to 1: (PC) + Displacement → (PC).

If equal bit is equal to 0: (PC) → (PC).

Refer to explanation of execution in paragraph 3.28.

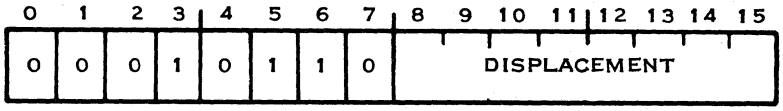
Application notes: Use the JEQ instruction to transfer control when the equal status bit is set and to test CRU bits.

3.36 JUMP IF NOT EQUAL JNE

Op Code: 1600

Addressing mode: Format II

Format:



Syntax definition:

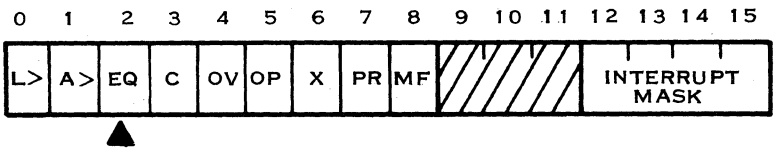
[<label>]b . . . JNEb . . . <exp>b . . . [<comment>]

Example:

```
LABEL   JNE   LOC2          JUMP TO LOC2 IF EQ=0
```

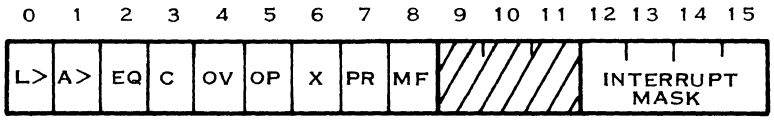
Definition: When the equal status bit is reset, add the signed displacement in the instruction word to the PC and replace the PC with the sum.

Status bits tested:



Jump if: EQ = 0

Status bits affected: None



Execution results: If equal bit is equal to 0: (PC) + Displacement → (PC).

If equal bit is equal to 1: (PC) → (PC).

Refer to explanation of execution in paragraph 3.28.

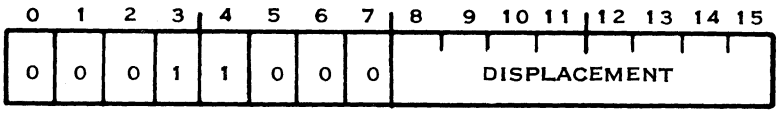
Application notes: Use the JNE instruction to transfer control when the equal status bit is reset. The JNE instruction is also useful in testing CRU bits.

3.37 JUMP ON CARRY JOC

Op Code: 1800

Addressing mode: Format II

Format:



Syntax definition:

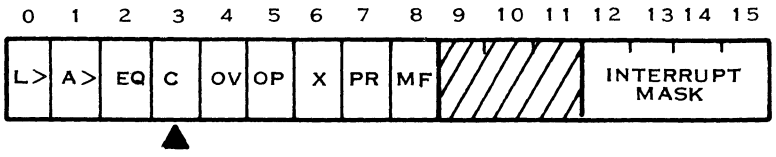
[<label>]b . . . JOCb . . . <exp>b . . . [<comment>]

Example:

```
LABEL   JOC   PROCED          IF C=1 SKIP TO PROCED
```

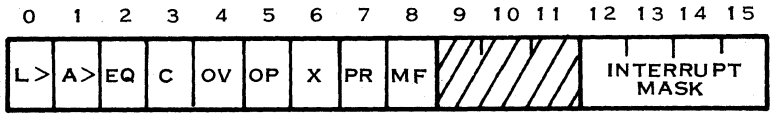
Definition: When the carry status bit is set, add the signed displacement in the instruction word to the PC and replace the PC with the sum.

Status bits tested:



Jump if: C = 1

Status bits affected: None



Execution results: If carry bit is equal to 1: (PC) + Displacement → (PC).

If carry bit is equal to 0: (PC) → (PC).

Refer to explanation of execution in paragraph 3.28.

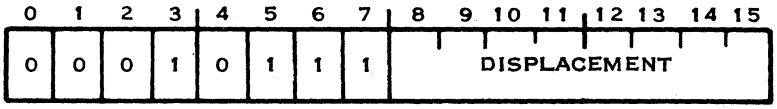
Application notes: Use the JOC instruction to transfer control when the carry status bit is set.

3.38 JUMP IF NO CARRY JNC

Op Code: 1700

Addressing mode: Format II

Format:



Syntax definition:

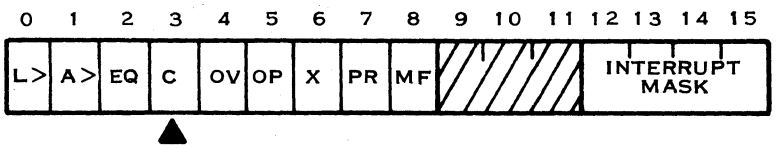
[<label>]b . . . JNCb . . . <exp>b . . . [<comment>]

Example:

```
LABEL   JNC   NONE          JUMP TO NONE IF C=0
```

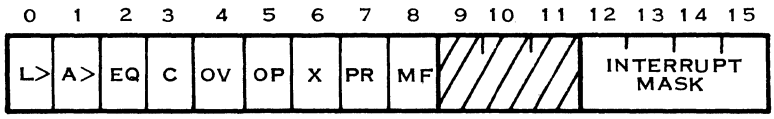
Definition: When the carry status bit is reset, add the signed displacement in the instruction word to the PC and replace the PC with the sum.

Status bits tested:



Jump if: C = 0

Status bits affected: None



Execution results: If carry bit is equal to 0: (PC) + Displacement → (PC).

If carry bit is equal to 1: (PC) → (PC).

Refer to explanation of execution in paragraph 3.28.

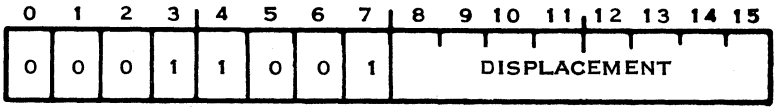
Application notes: Use the JNC instruction to transfer control when the carry status bit is reset.

3.39 JUMP IF NO OVERFLOW JNO

Op Code: 1900

Addressing mode: Format II

Format:



Syntax definition:

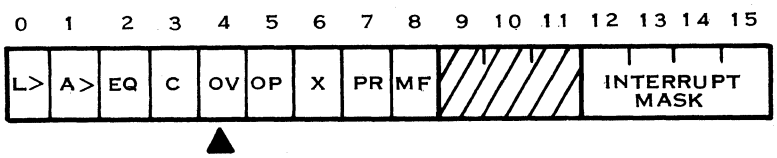
[<label>]b . . . JNOb . . . <exp>b . . . [<comment>]

Example:

```
LABEL   JNO   NORML          JUMP TO NORML IF OV=0
```

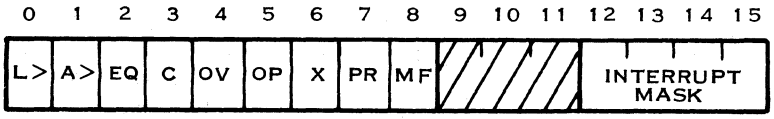
Definition: When the overflow status bit is reset, add the signed displacement in the instruction word to the PC and replace the PC with the sum.

Status bits tested:



Jump if: OV = 0

Status bits affected: None



Execution results: If overflow bit is equal to 0: (PC) + Displacement → (PC).

If overflow bit is equal to 1: (PC) → (PC).

Refer to explanation of execution in paragraph 3.28.

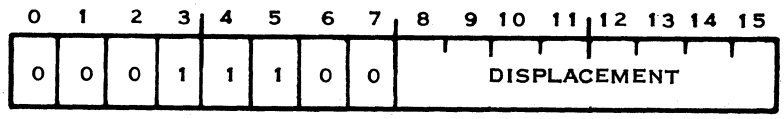
Application notes: Use the JNO instruction to transfer control when the overflow status bit is reset. JNO normally transfers control during arithmetic sequences where addition, subtraction, incrementing, and decrementing may cause an overflow condition. JNO may also be used following an SLA (Shift Left Arithmetic) operation. If, during the SLA execution, the sign of the workspace register being shifted changes (+ to -, - to +), the overflow status bit sets. This feature permits transfer, after a sign change, to error correction routines or to another functional code sequence.

3.40 JUMP IF ODD PARITY JOP

Op Code: 1C00

Addressing mode: Format II

Format:



Syntax definition:

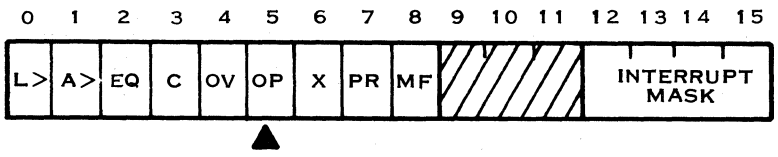
[<label>]b . . . JOPb . . . <exp>b . . . [<comment>]

Example:

```
LABEL   JOP   THERE          JUMP TO THERE IF OP=1
```

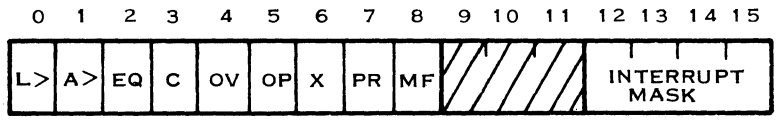
Definition: When the odd parity status bit is set, add the signed displacement in the instruction word to the PC and replace the PC with the sum.

Status bits tested:



Jump if: OP = 1

Status bits affected: None



Execution results: If odd parity bit is equal to 1: (PC) + Displacement → (PC).

If odd parity bit is equal to 0: (PC) → (PC).

Refer to explanation of execution in paragraph 3.28.

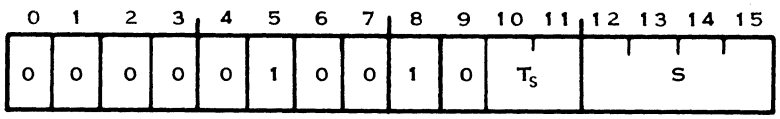
Application notes: Use the JOP instruction to transfer control when there is odd parity. Odd parity indicates that there is an odd number of logic one bits in the byte tested. JOP transfers control if the byte tested contains an odd number (sum) of logic one bits. This instruction may be used in data transmissions where the parity of the transmitted byte is used to ensure the validity of the received character at the point of reception.

3.41 EXECUTE X

Op Code: 0480

Addressing mode: Format VI

Format:



Syntax definition:

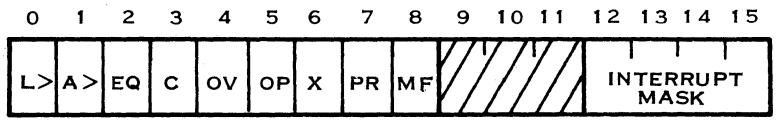
[<label>]b . . . Xb . . . <ga_s>b . . . [<comment>]

Example:

```
LABEL   X   2          EXECUTE THE CONTENTS OF WSR2
```

Definition: Execute the source operand as an instruction. When the source operand is not a single word instruction, the word or words following the execute instruction are used with the source operand as a 2-word or 3-word instruction. The source operand, when executed as an instruction, may affect the contents of the status register. The PC increments by either one, two, or three words depending upon the source operand. If the executed instruction is a branch, the branch is taken. If the executed instruction is a jump and if the conditions for a jump (i.e. the status test indicates a jump) are satisfied, then the jump is taken relative to the location of the X instruction.

Status bits affected: None, but substituted instruction affects status bits normally.



Execution results: An instruction at ga_s is executed instead of the X instruction.

Application notes: Use the X instruction to execute the source operand as an instruction. This is primarily useful when the instruction to be executed is dependent upon a variable factor. Refer to a subsequent paragraph for additional application notes.

3.42 COMPARE INSTRUCTIONS

Compare instructions have no effect other than the setting or resetting of appropriate status bits in the status register. The compare instructions perform both arithmetic and logical comparisons. The arithmetic comparison is of the two operands as two's complement values and the logical comparison is of the two operands as unsigned magnitude values. The instructions are:

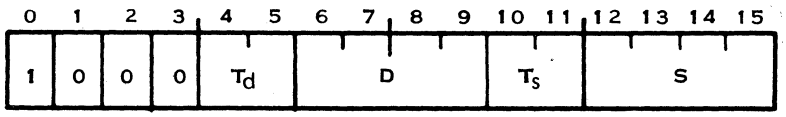
Instruction	Mnemonic	Paragraph
Compare Words	C	3.43
Compare Bytes	CB	3.44
Compare Immediate	CI	3.45
Compare Ones Corresponding	COC	3.46
Compare Zeros Corresponding	CZC	3.47

3.43 COMPARE WORDS C

Op Code: 8000

Addressing mode: Format I

Format:



Syntax definition:

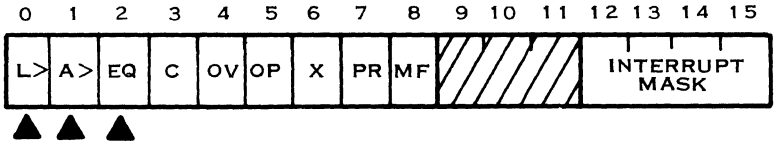
[<label>]b . . . Cb . . . <ga_s>,<ga_d>b . . . [<comment>]

Example:

```
LABEL   C   2, 3          COMPARE THE CONTENTS OF WSR2 AND WSR3
```

Definition: Compare the source operand (word) with the destination operand (word) and set/reset the status bits to indicate the results of the comparison. The arithmetic and equal comparisons compare the operand as signed, two's complement values. The logical comparison compares the two operands as unsigned, 16-bit magnitude values.

Status bits affected: Logical greater than, arithmetic greater than, and equal.



Execution results: $(ga_s) : (ga_d)$

Application notes: C compares the two operands as signed, two's complement values and as unsigned integers. Some examples are:

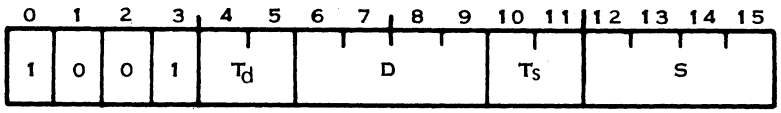
Source	Destination	Logical	Arithmetic	Equal
FFFF	0000	1	0	0
7FFF	0000	1	1	0
8000	0000	1	0	0
8000	7FFF	1	0	0
7FFF	7FFF	0	0	1
7FFF	8000	0	1	0

3.44 COMPARE BYTES CB

Op Code: 9000

Addressing mode: Format I

Format:



Syntax definition:

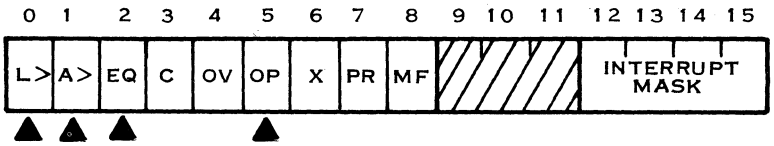
[<label>]b . . . CBb . . . <ga_s>,<ga_d>b . . . [<comment>]

Example:

```
LABEL   CB   2, 3          COMPARE THE LEFTMOST BYTES OF WSR2 AND
                           WSR3
```

Definition: Compare the source operand (byte) with the destination operand (byte) and set/reset the status bits according to the result of the comparison. CB uses the same comparison basis as does C. If the source operand contains an odd number of logic one bits, the odd parity status bit sets. The operands remain unchanged. If either operand is addressed in the workspace register mode, the byte addressed is the most significant byte.

Status bits affected: Logical greater than, arithmetic greater than, equal, and odd parity.



Execution results: $(ga_s) : (ga_d)$

Application notes: CB compares the two operands as signed, two's complement values or as unsigned integers. Some examples are:

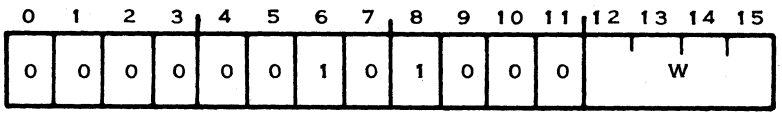
Source	Destination	Logical	Arithmetic	Equal	Odd Parity
FF	00	1	0	0	0
7F	00	1	1	0	1
80	00	1	0	0	1
80	7F	1	0	0	1
7F	7F	0	0	1	1
7F	80	0	1	0	1

3.45 COMPARE IMMEDIATE CI

Op Code: 0280

Addressing mode: Format VIII

Format:



Syntax definition:

[<label>]b . . . CIb . . . <wa>,<iop>b . . . [<comment>]

Example:

```
LABEL   CI   3, 7          COMPARE CONTENTS OF WSR3 TO 7
```

Definition: Compare the contents of the specified workspace register with the word in memory immediately following the instruction. Set/reset the status bits according to the comparison. CI makes the same type of comparison as does C.

Status bits affected: Logical greater than, arithmetic greater than, and equal.

Execution results: (wa) : iop

Application notes: Use the CI instruction to compare the workspace register to an immediate operand. For example, if the contents of workspace register 9 is 2183_{16}, then the instruction

```
CI      9,>F330
```

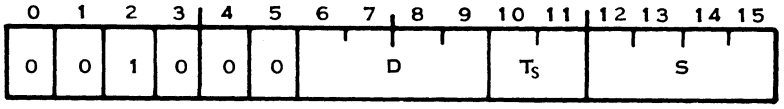
results in the arithmetic greater than status bit set and the logical greater than and equal status bits reset.

3.46 COMPARE ONES CORRESPONDING COC

Op Code: 2000

Addressing mode: Format III

Format:



Syntax definition:

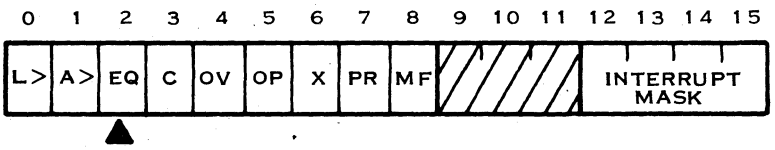
[<label>]b . . . COCb . . . <ga_s>,<wa_d>b . . . [<comment>]

Example:

```
LABEL   COC   @MASK, 2        DOES (WSR2) SATISFY MASK?
```

Definition: When the bits in the destination operand workspace register that correspond to the logic one bits in the source operand are equal to logic one, set the equal status bit. The source and destination operands are unchanged.

Status bit affected: Equal



Execution results: Equal bit set if all bits of (wa_d) that correspond to the bits of (ga_s) that are equal to 1 are also equal to 1.

Application notes: Use the COC instruction to test single/multiple bits within a word in a workspace register. For example, if TESTBI contains the word $C102_{16}$ and workspace register 8 contains the value $E306_{16}$, then the instruction

```
COC     @TESTBI,8
```

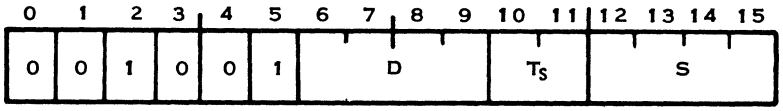
results in setting the equal status bit. If workspace register 8 were to contain $E301_{16}$, the equal status bit would reset. Use this instruction to determine if a workspace register has 1s in the bit positions indicated by 1s in a mask.

3.47 COMPARE ZEROS CORRESPONDING CZC

Op Code: 2400

Addressing mode: Format III

Format:



Syntax definition:

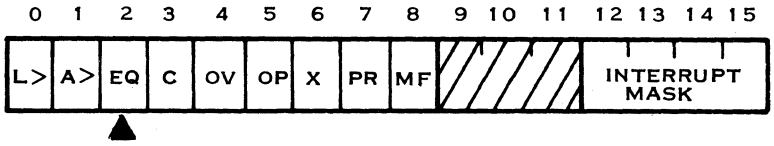
[<label>]b . . . CZCb . . . <ga_s>,<wa_d>b . . . [<comment>]

Example:

```
LABEL   CZC   @MASK, 2        DOES (WSR2) SATISFY THE MASK?
```

Definition: When the bits in the destination operand workspace register that correspond to the one bits in the source operand are all equal to a logic zero, set the equal status bit. The source and destination operands are unchanged.

Status bit affected: Equal



Execution results: Equal bit set if all bits of (wa_d) that correspond to the bits of (ga_s) that are equal to 1 are equal to 0.

Application notes: Use the CZC instruction to test single/multiple bits within a word in a workspace register. For example, if the memory location labeled TESTBI contains the value $C102_{16}$, and workspace register 8 contains 2301_{16}, then the instruction

```
CZC     @TESTBI, 8
```

results in the equal status bit reset. If workspace register 8 contained the value 2201_{16}, then the equal status bit would set. Use this instruction to determine if a workspace register has 0s in the positions indicated by 0s in a mask.

3.48 CONTROL AND CRU INSTRUCTIONS

Control instructions affect the operation of the Arithmetic Unit (AU) and the associated portions of the computer or microprocessor. CRU instructions affect the modules connected to the Communications Register Unit. The instructions are:

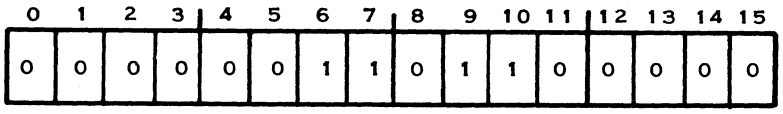
Instruction	Mnemonic	Paragraph
Clock Off	CKOF	3.51
Clock On	CKON	3.52
Load CRU	LDCR	3.57
Idle	IDLE	3.50
Load or Restart Execution	LREX	3.53
Reset	RSET	3.49
Set CRU Bit to Logic One	SBO	3.54
Set CRU Bit to Logic Zero	SBZ	3.55
Store CRU	STCR	3.58
Test Bit	TB	3.56

3.49 RESET RSET

Op Code: 0360

Addressing mode: Format VII

Format:



Syntax definition:

[<label>]b . . . RSETb . . . [<comment>]

Example:

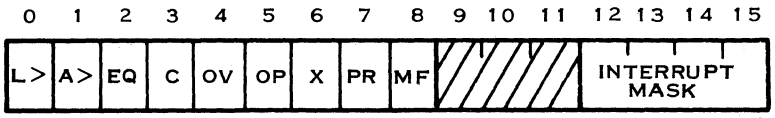
```
LABEL   RSET                    START OVER
```

Definition: The RSET instruction clears the interrupt mask, which disables all except level 0 interrupts. It also resets all directly connected input/output devices and those CRU devices that provide for reset in the interface with the CRU. RSET also resets all pending interrupts and turns the clock off.

TMS 9900 Microprocessor: Provides a signal that an RSET instruction is identified, but performs no processing. User may implement hardware to perform desired processing when the signal is present.

Model 990/10 Computer: When Privileged Mode bit (bit 7 of ST register) is set to 0, instruction executes normally. When Privileged Mode bit is set to 1, an error interrupt occurs when execution of an RSET instruction is attempted.

Status bits affected: None



Execution results: Clears the interrupt mask, resets directly connected I/O devices, resets the CRU devices that provide for reset in the interface with the CRU, resets pending interrupts, and turns the clock off.

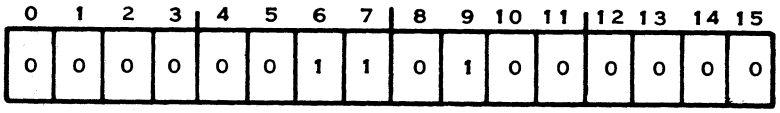
Application notes: Use the reset instruction to reset the interrupt mask to zero, turn off the clock, and (depending on the device and interface) clear any pending interrupt and reset interface electronics.

3.50 IDLE IDLE

Op Code: 0340

Addressing mode: Format VII

Format:



Syntax definition:

[<label>]b . . . IDLEb . . . [<comment>]

Example:

```
LABEL   IDLE                WAIT FOR INTERRUPT
```

Definition: Place the computer in the idle state. Note that the PC is incremented prior to the execution of this instruction and the contents of the PC point to the instruction word in memory immediately following the IDLE instruction. The computer will remain in the IDLE state until an interrupt, RESTART, or LOAD occurs.

TMS 9900 Microprocessor: Provides a signal that an IDLE instruction is being executed, and places the microprocessor in the idle mode. User may implement hardware to perform additional processing when the signal is present.

Model 990/10 Computer: When Privileged Mode bit (bit 7 of ST register) is set to 0, instruction executes normally. When Privileged Mode bit is set to 1, an error interrupt occurs when execution of an IDLE instruction is attempted.

Status bits affected: None

Execution results: Places the computer in the idle mode, suspending program execution until an interrupt occurs.

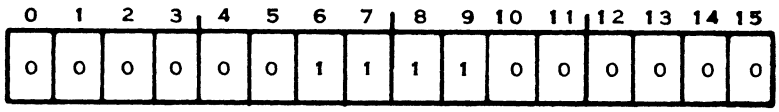
Application notes: Use the IDLE instruction to place the computer in the idle state. This instruction is useful in timing delays using the clock or in waiting for interrupt signals.

3.51 CLOCK OFF CKOF

Op Code: 03C0

Addressing mode: Format VII

Format:



Syntax definition:

[<label>]b . . . CKOFb . . . [<comment>]

Example:

```
STOCK   CKOF                    STOP THE CLOCK
```

Definition: Stop the line frequency clock (120 Hz). No status bits are changed and the clock interrupt will not occur as long as the clock is off.

TMS 9900 Microprocessor: Provides a signal that a CKOF instruction is identified, but performs no processing. User may implement hardware to perform desired processing when signal is present.

Model 990/10 Computer: When Privileged Mode bit (bit 7 of ST register) is set to 0, instruction executes normally. When Privileged Mode bit is set to 1, an error interrupt occurs when execution of a CKOF instruction is attempted.

990/4 Microcomputer: If a clock interrupt occurs during the execution of a CKOF instruction, the interrupt can be vectored incorrectly through level 15 instead of through the level to which it is connected. To avoid this situation, mask the clock interrupt before executing a CKOF instruction. The following sequence performs that function.

```
LIMI   0   Mask all interrupts
CKOF       Clock off
LIMI   n   Reset interrupt mask to desired level,n.
```

This sequence is not required if CKOF is used in the service routine for a clock interrupt because the clock interrupt causes the interrupt mask to be set to one level below the level of the clock interrupt.

Status bits affected: None

Execution results: Line frequency clock disabled, and the clock interrupt cleared.

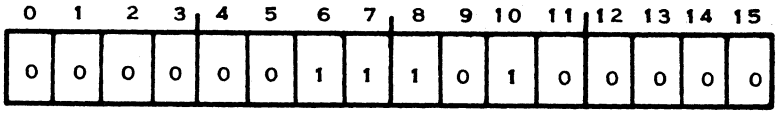
Application notes: Clock applications are described in paragraph 3.89.7.2.

3.52 CLOCK ON CKON

Op Code: 03A0

Addressing mode: Format VII

Format:



Syntax definition:

[<label>]b . . . CKONb . . . [<comment>]

Example:

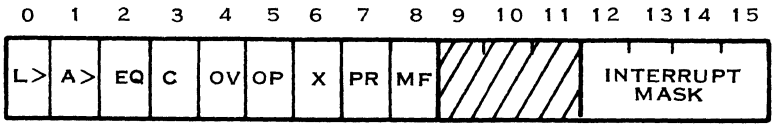
```
STRTC  CKON                START THE CLOCK
```

Definition: Enable the line frequency clock. If interrupt level five is enabled, an interrupt will occur every 8.33 ms after the initial interrupt, which may occur from 1μs to 8.33 ms after the clock is turned on. Interrupt five may be enabled/disabled by the interrupt mask as necessary.

TMS 9900 Microprocessor: Provides a signal that a CKON instruction is identified, but performs no processing. User may implement hardware to perform desired processing when signal is present.

Model 990/10 Computer: When Privileged Mode bit (bit 7 of ST register) is set to 0, instruction executes normally. When Privileged Mode bit is set to 1, an error interrupt occurs when execution of a CKON instruction is attempted.

Status bits affected: None



Execution results: Line frequency clock enabled.

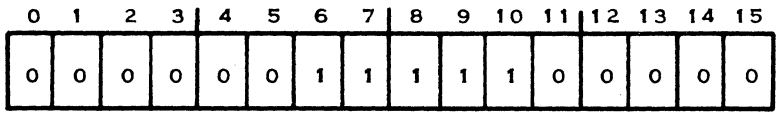
Application notes: Clock applications are described in paragraph 3.89.7.2.

3.53 LOAD OR RESTART EXECUTION LREX

Op Code: 03E0

Addressing mode: Format VII

Format:



Syntax definition:

[<label>]b . . . LREXb . . . [<comment>]

Example:

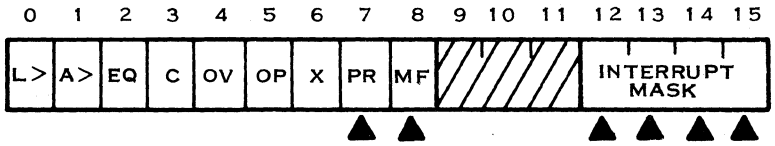
```
LABEL   LREX                    START ALL OVER
```

Definition: Place the contents of location $FFFC_{16}$ into the WP register and the contents of location $FFFE_{16}$ into the PC. Store the previous contents of the WP register, the PC, and the ST register into workspace registers 13, 14, and 15, respectively. Set the interrupt mask to 0, disabling all interrupt levels except level 0.

TMS 9900 Microprocessor: Provides a signal that an LREX instruction is identified, but performs no processing. User may implement hardware to perform desired processing when signal is present.

Model 990/10 Computer: The LREX instruction sets the Privileged Mode bit (bit 7) of the ST register to 0 in addition to performing the context switch. When the Privileged Mode bit is set to 0 prior to execution of an LREX instruction, the instruction executes normally. When the Privileged Mode bit is set to 1 and execution of an LREX instruction is attempted, an error interrupt occurs. When the map option is included, the LREX instruction also sets the Map File bit (bit 8) of the ST register to 0.

Status bits affected: Map File, Privilege, Interrupt Mask



Execution results:
(location $FFFC_{16}$) → (WP)
(location $FFFE_{16}$) → (PC)
(old WP) → (Workspace register 13)
(old PC) → (Workspace register 14)
(old ST) → (Workspace register 15)
0 → (Interrupt Mask) }
0 → (Map File) } Status Register
0 → (Privilege) }

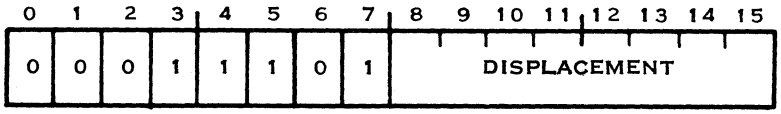
Application notes: Use the LREX instruction to perform a context switch using the transfer vector at location $FFFC_{16}$. Typically, the transfer vector transfers control to the front panel routine in Read Only Memory (ROM). Additional application information is included in a subsequent paragraph.

3.54 SET CRU BIT TO LOGIC ONE SBO

Op Code: 1D00

Addressing mode: Format II

Format:



Syntax definition:

[<label>]b . . . SBOb . . . <disp>b . . . [<comment>]

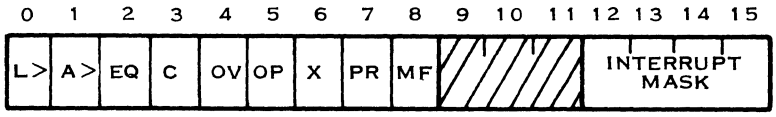
Example:

```
LABEL   SBO   7          SET BIT 7 ON CRU TO ONE
```

Definition: Set the digital output bit to a logic one on the CRU at the address derived from this instruction. The derived address is the sum of the user supplied signed displacement and the contents of workspace register 12, bits 3 through 14. The execution of this instruction does not affect the status register or the contents of workspace register 12.

Model 990/10 Computer: When the Privileged Mode bit (bit 7) of the ST register is set to 0, the SBO instruction executes normally. When bit 7 is set to 1 and the effective CRU address is equal to or greater than $E00_{16}$, an error interrupt occurs and the instruction is not executed.

Status bits affected: None



Execution results: CRU bit addressed by the sum of the contents of workspace register 12 + displacement is set to 1.

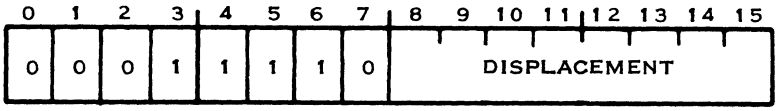
Application notes: Use the SBO instruction to set a CRU bit to a logic one. Refer to a subsequent paragraph for additional application notes.

3.55 SET CRU BIT TO LOGIC ZERO SBZ

Op Code: 1E00

Addressing mode: Format II

Format:



Syntax definition:

[<label>]b . . . SBZb . . . <disp>b . . . [<comment>]

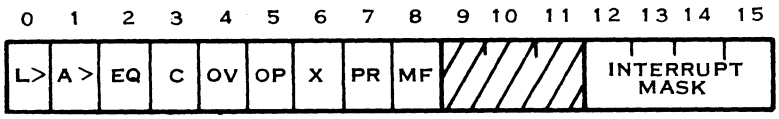
Example:

```
LABEL   SBZ   7              SET BIT 7 ON CRU TO ZERO
```

Definition: Set the digital output bit to a logic zero on the CRU at the address derived from this instruction. The derived address is the sum of the user supplied signed displacement and the contents of workspace register 12, bits 3 through 14. The execution of this instruction does not affect the status register or the contents of workspace register 12.

Model 990/10 Computer: When the Privileged Mode bit (bit 7) of the ST register is set to 0, the SBZ instruction executes normally. When bit 7 is set to 1 and the effective CRU address is equal to or greater than $E00_{16}$, an error interrupt occurs and the instruction is not executed.

Status bits affected: None



Execution results: CRU bit addressed by the sum of the contents of workspace register 12 (bits 3-14) + displacement is set to 0.

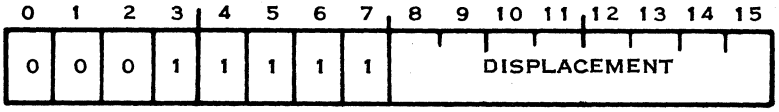
Application notes: Use the SBZ instruction to set a CRU bit to a logic zero. Refer to a subsequent paragraph for additional application notes.

3.56 TEST BIT TB

Op Code: 1F00

Addressing mode: Format II

Format:



Syntax definition:

[<label>]b . . . TBb . . . <disp>b . . . [<comment>]

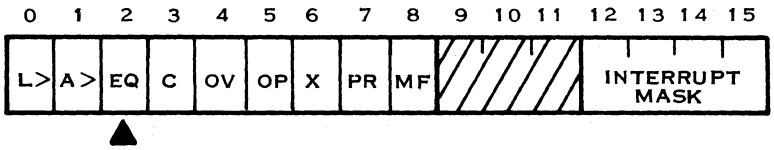
Example:

```
CHECK   TB   7          READ BIT 7 ON CRU AND SET EQUAL STATUS
                        BIT WITH THE VALUE READ
```

Definition: Read the digital input bit on the CRU at the address specified by the sum of the user supplied signed displacement and the contents of workspace register 12, bits 3 through 14 and set the equal status bit to the logic value read. The digital input bit and the contents of workspace register 12 are unchanged.

Model 990/10 Computer: When the Privileged Mode bit (bit 7) of the ST register is set to 0, the TB instruction executes normally. When bit 7 is set to 1 and the effective CRU address is equal to or greater than $E00_{16}$, an error interrupt occurs and the instruction is not executed.

Status bit affected: Equal



Execution results: Equal bit is set to the value of the CRU bit addressed by the sum of the contents of workspace register 12 (bits 3-12) + displacement.

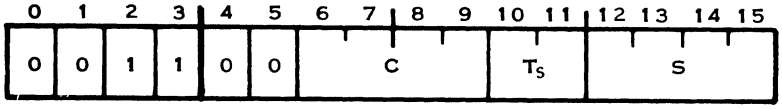
Application notes: TB CRU line logic level test transfers the logic level from the indicated CRU line to the equal status bit without modification. If the CRU line tested is set to a logic one, the equal status bit sets to a logic one and if the line is zero, sets to a zero. JEQ will then transfer control when the CRU line is a logic one and will not transfer control when the line is a logic zero. In addition, JNE will transfer control under the exact opposite conditions.

3.57 LOAD CRU LDCR

Op Code: 3000

Addressing mode: Format IV

Format:



Syntax definition:

[<label>]b . . . LDCRb . . . <ga_s>,<cnt>b . . . [<comment>]

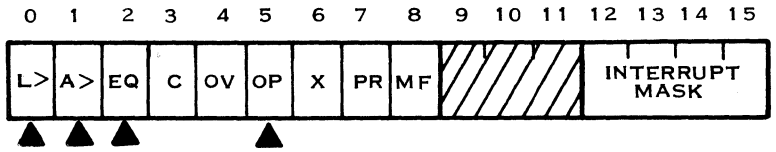
Example:

```
WRITE   LDCR   @BUFF, 15     SEND 15 BITS FROM BUFF TO CRU
```

Definition: Transfer the number of bits specified in the C field from the source operand to the CRU. The transfer begins with the least significant bit of the source operand. The CRU address is contained in bits 3 through 14 of workspace register 12. When the C field contains zero, the number of bits transferred is 16. If the number of bits to be transferred is from one to eight, the source operand address is a byte address. If the number of bits to be transferred is from 9 to 16, the source operand address is a word address. If the source operand address is odd, the address is truncated to an even address prior to data transfer. When the number of bits transferred is a byte or less, the source operand is compared to zero and the status bits are set/reset, according to the results of the comparison. The odd parity status bit sets when the bits in a byte (or less) to be transferred establish odd parity.

Model 990/10 Computer: When the Privileged Mode bit (bit 7) of the ST register is set to 0, the LDCR instruction executes normally. When bit 7 is set to 1 and the effective CRU address is equal to or greater than $E00_{16}$, an error interrupt occurs and the instruction is not executed.

Status bits affected: Logical greater than, arithmetic greater than, and equal. When C is less than 9, odd parity is also set or reset. Status is set according to the full word or byte, not just the transferred bits.



Execution results: Number of bits specified by C are transferred from memory at address ga_s to consecutive CRU lines beginning at the address in workspace register 12.

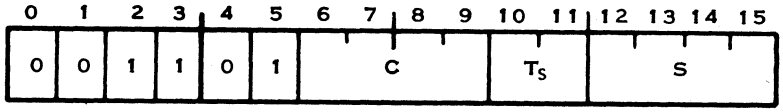
Application notes: Use the LDCR instruction to transfer a specific number of bits from memory to the CRU at the address contained in bits 3 through 14 of workspace register 12. Refer to a subsequent paragraph for a detailed example and explanation of the LDCR instruction.

3.58 STORE CRU STCR

Op Code: 3400

Addressing mode: Format IV

Format:



Syntax definition:

[<label>]b . . . STCRb . . . <ga_s>,<cnt>b . . . [<comment>]

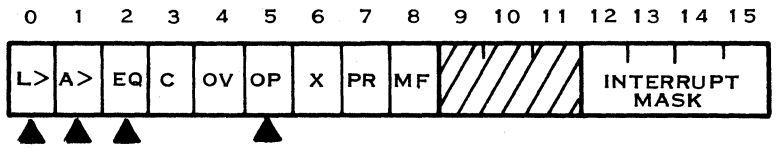
Example:

```
READ   STCR   @BUF, 9        READ 9 BITS FROM CRU AND STORE AT
                              LOCATION BUF
```

Definition: Transfer the number of bits specified in the C field from the CRU to the source operand. The transfer begins from the CRU address specified in bits 3 through 14 of workspace register 12 to the least significant bit of the source operand and fills the source operand toward the most significant bit. When the C field contains a zero, the number of bits to transfer is 16. If the number of bits to transfer is from one to eight, the source operand address is a byte address. Any bit in the memory byte not filled by the transfer is reset to a zero. When the number of bits to transfer is from 9 to 16, the source operand address is a word address. If the source operand address is odd, the address is truncated to an even address prior to data transfer. If the transfer does not fill the entire memory word, unfilled bits are reset to zero. When the number of bits to transfer is a byte or less, the bits transferred are compared to zero and the status bits set/reset to indicate the results of the comparison. Also, when the bits to be transferred are a byte or less, the odd parity bit sets when the bits establish odd parity.

Model 990/10 Computer: When the Privileged Mode bit (bit 7) of the ST register is set to 0, the STCR instruction executes normally. When bit 7 is set to 1 and the effective CRU address is equal to or greater than $E00_{16}$, an error interrupt occurs and the instruction is not executed.

Status bits affected: Logical greater than, arithmetic greater than, and equal. When C is less than 9, odd parity is also set or reset. Status is set according to the full word or byte, not just those bits transferred.



Execution results: Number of bits specified by C are transferred from consecutive CRU lines beginning at the address in workspace register 12 to memory at address ga_s.

Application notes: Use the STCR instruction to transfer a specified number of CRU bits from the CRU to memory location supplied by the user as the source operand. Note that the CRU base address must be in workspace register 12 prior to the execution of this instruction. Refer to a subsequent paragraph for a detailed explanation and examples of the use of the STCR instruction.

3.59 LOAD AND MOVE INSTRUCTIONS

Load and move instructions permit the user to establish the execution environment and the execution results. These instructions manipulate data between memory locations and between hardware registers and memory locations. The instructions are:

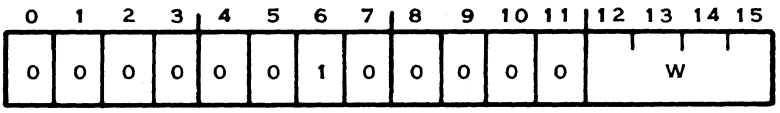
Instruction	Mnemonic	Paragraph
Load Immediate	LI	3.60
Load Interrupt Mask Immediate	LIMI	3.61
Load Memory Map File	LMF	3.63
Load Workspace Pointer Immediate	LWPI	3.62
Move Words	MOV	3.64
Move Bytes	MOVB	3.65
Store Status	STST	3.67
Store Workspace Pointer	STWP	3.68
Swap Bytes	SWPB	3.66

3.60 LOAD IMMEDIATE LI

Op Code: 0200

Addressing mode: Format VIII

Format:



Syntax definition:

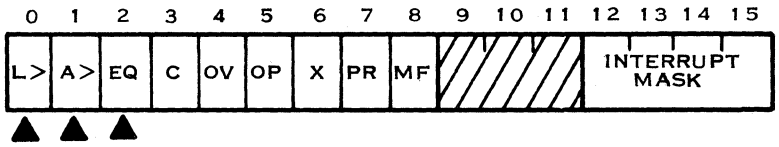
[<label>]b . . . LIb . . . <wa>,<iop>b . . . [<comment>]

Example:

```
GETIT   LI   3, >17          LOAD WSR3 WITH 17HEX=23
```

Definition: Place the immediate operand (the word of memory immediately following the instruction) in the user specified workspace register (W field). The immediate operand is not affected by the execution of this instruction. The immediate operand is compared to 0 and the L>, A>, and EQ status bits are set or reset according to the result of the comparison.

Status bits affected: Logical greater than, arithmetic greater than, and equal.



Execution results: iop → (wa)

Application notes: Use the LI instruction to place an immediate operand in a specified workspace register. This is useful for initializing a workspace register as a loop counter. For example, the instruction

```
LI      7,5
```

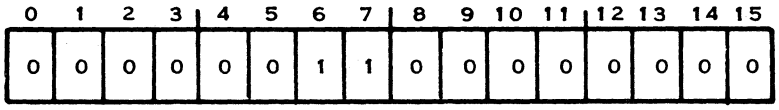
initializes workspace register 7 with the value 0005_{16}. L> and A> are set while EQ is reset in this example.

3.61 LOAD INTERRUPT MASK IMMEDIATE LIMI

Op Code: 0300

Addressing mode: Format VIII

Format:



Syntax definition:

[<label>]b . . . LIMIb . . . <iop>b . . . [<comment>]

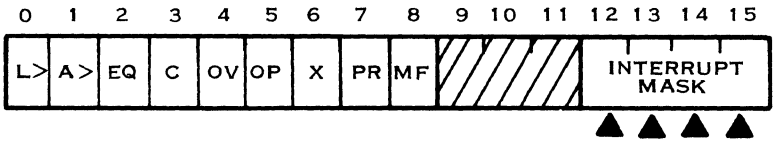
Example:

```
LABEL   LIMI  3             MASK LEVEL 3 AND BELOW
```

Definition: Place the low order four bits (bits 12-15) of the contents of the immediate operand (the next word after the instruction) in the interrupt mask of the status register. The remaining bits of the status register (0 through 11) are not affected.

Model 990/10 Computer: When Privileged Mode bit (bit 7 of ST register) is set to 0, instruction executes normally. When Privileged Mode bit is set to 1, an error interrupt occurs when execution of an LIMI instruction is attempted and the interrupt mask is not loaded.

Status bits affected: Interrupt Mask



Execution results: Places the four least significant bits of iop into the interrupt mask, the four least significant bits of the ST register.

Application notes: Use the LIMI instruction to initialize the interrupt mask for a particular level of interrupt to be accepted. For example, the instruction

```
LIMI  3
```

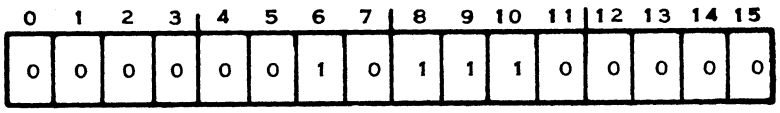
sets the interrupt mask to level three and enables interrupts at level 0, 1, 2, and 3.

3.62 LOAD WORKSPACE POINTER IMMEDIATE LWPI

Op Code: 02E0

Addressing mode: Format VIII

Format:



Syntax definition:

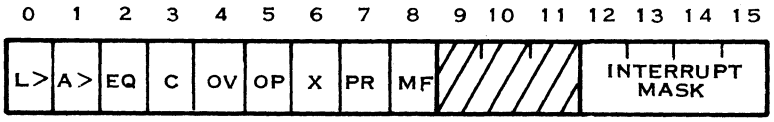
[<label>]b . . . LWPIb . . . <iop>b . . . [<comment>]

Example:

```
NEWWP  LWPI  02F2          02F2=NEWWP
```

Definition: Replace the contents of the WP with the immediate operand. The immediate operand is the word of memory immediately following the LWPI instruction.

Status bits affected: None



Execution results: iop → (WP)

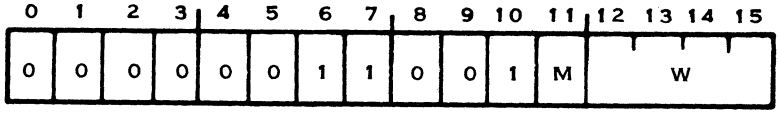
Application notes: Use the LWPI instruction to initialize or change the WP register to alter the workspace environment of the program module. The user should use either a BLWP or a LWPI instruction prior to the use of any workspace register in a program module.

3.63 LOAD MEMORY MAP FILE LMF

Op Code: 0320

Addressing mode: Format IX

Format:



This instruction is only available on the Model 990/10 Computer with map option.

Syntax definition:

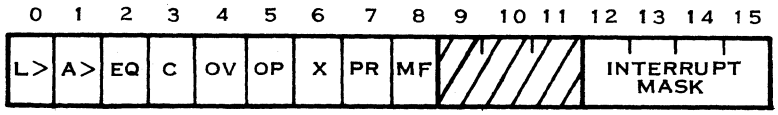
[<label>]b . . . LMFb . . . <wa>,<m>b . . . [<comment>]

Example:

```
NMAP   LMF   3,1              LOAD MAP FILE 1
```

Definition: Place the contents of a six-word area of memory at the address in the workspace register specified by wa into the memory map file designated by m.

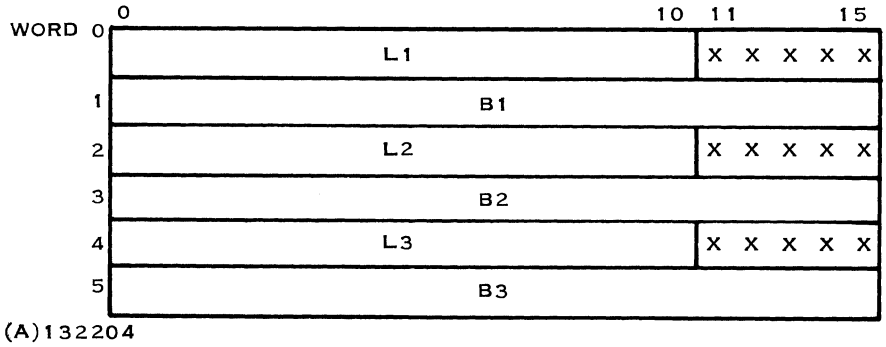
Status bits affected: None



Execution results: When Privileged Mode bit (bit 7 of ST register) is set to 0: the contents of a six-word area at address in wa are placed in map file m.

When Privileged Mode bit is set to 1, an error interrupt occurs.

Application notes: Use the LMF instruction to load either map file 0 or 1 (map file 2 is loaded by the long distance instructions). The map file is a set of six registers that maps the 32K word addresses of the AU into the desired addresses of memory having a larger capacity. The six-word area contains the following:



Words 0, 2, and 4 contain values that are placed in limit registers L1, L2, and L3

To determine values to be placed in the limit registers, the following considerations apply:

- The 11 most significant bits of each memory word are placed in the 11-bit limit registers.
- The 5 least significant bits may be any value. (They are ignored.)
- The one's complement of the limit is placed in the memory word, and in the map file.

The values in words 1, 3, and 5 are the 16 most significant bits of the bias register values, and are placed in registers B1, B2, and B3.

To determine the values to be placed in the six-word memory area, consider the following:

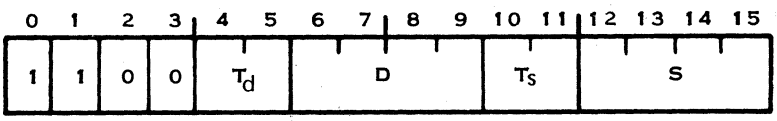
- All addresses from 0 through limit 1 are contiguous in memory.
- All addresses greater than limit 1, up through limit 2 are contiguous in memory.
- All addresses greater than limit 2, up through limit 3 are contiguous in memory.
- All addresses greater than limit 3 are protected addresses.
- Place the one's complements of the limit values in words 0, 2, and 4.
- Place the 16 most significant bits of the bias address for the lowest group in the second word.
- Place the 16 most significant bits of the bias address for the next group in the fourth word.
- Place the 16 most significant bits of the bias address for the highest group in the sixth word.

3.64 MOVE WORD MOV

Op Code: C000

Addressing mode: Format I

Format:



Syntax definition:

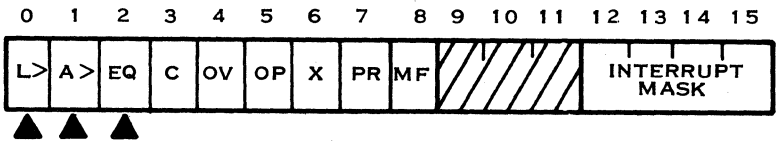
[<label>]b . . . MOVb . . . <ga_s>,<ga_d>b . . . [<comment>]

Example:

```
GET   MOV   @WORD, 2        GET A COPY OF WORD INTO WSR2
```

Definition: Replace the destination operand with a copy of the source operand. The AU compares the resulting destination operand to zero and sets/resets the status bits according to the comparison.

Status bits affected: Logical greater than, arithmetic greater than, and equal.



Execution results: (ga_s) → (ga_d)

Application notes: MOV is used to move 16-bit words as follows:

Memory-to-memory (non register)

Load register (memory-to-register)

Register-to-register

Register-to-memory

MOV may also be used to compare a memory location to zero by the use of

```
MOV     7,7
JNE     TEST
```

which would move register 7 to itself and compare the contents of register 7 to zero. If the contents are not equal to zero, the equal status bit is reset and control transfers to TEST. Another use of MOV, for example, is if workspace register 9 contains 3416_{16} and location ONES contains $FFFF_{16}$, then

```
MOV     @ONES,9
```

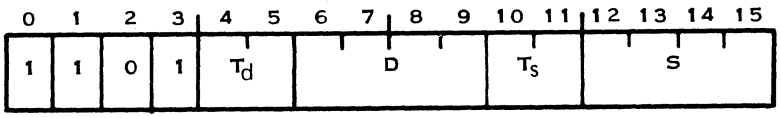
changes the contents of workspace register 9 to $FFFF_{16}$, while the contents of location ONES is not changed. For this example, the logical greater than status bit sets and the arithmetic greater than and equal status bits reset.

3.65 MOVE BYTE MOVB

Op Code: D000

Addressing mode: Format I

Format:



Syntax definition:

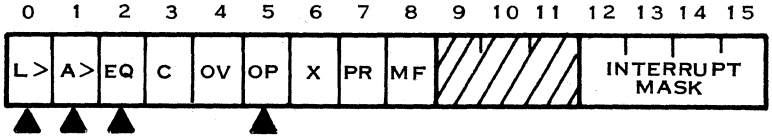
[<label>]b . . . MOVBb . . . <ga_s>,<ga_d>b . . . [<comment>]

Example:

```
NEXT    MOVB    2, @BUFF (3)    STORE CHARACTER IN EFFECTIVE BUFFER
                                ADDRESS
```

Definition: Replace the destination operand (byte) with a copy of the source operand (byte). If either operand is addressed in the workspace register mode, the byte addressed is the most significant byte of the word (bits 0-7) and the least significant byte (bits 8-15) is not affected by this instruction. The AU compares the destination operand to zero and sets/resets the status bits to indicate the result of the comparison. The odd parity bit sets when the bits in the destination operand establish odd parity.

Status bits affected: Logical greater than, arithmetic greater than, equal, and odd parity.



Execution results: $(ga_s) \rightarrow (ga_d)$

Application notes: MOVB is used to move bytes in the same combinations as the MOV instruction moves words. For example, if memory location $1C14_{16}$ contains a value of 2016_{16} and TEMP is located at $1C15_{16}$, and if workspace register 3 contains $542B_{16}$, then the instruction

```
MOVB        @TEMP,3
```

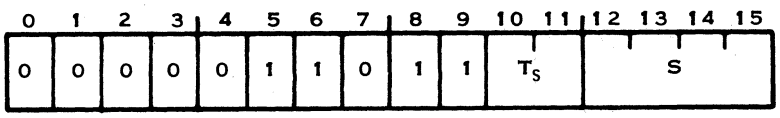
Changes the contents of workspace register 3 to $162B_{16}$. The logical greater than, arithmetic greater than, and odd parity status bits set while the equal status bit resets.

3.66 SWAP BYTES SWPB

Op Code: 06C0

Addressing mode: Format VI

Format:



Syntax definition:

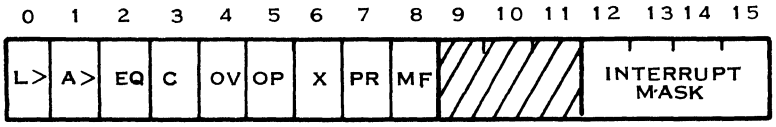
[<label>]b . . . SWPBb . . . <ga_s>b . . . [<comment>]

Example:

```
SWITCH   SWPB   3          BYTE REVERSE WSR3
```

Definition: Replace the most significant byte (bits 0-7) of the source operand with a copy of the least significant byte (bits 8-15) of the source operand and replace the least significant byte with a copy of the most significant byte.

Status bits affected: None



Execution results: Exchanges left and right bytes of word (ga_s).

Application notes: Use the SWPB instruction to interchange bytes of an operand prior to executing various byte instructions. For example, if workspace register 0 contains 2144_{16} and memory location 2144_{16} contains the value $F312_{16}$, then the instruction

```
SWPB   *0+
```

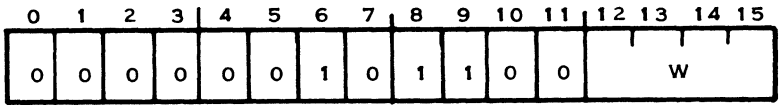
Changes the contents of workspace register 0 to 2146_{16} and the contents of memory location 2144_{16} to $12F3_{16}$. The status register remains unchanged.

3.67 STORE STATUS STST

Op Code: 02C0

Addressing mode: Format VIII

Format:



Syntax definition:

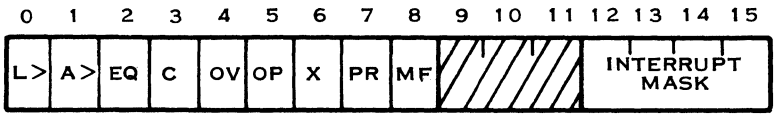
[<label>]b . . . STSTb . . . <wa>b . . . [<comment>]

Example:

```
LABEL   STST   7          STORE STATUS IN WSR7
```

Definition: Store the status register contents in the specified workspace register.

Status bits affected: None



Execution results: (ST) → (wa)

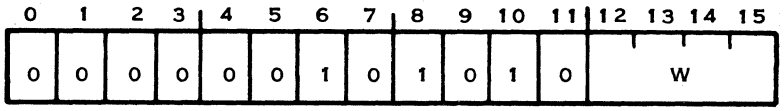
Application notes: Use the STST instruction to store the ST register contents when applicable.

3.68 STORE WORKSPACE POINTER STWP

Op Code: 02A0

Addressing mode: Format VIII

Format:



Syntax definition:

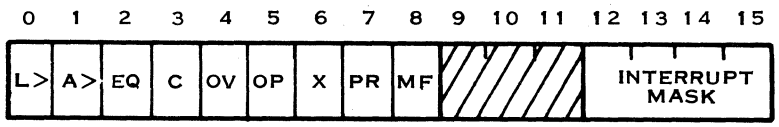
[<label>]b . . . STWPb . . . <wa>b . . . [<comment>]

Example:

```
LABEL   STWP   6          STORE WKSP POINTER IN WSR6
```

Definition: Place a copy of the workspace pointer contents in the specified workspace register.

Status bits affected: None



Execution results: (WP) → (wa)

Application notes: Use the STWP instruction to store the contents of the WP register as applicable.

3.69 LOGICAL INSTRUCTIONS

The set of logical instructions permits the user to perform various logical operations on memory locations and/or workspace registers. The instructions are:

Instruction	Mnemonic	Paragraph
AND Immediate	ANDI	3.70
Clear	CLR	3.74
Invert	INV	3.73
OR Immediate	ORI	3.71

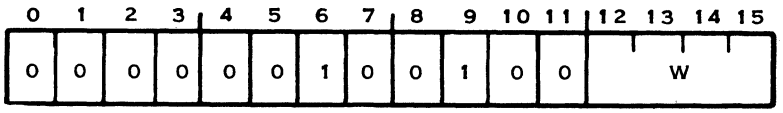
Instruction	Mnemonic	Paragraph
Set to One	SETO	3.75
Set Ones Corresponding (OR)	SOC	3.76
Set Ones Corresponding, Byte (OR)	SOCB	3.77
Set Zeros Corresponding	SZC	3.78
Set Zeros Corresponding, Byte	SZCB	3.79
Exclusive OR	XOR	3.72

3.70 AND IMMEDIATE ANDI

Op Code: 0240

Addressing mode: Format VIII

Format:



Syntax definition:

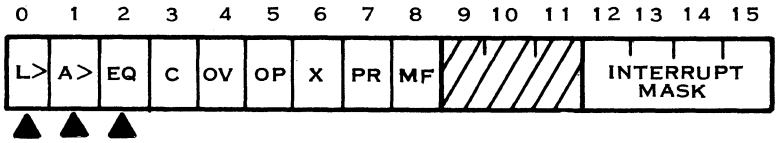
[<label>]b . . . ANDIb . . . <wa>,<iop>b . . . [<comment>]

Example:

```
LABEL   ANDI   3, >FFF0        SET LOWER 4 BITS OF WSR3 TO ZERO
```

Definition: Perform a bit-by-bit AND operation of the 16 bits in the immediate operand and the corresponding bits of the workspace register. The immediate operand is the word in memory immediately following the instruction word. Place the result in the workspace register. The AU compares the result to zero and sets/resets the status bits according to the results of the comparison.

Status bits affected: Logical greater than, arithmetic greater than, and equal.



Execution results: (wa) AND iop → (wa)

Application notes: Use the ANDI instruction to perform a logical AND with an immediate operand and a workspace register. Each bit of the 16-bit word of both operands follows the truth table

Immediate Operand Bit	Workspace Register Bit	AND Result
0	0	0
0	1	0
1	0	0
1	1	1

For example, if workspace register 0 contains $D2AB_{16}$, the instruction

```
ANDI   0,>6D03
```

results in workspace register 0 changing to 4003_{16}. This AND operation on a bit-by-bit basis is

```
0 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1    (Immediate operand)
1 1 0 1 0 0 1 0 1 0 1 0 1 0 1 1    (Workspace register 0)
-------------------------------
0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1    (Workspace register 0 result)
```

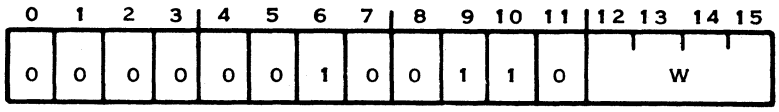
For this example, the logical greater than and arithmetic greater than status bits set while the equal status bit resets. ANDI is also useful for masking out bits of a workspace register.

3.71 OR IMMEDIATE ORI

Op Code: 0260

Addressing mode: Format VIII

Format:



Syntax definition:

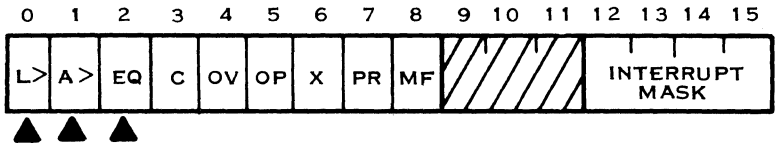
[<label>]b . . . ORIb . . . <wa>,<iop>b . . . [<comment>]

Example:

```
LABEL   ORI   3, >F000        SET HIGH ORDER 4 BITS OF WSR3 TO ONES
```

Definition: Perform an OR operation of the 16-bit immediate operand and the corresponding bits of the workspace register. The immediate operand is the memory word immediately following the ORI instruction. Place the result in the workspace register. The AU compares the result to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected: Logical greater than, arithmetic greater than, and equal.



Execution results: (wa) OR iop → (wa)

Application notes: Use the ORI instruction to perform a logical OR with the immediate operand and a specified workspace register. Each bit of the 16-bit word of both operands is OR'd using the truth table

Immediate Operand	Workspace Register	OR Result
0	0	0
1	0	1
0	1	1
1	1	1

For example, if workspace register 5 contains $D2AB_{16}$, then the instruction

```
ORI     5,>6D03
```

results in workspace register 5 changing to $FFAB_{16}$. This OR operation on a bit-by-bit basis is

```
0 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1   (Immediate operand)
1 1 0 1 0 0 1 0 1 0 1 0 1 0 1 1   (Workspace register 5)
-------------------------------
1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1   (Workspace register 5 result)
```

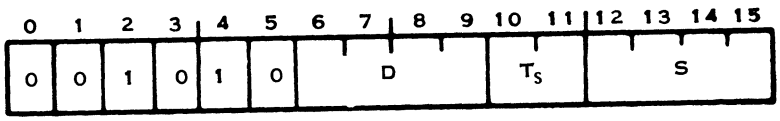
For this example, the logical greater than status bit sets, and the arithmetic greater than and equal status bits reset.

3.72 EXCLUSIVE OR XOR

Op Code: 2800

Addressing mode: Format III

Format:



Syntax definition:

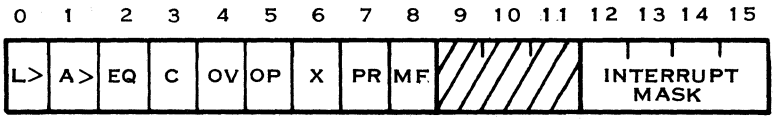
[<label>]b . . . XORb . . . <ga_s>,<wa_d>b . . . [<comment>]

Example:

```
LABEL   XOR   @WORD, 3      EXCLUSIVE OR THE CONTENTS OF WORD
                            AND WSB
```

Definition: Perform a bit-by-bit exclusive OR of the source and destination operands, and replace the destination operand with the result. This exclusive OR is accomplished by setting the bits in the resultant destination operand to a logic one when the corresponding bits of the two operands are not equal. The bits in the resultant destination operand are reset to zero when the corresponding bits of the two operands are equal. The AU compares the resultant destination operand to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected: Logical greater than, arithmetic greater than, and equal.



Execution results: (ga_s) XOR (wa_d) → (wa_d)

(i.e. [(ga_d) AND NOT (wa_d)] OR [(wa_d) AND NOT (ga_d)] → (wa_d)

Application notes: Use the XOR instruction to perform an exclusive OR on two word operands. For example, if workspace register 2 contains $D2AA_{16}$ and location CHANGE contains the value $6D03_{16}$, then the instruction

```
XOR     @CHANGE,2
```

results in the contents of workspace register 2 changing to $BFA9_{16}$. Location CHANGE remains $6D03_{16}$. This is shown as

```
0 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1    (Source operand)
1 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0    (Destination operand)
-------------------------------
1 0 1 1 1 1 1 1 1 0 1 0 1 0 0 1    (Destination operand result)
```

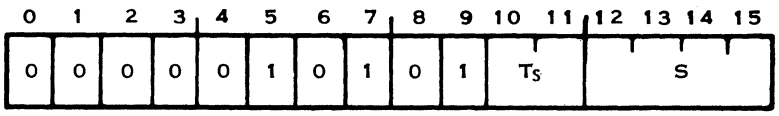
For this example, the logical greater than status bit sets while the arithmetic greater than and equal status bits reset.

3.73 INVERT INV

Op Code: 0540

Addressing mode: Format VI

Format:



Syntax definition:

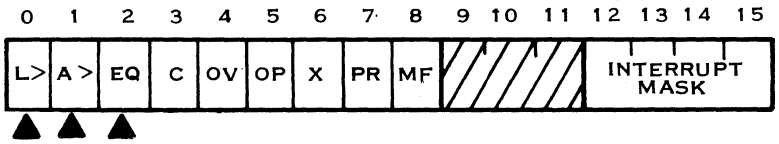
[<label>]b . . . INVb . . . <ga_s>b . . . [<comment>]

Example:

```
COMPL   INV   @BUFF(2)      REPLACE BUFFER WORD WITH ONEs COMPLEMENT
                            OF DATA
```

Definition: Replace the source operand with the one's complement of the source operand. The one's complement is equivalent to changing each zero in the source operand to a logic one and each logic one in the source operand to a logic zero. The AU compares the result to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected: Logical greater than, arithmetic greater than, and equal.



Execution results: The one's complement of (ga_s) is placed in (ga_s).

Application notes: INV changes each logic zero in the source operand to a logic one and each logic one to a logic zero. For example, if workspace register 11 contains $A54B_{16}$, then the instruction

```
INV     11
```

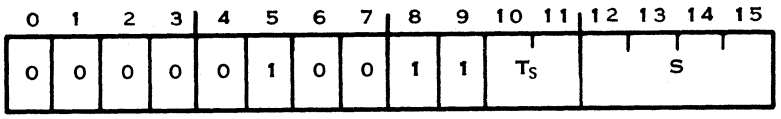
changes the contents of workspace register 11 to $5AB4_{16}$. The logical greater than and arithmetic greater than status bits set and the equal status bit resets.

3.74 CLEAR CLR

Op Code: 04C0

Addressing mode: Format VI

Format:



Syntax definition:

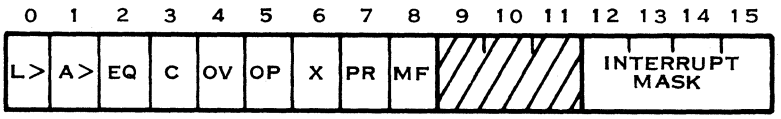
[<label>]b . . . CLRb . . . <ga_s>b . . . [<comment>]

Example:

```
PRELM   CLR   @BUFF(2)      CLEAR EFFECTIVE BUFFER ADDRESS
```

Definition: Replace the source operand with a full, 16-bit word of zeros.

Status bits affected: None



Execution results: 0 → (ga_s)

Application notes: Use the CLR instruction to set a full, 16-bit, memory addressable word to zero. For example, if workspace register 11 contains the value 2001_{16}, then the instruction

```
CLR     *>B
```

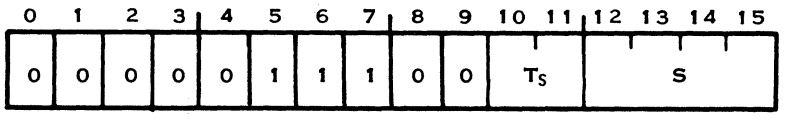
results in the contents of memory location 2000_{16} being set to 0. Workspace register 11 and the status register are unchanged.

3.75 SET TO ONE SETO

Op Code: 0700

Addressing mode: Format VI

Format:



Syntax definition:

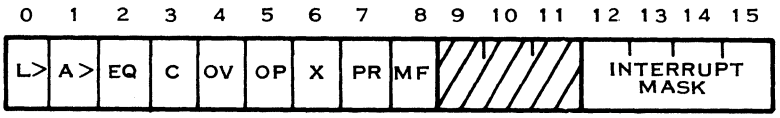
[<label>]b . . . SETOb . . . <ga_s>b . . . [<comment>]

Example:

```
LABEL   SETO   3              SET WSR3 TO -1
```

Definition: Replace the source operand with a 16-bit word logic one value.

Status bits affected: None



Execution results: $FFFF_{16} \rightarrow (ga_s)$

Application notes: Use the SETO instruction to initialize an addressable memory to a -1 value. For example, the instruction

```
SETO   3
```

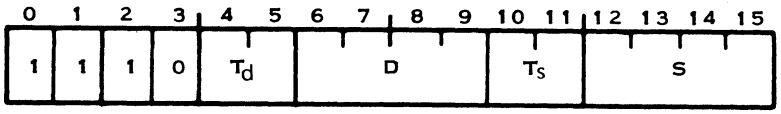
initializes workspace register 3 to a value of $FFFF_{16}$. The contents of the status register is unchanged. This is a useful means of setting flag words.

3.76 SET ONES CORRESPONDING SOC

Op Code: E000

Addressing mode: Format I

Format:



Syntax definition:

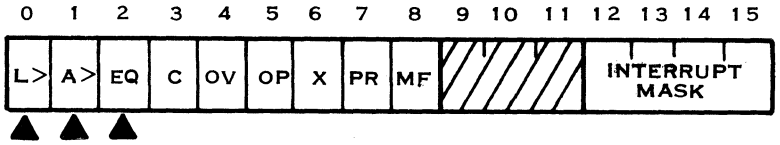
[<label>]b . . . SOCb . . . <ga_s>,<ga_d>b . . .[<comment>]

Example:

```
LABEL   SOC   3, 2           OR WSR3 INTO WSR2
```

Definition: Set to a logic one the bits in the destination operand that correspond to any logic one bit in the source operand. Leave unchanged the bits in the destination operand that are in the same bit positions as the logic zero bits in the source operand. The changed destination operand replaces the original destination operand. This operation is an OR of the two operands. The AU compares the result to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected: Logical greater than, arithmetic greater than, and equal.



Execution results: Bits of (ga_d) corresponding to bits of (ga_s) equal to 1 are set to 1.

Application notes: Use the SOC instruction to OR the 16-bit contents of two operands. For example, if workspace register 3 contains $FF00_{16}$ and location NEW contains $AAAA_{16}$, then the instruction

```
SOC     3,@NEW
```

changes the contents of location NEW to $FFAA_{16}$ while the contents of work space register 3 is unchanged. This is shown as

```
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0    (Source operand)
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0    (Destination operand)
-------------------------------
1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 0    (Destination operand result)
```

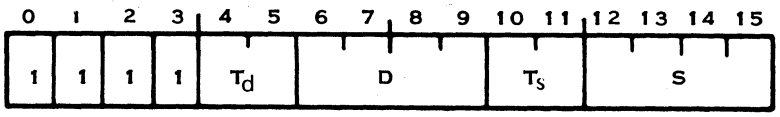
For this example, the logical greater than status bit sets and the arithmetic greater than and equal status bits reset.

3.77 SET ONES CORRESPONDING, BYTE SOCB

Op Code: F000

Addressing mode: Format I

Format:



Syntax definition:

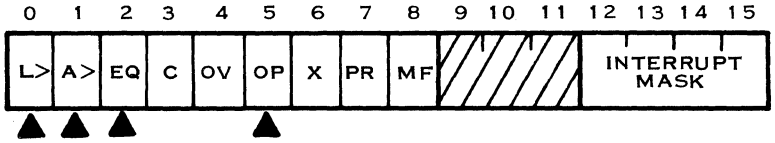
[<label>]b ... SOCBb ... <ga_s>,<ga_d>b ... [<comment>]

Example:

```
LABEL   SOCB   3, @DET        OR WSR3 INTO BYTE AT LOCATION DET
```

Definition: Set to a logic one the bits in the destination operand byte that correspond to any logic one in the source operand byte. Leave unchanged the bits in the destination operand that are in the same bit positions as the logic zero bits in the source operand byte. The changed destination operand byte replaces the original destination operand byte. This operation is an OR of the two operand bytes. The AU compares the resulting destination operand byte to zero and sets/resets the status bits to indicate the results of the comparison. The odd parity status bit sets when the bits in the resulting byte establish odd parity.

Status bits affected: Logical greater than, arithmetic greater than, equal, and odd parity.



Execution results: Bits of (ga_s) corresponding to bits of (ga_s) equal to 1 are set to 1.

(i.e. (ga_d) OR (ga_s) → (ga_d))

Application notes: Use the SOCB instruction to OR two byte operands. For example, if workspace register 5 contains the value $F013_{16}$ and workspace register 8 contains the value $AA24_{16}$, then the instruction

```
SOCB   5,8
```

changes the contents of workspace register 8 to $FA24_{16}$, while the contents of workspace register 5 is unchanged. This is shown as

```
1 1 1 1 0 0 0 0 0 0 0 1 0 0 1 1   (Source operand)
1 0 1 0 1 0 1 0 0 0 1 0 0 1 0 0   (Destination operand)
-------------------------------
1 1 1 1 1 0 1 0 0 0 1 0 0 1 0 0   (Destination operand result)
          ---------------------
          (Unchanged)
```

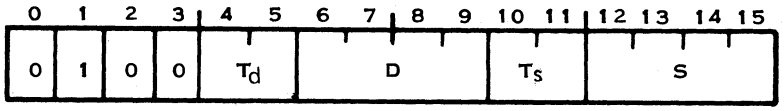
For this example, the logical greater than status bit sets while the arithmetic greater than, equal, and odd parity status bits reset.

3.78 SET ZEROS CORRESPONDING SZC

Op Code: 4000

Addressing mode: Format I

Format:



Syntax definition:

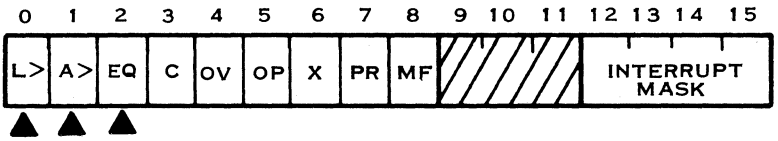
[<label>]b . . . SZCb . . . <ga_s>,<ga_d>b . . . [<comment>]

Example:

```
LABEL   SZC   @MASK, 2        RESET BITS OF WSR2 INDICATED BY MASK
```

Definition: Set to a logic zero the bits in the destination operand that correspond to the bit positions equal to a logic one in the source operand. This operation is effectively an AND operation of the one's complement of the source operand and the destination operand. The AU compares the resulting destination operand to zero and sets/resets the status bits to indicate the results of the comparison.

Status bits affected: Logical greater than, arithmetic greater than, and equal.



Execution results: Bits of (ga_d) corresponding to bits of (ga_s) equal to 1 are set to 0.

(i.e. [NOT (ga_s) AND (ga_d)] → (ga_d))

Application notes: Use the SZC instruction to turn off flag bits or AND the contents of the one's complement of the source operand and the destination operand. For example, if workspace register 5 contains $6D03_{16}$ and workspace register 3 contains $D2AA_{16}$, then the instruction

```
SZC     5,3
```

changes the contents of workspace register 3 to $92A8_{16}$ while the contents of workspace register 5 remain unchanged. This is shown as

```
0 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1    (Source operand)
1 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0    (Destination operand)
-------------------------------
1 0 0 1 0 0 1 0 1 0 1 0 1 0 0 0    (Destination operand result)
```

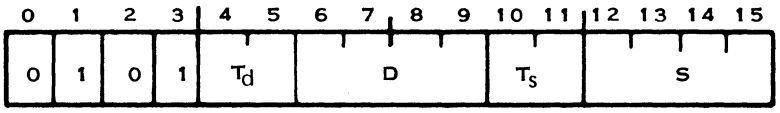
For this example, the logical greater than status bit sets while the arithmetic greater than and equal status bits reset.

3.79 SET ZEROS CORRESPONDING, BYTE SZCB

Op Code: 5000

Addressing mode: Format I

Format:



Syntax definition:

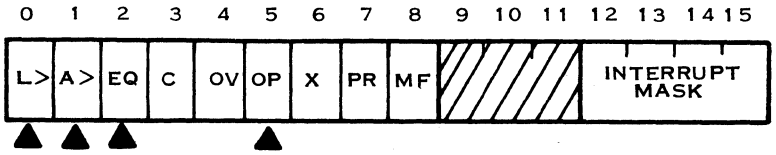
[<label>]b . . . SZCBb . . . <ga_s>,<ga_d>b . . . [<comment>]

Example:

```
LABEL   SZCB   @MASK, @CHAR   RESET BITS OF CHAR INDICATED BY MASK
```

Definition: Set to a logic zero the bits in the destination operand byte that correspond to the bit positions equal to a logic one in the source operand byte. This operation is effectively an AND operation of the one's complement of the source operand byte and the destination operand byte. The AU compares the resulting destination operation byte to zero and sets/resets the status bits to indicate the result of the comparison. The odd parity status bit sets when the bits in the resulting destination operand byte establish odd parity. When the destination operand is addressed in the workspace register mode, the least significant byte (bits 8-15) is unchanged.

Status bits affected: Logical greater than, arithmetic greater than, equal, and odd parity.



Execution results: Bits of (ga_d) corresponding to bits of (ga_s) equal to 1 are set to 0.

(i.e. [NOT (ga_s) AND (ga_d)] → (ga_d))

Application notes: The SZCB instruction is used for the same applications as SZC except bytes are used instead of words. For example, if location BITS contains the value $F018_{16}$, and location TESTVA contains the value $AA24_{16}$, then

```
SZCB   @BITS,@TESTVA
```

changes the contents of TESTVA to $0A24_{16}$ while BITS remains unchanged. This is shown as

```
1 1 1 1 0 0 0 0 0 0 0 1 1 0 0 0   (Source operand)
1 0 1 0 1 0 1 0 0 0 1 0 0 1 0 0   (Destination operand)
-------------------------------
0 0 0 0 1 0 1 0 0 0 1 0 0 1 0 0   (Destination operand result)
                  ---------------
                  (Unchanged)
```

For this example, the logical greater than and arithmetic greater than status bits set while the equal and odd parity status bits reset.

3.80 WORKSPACE REGISTER SHIFT INSTRUCTIONS

Workspace register shift instructions permit the shifting of the contents of a specified workspace register from one to sixteen bits. The shifting instructions are:

Instruction	Mnemonic	Paragraph
Shift Right Arithmetic	SRA	3.81
Shift Right Logical	SRL	3.83
Shift Left Arithmetic	SLA	3.82
Shift Right Circular	SRC	3.84

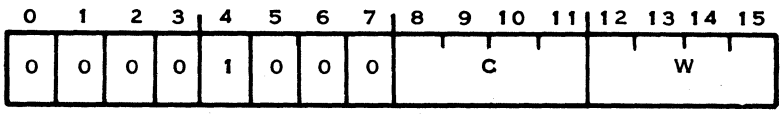
For each of these instructions, if the shift count in the instruction is zero, the shift count is taken from workspace register 0, bits 12 through 15.If the four bits of workspace register 0 are equal to zero, the shift count is 16 bit positions. The value of the last bit shifted out of the workspace register is placed in the carry bit of the ST register. The result is compared to zero and the results of the comparison are shown in the logical greater than, arithmetic greater than, and equal bits (bits 0 through 2) in the ST register. If a shift count greater than 15 is supplied, the assembler fills in the four-bit field with the least significant four bits of the shift count. SDSMAC gives a warning message when this occurs.

3.81 SHIFT RIGHT ARITHMETIC SRA

Op Code: 0800

Addressing mode: Format V

Format



Syntax definition:

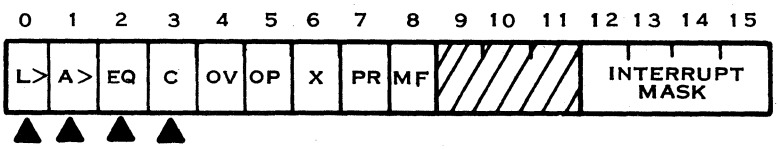
[<label>]b . . . SRAb . . . <wa>,<scnt>b . . . [<comment>]

Example:

```
LABEL   SRA   2, 3          SHIFT WSR2 RIGHT THREE BIT LOCATIONS
```

Definition: Shift the contents of the specified workspace register to the right for the specified number of bit positions, filling vacated bit positions with the sign bit.

Status bits affected: Logical greater than, arithmetic greater than, equal, and carry.



Execution results: Shift the bits of (wa) to the right, extending the sign bit to fill vacated bit positions. When SCNT is greater than 0, shift the number of bit positions specified by SCNT. If SCNT is equal to 0, shift the number of bit positions contained in the four least significant bits of workspace register 0. When SCNT and the four least significant bits of workspace register 0 both contain 0, shift 16 bit positions.

Application notes: An example of an arithmetic right shift is: If workspace register 5 contains the value 8224_{16} , and workspace register 0 contains the value $F326_{16}$, then the instruction

```
SRA     5,0
```

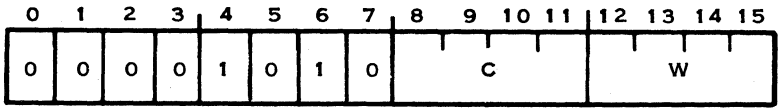
changes the contents of workspace register 5 to $FE08_{16}$. The logical greater than and carry status bits set while the arithmetic greater than and equal status bits reset. Additional examples are shown in a subsequent paragraph.

3.82 SHIFT LEFT ARITHMETIC SLA

Op Code: 0A00

Addressing mode: Format V

Format:



Syntax definition:

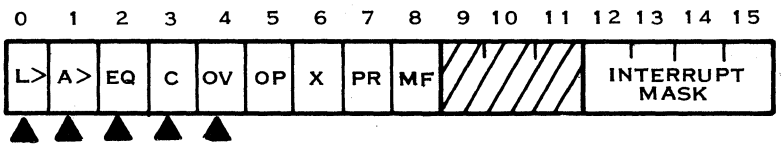
[<label>]b . . . SLAb . . . <wa>,<scnt>b . . . [<comment>]

Example:

```
LABEL   SLA   2, 1          SHIFT WSR2 LEFT ONE BIT LOCATION
```

Definition: Shift the contents of the specified workspace register to the left for the specified number of bit positions while filling the vacated bit positions with logic zero values. Note that the overflow status bit sets when the sign of the word changes during the shifting operation.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results: Shift the bits of (wa) to the left, filling the vacated bit positions with zeros. When SCNT is greater than 0, shift the number of bit positions specified by SCNT. If SCNT is equal to 0, shift the number of bit positions contained in the four least significant bits of workspace register 0. When SCNT and the four least significant bits of workspace register 0 both contain 0, shift 16 bit positions.

Application notes: An example of an arithmetic left shift is: If workspace register 10 contains the value 1357_{16} , then the instruction

```
SLA     10,5
```

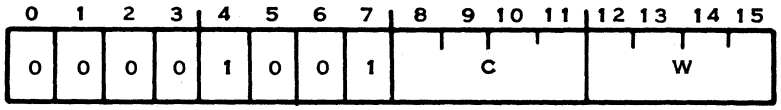
changes the contents of workspace register 10 to $6AE0_{16}$. The logical greater than, arithmetic greater than, and overflow status bits set while the equal and carry status bits reset. Refer to a subsequent paragraph for additional examples.

3.83 SHIFT RIGHT LOGICAL SRL

Op Code: 0900

Addressing mode: Format V

Format:



Syntax definition:

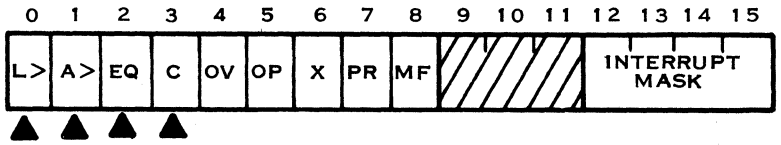
[<label>]b . . . SRLb . . . <wa>,<scnt>b . . . [<comment>]

Example:

```
LABEL   SRL   3, 7          SHIFT WSR3 RIGHT SEVEN BIT LOCATIONS
```

Definition: Shift the contents of the specified workspace register to the right for the specified number of bits while filling the vacated bit positions with logic zero values.

Status bits affected: Logical greater than, arithmetic greater than, equal, and carry.



Execution results: Shift the bits of (wa) to the right, filling the vacated bit positions with zeros. When SCNT is greater than 0, shift the number of bit positions specified by SCNT. If SCNT is equal to 0, shift the number of bit positions contained in the four least significant bits of workspace register 0. When SCNT and the four least significant bits of workspace register 0 both contain 0, shift 16 bit positions.

Application notes: An example of a logical right shift is: If workspace register zero contains the value $FFEF_{16}$, then the instruction

```
SRL   0,3
```

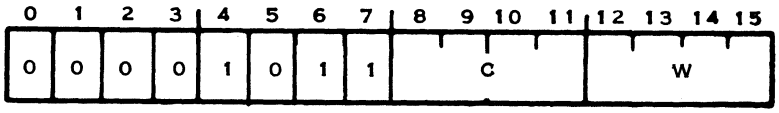
Changes the contents of workspace register 0 to $1FFD_{16}$. The logical greater than, arithmetic greater than and carry status bits set while the equal status bit resets. Additional examples are shown in a subsequent paragraph.

3.84 SHIFT RIGHT CIRCULAR SRC

Op Code: 0B00

Addressing mode: Format V

Format:



Syntax definition:

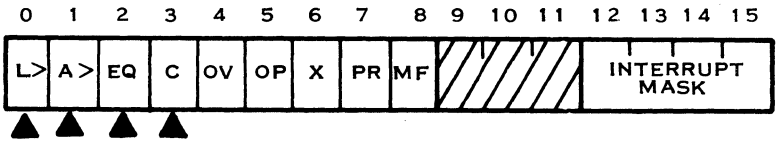
[<label>]b . . . SRCb . . . <wa>,<scnt>b . . . [<comment>]

Example:

```
LABEL   SRC   7, 16-3          SHIFT CIRC WSR7  3 BIT LOCATIONS LEFT
```

Definition: Shift the specified workspace register to the right for the specified number of bit positions while filling vacated bit positions with the bit shifted out of position 15.

Status bits affected: Logical greater than, arithmetic greater than, equal, and carry.



Execution results: Shift the bits of (wa) to the right, filling the vacated bit positions with the bits shifted out at the right. When SCNT is greater than 0, shift the number of bit positions specified by SCNT. If SCNT is equal to 0, shift the number of bit positions contained in the four least significant bits of workspace register 0. When SCNT and the four least significant bits of workspace register 0 both contain 0, shift 16 bit positions.

Application notes: An example of a circular right shift is: If workspace register 2 contains the value $FFEF_{16}$, then the instruction

```
SRC   2,7
```

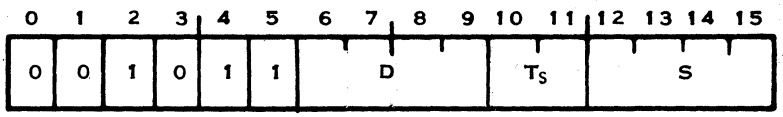
changes the contents of workspace register 2 to $DFFF_{16}$. The logical greater than and carry status bits set while the arithmetic greater than and equal status bits reset. Shift left circular is not implemented since SRC can perform the same function: SLC x,n = SRC x,16-n. Refer to a subsequent paragraph for additional application notes.

3.85 EXTENDED OPERATION XOP

Op Code: 2C00

Addressing mode: Format IX

Format:



Syntax definition:

[<label>]b . . . XOPb . . . <ga_s>,<op>b . . . . [<comment>]

Example:

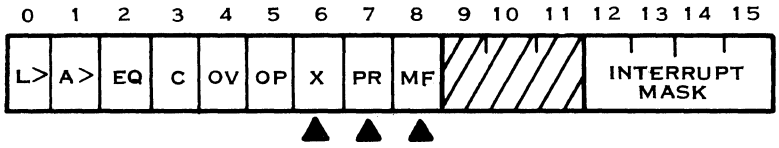
```
LABEL   XOP   @BUFF(4), 12   DO XOP12 ON WORD OF BUFFER SPECIFIED
                              BY WSR4
```

Definition: The op field specifies the extended operation transfer vector in memory. The two memory words at that location contain the WP contents and PC contents for the software implemented XOP instruction subroutine. The memory location for these two words is derived by multiplying the op field contents by four and adding the product to 0040_{16}. Note that the two memory words at this location must contain the necessary WP and PC values prior to the XOP instruction execution for software implemented instructions.

The effective address of the source operand is placed in workspace register 11 of the XOP workspace. The WP contents are placed in workspace register 13 of the XOP workspace. The PC contents are placed in workspace register 14 of the XOP workspace. The ST contents are placed in workspace register 15 of the XOP workspace. The ST contents are placed in workspace register 15 of the XOP workspace. Control is transferred to the new PC address and the software implemented XOP is executed. (XOP execution of software implemented XOP instruction is similar to an interrupt trap execution.)

Model 990/10 Computer: An extended operation may be alternatively implemented by user-supplied hardware. When hardware is connected for the specified operation no context switch occurs, and the hardware performs the operation. When a Model 990/10 Computer performs a software-implemented extended operation, the Privileged Mode bit is set to 0. When the map option is included, the Map File bit is set to 0 also.

Status bits affected: Extended operation



Execution results:

$ga_s \rightarrow$ (workspace register 11)
$(0040_{16} + (op)*4) \rightarrow (WP)$
$(0042_{16} + (op)*4) \rightarrow (PC)$
(WP) → (workspace register 13)
(PC) → (workspace register 14)
(ST) → (workspace register 15)
0 → ST8
0 → ST7
1 → ST6
} 990/10

Application notes: Refer to a subsequent paragraph for a detailed example of the execution of a software implemented XOP instruction.

3.86 LONG DISTANCE ADDRESSING INSTRUCTIONS

The long distance addressing instructions are available in the Model 990/10 Computer with the map option. These instructions enable accesses outside of the current memory map for a single address. The instructions are:

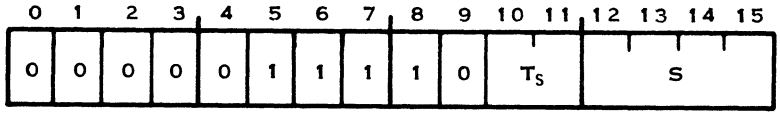
Instruction	Mnemonic	Paragraph
Long Distance Source	LDS	3.87
Long Distance Destination	LDD	3.88

3.87 LONG DISTANCE SOURCE LDS

Op Code: 0780

Addressing mode: Format VI

Format:



Syntax definition:

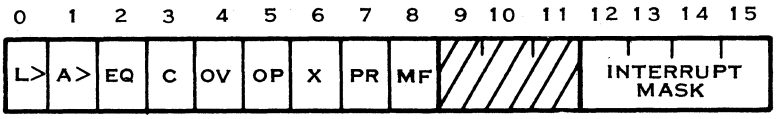
[<label>]b . . . LDSb . . . <ga_s>b . . . [<comment>]

Example:

```
LABEL   LDS   @SIXWD          PREPARE TO USE LONG DISTANCE SOURCE
```

Definition: Place the contents of a six-word area of memory into map file 2, and use map file 2 in developing the source address of the next instruction. The instruction places the contents of the six-word memory area at the effective address of the source operand in map file 2 in all cases; the map file is not used when the source address of the following instruction is a workspace register address, or when the following instruction is a B, BL, or BLWP instruction. The instruction inhibits all interrupts until the following instruction is executed.

Status bits affected: None.



Execution results: When Privileged Mode bit (bit 7 of ST register) is set to 0: The contents of a six-word area at address ga_s are placed in map file 2, and the source address of the following instruction is mapped with map file 2. (If T_s of the following instruction is equal to 0, or if following instruction is B, BL, or BLWP instruction, new map is not used.)

When Privileged Mode bit is set to 1: Error interrupt.

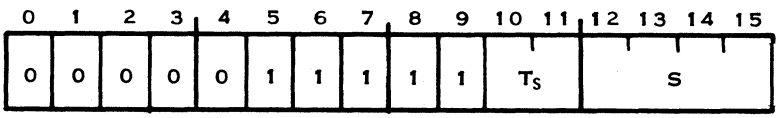
Application notes: Use the LDS instruction in the Privileged Mode to access an address outside of the current map. The contents of the six-word area are placed in the L1, L2, L3, B1, B2, and B3 registers of map file 2 as shown in paragraph 3.63. The address to which the map file applies is the source address of the next instruction. Placing an LDS instruction prior to an instruction that has no destination operand, or an instruction having a workspace register address for the destination operand does not result in an access outside of the current map.

3.88 LONG DISTANCE DESTINATION LDD

Op Code: 07C0

Addressing mode: Format VI

Format:



Syntax definition:

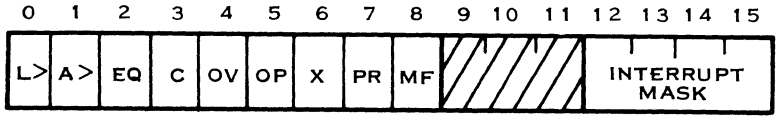
[<label>]b . . . LDDb . . . <ga_s>b . . . [<comment>]

Example:

```
LABEL   LDD   @SIXWD          PREPARE TO STORE LONG DISTANCE
```

Definition: Place the contents of a six-word area of memory into map file 2, and use map file 2 in developing the destination address of the next instruction. The instruction places the contents of the six-word memory area at the effective address of the source operand in map file 2 in all cases; the map file is not used when the following instruction has no destination operand, or when the destination address has a workspace register address. The instruction inhibits all interrupts until the following instruction is executed.

Status bits affected: None.



Execution results: When Privileged Mode bit (bit 7 of ST register) is set to 0: the contents of a six-word area at address ga_s are placed in map file 2, and the destination address of the following instruction is mapped with map file 2. (If T_d of the following instruction is equal to 0, or if the destination address is a workspace register address, the new map is not used.)

When Privileged Mode bit is set to 1: Error interrupt.

Application notes: Use the LDD instruction in the Privileged Mode to access an address outside of the current map. The contents of the six-word area are placed in the L1, L2, L3, B1, B2, and B3 registers of map file 2 as shown in paragraph 3.63. The address to which the map file applies is the destination address of the next instruction. Placing an LDD instruction prior to an instruction that has no destination operand, or an instruction having a workspace register address for the destination operand does not result in an access outside of the current map.

3.89 PROGRAMMING EXAMPLES

The remaining paragraphs of this section describe programming examples that supplement the application notes in the instruction descriptions. Programming examples are only included for those instructions for which the application notes require additional explanation.

3.89.1 **ABS INSTRUCTION.** Since the ABS instruction compares the operand to zero prior to any modification of the operand, the ABS instruction may be used to test a switch. The following example program illustrates this use of the instruction. A word of memory at location SWITCH is used to indicate whether or not a subroutine at location SUBR is being executed. Subroutine SUBR is used by several programs, but only one may use it at a time. When the subroutine is in use, location SWITCH contains one, and other programs may not transfer control to location SUBR. When control returns from the subroutine, location SWITCH is set to -1, making subroutine SUBR available again.

The first instruction would be used in the initialization portion, to make the subroutine available initially. The four instructions at location TEST would be included in each program that calls the subroutine. These instructions branch to location CALL when location SWITCH contains -1, setting location SWITCH to +1 after testing its value. Any attempt to access the subroutine before its completion results in the program entering a delay mode, retesting following each delay interval.

A BL instruction at location CALL transfers control to the subroutine, and stores the address of the SETO instruction in workspace register 11. When the subroutine returns control, the SETO instruction sets location SWITCH to -1, so that the next time any calling program tests the location, a transfer to the subroutine occurs. The code is as follows.

```
        SETO    @SWITCH     INITIALIZES SWITCH NEGATIVE[1]
        .
        .
        .
        .
TEST    ABS     @SWITCH     TEST SWITCH[2]
        JLT     CALL        IF NEGATIVE, TRANSFER[3]
        XOP     @TMDLY,15   IF NOT, WAIT[4]
        JMP     TEST        TEST AGAIN
        .
        .
        .
        .
CALL    BL      @SUBR       USE SUBROUTINE
        SETO    @SWITCH     RESET SWITCH[5]
        .
        .
        .
        .
SUBR    ....    SUBROUTINE ENTRY
        .
        .
        .
        .
        B       *11         SUBROUTINE RETURN
SWITCH  DATA    0           STORAGE AREA FOR SWITCH
TIMDLY  DATA    >200,10     TIME DELAY SUPERVISOR
                            CALL BLOCK
```

NOTE

1. Set SWITCH to all ones, making it negative.

2. If SWITCH negative, set to positive value to prevent subsequent entry.

3. If value in SWITCH was negative, the JLT instruction transfers control.

4. Supervisor call pointing to data block defining time delay request. Used to wait for a time period before retesting SWITCH. While in

a time delay, other programs can be executed, thus leaving SUBR available for use. Time Delay Supervisor calls are supported by the DX10 and TX990 Operating Systems. Reference either the *DX10 Operating System Programmer's Guide* or the *TX990 Operating System Programmer's Guide*.

5. Upon return, reset SWITCH to negative value to permit future use.

3.89.2 **SHIFTING INSTRUCTIONS.** There are 4 shifting instructions available with the Model 990 Computer that permit the user to shift the contents of a specified workspace register from one to sixteen consecutive bit positions.

The four shifting instructions are:

- Shift Left Arithmetic (SLA)
- Shift Right Arithmetic (SRA)
- Shift Right Circular (SRC)
- Shift Right Logical (SRL).

3.89.2.1 **Shift Left Arithmetic.** This shifting instruction shifts the indicated workspace register a specified number of bits to the left. For example, the instruction

```
SLA   5,1
```

would shift the contents of register five one bit to the left. The carry status bit contains the value shifted out of bit position zero and the jump instructions JOC and JNC permit the user to test the shifted bit. The overflow status bit sets when the sign of the contents of the register being shifted changes during the shift operation. If register five contained

```
0100111100000111
```

before the above instruction, the results of the instruction execution would be

```
1001111000001110
```

and the carry status bit would contain a zero and the overflow status bit would set because the contents changed from positive to negative (bit zero equal to zero changed to equal to one). If this shift sign change is important, the user could insert a JNO instruction to test the overflow condition. If there is no overflow, control transfers to the normal program sequence. Otherwise, the next instruction is then executed.

It is possible to construct double-length shifts with the SLA instruction, which could shift two or more words in a workspace. The following code will shift two consecutive workspace registers.

- Assumptions:

  1. The contents of workspace registers 1 and 2 are shifted one bit position.
  2. Additional code could be included to execute the code once for each bit shift required, when shifts of more than one bit position are required. The additional code must include a means of testing that the desired number of shifts are performed.
  3. Additional code tests for overflow from workspace register 1, to branch to an error routine at location ERR when overflow occurs.

- Code:

```
        SLA     1,1     SHIFT W1 ONE BIT
        JOC     ERR
        SLA     2,1     SHIFT W2 ONE BIT
        JNC     EXIT    TRANSFER IF NO CARRY
        INC     1       TRANSFER BIT FROM W2 to W1
          .
          .
          .
EXIT    INC     1       CONTINUE WITH PROGRAM
          .
          .
ERR     NOP
          .
```

3.89.2.2 Shift Right Arithmetic. This shifting instruction shifts the contents of a workspace register right a specified number of bits and extends the sign bit (bit zero) at the logic level that existed prior to the shift. The carry status bit contains the last bit shifted out of bit 15 of the workspace register. For example, the instruction

```
SRA  5,3
```

would shift the contents of workspace register five three bits to the right. If workspace register five contained

1100000011110000

prior to the shift, the results of this instruction would be

1111100000011110

and the carry status bit would contain a logic zero for the last shifted bit.

3.89.2.3 Shift Right Circular. The SRC instruction shifts the contents of a workspace register a specified number of bits to the right and transfers the bits shifted off the right end of the workspace into the left end of the workspace register. The carry status bit contains the last bit shifted out of bit 15 of the workspace register. For example, the instruction

```
SRC  6,5
```

would shift the contents of register six, five bits to the right and transfer the five bits shifted off the right end to the first five bits of workspace register six. For this example, if workspace register six contained

1100110011110101

before this instruction was executed, workspace register six would contain

1010111001100111

and the carry status bit would contain a logic one from the last bit shifted in workspace register six.

3.89.2.4 **Shift Right Logical.** The SRL instruction shifts the contents of a special workspace register to the right for a specified number of bits and fills the vacated bit positions on the left end of the workspace with zeros. The carry status bit contains the last bit shifted out of bit 15 of the workspace register. For example, the instruction.

```
SRL   5,8
```

would shift the contents of workspace register five eight bits to the right and would fill the first eight bits of the word with zeros. If the workspace register contained

1000100011111000

prior to the SRL instruction, the contents of workspace register five would be

0000000010001000

and the carry status bit would contain a logic one for the last bit shifted off the right end of workspace register five.

3.89.3 **INCREMENTING AND DECREMENTING.** There are two decrement and two increment instructions that may be used for various types of control when passing through a loop, indexing through an array, or operating within a group of instructions.

The four incrementing and decrementing instructions available for use with the 990 Computer are:

- Decrement (DEC)
- Decrement By Two (DECT)
- Increment (INC)
- Increment By Two (INCT).

The increment and decrement instructions are useful for indexing byte arrays and for counting byte operations. The increment and decrement by two instructions are useful for indexing word arrays and for counting word operations. The following paragraphs provide some examples of these operations.

3.89.3.1 **Increment Instruction Example.** Since the INC instruction is useful in byte operations, an example problem searches a character array for a character with odd parity. The last character contains zero to terminate the search. Begin the search at the lowest address of the array and maintain an index in a workspace register. The character array for this example is called A1 (also the relocatable address of the array). The code for a solution to this problem is:

```
        SETO    1           SET COUNTER INDEX TO -1
SEARCH  INC     1           INCREMENT INDEX
        MOVB    @A1(1),2    GET CHARACTER
        JOP     ODDP        JUMP IF FOUND
        JNE     SEARCH      CONTINUE SEARCH IF NOT ZERO
          .
          .
          .
          .
ODDP    ...     ....
```

3.89.3.2 **Decrement Instruction Example.** To illustrate the use of a DEC instruction in a byte array, this example problem inverts a byte array and places the results in another array of the same size. This example inverts a 26-character array called A1 and places the results in array A2. The contents of A1 are defined with a data TEXT statement to be as follows:

```
A1      TEXT    'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Array A2 is defined with the BSS statement as follows:

```
A2      BSS     26
```

The sample code for the solution is:

```
        LI      5,26            COUNTER AND INDEX FOR A1
        LI      4,A2            ADDRESS OF A2
INVRT   MOVB    @A1(5),*4+      INVERT ARRAY[1]
        DEC     5               REDUCE COUNTER
        JGT     INVRT           CONTINUE IF NOT COMPLETE
         .
         .
         .
```

NOTE

1. @A1(5) addresses elements of array A1 in descending order as workspace register five is decremented. *4+ addresses array A2 in ascending order as workspace register four is incremented.

Array A2 would contain the following as a result of executing this sequence of code:

```
A2  ZYXWVUTSRQPONMLKJIHGFEDCBA
```

Even though the result of this sequence of code is trivial, the example use of the MOVB instruction, with indexing by workspace register five, and the result incrementally placed into A2 with the auto-increment function can be useful in other applications.

The JGT instruction used to terminate the loop allows workspace register 5 to serve both as a counter and as an index register.

A special quality of the DEC instruction allows the programmer to simulate a jump greater than or equal to zero instruction. Since DEC always sets the carry status bit except when changing from zero to minus one, it can be used in conjunction with a JOC instruction to form a JGE loop. The example below performs the same function as the preceding example:

```
A1      TEXT    'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
A2      BSS     26
        LI      5,25            COUNTER AND INDEX FOR A1†
        LI      4,A2            ADDRESS OF A2
INVRT   MOVB    @A1(5),*4+      INVERT ARRAY
        DEC     5               REDUCE COUNTER
        JOC     INVRT           CONTINUE IF NOT COMPLETE
         .
         .
         .
```

†NOTE

Since the use of JOC makes the loop execute when the counter is zero, the counter is initialized to 25 rather than 26 as in the preceding example.

3.89.3.3 Decrement By Two Instruction Example. To illustrate the use of a DECT instruction in processing word arrays, the example problem adds the elements of a word array to the elements of another word array and places the results in the second array. The contents of the two arrays are initialized as follows:

```
A1      DATA    500,300,800,1000,1200,498,650,3,27,0
A2      DATA    36,192,517,29,315,807,290,40,130,1320
```

The sample code that adds the two arrays is as follows:

```
        LI      4,20                INITIALIZE COUNTER1
SUMS    A       @A1-2(4),@A2-2(4)   ADD ARRAYS2
        DECT    4                   DECREMENT COUNTER BY TWO
        JGT     SUMS                REPEAT ADDITION
```

NOTE

Addressing of the two arrays through the use of the @ sign is indexed by the counter, which is decremented after each addition.

The contents of the A2 array after the addition process is as follows:

```
A2  536,492,1317,1029,1515,1305,940,43,157,1320
```

There is another method by which this addition process may be accomplished. This method is shown in the following code:

```
        LI      4,10        INITIALIZE COUNTER1
        LI      5,A1-2      LOAD ADDRESS OF A12
        LI      6,A2-2      LOAD ADDRESS OF A22
SUMS    A       *5+,*6+     ADD ARRAYS3
        DEC     4           DECREMENT COUNTER
        JGT     SUMS        REPEAT ADDITION4
```

NOTE

1. Counter preset to 10 (the number of elements in the array).

2. This address will be incremented each time an addition takes place. The increment is via the auto-increment function (+).

3. The * indicates that the contents of the register is to be used as an address and the + indicates that it will be automatically incremented by two each time the instruction is executed.

4. Workspace register four will only be greater than zero for ten executions of the DEC instruction and control will be transferred to SUMS nine times after the initial execution.

The contents of array A2 are the same for this method as for the first.

3.89.4 SUBROUTINES. There are two types of subroutine linkage available with the Model 990 Computer. One type uses the same set of workspace registers that the calling routine uses, and is called a common workspace subroutine. The BL instruction stores the contents of the program counter and transfers control to the subroutine. Another type is called a context switch subroutine. The BLWP instruction stores the contents of the WP register, the program counter, and the status register. The instruction makes the subroutine workspace active and transfers control to the subroutine.

3.89.4.1 Common Workspace Subroutine Example. Figure 3-1 shows an example of memory contents prior to a BL call to a subroutine. The contents of workspace register 11 is not important to the main routine. When the BL instruction is executed, the CPU stores the contents of the PC in workspace register 11 of the main routine and transfers control to the instruction located at the address indicated by the operand of the BL instruction. This type of subroutine uses the main program workspace. Figure 3-2 shows the memory contents after the call to the subroutine with the BL instruction.

When the instruction at location 1130_{16} is executed (BL @RAD), the present contents of the PC, which point to the next instruction, are saved in workspace register 11. WR11 would then contain an address of 1134_{16}. The PC is then loaded with the address of label RAD, which is address 2220_{16}. This example subroutine returns to the main program with a branch to the address in WR11 (B *11).

3.89.4.2 Context Switch Subroutine Example. Figure 3-3 shows the example memory contents prior to the call to the subroutine. The contents of workspace registers 13, 14, and 15 are not significant. When the BLWP instruction is executed at location 0300, there is a context switch from the main program to the subroutine. The context switch then places the main program PC, WP, and ST register contents in workspace registers 13, 14, and 15 of the subroutine. This saves the environment of the main program for return. The operand of the BLWP instruction specifies that the address vector for the context switch is in workspace registers 5 and 6. The address in workspace register 5 is placed in the WP register and the address in workspace register 6 is placed in the PC.

After the instruction at location 0300 is executed, the memory contents are shown in figure 3-4. This illustration shows the subroutine in control, with the WP pointing to the subroutine workspace and the PC pointing to the first instruction of the subroutine. The contents of the status register are not reset prior to the execution of the first instruction of the subroutine, so the status indicated will actually be the status of the main program execution. A subroutine may then execute in accordance with the status of the main program.

This example subroutine contains a RTWP return from the subroutine. The results of executing the RTWP instruction are shown in figure 3-5. Control is transferred to the main program at the instruction following the BLWP to the subroutine. The status register is restored from workspace register 15 and the workspace pointer points to the workspace of the main program.

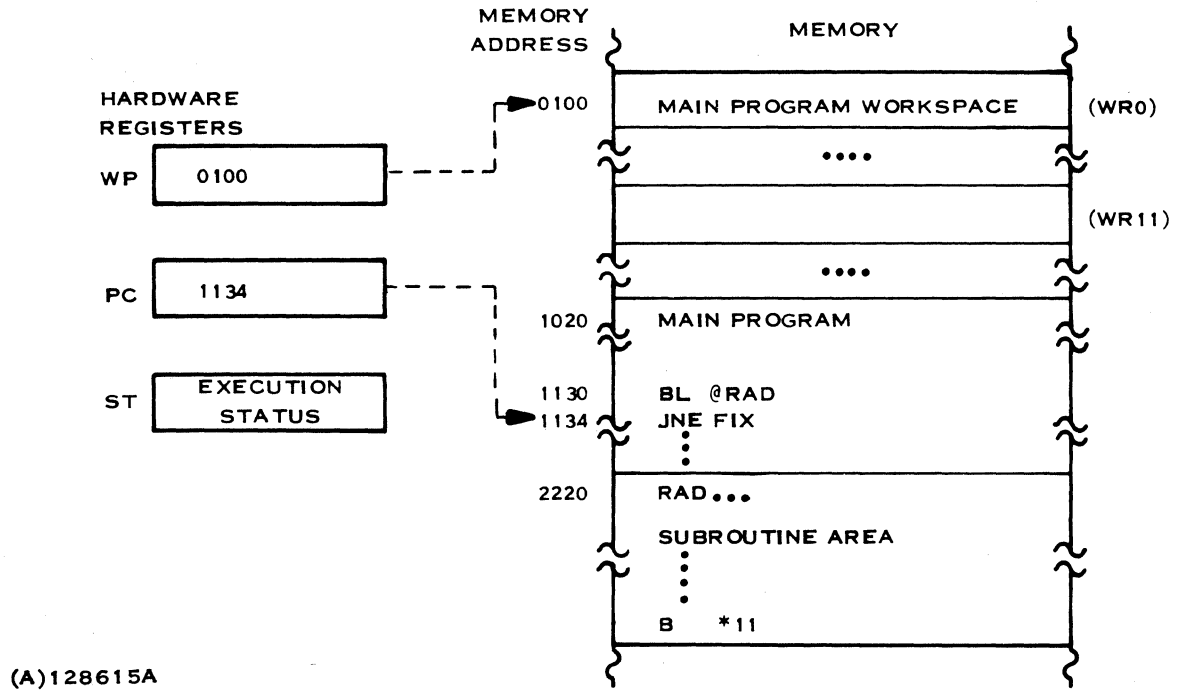


Figure 3-1. Common Workspace Subroutine Example

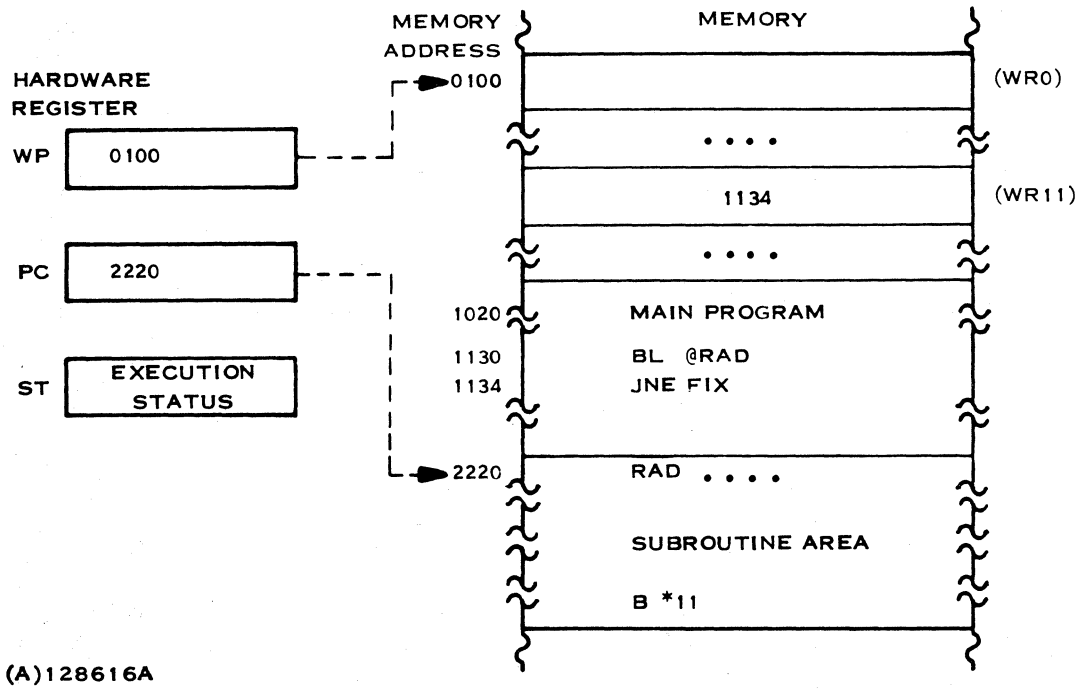


Figure 3-2. PC Contents after BL Instruction Execution

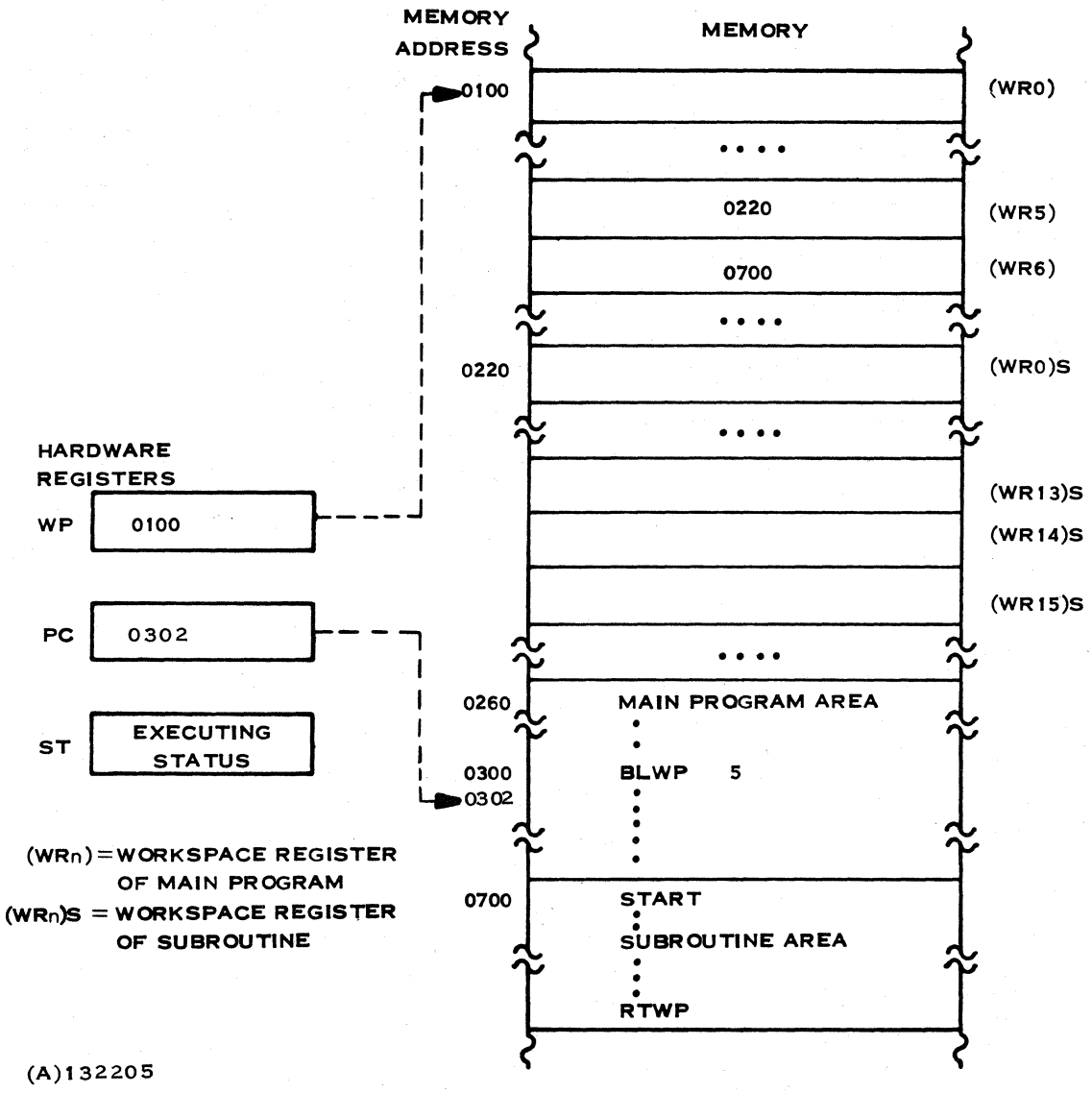


Figure 3-3. Context Switch Subroutine Example

When the calling program's workspace contains data for the subroutine, this data may be obtained by using the indexed memory address mode indexed by workspace register 13. The address used is equal to two times the number of the workspace register that contains the desired data. The following instruction is an example:

```
MOV   @10(13),R10
```

3.89.4.3 **Passing Data to Subroutines.** When a subroutine is entered with a context switch (BLWP) data may be passed using either the contents of workspace register 13 or 14 of the subroutine workspace. Workspace register 13 contains the memory address of the calling program's workspace. The calling program's workspace may contain data to be passed to the subroutine. Workspace register 14 contains the memory address of the next memory location following the BLWP instruction. This location and following locations may contain data to be passed to the subroutine.

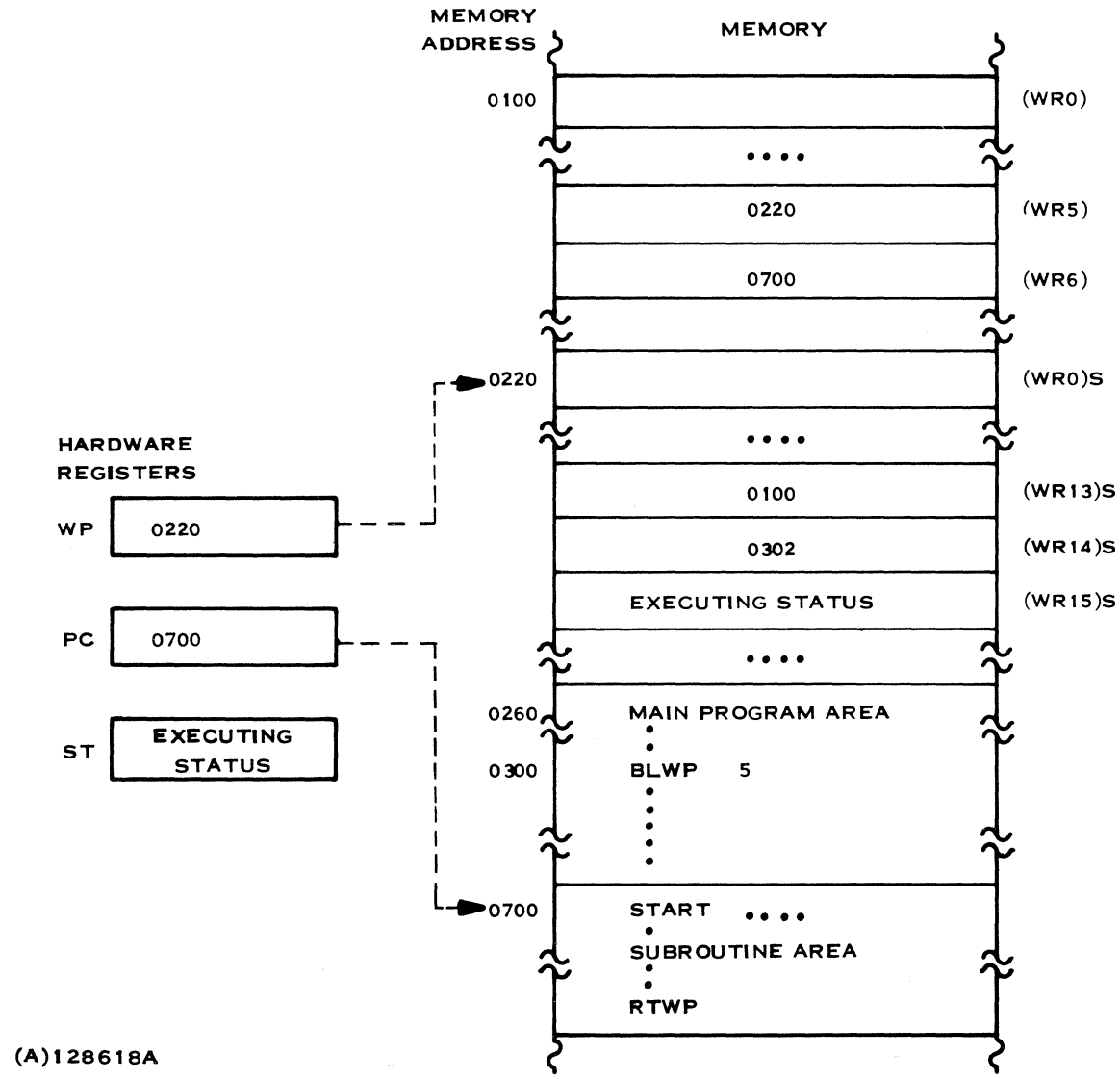


Figure 3-4. After Execution of BLWP Instruction

The contents of workspace register 5 of the calling program's workspace (bytes 10 and 11 relative to the workspace address) are placed in workspace register 10 of the subroutine workspace. This method of data access by subroutines is appropriate for re-entrant procedures.

The following example shows passing of data to a subroutine by placing the data following the BLWP instruction:

```
BLWP    @SUB    SUBROUTINE CALL
DATA    V1      DATA
DATA    V2      DATA
DATA    V3      DATA
JEQ     ERROR   RETURN FROM SUBROUTINE, TEST
  .               FOR ERROR (Subroutine sets the
  .               EQUAL status bit to one for error.)
```

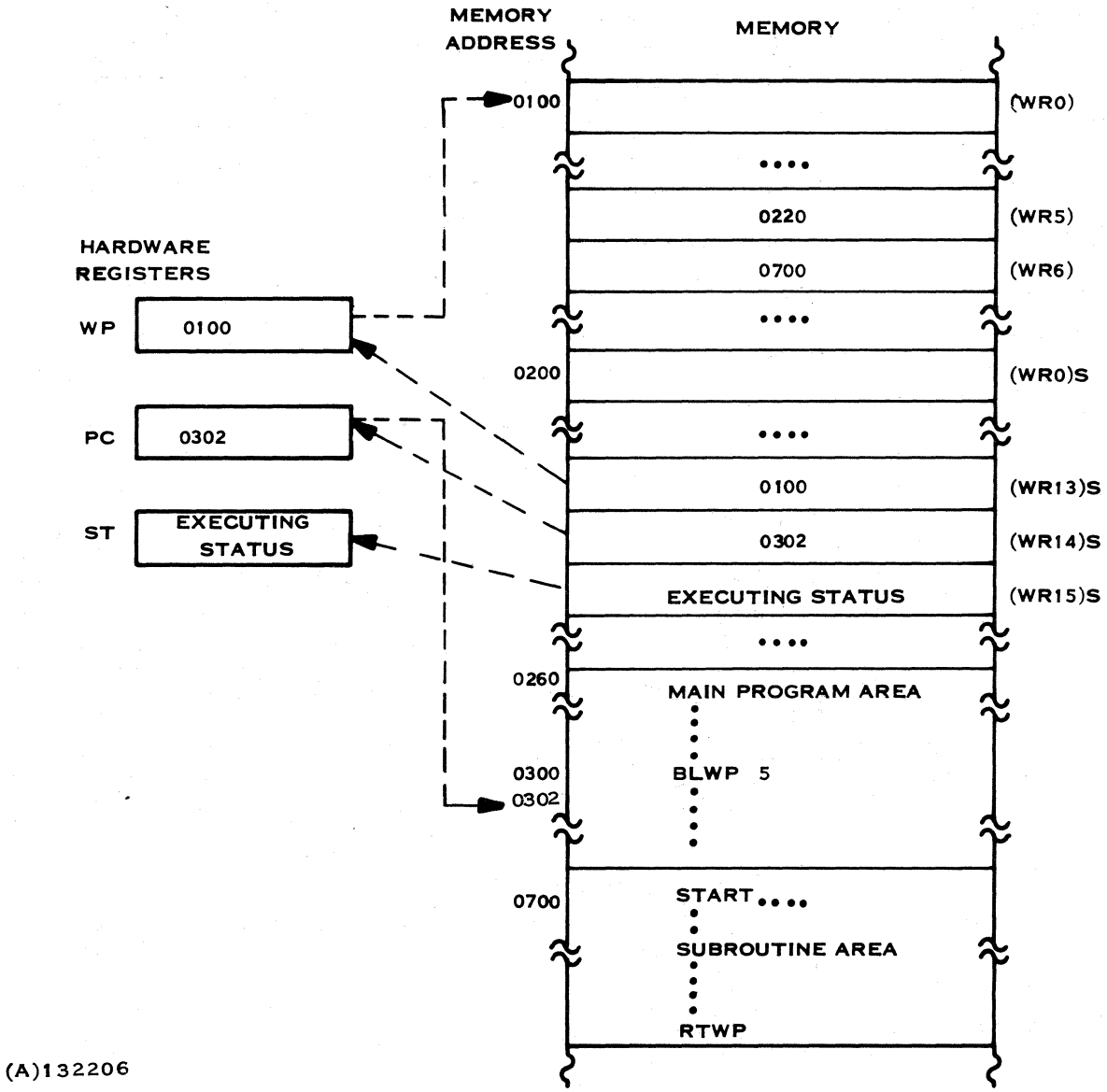


Figure 3-5. After Return Using the RTWP Instruction

```
SUB       DATA    SUBWS,SUBPRG     ENTRY POINT FOR SUB
                                   AND SUB WRKSPCE
            .
            .
            .
            .
SUBWS     BSS     32
SUBPRG    MOV     *14+,1           FETCH V1 PLACED IN WR1
          MOV     *14+,2           FETCH V2 PLACED IN WR2
          MOV     *14+,3           FETCH V3 PLACED IN WR3
            .
            .
            .
          RTWP                     RETURN FROM SUBROUTINE
```

The three MOV instructions retrieve the variables from the main program module and place them in workspace registers one, two, and three of the subroutine.

When the BLWP instruction is executed, the main program module status is stored in workspace register 15 of the subroutine. If the subroutine returns with a RTWP instruction, this status is placed in the status register after the RTWP instruction is executed. The subroutine may alter the status register contents prior to executing the RTWP instruction. The calling program can then test the appropriate bit of the status word, the equal bit in this example, with jump instructions.

A BL instruction can also be used to pass parameters to a subroutine. When using this instruction, the originating PC value is placed in workspace register 11. Therefore, the subroutine must fetch the parameters relative to the contents of workspace register 11 rather than the contents of workspace register 14 as in the BLWP example. The following example demonstrates parameter passing with a BL instruction:

```
        BL      @SUBR           BRANCH TO SUBROUTINE
        DATA    PARM1,PARM2     PASSED PARAMETERS STORED IN NEXT TWO
                                MEMORY WORDS
        JEQ     ERROR           TEST FOR ERROR (Subroutine sets the Equal
                                status bit to one for error)
         .
         .
         .

SUBR    EQU     $
        MOV     *R11+,R0        GET VALUE OF FIRST PARAMETER AND PUT
                                IN WR0
        MOV     *R11+,R1        GET VALUE OF SECOND PARAMETER AND PUT
                                IN WR1 (R11 is incremented past the locations of
                                the two data words and now indicates the address of
                                the next instruction in main program)
         .
         .
         .
        B       *11
```

3.89.5 INTERRUPTS. Either eight (990/4, TMS 9900) or sixteen (990/10) priority vectored interrupt levels are implemented in the Model 990 Computer. The contents of the interrupt mask in the status register define the interrupt level. Low-order memory, address as 0 through 3F, is reserved for transfer vectors used by the interrupts (table 3-4). When an interrupt request at an enabled level occurs, the contents of the transfer vector corresponding to the level are used to enter a subroutine to serve the interrupt.

The reserved memory locations are shown on the memory map (figure 2-3). Two memory words are reserved for each interrupt level. The first of the two words for a given level contains an address that is placed in the WP when the interrupt is requested and enabled. The second contains the entry point of the interrupt subroutine for that level; its contents are placed in the PC. If an executive is in use, it places the transfer vectors for pre-defined interrupts and for devices supported by the executive in the reserved memory locations. The user need not be concerned with transfer vectors for interrupts except for programs that do not execute under an executive or for external devices not supported by the executive. Similarly, the executive includes interrupt subroutines for pre-defined interrupts and for supported devices. The user must supply interrupt subroutines when the executive is not used and for devices that are not supported by the executive.

Table 3-4. Interrupt Vector Addresses

Memory Address	Interrupt Vector	Vector Contents	Typical Assignment
0000	0	WP address for interrupt 0	Power On
0002	0	PC address for interrupt 0	
0004	1	WP address for interrupt 1	Power Failing
0006	1	PC address for interrupt 1	
0008	2	WP address for interrupt 2	Error
000A	2	PC address for interrupt 2	
000C	3	WP address for interrupt 3	External Device
000E	3	PC address for interrupt 3	
0010	4	WP address for interrupt 4	External Device
0012	4	PC address for interrupt 4	
0014	5	WP address for interrupt 5	External Device or Line Frequency Clock
0016	5	PC address for interrupt 5	
0018	6	WP address for interrupt 6	External Device
001A	6	PC address for interrupt 6	
001C	7	WP address for interrupt 7	External Device or Line Frequency Clock (990/4)
001E	7	PC address for interrupt 7	
0020	8	WP address for interrupt 8	External Device
0022	8	PC address for interrupt 8	
0024	9	WP address for interrupt 9	External Device
0026	9	PC address for interrupt 9	
0028	10	WP address for interrupt 10	External Device
002A	10	PC address for interrupt 10	
002C	11	WP address for interrupt 11	External Device
002E	11	PC address for interrupt 11	
0030	12	WP address for interrupt 12	External Device
0032	12	PC address for interrupt 12	
0034	13	WP address for interrupt 13	External Device
0036	13	PC address for interrupt 13	
0038	14	WP address for interrupt 14	External Device
003A	14	PC address for interrupt 14	
003C	15	WP address for interrupt 15	External Device or Line Frequency Clock (990/10)
003E	15	PC address for interrupt 15	

3.89.5.1 General Interrupt Structure. The interrupt levels, numbered 0 through 15, determine the interrupt priority. Level 0 has the highest priority and level 15 the lowest. The contents of the interrupt mask, bits 12 through 15 of the ST register, determine the enabled interrupt levels. Table 3-5 shows the interrupt levels enabled by the contents of the interrupt mask. Note that level 0 cannot be disabled since the level contained in the mask is always enabled.

Table 3-5. Interrupt Mask

Status Register Bits 12-15	Interrupt Levels Enabled	Mask Set By Interrupt Level
0	0	0, 1
1	0, 1	2
2	0, 1, 2	3
3	0, 1, 2, 3	4
4	0, 1, 2, 3, 4	5
5	0, 1, 2, 3, 4, 5	6
6	0, 1, 2, 3, 4, 5, 6	7
7	0, 1, 2, 3, 4, 5, 6, 7	8
8	0, 1, 2, 3, 4, 5, 6, 7, 8	9
9	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	10
A	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10	11
B	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	12
C	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	13
D	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13	14
E	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14	15
F	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15	–

3.89.5.2 **Interrupt Sequence.** The level of the highest priority pending interrupt request is continually compared with the interrupt mask contents. When the level of the pending request is equal to or less than the mask contents (equal or higher priority) the interrupt is taken after the currently executing instruction has completed.

The workspace defined for the interrupt subroutine becomes active and the entry point is placed in the program counter. The CPU also stores the previous contents of the WP register in the new workspace register 13, the previous contents of the program counter in the new workspace register 14, and the contents of the ST register in the new workspace register 15. This preserves the program environment existing when the interrupt is taken. No additional interrupt is taken until the first instruction of the interrupt subroutine is completed. Thereafter, interrupts of higher priority can interrupt processing of the current interrupt.

After storing the ST register contents, the CPU subtracts one from the level of the interrupt taken and places the result in the interrupt mask, disabling the current interrupt level, and leaving only higher priority levels enabled. Should a higher priority level interrupt be taken, and the original interrupt request remain active when the return from the higher priority level interrupt subroutine occurs, the original interrupt remains disabled and is not taken again. Control returns to the interrupt subroutine at the point at which the higher priority interrupt occurred.

3.89.5.3 **Pre-Defined Interrupts.** Level 0 is pre-defined as the power on interrupt in the TMS 9900 Microprocessor and all Model 990 Computers. The other pre-defined levels vary in the Model 990 Computers. Refer to the Model 990 Computer Hardware Reference Manual for the levels that are pre-defined in each model. The total number of levels is 8 in the 990/4 Computer and is 16 in the Model 990/10 or TMS 9900 Microprocessor. The available interrupt levels that are not pre-defined are available for assignment to devices on the CRU, or on the CRU and the TILINE in the case of the Model 990/10. Several interrupt lines may be combined at one level. Any interrupt request must remain active until the interrupt is taken, and must be reset before the interrupt subroutine is completed.

3.89.5.4 **CPU Error Interrupt.** A CPU error interrupt is defined as an interrupt level two. On the 990/4 Computer, two errors cause a CPU error interrupt: a memory parity error or a memory protection violation. Either an SBO or SBZ instruction to bit 12 of the Programmer's panel base address clears a memory parity error interrupt. The base address is selected by placing a $1FE0_{16}$ in register 12.

If the optional write-protect hardware is installed, a CPU interrupt may be caused by a write-protect violation as well as a memory parity error. To determine which condition caused the interrupt, the bit at CRU base address $1FA0_{16}$ can be sensed. If the bit is zero, a parity error has occurred and can be cleared as previously described. If it is a one, a write-protect error has occurred. This error is cleared by setting any of the sixteen bits at CRU base address $1FA0_{16}$ to a one.

On the 990/10 Computer, any one of five conditions can cause a CPU error interrupt. Table 3-6 contains a list of these conditions. To isolate the cause of the error, read the CRU Error Register. The CRU register is addressed by placing a $1FC0_{16}$ in register 12. An individual error is cleared by addressing the appropriate CRU bits as listed in table 3-6. The memory mapping error is cleared by addressing bit 4 at the CRU base address $1FA0_{16}$, the CRU base address used to control the mapping hardware. Either the SBZ or SBO instruction is used to clear the interrupts. To allow software compatability, the memory parity error interrupt in the 990/10 can also be cleared in the manner described for the 990/4.

3.89.5.5 **Interrupt Processing Example.** Refer to figure 3-6 for the following discussion. Prior to the example interrupt (eight for this example), the PC contains 1022 for the executing program, the WP contains 780 for the executing program workspace, and the ST register contains the executing program status. At this point, the example external interrupt, number eight, occurs and there is a context switch from the executing program to the interrupt subroutine. The two words of memory required for external interrupt eight are found in memory locations 0020 and 0022. Figure 3-6 shows that these two words of memory contain 0270 and 0290, respectively, for the WP and PC that are to be used by the interrupt subroutine.

At the point of interrupt, the CPU transfers the present WP, PC, and ST register contents to the interrupt routine workspace in workspace registers 13, 14 and 15, respectively. Once these are stored, the CPU transfers the interrupt subroutine WP and PC into the WP and PC registers.

When these actions are completed, the contents of memory and the registers are as shown in figure 3-7.

After the completion of the interrupt subroutine, the CPU restores the executing program WP, PC, and ST registers. Completion of the interrupt subroutine occurs when the RTWP instruction in the interrupt subroutine is executed.

3.89.6 **EXTENDED OPERATIONS.** Extended operation instructions permit the extension of the existing instruction set to include additional instructions. In the TMS 9900 Microprocessor and the Model 990/4 Computer, these additional instructions are implemented by software routines. In the Model 990/10 Computer, the instructions may be implemented by user-supplied hardware or software routines. Interface between a user program and the standard TI executives is implemented as XOP 15.

Memory locations 0040_{16} through $007E_{16}$ are used for XOP vectors for software implemented XOPs. Vector contents are user supplied WP and PC addresses for the XOP routine workspace and starting address. Table 3-7 contains the addresses and contents of the 16 XOP vectors. Note that these vectors must be supplied and loaded prior to the XOP instruction execution.

Table 3-6. Error Interrupt Logic CRU Bit Assignments

Input Bit	Output Bit	Error Condition
11		Memory Mapping Error
12	12	Error from TILINE memory (parity/error correcting)
13	13	Illegal Operation
14	14	Privileged instruction fetch with privileged mode off
15	15	TILINE timeout

When the program module contains an XOP instruction that is software implemented, the AU locates the XOP WP and PC words in the XOP reserved memory locations and loads the WP and PC. When the WP and PC are loaded, the AU transfers control to the XOP instruction set through a context switch. When the context switch is complete, the XOP workspace contains the calling routine return data in WRs 13, 14, and 15.

The XOP instruction passes one operand to the XOP (input to the XOP routine in workspace register 11 of the XOP workspace). At the completion of the software XOP, the XOP routine should return to the calling routine with an RTWP instruction that will restore the execution environment of the calling routine to that in existence at the call to the XOP.

An example of a software implemented XOP, shown in figure 3-8, causes XOP number two to be executed on the data stored at the address contained in workspace register 1 of the calling program module. Prior to the execution of the XOP, the PC contains the address of the XOP *1, 2 instruction and the WP contains the address of the calling program workspace. At this point, the PC increments by two, to 922, and the XOP is executed. This execution is a context switch in which the XOP routine gains control of the execution sequence. Note that workspace register 1 of the calling program module contains the data address for the operand that is passed to the XOP routine.

After the context switch is complete and the XOP subroutine is in control (figure 3-9), the PC contains the starting address of the XOP subroutine and the WP contains the address of the XOP subroutine workspace. Workspace register 11 of the XOP subroutine contains the effective address of the data to be used as an operand. Workspace registers 13, 14, and 15 contain the return control information, which is used to return control to the main program module when the XOP subroutine completes execution.

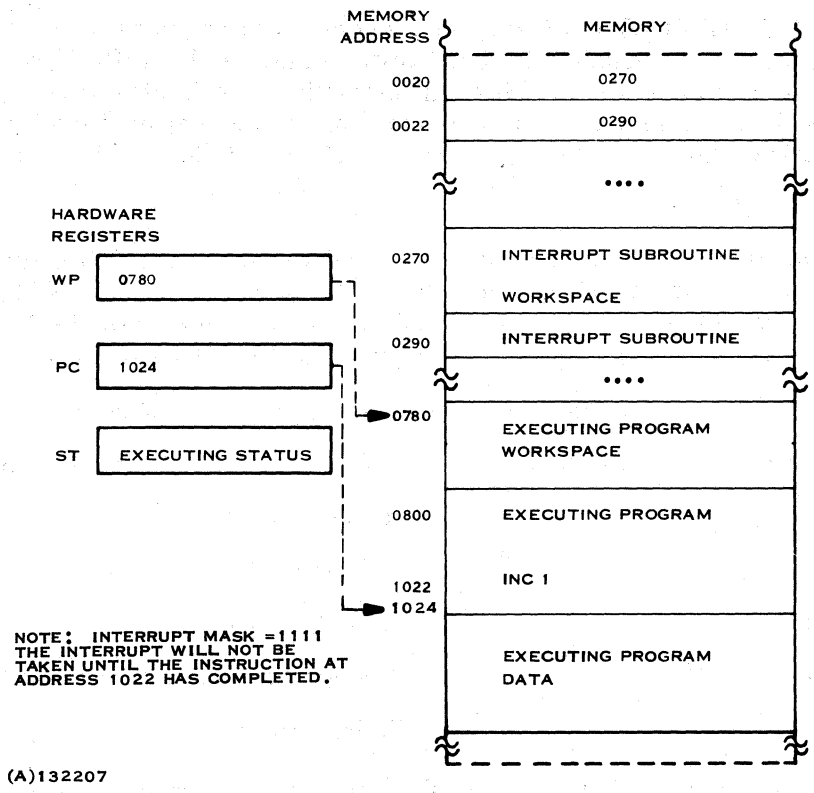


Figure 3-6. Interrupt Processing Example

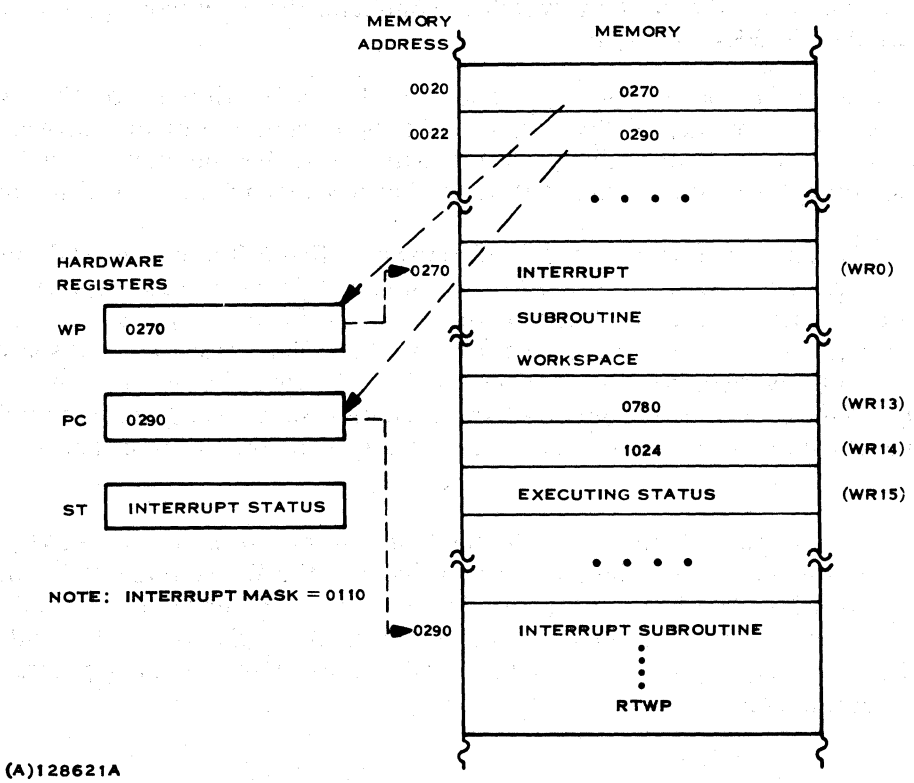


Figure 3-7. Memory Contents After Interrupt

Table 3-7. XOP Vectors

Memory Address	XOP Number	Vector Contents
0040	0	WP address for XOP workspace
0042	0	PC address for XOP routine
0044	1	WP address for XOP workspace
0046	1	PC address for XOP routine
0048	2	WP address for XOP workspace
004A	2	PC address for XOP routine
004C	3	WP address for XOP workspace
004E	3	PC address for XOP routine
0050	4	WP address for XOP workspace
0052	4	PC address for XOP routine
0054	5	WP address for XOP workspace
0056	5	PC address for XOP routine
0058	6	WP address for XOP workspace
005A	6	PC address for XOP routine
005C	7	WP address for XOP workspace
005E	7	PC address for XOP routine
0060	8	WP address for XOP workspace
0062	8	PC address for XOP routine
0064	9	WP address for XOP workspace
0066	9	PC address for XOP routine
0068	10	WP address for XOP workspace
006A	10	PC address for XOP routine
006C	11	WP address for XOP workspace
006E	11	PC address for XOP routine
0070	12	WP address for XOP workspace
0072	12	PC address for XOP routine
0074	13	WP address for XOP workspace
0076	13	PC address for XOP routine
0078	14	WP address for XOP workspace
007A	14	PC address for XOP routine
007C	15	WP address for XOP workspace
007E	15	PC address for XOP routine

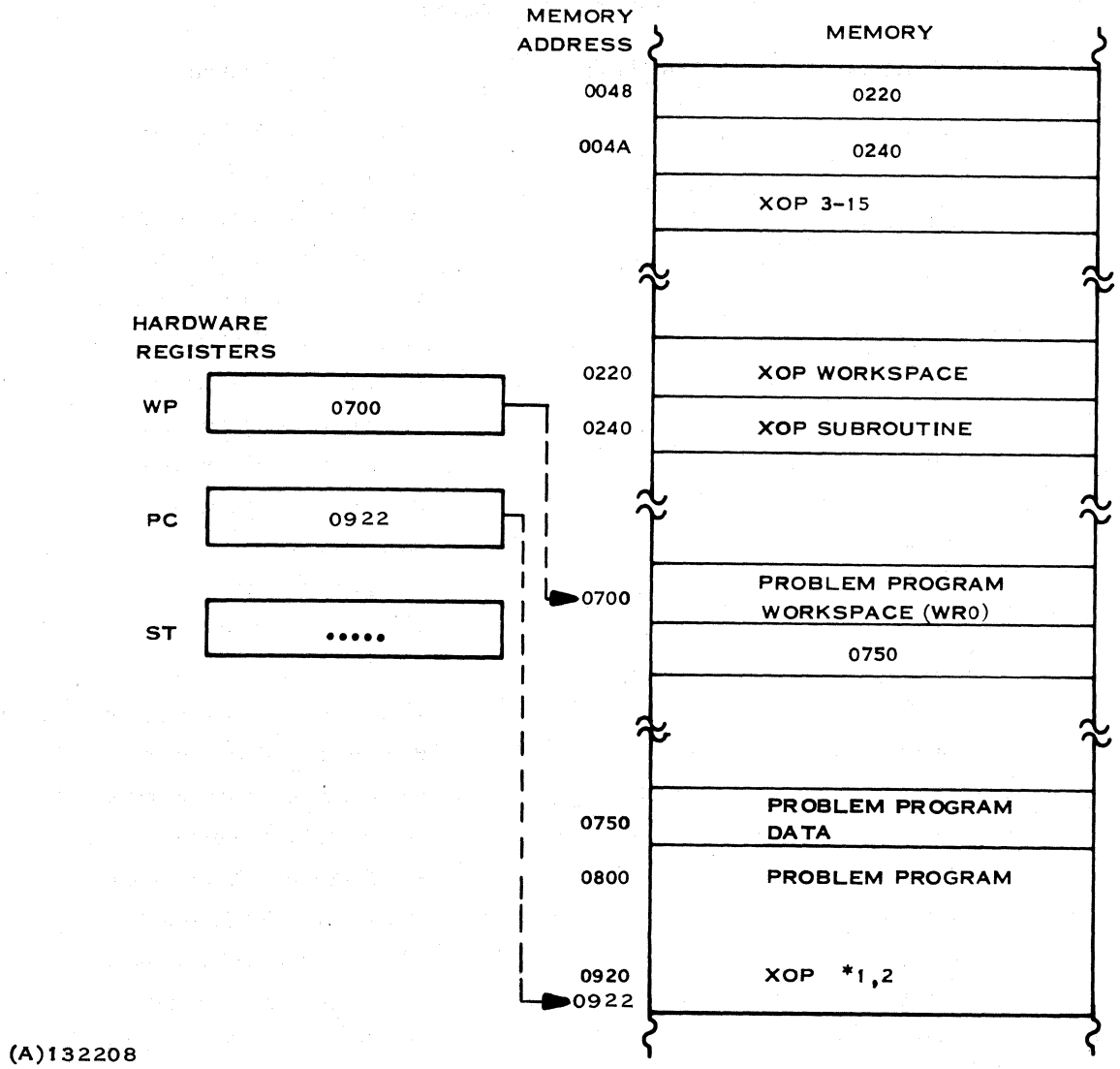


Figure 3-8. Extended Operation Example

3.89.7 **SPECIAL CONTROL INSTRUCTIONS.** There are five special control instructions that permit the programmer to control the state of the execution process of the 990 Computer. These instructions are:

Instruction	Mnemonic
Load or Restart Execution	LREX
Clock On and Clock Off	CKON/CKOF
Reset	RSET
Execute	X
Idle	IDLE

CAUTION

In Model 990/4 Computers, executing any of these instructions except Execute in a program executing under an executive may

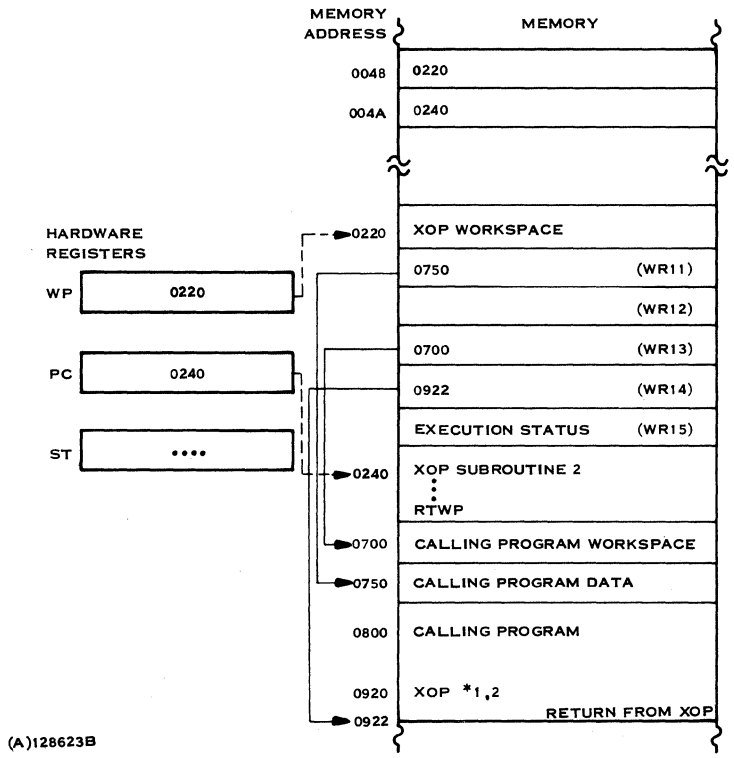


Figure 3-9. Extended Operation Example after Context Switch

drastically interfere with the executive's operation. Executives running in a Model 990/10 Computer allow program execution only in a nonpriviledged mode. Attempting to execute these instructions in a nonprivileged mode generates a error/interrupt.

In the TMS 9900 Microprocessor, only the Execute instruction applications apply. The other instructions perform no processing in the microprocessor, but may be implemented in the users hardware to perform any desired functions.

3.89.7.1 LREX Applications. The LREX instruction may be used to activate any desired function by placing a transfer vector for that function in addresses $FFFC_{16}$ and $FFFE_{16}$ and placing a subroutine and workspace to perform that function in the locations specified in the transfer vector. Typically, these locations are ROM locations, and the LREX instruction activates a programmer's panel and loader function. Other functions could be performed either by using different ROM's in these locations, or by using RAM in these locations and loading the desired data into the locations.

The LREX instruction is not implemented in the TMS 9900 Microprocessor, and is a Privileged Mode instruction in the Model 990/10 Computer.

3.89.7.2 CKON/CKOF Applications. These two instructions are used to turn on and turn off the clock, respectively. Through the use of these two instructions, the programmer may use the clock for timing operations. As an example, the clock may be used to time-out I/O procedures by turning the clock on, counting the clock interrupts until the desired time is passed, and turning the clock off. This is possible only if the interrupt level for the real time clock has previously been enabled.

The clock interrupt is normally attached to level 5, or optionally at level 7 on the 990/4 Computer or level 15 on the 990/10 Computer. The interrupt is normally cleared in the Clock Interrupt Service Routine with a CKOF/CKON instruction sequence.

The RSET instruction also clears an interrupt.

When a program executes under an executive, the executive uses the clock for timing various executive and user program functions. Executing either a CKON or a CKOF instruction interferes with normal operation of the executive. I/O timeout is part of the support provided by the executive, and is not a user function. Refer to the user's guide for the appropriate executive for methods of timing user program functions supported by that executive.

The CKON and CKOF instructions are not implemented on the TMS 9900 Microprocessor, and are Privileged Mode instructions in the Model 990/10 Computer with map option.

3.89.7.3 RSET Applications. RSET is primarily used to initialize the state of the computer and has the effect of clearing any pending interrupts. This instruction is useful at the start of a program to clear the state in existence so that the new application will not be adversely affected by the previous state of the computer.

When a program executes under an executive, the executive processes internal interrupts and external interrupts for supported devices. Execution of an RSET instruction interferes with normal operation of the executive. Refer to the user's guide for the appropriate executive for permissable changes in the enabled interrupt level.

The RSET instruction is not implemented in the TMS 9900 Microprocessor, and is a Privileged Mode instruction in the Model 990/10 with map option.

3.89.7.4 X Applications. The execute instruction may be used to execute an instruction that is not in sequence without transferring control to the desired instruction. One useful application is to execute one of a table of instructions, selecting the desired instruction by using an index into the table of instructions. The computed value of the index determines which instruction is executed.

A table of shift instructions is an example of the use of the X instruction. Place the following instructions at location TBLE:

```
TBLE    SLA     R6,3      SHIFT WORKSPACE REGISTER 6
        SLA     R7,3      SHIFT WORKSPACE REGISTER 7
        SLA     R8,3      SHIFT WORKSPACE REGISTER 8
TABEND  EQU     $
```

A character is placed in the most significant byte of workspace register 5 to select the workspace register to be shifted to the left 3 bit positions. ASCII characters A, B, and C specify shifting workspace registers 6, 7, and 8, respectively. Other characters are ignored. The following code performs the selection of the shift desired:

```
        SRL     R5,8      MOVE TO LOWER BYTE
        AI      R5,-'A'   SUBTRACT TABLE BIAS
        JLT     NOSHFT    ILLEGAL
        SLA     R5,1      MAKE IT A WORD INDEX
        CI      R5, TABEND - TBLE
        JGT     NOSHFT    ILLEGAL
        X       @TBLE(R5)
NOSHFT  EQU     $
```

When using the X instruction, if the substituted instruction contains a T_s field or a T_d field that results in a two word instruction, the computer accesses the word following the X instruction as the second word, not the word following the substituted instruction. When the substituted instruction is a jump instruction with a displacement, the displacement must be computed from the X instruction, not from the substituted instruction.

3.89.8 CRU INPUT/OUTPUT. The communications register unit (CRU) performs single and multiple bit programmed input/output in the Model 990 Computer. All input consists of reading CRU line logic levels into memory and output consists of setting CRU output lines to bit values from a word or byte of memory. The CRU provides a maximum of 4096 input and output lines that may be individually selected by a 12-bit address. The 12-bit address is located in bits 3 through 14 of workspace register 12 and is the base address for all CRU communications.

When a program executes under an executive, I/O to supported devices is provided through the use of I/O supervisor calls. For these CRU devices, it is not necessary to use the instructions described in the following paragraphs. Refer to the appropriate user's guide for information on the use of the I/O supervisor call to the desired device under that executive.

3.89.8.1 CRU I/O Instructions. There are five instructions for communications with CRU lines. They are:

- SBO - Set CRU Bit To One. This instruction sets a CRU output line to a logic one. If the device on the CRU line is a data module, SBO results in zero volts at the data module terminal corresponding to the addressed bit.

- SBZ - Set CRU Bit To Zero. This instruction sets a CRU output line to a logic zero. If the device on the CRU line is a data module, SBZ results in a float (no signal applied) at the data module terminal corresponding to the addressed bit.

- TB - Test CRU Bit. This instruction reads the digital input bit and sets the equal status bit (bit 2) to the value of the digital input bit.

NOTE

The CRU address of the SBO, SBZ, and TB instructions is determined as follows:

Bits 3-14 of workspace register 12 equal the CRU base address

+

The user supplied displacement in the instruction with sign bit extended

=

Effective CRU address

- LDCR - Load Communications Register. This instruction transfers the number of bits (1-16) specified by the C field of the instruction onto the CRU from the source operand. When less than nine bits are specified, the source operand address is a byte address. When more than eight bits are specified, the source operand is a word address. The CRU address is the address of the first CRU digital output affected. The CRU address is determined by the contents of workspace register 12, bits 3 through 14.

- STCR - Store Communications Register. This instruction transfers the number of bits specified by the C field of the instruction from the CRU to the source operand. When less than nine bits are specified, the source operand address is a byte address. When there are nine or more bits specified, the source operand address is a word address. The CRU address is determined by workspace register 12, bits 3 through 14.

3.89.8.2 **SBO Example.** Assume that a control device that turns on a motor when the computer sets a one on CRU line $10F_{16}$, and that workspace register 12 contains 0200_{16}, making the base address in bits 3 through 14 equal to 100_{16}. The following instruction sets CRU line $10F_{16}$ to one:

```
SBO     15
```

If a data module were connected as the CRU device, the instruction would place zero volts on output line 15 of the module without affecting other lines.

3.89.8.3 **SBZ Example.** Assume that a control device that shuts off a valve when the computer sets a zero on a CRU line is connected to CRU line 2, and that workspace register 12 contains zero. The following instructions sets CRU line 2 to zero:

```
SBZ     2
```

If a data module were connected as the CRU device, output line 2 of that module would float at a voltage determined by the characteristics of the control device. No other CRU line would be affected by the instruction.

3.89.8.4 **TB Example.** Assume that workspace register 12 contains 0140_{16}, making the base address in bits 3 through 14 equal to $A0_{16}$. The following instructions would test the input on CRU line $A4_{16}$ and execute the instructions beginning at location RUN when the CRU line is set to one. When the CRU line is set to zero, execute the instructions beginning at location WAIT:

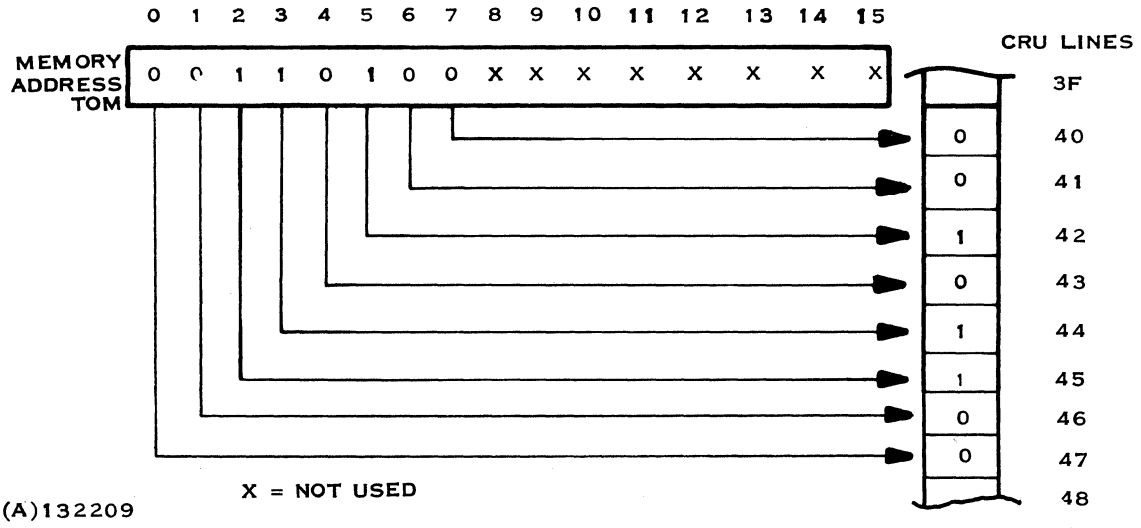
```
        TB      4       TEST CRU LINE 4
        JEQ     RUN     IF ON, GO TO RUN
WAIT    .               IF OFF, CONTINUE
        .
        .
RUN     .
```

The TB instruction sets the logic level of the Equal bit of the ST register to the level on line 4 of the CRU device.

3.89.8.5 **LDCR Example.** Assume that a 913 CRT Display Terminal is connected to the CRU and that the base address in workspace register 12 is set to CRU line 48_{16}. The following instructions display a character in an even address at location TOM on the screen of the CRT. Output CRU lines 40_{16} through 47_{16} must be set to the bit configuration of the character, which requires that the base address in bits 3 through 14 of workspace register 12 be modified. The instructions are:

```
AI      R12,-16     MODIFY BASE ADDRESS BY 8
LDCR    @TOM,8      TRANSFER CHARACTER
AI      R12,16      RESTORE BASE ADDRESS
```

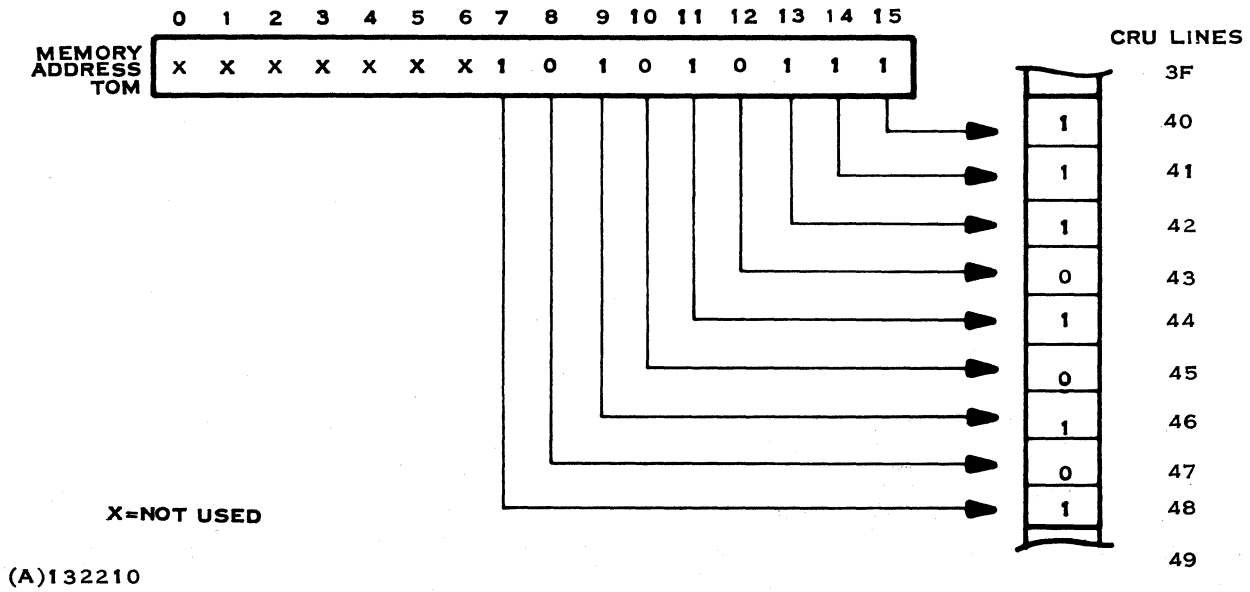
The operand required in the first instruction is -16 because the least significant bit of workspace register 12 is not included in the base address. The base address must be decremented by 8, so 16 must be subtracted. The following diagram shows the transfer of data, which places the character in the proper register of the CRT controller. The Write Data Strobe line, CRU output line 48_{16}, must be set to actually display the character.



If the LDCR instruction were changed as follows:

```
LDCR    @TOM,9
```

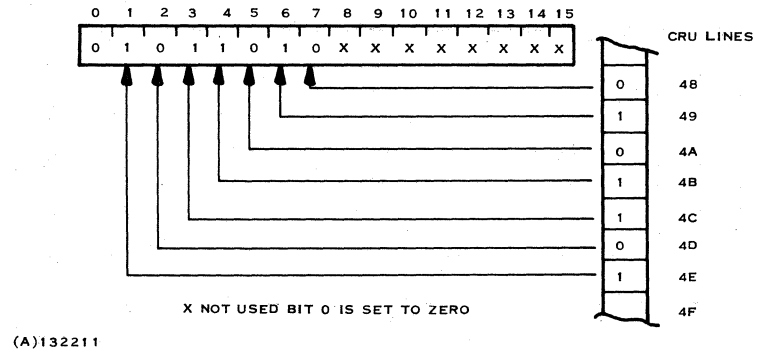
there would be a transfer of 9 bits beginning with the least significant bit of address TOM to nine CRU lines, 40_{16} through 48_{16}. Setting bit 48_{16} to either a value of 0 or 1 causes the character to be displayed on the screen. The following diagram shows the data transfer:



3.89.8.6 STCR Example. The last AI instruction of the LDCR example in the preceding paragraph left the base address in workspace register 12 set for a keyboard input operation. The following instruction places the seven bits of the keyboard character into the seven least significant bits of the byte at the address in workspace register 2:

```
STCR   *R2,7        READ CHARACTER
```

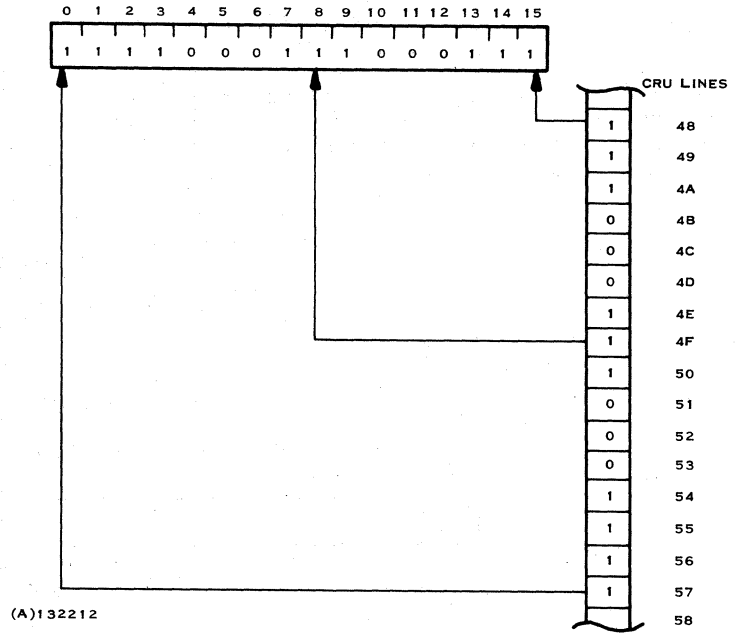
The STCR instruction stores the bits as shown in the following diagram:



If the STCR instruction were changed to:

```
STCR   *R12,0
```

sixteen bits would be transferred from the CRU lines specified by workspace register 12 to the address that is specified by the contents of workspace register 2. The transfer of data is shown in the following diagram:



The keyboard character is placed in the least significant byte.

3.89.9 **TILINE INPUT/OUTPUT.** (990/10 ONLY). The set of machine instructions that communicate with the memory may be used to communicate with devices connected to the TILINE, as illustrated in appendix I. To communicate with the TILINE device, these instructions must be coded with the TILINE addresses for the device. The hardware supplies the five most significant bits, each having the value of one, to convert the upper 1024 memory byte addresses to TILINE addresses. The actual TILINE addresses for a device and the significance of data transferred to these addresses are device dependent.

The Disc Executive DX10 supports I/O to the available disc units. The user programs that execute under DX10 use the I/O supervisor call to perform I/O to the disc. Refer to the user's guide for the appropriate executive for a description of the I/O supervisor call and for a list of supported devices.

3.89.10 **RE-ENTRANT PROGRAMMING.** Re-entrant programming is a technique that allows the same program code to be used for several different applications while maintaining the integrity of the data used with each application. The common program code and its associated constants are stored in one area in memory. Each function that uses that code is then assigned a unique workspace and data area so that as it executes the common code, its variable data is developed without affecting the variable data associated with any of the other functions that use the program. With this arrangement one function can execute the common code routine and be interrupted in the middle of the routine by another function that also uses the same routine. The second function then uses the routine for its purpose and returns control to the first function so that it can proceed from the point of interruption without returning to the start of the routine. Re-entrant programming of this type lends itself well to servicing similar peripheral devices that interface with the computer at different priority levels. The following characteristics apply to a re-entrant procedure:

- The procedure does not contain data except data common to all tasks.
- The procedure does not alter the contents of any word in the procedure whether that word contains data or an instruction.
- Data that is unique to one or more tasks is in the data division for the task and is either in a workspace or is indirectly addressed.

A very important application of a re-entrant procedure is one that controls a process using several sets of identical control devices through identical sets of CRU lines. Each task using the re-entrant procedure addresses a unique set of control devices that controls a set of equipment to perform the same process concurrently. The workspace for each task contains the CRU base address in workspace register 12 for the set of control devices for the task. The procedure addresses a control device by a displacement from the base address. For each task, the base address in workspace register 12 of its workspace controls the proper device. Figure 3-10 shows a procedure common to sixteen tasks, each of which uses an identical set of CRU lines at different CRU base addresses.

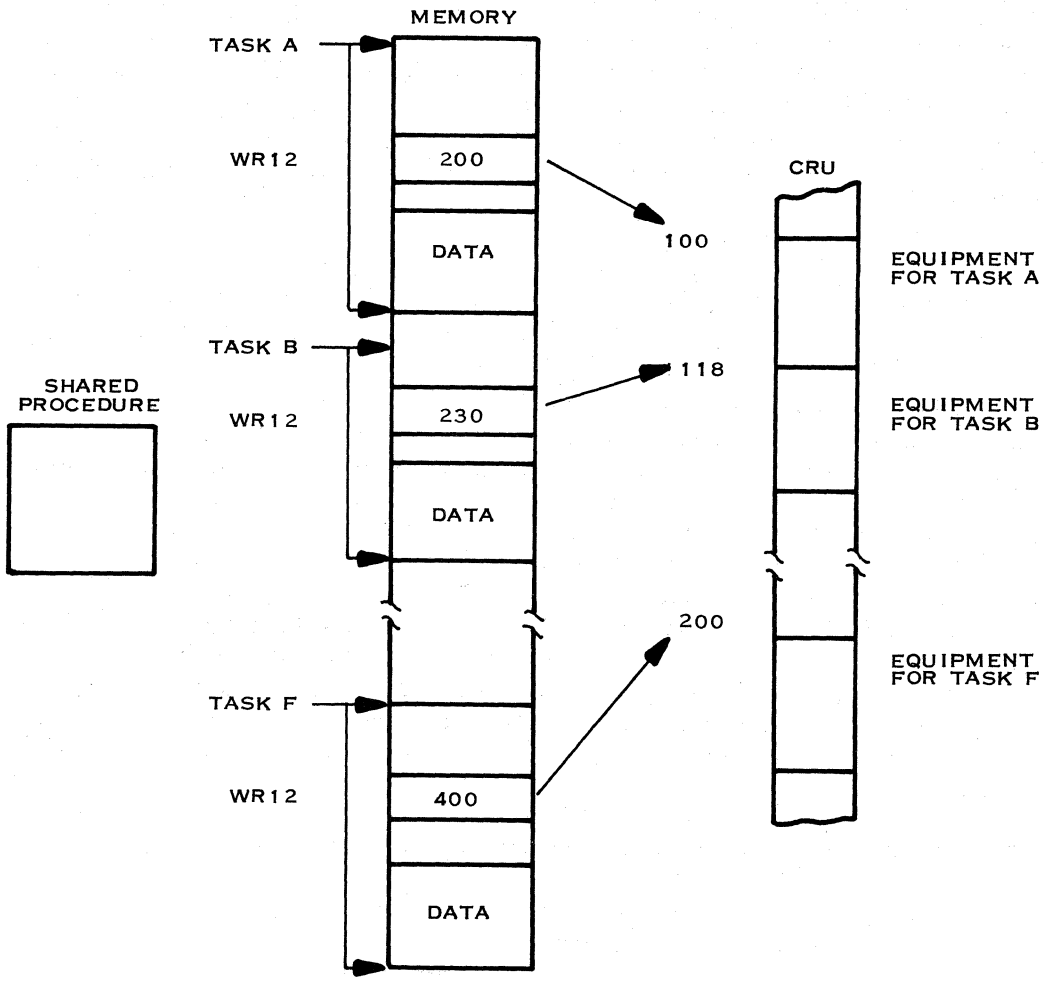


Figure 3-10. Re-entrant Procedure for Process Control

The following is an example of re-entrant code. The following assumptions apply:

- Workspace register 14 contains the address of a word that contains the size of a buffer, in bytes.
- Workspace register 9 contains the start address of that buffer.
- Label NOTFND is the location that contains the first instruction of a routine that is to be executed if the buffer does not contain a carriage return character.
- Label FOUND is the location of the first instruction of a routine that is to be executed when the buffer contains a carriage return.

The re-entrant code is as follows:

```
ENTER     MOV     *14,3          GET BUFFER SIZE
          MOV     9,8            GET START ADDRESS
          A       3,8            POINT TO END OF BUFFER
LOOK      C       9,8            CHECK FOR END
          JH      NOTFND         BRANCH AT END
          CB      *9+,@CARRET    CHECK CHARACTER
          JNE     LOOK           BRANCH WHEN NOT FOUND
FOUND                            CHARACTER FOUND
            .
            .
            .
CARRET    BYTE    >D
```

The code is re-entrant because it is not altered during execution of the code. Also, when execution resumes following an interruption, the workspace for the code again becomes active, and contains the correct values for resuming the execution as if execution had not be interrupted.

Another possible version of the same code is as follows:

```
ENTER     MOV     *14,@ADDLOC
          MOV     9,8
          AI      8,$-$
ADDLOC    EQU     $-2
LOOK      C       9,8
          JH      NOTFND
          CB      *9+,@CARRET
          JNE     LOOK
FOUND
            .
            .
            .
CARRET    BYTE    >D
```

The code performs the same function by storing the buffer length in the word that contains the immediate operand of an AI instruction. As long as only one task using this code is active, there would be no problem. However, if one task is interrupted after storing a value in ADDLOC and before executing the AI instruction, and another task executes the code, the size of the buffer for the first task is lost. The code is not re-entrant because it alters data within itself.

SECTION IV

ASSEMBLER DIRECTIVES

4.1 INTRODUCTION

Assembler directives and machine instructions in source programs supply data to be included in the program and control of the assembly process. The Model 990 Computer Assemblers support a number of directives in the following categories:

- Directives that affect the Location Counter
- Directives that affect the assembler output
- Directives that initialize constants
- Directives that provide linkage between programs
- Miscellaneous directives.

4.2 DIRECTIVES THAT AFFECT THE LOCATION COUNTER

As an assembler reads the source statements of a program, a component of the assembler called the location counter advances to correspond to the memory locations assigned to the resulting object code. The first nine of the assembler directives listed below initialize the location counter and define the value as relocatable, absolute, or dummy. The last three directives advance the location counter to provide a block or an area of memory for the object code to follow. The word boundary directive also ensures a word boundary (even address). The directives are:

- Absolute Origin
- Relocatable Origin
- Dummy Origin
- Data Segment
- Data Segment End
- Common Segment
- Common Segment End
- Program Segment
- Program Segment End
- Block Starting With Symbol
- Block Ending With Symbol
- Word Boundary

NOTE

The following are not supported by the PX9ASM.

- Data Segment
- Data Segment End
- Common Segment
- Common Segment End
- Program Segment
- Program Segment End

4.2.1 ABSOLUTE ORIGIN AORG

Syntax definition:

[<label>]b . . . AORGb . . . <wd-exp>b . . . [<comment>]

AORG places a value in the location counter and defines the succeeding locations as absolute. Use of the label field is optional. When a label is used, it is assigned the value that the directive places in the location counter. The operation field contains AORG. The operand field contains a well-defined expression (wd-exp). The assembler places the value of the well-defined expression in the location counter. Use of the comment field is optional. When no AORG directive is entered, no absolute addresses are included in the object program.

The following example shows an AORG directive:

```
AORG    >1000+X
```

Symbol X must be absolute and must have been previously defined. If X has a value of 6, the location counter is set to 1006_{16} by this directive. Had a label been included, the label would have been assigned the value 1006_{16}.

4.2.2 RELOCATABLE ORIGIN RORG

Syntax definition:

[<label>] b. . .RORG b. . .[<exp>] b. . .[<comment>]

RORG places a value in the location counter; if encountered in absolute code, it also defines succeeding locations as program-relocatable. When a label is used, it is assigned the value that the directive places into the location counter. The operation field contains RORG, and the operand field is optional. The comment field may be used only when the operand field is used.

When the operand field is not used, the length of the program segment, data segment, or specific common segment of a program replaces the value of the location counter. For a given relocation type X, the length of the X-relocatable segment at any time during an assembly is either of the following values:

- The maximum value the location counter has ever attained as a result of the assembly of any preceding block of X-relocatable code.
- Zero, if no X-relocatable code has been previously assembled.

Clearly, since the location counter begins at zero, the length of a segment and the "next available" address within that segment are identical.

If the RORG directive appears in absolute- or program-relocatable code and the operand field is not used, the location counter value is replaced by the current length of the program segment of that program. If the directive appears in data-relocatable code without an operand, the location counter value is replaced by the length of the data segment. Likewise, in common-relocatable code, the RORG directive without an operand causes the length of the appropriate common segment to be loaded into the location counter.

When the operand field is used, the operand must be an absolute or relocatable expression (exp) that contains only previously defined symbols. If the directive is encountered in absolute code, a relocatable operand must be program-relocatable; in relocatable code, the relocation type of the operand must match that of the current location counter. When it appears in absolute code, the RORG directive changes the location counter to program-relocatable and replaces its value with the operand value. In relocatable code, the operand value replaces the current location counter value, and the relocation type of the location counter remains unchanged.

The following example shows an RORG directive:

```
RORG    $-20    OVERLAY TEN WORDS
```

The $ symbol refers to the location following the preceding relocatable location of the program. This has the effect of backing up the location counter ten words. The instructions and directives following the RORG directive replace the ten previously assembled words of relocatable code, permitting correcting of the program without removing source records. Had a label been included, the label would have been assigned the value placed in the location counter.

An example of a RORG directive with no operand field is as follows:

```
SEG2    RORG
```

The location counter contents depend upon preceding source statements. Assume that after defining data for a program, which occupied 44_{16} bytes, an AORG directive initiated an absolute block of code. The absolute block is followed by the RORG directive in the above example. This places 0044_{16} in the location counter and defines the location counter as relocatable. Symbol SEG2 is a relocatable value, 0044_{16}. The RORG directive in the above example would have no effect except at the end of an absolute block or a dummy block, described in the next paragraph.

4.2.3 DUMMY ORIGIN DORG

Syntax definition:

[<label>]b . . . DORGb . . . <exp>b . . . [<comment>]

DORG places a value in the location counter and defines the succeeding locations as a dummy block or section. When assembling a dummy section, the assembler does not generate object code, but operates normally in all other respects. The result is that the symbols that describe the layout of the dummy section are available to the assembler during assembly of the remainder of the program. The label is assigned the value that the directive places in the location counter. The operation field contains DORG. The operand field contains an expression (exp), which may be either absolute or relocatable. Any symbol in the expression must have been previously defined. When the operand is absolute, the location counter contents are absolute; when the operand is relocatable, the location counter contents are relocatable.

The following example shows a DORG directive:

```
DORG    0
```

The effect of this directive is to cause the assembler to assign values relative to the start of the dummy section to the labels within the dummy section. It is assumed that the code corresponding to the dummy section is assembled in another program module.

The example directive would be appropriate in the executable portion (procedure division) of a disc-resident task that is common to more than one task, and which executes under the disc executive. The dummy section of the procedure should contain the directives of the data division, and the executable portion of the module (following a RORG directive) should use the labels of the dummy section as indexed addresses. In this manner, the data is available to the procedure regardless of the memory area into which the data is loaded.

The DORG directive may also be used with data-relocatable or common-relocatable operands to specify dummy data or common segments. The following example illustrates this usage:

```
        CSEG 'COM1'

        DORG $    "$" HAS A COMMON-RELOCATABLE VALUE
          .
          .
          .
          .
LAB1 DATA $

MASK DATA >F000
          .
          .
          .
          .
        CEND

        SZC @MASK,@LAB1(R3)
```

In the example, no object code is generated to initialize the common segment, COM1, but all common-relocatable labels describing the structure of the common block (including LAB1 and MASK) are available for use throughout the program.

4.2.4 BLOCK STARTING WITH SYMBOL BSS

Syntax definition:

[<label>]b . . . BSSb . . . <wd-exp>b . . . [<comment>]

BSS advances the location counter by the value of the well-defined expression (wd-exp) in the operand field. Use of the label field is optional. When a label is used, it is assigned the value of the location of the first byte in the block. The operation field contains BSS. The operand field contains a well-defined expression that represents the number of bytes to be added to the location counter. The comment field is optional.

The following example shows a BSS directive:

```
BUFF1   BSS 80    CARD INPUT BUFFER
```

This directive reserves an 80-byte buffer at location BUFF1.

4.2.5 BLOCK ENDING WITH SYMBOL BES

Syntax definition:

[<label>]b . . . BESb . . . <wd-exp>b . . . [<comment>]

BES advances the location counter according to the value in the operand field, and assigns the new location counter value to the symbol in the label field, when there is a symbol in the label field. Use of the label field is optional. The label is assigned the value of the location following the block when the label is entered. The operation field contains BES. The operand field contains a well-defined expression that represents the number of bytes to be added to the location counter. The comment field is optional.

The following example shows a BES directive:

```
BUFF2   BES   >10
```

The directive reserves a 16-byte buffer. Had the location counter contained 100_{16} when the assembler processed this directive, BUFF2 would have been assigned the value 110_{16}.

4.2.6 WORD BOUNDARY EVEN

Syntax definition:

[<label>]b . . . EVENb . . . [<comment>]

EVEN places the location counter on the next word boundary (even) byte address. When the location counter is already on a word boundary, the location counter is not altered. Use of the label field is optional. When a label is used, the value in the location counter before processing the directive is assigned to the label. The operation field contains EVEN. The operand field is not used, and the comment field is optional.

The following example shows an EVEN directive:

```
WRF1    EVEN    WORKSPACE REGISTER FILE ONE
```

The directive assigns the location counter address to label WRF1, and assures that the location counter contains a word boundary address. Use of an EVEN directive preceding or following a machine instruction or a DATA directive is redundant. The assembler advances the location counter to an even address when it processes a machine instruction or a DATA directive.

4.2.7 DATA SEGMENT DSEG

Syntax definition:

[<label>] b. . .DSEG b. . .[<comment>]

NOTE

This directive does not apply to the PX9ASM assembler.

DSEG places a value in the location counter and defines succeeding locations as data-relocatable. Use of the label field is optional. When a label is used, it is assigned the data-relocatable value that the directive places in the location counter. The operation field contains DSEG. The operand field is not used, and the comment field is optional. Either of the following values is placed in the location counter:

- The maximum value the location counter has ever attained as a result of the assembly of any preceding block of data-relocatable code
- Zero, if no data-relocatable code has been previously assembled.

The DSEG directive defines the beginning of a block of data-relocatable code. The block is normally terminated with a DEND directive (see paragraph 4.2.8). If several such blocks appear throughout the program, they together comprise the data segment of the program. The entire data segment may be relocated independently of the program segment at link edit time and therefore provides a convenient means of separating modifiable data from executable code.

In addition to the DEND directive, the following directives will properly terminate the definition of a block of data-relocatable code: PSEG, CSEG, AORG, and END. The PSEG directive, like DEND, indicates that succeeding locations are program-relocatable. The CSEG and AORG directives effectively terminate the data segment by beginning a common segment or absolute segment, respectively. The END directive terminates the data segment as well as the program.

The following example illustrates the use of both the DSEG and the DEND directives.

```
RAM DSEG   START OF DATA AREA
      :
      :
   <Data-relocatable code>
      :
      :
ERAM DEND

LRAM EQU ERAM-RAM
```

The block of code between the DSEG and DEND directives is data-relocatable. RAM is the symbolic address of the first word of this block; ERAM is the data-relocatable byte address of the location following the code block. The value of the symbol LRAM is the length in bytes of the block.

4.2.8 DATA SEGMENT END DEND

Syntax definition:

[<label>] ƀ. . . .DEND ƀ. . .[<comment>

NOTE

This directive does not apply to the PX9ASM assembler.

DEND terminates the definition of a block of data-relocatable code by placing a value in the location counter and defining succeeding locations as program-relocatable. Use of the label field is optional. When a label is used, it is assigned the value of the location counter *prior* to modification. The operation field contains DEND. The operand field is not used, and the comment field is optional. Either of the following values is placed in the location counter as a result of this directive:

- The maximum value the location counter has ever attained as a result of the assembly of any preceding block of program-relocatable code.
- Zero, if no program-relocatable code has been previously assembled.

If encountered in common-relocatable or program-relocatable code, this directive functions as a CEND or PEND (and a warning message is issued); like CEND and PEND, it is invalid when used in absolute code. The following example illustrates the use of both DSEG and DEND directive.

```
RAM DSEG   START OF DATA AREA
        :
        :
      <Data-relocatable code>
        :
        :
ERAM DEND

LRAM EQU ERAM-RAM
```

4.2.9 COMMON SEGMENT CSEG

Syntax description:

[<label>] ƀ. . .CSEGƀ. . .['<string>'] ƀ. . .[<comment>]

NOTE

This directive does not apply to the PX9ASM assembler.

CSEG places a value in the location counter and defines succeeding locations as common-relocatable (i.e., relocatable with respect to a common segment). Use of the label field is optional. When a label is used, it is assigned the value that the directive places in the location counter. The operation field contains CSEG, and the operand field is optional. The comment field may be used only when the operand field is used.

If the operand field is not used, the CSEG directive defines the beginning of (or continuation of) the "blank common" segment of the program. When the operand field is used, it must contain a character string of up to six characters, enclosed in quotes. (If the string is longer than six characters, the assembler prints a truncation error message and retains the first six characters of the string.) If this string has not previously appeared as the operand of a CSEG directive, the assembler associates a new relocation type with the operand, sets the location counter to zero, and defines succeeding locations as relocatable with respect to the new relocatable type. When the operand string has been previously used in a CSEG, the succeeding code represents a continuation of that particular common segment associated with the operand. The location counter is restored to the maximum value it previously attained during the assembly of any portion of the particular common segment.

The following directives will properly terminate the definition of a block of common-relocatable code: CEND, PSEG, DSEG, AORG, and END. The block is normally terminated with a CEND directive (see paragraph 4.2.10). The PSEG directive, like CEND, indicates that succeeding locations are program-relocatable. The DSEG and AORG directives effectively terminate the common segment by beginning a data segment or absolute segment. The END directive terminates the common segment as well as the program.

The CSEG directive permits the construction and definition of independently relocatable segments of data which several programs may access or reference at execution time. The segments are the assembly language counterparts of FORTRAN blank COMMON and labeled COMMON, and in fact permit assembly language programs to communicate with FORTRAN programs which use COMMON. Information placed in the object code by the assembler permits the linkage editor to relocate all common segments independently and to make appropriate adjustments to all addresses which reference locations within common segments. Locations within a particular common segment may be referenced by several different programs if each contains a CSEG directive with the same operand or no operand.

The following example illustrates the use of both the CSEG and the CEND directives:

```
COM1A CSEG 'ONE'

          <Common-relocatable code, type 'ONE' >
               .
               .
               .
          CEND
COM2A CSEG 'TWO'
               .
               .
               .
          <Common-relocatable code, type 'TWO'>
               .
               .
               .
COM2B CEND
COM1C CSEG 'ONE'
               .
               .
               .
          <Common-relocatable code, type 'ONE'>
               .
               .
               .
COM1B  CEND
```

```
COM1L   DATA  COM1B - COM1A   LENGTH OF SEGMENT 'ONE'
COM2L   DATA  COM2B - COM2A   LENGTH OF SEGMENT 'TWO'
```

The three blocks of code between the CSEG and CEND directives are common-relocatable. The first and third blocks are relocatable with respect to one common relocation type; the second is relocatable with respect to another. The first and third blocks comprise the common segment 'ONE', and the value of the symbol COM1L is the length in bytes of this segment. The symbol COM2A is the symbolic address of the first word of common segment 'TWO'; COM2B is the common-relocatable (type 'TWO') byte address of the location following segment. (Note that the symbols COM2B and COM1C are of different relocation types and possibly different values.) The value of the symbol COM2L is the length in bytes of common segment 'TWO'.

4.2.10 COMMON SEGMENT END CEND

Syntax definition:

[<label>] b. . .CEND b. . .[<comment>]

NOTE

This directive does not apply to the PX9ASM assembler.

CEND terminates the definition of a block of common-relocatable code by placing a value in the location counter and defining succeeding locations as program-relocatable. Use of the label field is optional. When a label is used, it is assigned the value of the location counter prior to modification. The operation field contains CEND. The operand field is not used, and the comment field is optional. Either of the following values is placed in the location counter as a result of this directive:

- The maximum value the location counter has ever attained as a result of the assembly of any preceding block of program-relocatable code.
- Zero, if no program-relocatable code has been previously assembled.

If encountered in common- or program-relocatable code, this directive functions as a DEND or PEND (and a warning message is issued); like DEND and PEND, it is invalid when used in absolute code. See paragraph 4.2.9 for an example of the use of the CEND directive.

4.2.11 PROGRAM SEGMENT PSEG

Syntax definition:

[<label>] b. . .PSEG b. . .[<comment>]

NOTE

This directive does not apply to the PX9ASM assembler.

PSEG places a value in the location counter and defines succeeding locations as program-relocatable. When a label is used, it is assigned the value that the directive places in the location counter. The operation field contains PSEG. The operand field is not used and the comment field is optional. Either of the following values is placed in the location counter:

- The maximum value the location counter has ever attained as a result of the assembly of any preceding block of program-relocatable code.
- Zero, if no program-relocatable code has been previously assembled.

The PSEG directive is provided as the program-segment counterpart to the DSEG and CSEG directives. Together, the three directives provide a consistent method of defining the various types of relocatable segments. The following sequences of directives are functionally identical:

```
DSEG                              DSEG
  .                                 .
  .                                 .
  .                                 .
<Data-relocatable code>           <Data-relocatable code>
  .                                 .
  .                                 .
  .                                 .
DEND                                .
CSEG                              CSEG
  .                                 .
  .                                 .
  .                                 .
<Common-relocatable code>         <Common-relocatable code>
  .                                 .
  .                                 .
  .                                 .
CEND                                .
PSEG                              PSEG
  .                                 .
  .                                 .
  .                                 .
<Program-relocatable code>        <Program-relocatable code>
  .                                 .
  .                                 .
PEND                                .
  .                                 .
END                               END
```

4.2.12 PROGRAM SEGMENT END PEND

Syntax definition:

[<label>] ƀ. . .PEND ƀ. . .[<comment>]

NOTE

This directive does not apply to the PX9ASM assembler.

The PEND directive is provided as the program-segment counterpart to the DEND and CEND directives. Like those directives, it places a value in the location counter and defines succeeding locations as program-relocatable (however, since PEND properly appears only in program-relocatable code, the relocation type of succeeding locations remains unchanged.) Use of the label field is optional. When a label is used, it is assigned the value of the location counter *prior* to modification. The operation field contains PEND. The operand field is not used, and the comment field is optional. The value placed in the location counter by this directive is simply the maximum value ever attained by the location counter as a result of the assembly of all preceding program-relocatable code. If encountered in data- or common relocatable code, this directive functions as a DEND or CEND (and a warning message is issued), like DEND and CEND, it is invalid when used in absolute code. See paragraph 4.2.11 for an example of the use of the PEND directive.

4.3 DIRECTIVES THAT AFFECT THE ASSEMBLER OUTPUT

This category includes the directive that specifies optional output for the Cross Assembler and the directive that supplies a program identifier in the object code. In addition four directives affect the source listing. The directives in this category are:

- Output Options
- Program Identifier
- Page Title
- List Source
- No Source List
- Page Eject

4.3.1 OUTPUT OPTIONS. This directive does not apply to the PX9ASM or TXMIRA assembler.

Syntax definition:

b . . . OPTIONb . . . <keyword>[,<keyword>] . . . b . . . [<comment>]

OPTION specifies output and list options to the assembler. No label is entered with the OPTION directive. The operation field contains OPTION. The operand field contains one or more keywords to specify the desired options. The comment field is optional.

The keywords supported by the Cross Assembler and SDSMAC, and their meanings are as follows:

- XREF - Print a cross reference listing at the end of the source and object listing.
- OBJ - Print a hexadecimal listing of the object code at the end of the source and object listing or the cross reference listing (not supported by SDSMAC).
- SYMT - Output a symbol table in the object code that contains all symbols in the program.

Additional keywords are supported by SDSMAC, as described in Section VI.

The following example shows an OPTION directive:

```
OPTION    XREF,SYMT
```

The directive in the example specifies the printing of a cross reference listing and the output of a symbol table with the object code.

4.3.2 PROGRAM IDENTIFIER IDT

Syntax definition:

[<label>]b . . . IDTb . . . '<string>'b . . . [<comment>]

IDT assigns a name to the program. An IDT directive must precede any machine instruction or assembler directive that results in object code. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains IDT. The operand field contains the program name (string), a character string of up to eight characters. When a character string of more than eight characters is entered, the assembler prints a truncation error message, and retains the first eight characters as the program name. The comment field is optional.

The following example shows an IDT directive:

```
IDT   'CONVERT'
```

The directive assigns the name CONVERT to the program to be assembled. The program name is printed in the source listing as the operand of the IDT directive, but does not appear in the page heading of the source listing. The program name is placed in the object code, but serves no purpose during the assembly.

NOTE

Although SDSMAC will accept lower case letters and special characters within the quotes, ROM loaders, etc., will not. Therefore only upper case letters and numerals are recommended.

4.3.3 PAGE TITLE TITL

Syntax definition:

[<label>]b . . . TITLb . . . '<string>'b [<comment>]

TITL supplies a title to be printed in the heading of each page of the source listing. When a title is desired in the heading of the first page of the source listing, a TITL directive must be the first source statement submitted to the assembler. This directive is not printed in the source listing. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains TITL. The operand field contains the title (string), a character string of up to 50 characters. When more than 50 characters are entered, the assembler retains the first 50 characters as the title, and prints a truncation error message. The comment field is optional; the assembler does not print the comment, but does increment the line counter.

The following example shows a TITL directive:

```
TITL   '**REPORT GENERATOR**'
```

This directive causes the title **REPORT GENERATOR** to be printed in the page headings of the source listing. When a TITL directive is the first source statement in a program, the title is printed on all pages until another TITL directive is processed. Otherwise, the title is printed on the next page after the directive is processed, and on subsequent pages until another TITL directive is processed.

NOTE

The maximum source record length is 60 characters. If a full 50-character title is desired, the operand field must be started at or before column 11 of the source record.

4.3.4 LIST SOURCE LIST

Syntax definition:

[<label>]b . . . LISTb . . . [<comment>]

LIST restores printing of the source listing. This directive is required only when a No Source List directive is in effect, to cause the assembler to resume listing. This directive is not printed in the source listing, but the line counter increments. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains LIST. The operand field is not used. Use of the comment field is optional, but the assembler does not print the comment.

The following example shows a LIST directive:

```
LIST
```

The directive causes the source listing to be resumed with the next source statement.

4.3.5 NO SOURCE LIST UNL

Syntax definition:

[<label>]b . . . UNLb . . . [<comment>]

UNL inhibits printing of the source listing. The UNL directive is not printed in the source listing, but the line counter increments. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains UNL. The operand field is not used. Use of the comment field is optional, but the assembler does not print the comment.

The following example shows a UNL directive:

```
UNL
```

The directive inhibits printing of the source listing. Use of the UNL directive to inhibit printing reduces assembly time and the size of the source listing.

4.3.6 PAGE EJECT PAGE

Syntax definition:

[<label>]b . . . PAGEb . . . [<comment>]

PAGE causes the assembler to continue the source program listing on a new page. The PAGE directive is not printed in the source listing, but the line counter increments. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains PAGE. The operand field is not used. Use of the comment field is optional, but the assembler does not print the comment.

The following example shows a PAGE directive:

```
PAGE
```

The directive causes the assembler to begin a new page of the source listing. The next source statement is the first statement listed on the new page. Use of the page directive to begin new pages of the source listing at the logical divisions of the program improves documentation of the program.

4.4 DIRECTIVES THAT INITIALIZE CONSTANTS

This category includes directives that place values in successive bytes or words of the object code, and a directive that places characters of text in the object code to be displayed or printed. It also includes a directive that initializes a constant for use during the assembly process. The directives are:

- Initialize Byte
- Initialize Word
- Initialize Text
- Define Assembly-Time Constant

4.4.1 INITIALIZE BYTE BYTE

Syntax definition:

[<label>]b . . . BYTEb . . . <exp>[,<exp>] . . . b . . . [<comment>]

BYTE places one or more values in one or more successive bytes of memory. Use of the label field is optional. When a label is used, the location at which the assembler places the first byte is assigned to the label. The operation field contains BYTE. The operand field contains one or more expressions separated by commas. The expressions must contain no external references. The assembler evaluates each expression and places the value in a byte as an eight-bit two's complement number. When truncation is required, the assembler prints a truncation error message and places the rightmost portion of the value in the byte. The comment field is optional.

The following example shows a BYTE directive:

```
KONS    BYTE    >F+1,-1,'D'-'=',0,'AB'-'AA'
```

The directive initializes five bytes, starting with a byte at location KONS. The contents of the resulting bytes is 00010000, 11111111, 00000111, 00000000, and 00000001.

4.4.2 INITIALIZE WORD DATA

Syntax definition:

[<label>]b . . . DATAb . . . <exp>[,<exp>] . . . b . . . [<comment>]

DATA places one or more values in one or more successive words of memory. The assembler advances the location counter to a word boundary (even) address. Use of the label field is optional. When a label is used, the location at which the assembler places the first word is assigned to the label. The operation field contains DATA. The operand field contains one or more expressions separated by commas. The assembler evaluates each expression and places the value in a word as a sixteen-bit two's complement number. The comment field is optional.

The following example shows a DATA directive:

```
KONS1   DATA    3200,1+'AB',-'AF',>F4A0,'A'
```

The directive initializes five words, starting with a word at location KONS1. The contents of the resulting words are $0C80_{16}$, 4143_{16}, $BEBA_{16}$, $F4A0_{16}$, and 0041_{16}. Had the location counter contents been $010F_{16}$ prior to processing this directive, the value assigned to KONS1 would be 0110_{16}.

4.4.3 INITIALIZE TEXT TEXT

Syntax definition:

[<label>]b . . . TEXTb . . [-]'<string>'b . . . [<comment>]

TEXT places one or more characters in successive bytes of memory. The assembler negates the last character of the string when the string is preceded by a minus (-) sign (unary minus). Use of the label field is optional. When a label is used, the location at which the assembler places the first character is assigned to the label. The operation field contains TEXT. The operand field contains a character string of up to 52 characters, which may be preceded by a unary minus sign. The comment field is optional.

The following example shows a TEXT directive:

```
MSG1    TEXT  'EXAMPLE'    MESSAGE HEADING
```

The directive places the eight-bit ASCII representations of the characters in successive bytes. When the location counter is on an even address, the result, in hexadecimal representation, is 4558, 414D, 504C, and 45XX. XX represents the contents of the rightmost byte of the fourth word, which are determined by the next source statement. The label MSG1 is assigned the value of the first byte address in which 45 is placed. Another example, showing the use of a unary minus, is as follows:

```
MSG2    TEXT  -'NUMBER'
```

When the location counter is on an even address, the result, in hexadecimal representation, is 4E55, 4D42, and 45AE. The label MSG2 is assigned the value of the byte address in which 4E is placed.

NOTE

PX9ASM prints only the first character in a text string.

4.4.4 DEFINE ASSEMBLY-TIME CONSTANT EQU

Syntax definition:

<label>b . . . EQUb . . . <exp>b . . . [<comment>]

NOTE

<exp> may not be a REF'd symbol. TXMIRA does not allow forward references in the <exp>.

EQU assigns a value to a symbol. The label field contains the symbol. The operation field contains EQU. The operand field contains an expression in which all symbols have been previously defined. Use of the comment field is optional.

The following example shows an EQU directive:

```
SUM    EQU   5    WORKSPACE REGISTER 5
```

The directive assigns an absolute value to the symbol SUM, making SUM available to use as a workspace register address. Another example of an EQU directive is:

```
TIME   EQU   HOURS
```

The directive assigns the value of previously defined symbol HOURS to symbol TIME. When HOURS appears in the label field of a machine instruction in a relocatable block of the program, the value is a relocatable value. The two symbols may be used interchangeably. SYMBOLS in the operand field need not have been previously defined when using SDSMAC.

4.5 DIRECTIVES THAT PROVIDE LINKAGE BETWEEN PROGRAMS

This category consists of two directives that enable program modules to be assembled separately and integrated into an executable program. One directive places one or more symbols defined in the module into the object code, making them available for linking. The other directive places symbols used in the module but defined in another module into the object code, allowing them to be linked. The directives are:

- External Definition
- External Reference
- Secondary Reference
- Force Load

4.5.1 EXTERNAL DEFINITION DEF

Syntax definition:

[<label>]b . . . DEFb . . . <symbol>[,<symbol>] . . . b . . . [<comment>]

DEF makes one or more symbols available to other programs for reference. The use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains DEF. The operand field contains one or more symbols, separated by commas, to be defined in the program being assembled. The comment field is optional.

The following example shows a DEF directive:

```
DEF    ENTER,ANS
```

The directive causes the assembler to include symbols ENTER and ANS in the object code so that these symbols are available to other programs. When the DEF directive does not precede the

source statements that contain the symbols, the assembler identifies the symbols as multiply defined symbols.

4.5.2 EXTERNAL REFERENCE REF

Syntax definition:

[<label>]ƀ . . . REFƀ . . . <symbol>[,<symbol>] . . . ƀ . . . [<comment>]

REF provides access to one or more symbols defined in other programs. The use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains REF. The operand field contains one or more symbols, separated by commas, to be used in the operand field of a subsequent source statement. The comment field is optional.

The following example shows a REF directive:

```
REF     ARG1,ARG2
```

The directive causes the assembler to include symbols ARG1 and ARG2 in the object code so that the corresponding addresses may be obtained from other programs. The Prototyping System Assembler, PX9ASM, requires that a REF directive precede the first use of a REF'd symbol.

If a sumbol is listed in the REF statement then a corresponding symbol must also be present in a DEF statement in another source module. If a one-to-one matching of symbols does not occur then an error occurs at Link Edit time. The system will generate a summary list of all "unresolved references".

NOTE

If a symbol in the operand field of an REF directive is the first operand of a DATA directive, the assembler places the value of the symbol at location 0 of the routine. If that routine is loaded at absolute location 0, the symbol will not be linked correctly. Use of the symbol at other locations will be correctly linked.

4.5.3 SECONDARY EXTERNAL REFERENCE SREF

Syntax definition:

[<label>]ƀ . . . SREF . . . <symbol>,[<symbol>] . . . ƀ . . . [<comment>]

SREF provides access to one or more symbols defined in other programs. The use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains SREF. The operand field contains one or more symbols, separated by commas, to be used in the operand field of a subsequent source statement. The comment field is optional.

The following example shows a REF directive:

```
SREF    ARG1,ARG2
```

The directive causes the Link Editor to include symbols ARG1 and ARG2 in the object code so that the corresponding addresses may be obtained from other programs.

SREF unlike REF does not require a symbol to have a corresponding symbol listed in a DEF statement of another source module. But the symbol will be an unresolved reference.

NOTE

SREF is supported by SDSMAC and TXMIRA only.

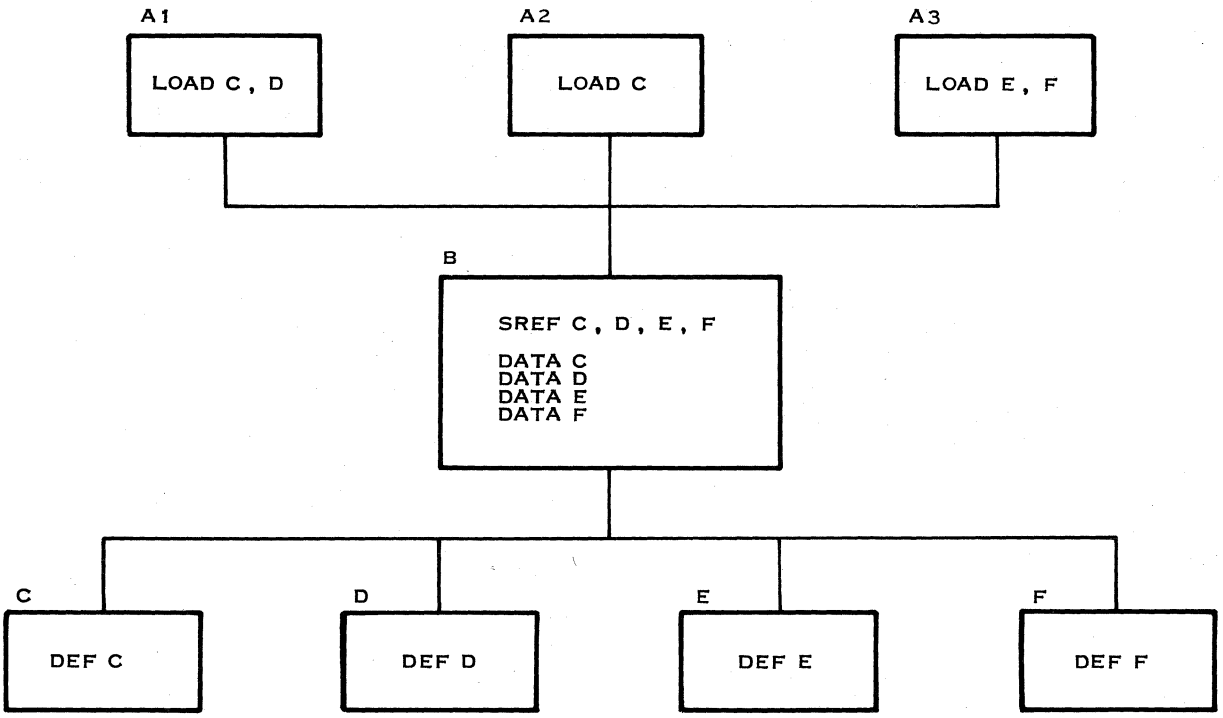
4.5.4 FORCE LOAD LOAD

Syntax definition:

[<label>]b . . . LOADb . . . <symbol> [,<symbol>] . . . b . . . [<comment>]

The LOAD directive is like a REF, but the symbol does not need to be used in the module containing the LOAD. The symbol used in the LOAD must be DEFed in some other module. LOADs are used with SREFs. If a one-to-one matching of LOAD-SREF pairs and DEF symbols does not occur, then unresolved references will occur during link editing.

The following example shows the use of the SREF and the LOAD directives:



Modules A1 uses a branch table in module B to get one of the modules C, D, E, or F. Module A1 knows which of the modules C, D, E, and F it will need. Module B has SREF for C, D, E, and F. Module C has a DEF for C. Module D has a DEF for D. Module E has a DEF for E. Module F has a DEF for F. Module A1 has a LOAD for one or more of modules C, D, E, and F as needed.

The LOAD and SREF directives permit module B to be written to handle the most involved case and still be linked together without unneeded modules, since A1 only has LOAD directives for the modules it needs.

When a link edit is performed, automatic symbol resolution will pull in the modules appearing in a LOAD directive. (See Section 2.5.1 of the *Model 990 Computer Link Editor Reference Manual,* part number 949617-9701 for more details on automatic symbol resolution.)

If the link control file included A1 and A2, modules C and D would be pulled in, while modules E and F would not be pulled in. If the link control file included A3, modules E and F would be pulled in, while modules C and D would not be pulled in. If the link control file included A2, module C would be pulled in, while modules D, E, and F would not be pulled in.

4.6 MISCELLANEOUS DIRECTIVES

This category includes a directive that defines a symbol for an extended operation, and a directive that terminates a source program. The directives are:

- Define Extended Operation
- Program End

4.6.1 DEFINE EXTENDED OPERATION DXOP

Syntax definition:

[<label>]b . . . DXOPb . . . <symbol>,<term>b . . . [<comment>]

DXOP assigns a symbol to be used in the operator field to specify an extended operation. The use of the label field is optional. When a label is used, the current value in the location counter is assigned to the label. The operation field contains DXOP. The operand field contains a symbol followed by a comma and a term. The symbol assigned to an extended operation must not be used in the label or operand field of any other statement. The assembler assigns the symbol to an extended operation specified by the term, which must have a value in the range of 0 to 15. The comment field is optional.

The following example shows a DXOP directive:

```
DXOP   DADD,13
```

The directive defines DADD as extended operation 13. When the assembler recognizes the symbol DADD in the operator field, it assembles an XOP instruction that specifies extended operation 13. The following example shows the use of the symbol DADD in a source statement:

```
DADD   @LABEL1(4)
```

The assembler places the operand field contents in the T_s and S fields of an XOP instruction, and places 13 in the D field.

4.6.2 PROGRAM END END

Syntax definition:

[<label>]b . . . ENDb . . . [<symbol>]b . . . [<comment>]

END terminates the assembly. The last source statement of a program is the END directive. When any source statements follow the END directive, they are ignored. Use of the label field is optional. When a label is used, the current value in the location counter is assigned to the symbol. The operation field contains END. Use of the operand field is optional. When the operand field is used, it contains a program relocatable or absolute symbol that specifies the entry point of the program. When the operand field is not used, no entry point is placed in the object code. The comment field may be used only when the operand field is used.

The following example shows an END directive:

```
END    START
```

The directive causes the assembler to terminate the assembly of this program. The assembler also places the value of START in the object code as an entry point.

When a program executes in a stand-alone mode, and is loaded by the ROM loader, it must supply an entry point to the loader. When no operand is included in the END directive, and that program is loaded by the ROM loader, the loader transfers control to the entry point of the loader, and attempts to load another object program.

START darf nicht Extern definiert sein!!

SECTION V

PSEUDO-INSTRUCTIONS

5.1 GENERAL

A pseudo-instruction is a convenient way to code an operation that is actually performed by a machine instruction with a specific operand. The Model 990 Computer Assembly Language includes two pseudo-instructions. The pseudo-instructions are:

- No Operation
- Return

5.2 NO OPERATION NOP

Syntax definition:

[<label>]b . . . NOPb . . . [<comment>]

NOP places a machine instruction in the object code which has no effect on execution of the program other than execution time. Use of the label field is optional. When the label field is used, the label is assigned the location of the instruction. The operation field contains NOP. The operand field is not used. Use of the comment field is optional.

Enter the NOP pseudo-instruction as shown in the following example:

```
MOD    NOP
```

Location MOD contains a NOP pseudo-instruction when the program is loaded. Another instruction may be placed in location MOD during execution to implement a program option. The assembler supplies the same object code as if the source statement had contained the following:

```
MOD    JMP    $+2
```

5.3 RETURN RT

Syntax definition:

[<label>]b . . . RTb . . . [<comment>]

RT places a machine instruction in the object code to return control to a calling routine from a subroutine. Use of the label field is optional. When the label field is used, the label is assigned the location of the instruction. The operation field contains RT. The operand field is not used. Use of the comment field is optional.

Enter the RT pseudo-instruction as shown in the following example:

```
RT
```

The assembler supplies the same object code as if the source statement had contained the following:

```
B    *11
```

When control is transferred to a subroutine by execution of a BL instruction, the link to the calling routine is stored in workspace register 11. An RT pseudo-instruction returns control to the instruction following the BL instruction in the calling routine.

SECTION VI

ASSEMBLERS

6.1 GENERAL

Four assemblers process the Model 990 Computer Assembly Language. These four assemblers are described in this section. In addition, this section describes the extended capabilities of the Program Development System Assembler SDSMAC.

6.2 PROTOTYPING SYSTEM ASSEMBLER

The Prototyping System Assembler PX9ASM is a one-pass assembler that executes in a Model 990 Computer under the PX990 Executive, and is a part of the Prototyping System and the 733 ASR Program Development System. PX9ASM assembles object code for the TMS 9900 Microprocessor and the Model 990 Computer.

A one-pass assembler reads the source statements of a program one time only. The assembler maintains a location counter as it reads the statements, and assigns a location counter value to a label (symbol in the label field). The assembler builds a symbol table using these symbols and the assigned values. The assembler also evaluates the expression in the operand field using the values in the symbol table for any symbols in the expression. Then the assembler assembles the appropriate object code according to the operation codes and the values of the operands. Because the source statements are read only once, there must be limitations on the use of a symbol in operand fields prior to the statement that has the symbol in the label field (forward reference).

PX9ASM supports the assembly language as previously described. The limitations on the use of forward references in PX9ASM are included in the description of expressions in a preceding paragraph. PX9ASM provides a listing of the source and object code, and the machine language object code.

6.2.1 TERMINAL EXECUTIVE DEVELOPMENT SYSTEM ASSEMBLER. The terminal executive development system assembler TXMIRA is a two-pass assembler that executes in a Model 990 computer as part of the terminal executive development system, running under the TX990 executive.

TXMIRA reads the source statements of a program twice. On the first pass, the assembler maintains a location counter and builds a symbol table. For the second pass, the source statements are read in again after rewinding the input file. During the second pass, the assembler generates the object code using the source statements and the symbol table data developed during the first pass.

The TXMIRA assembler supports the assembly language as previously described. Because it is a two-pass assembler, the restrictions on forward references are relaxed. The TXMIRA assembler optionally produces a list of the source and object code, and the symbol table, and predefines registers.

6.3 CROSS ASSEMBLER

The Cross Assembler is a two-pass cross assembler that assembles object code for the TMS 9900 Microprocessor and Model 990 Computers. The Cross Assembler executes on an IBM System 3X0, and is available on several nationwide timesharing services.

A two-pass assembler reads the source statements of a program two times. The first time (first pass), the assembler maintains the location counter and builds a symbol table similar to those in a one-pass assembler. The two-pass assembler also copies the source statements for reading during the second pass, but does not assemble any object code. During the second pass, the assembler reads the copy of the source statements, and assembles the object code using the operation codes and the symbol table completed during the first pass.

The Cross Assembler also supports the assembly language as previously described. The restrictions on forward references are not as great for PX9ASM, because it is a two-pass assembler. These restrictions are included in the descriptions of expressions in a preceding paragraph. The Cross Assembler produces a listing of the source and object code, and the machine language object code. Optionally, the Cross Assembler prints a Cross Reference listing of the symbols in the program, and a listing of the object code. Also, optionally, the Cross Assembler includes the symbols used in the program and their values with the object code.

Finally, the Cross Assembler supports additional directives which permit the programmer to distinguish between program segment, data segment, and common segments. Program segment is the relocatable code normally generated by all three assemblers. Data segment is relocatable code which normally includes only modifiable storage. Common segments correspond to the blank common and labeled common blocks of a FORTRAN program.

6.4 PROGRAM DEVELOPMENT SYSTEM ASSEMBLER

The Program Development System Assembler SDSMAC is a two-pass assembler that assembles object code for the Model 990 Computer and the TMS 9900 Microprocessor. SDSMAC executes on a Model 990 Computer under the DX10 Disc Executive and is a part of the Program Development System.

The only restrictions on forward references are instances in which the value of the symbol affects the location counter.

SDSMAC supplies the additional capability of Macro-instructions or MACROs. A macro is a user-defined set of assembly language source statements. Macro definitions assign a name to the macro and define the source statements of the macro. The macro name may then be used in the operation field of a source statement of the program to cause the assembler to insert the pre-defined source statements and assemble them along with the other source statements of the program. The macro capability of SDSMAC allows the user to:

- Define macros to specify frequently used sequences of source code.
- Define macros for problem-oriented sequences of instructions to provide a means of programming that may be more meaningful to users who are not computer-oriented.

Macros are defined in a macro language consisting of eleven verbs described in Section VII. In addition to the macro language SDSMAC supports a number of extended capabilities described in subsequent paragraphs of this section.

SDSMAC supports the assembly language as described previously, and produces the source and object code listing and machine language object code that the other assemblers produce. The output options (cross reference listing, object code listing, and symbol table output) of the Cross Assembler are also available with SDSMAC. In addition, the user may suppress all printed output of SDSMAC or request SDSMAC to only produce a copy of the expanded source program.

In addition to the macro capability, SDSMAC supports the following capabilities beyond those of PX9ASM, TXMIRA and the Cross Assembler:

- Use of parentheses in expressions
- An additional arithmetic operator ('//') to perform right shifts
- Logical operators in expressions
- Relational operators in expressions
- Six additional output options
- Workspace pointer directive
- Copy source file directive
- Conditional Assembly Directives
- Define operation directive
- Transfer vector pseudo-instruction
- Define maximum macro level

The capabilities, and the use of symbolic addresses with SDSMAC are described in the following paragraphs.

6.4.1 USE OF PARENTHESES IN EXPRESSIONS. SDSMAC supports the use of parentheses in expressions to alter the order of evaluation of the expression. Nesting of pairs of parentheses within expressions is also supported. When parentheses are used, the portion of the expression within the innermost parentheses is evaluated first. Then the portion of the expression within the next-innermost pair is evaluated. When evaluation of the portions of the expression within all parentheses has been completed, the evaluation is completed from left to right. Evaluation of portions of an expression within parentheses at the same nesting level may be considered to be simultaneous.

For example, the use of parentheses in the expression LAB1 + ((4+3)*7) would result in the addition of 4 and 3. The result, 7, would be multiplied by 7, giving 49. The complete evaluation would be the value of LAB1 plus 49. Without parentheses, 4 would have been added to the value of LAB1, and 3 would have been added to the sum. The sum of the second addition would have been multiplied by 7 if LAB1 had an absolute value. If LAB1 had a relocatable value, the expression would have been illegal without the parentheses.

6.4.2 RIGHT SHIFT OPERATOR. In addition to the standard arithmetic operators used in expressions (add, subtract, multiply, divide), SDSMAC supports the operator // (double slash) to perform right shifts. The operator is used in the expression as follows:

<operand> // <shift count>

The operand may be an immediate value, a previously defined symbol, or a forward referenced symbol. The expression can not be relocatable. The precedence of this operator follows the normal left-to-right precedence unless the expression is modified by parentheses.

6.4.3 LOGICAL OPERATORS IN EXPRESSIONS. SDSMAC supports logical operations in expressions, which are the bit-by bit logical operations between the values of the symbols and/or constants. The logical operators are as follows:

- & for AND
- && for exclusive OR
- ++ for OR
- # for NOT (logical complement)

The order of evaluation of expressions that contain logical operators is similar to that of expressions that contain only arithmetic operators. Like the unary minus, the logical complement takes precedence over other operations regardless of position, except as altered by parentheses.

The following are examples of expressions that contain logical operators:

BLUE&&255	Specifies the result of an exclusive OR operation between the bits of the value of symbol BLUE and the bits of constant value 255.
GREEN++15	Specifies the result of an OR operation between the bits of the value of symbol GREEN and the bits of constant value 15.
RED&#255	Specifies the result of an AND operation between the bits of the value of symbol RED and the inversion of the bits of constant value 255.
RED&#255++(BLUE&255)	AND the value of BLUE with the constant 255. AND the value of RED with the 1's complement of 255. OR the two AND results to get the value of the expression.

6.4.4 RELATIONAL OPERATORS IN EXPRESSIONS. SDSMAC supports six relational operators that represent the relationship between the two constants and/or symbols, the result of comparing the constants and/or symbols. When the relationship corresponding to the operator exists (is true), the value of the combination is 1. When the relationship corresponding to the operator does not exist (is not true), the value of the combination is 0. The result may be used as an arithmetic value or as a logical value. The relational operators are as follows:

- = for equal
- < for less than
- > for greater than

- <= for less than or equal
- >= for greater than or equal
- #= for not equal.

NOTE

The greater than character (>) is also used to identify hexadecimal constants. The context determines the meaning of the greater than character in each statement.

The following are examples of expressions that contain relational operators:

BLUE#=GREEN	Compares the value of symbol BLUE to the value of symbol GREEN. When the values are not equal, the combination has a value of one. When the values are equal, the combination has a value of zero.
WHITE<BLACK	Compares the value of symbol WHITE to the value of symbol BLACK. When the value of WHITE is less than the value of BLACK, the combination has a value of one. Otherwise, the value of the combination is zero.
RED*(GREEN=0)	Compares the value of symbol GREEN to zero. When GREEN equals zero, the value of symbol RED is multiplied by 1, and the value of the expression is that of symbol RED. When GREEN is not equal to zero, the multiplier is zero, and the value of the expression is zero.
BLUE>=RED	Compares the value of symbol BLUE to the value of symbol RED. When BLUE is greater than or equal to RED, the combination is equal to one. When BLUE is less than RED, the combination is equal to zero.

6.4.5 OUTPUT OPTIONS. SDSMAC supports six options in addition to those listed in the description of the OPTION directive. The additional options are specified by entering keywords in an OPTION directive. The additional keywords and their meanings are as follows:

- NOLIST - Suppress printing of any listing. Overrides other directives and keywords.
- TUNLST - Limit the listing for text directives to a single line.

- DUNLST - Limit the listing for data directives to a single line.
- BUNLST - Limit the listing for byte directives to a single line.
- MUNLST - Limit the listing for a macro expansion to a single line.
- FUNL - Overrides unlist directives.

6.4.6 WORKSPACE POINTER. Only SDSMAC supports this directive. WPNT

Syntax definition:

[<label>]b . . . WPNTb . . . <label>b . . . [<comment>]

WPNT defines the current workspace to the assembler. WPNT generates no object code. The user must provide a machine instruction to actually place the value in the workspace register. The symbol in the label field, when used, must represent a word (even) address and must have been previously defined. The operation field contains WPNT. The operand field contains the label assigned to the workspace. The comment field is optional.

The following example shows a WPNT directive:

```
WPNT    WORK
```

The directive in the example is appropriate when the workspace at location WORK is the active workspace. The assembler stores the value of label WORK as the current workspace address, and from this information identifies symbolic addresses as workspace registers when the symbolic addresses have values greater than WORK by 15 or less. The assembler also recognizes WORK or a label equal to WORK as workspace register 0. Symbolic addresses having values outside this range are considered to be symbolic memory addresses.

6.4.7 COPY SOURCE FILE. Only SDSMAC supports this directive. COPY

Syntax definition:

[<label>]b . . . COPYb . . . <file name>b . . . [<comment>]

COPY changes the source input for the assembler. Use of the label field is optional. The operation field contains COPY. The operand field contains a file name from which the source statements are to be read. The file name may be:

- An access name recognized by DX10 operating system.
- A synonym form of an access name.

The comment field is optional.

The following example shows a COPY directive:

```
COPY    .SFILE
```

The directive in the example causes the assembler to take its source statements from a file SFILE. At the end-of-file of SFILE, the assembler resumes taking source statements from the file or device from which it was taking source statements when the COPY directive was processed. A COPY directive may be placed in a file being copied, which results in nested copying of files.

6.4.8 CONDITIONAL ASSEMBLY DIRECTIVES. Only SDSMAC supports these directives.

Syntax definition:

[<label>] ƀ . . ASMIFƀ . . <wd-exp> ƀ . . . [<comment>]

Assembly language statements

ƀ . . . ASMELSƀ . . [<comment>]

Assembly language statements

ƀ . . . ASMENDƀ . . [<comment>]

Three directives, ASMIF, ASMELS and ASMEND, furnish conditional assembly capability in SDSMAC. The three function as IF-THEN-ELSE brackets for blocks of assembly language statements. When the expression in the operand field of an ASMIF evaluates to a non-zero (or true) value, the block of statements enclosed by either ASMIF-ASMEND or ASMIF - ASMELS is assembled. If the block is terminated by ASMELS, the block enclosed by ASMELS - ASMEND is not assembled. When the expression on an ASMIF evaluates to zero (or false), the block of statements immediately following ASMIF is not assembled. If an alternate ASMELS block occurs, it is assembled. Statements not assembled are treated as comments. The ASMIF expression must be well defined when it is encountered.

WARNING

ASMIF, ASMELS and ASMEND may not appear as macro model statements. ASMIF - ASMELS - ASMEND constructs may be nested.

The following example shows the use of conditional assembly.

```
             SDSMAC 947075 *E    00:01:36
                                                                    PAGE 0001

0001                 *
0002                 *       THIS IS AN EXAMPLE OF A USE OF CONDITIONAL ASSEMBLY
0003                 *       TO INCLUDE VARIOUS LEVELS OF DEBUG INFORMATION.
0004                 *
0005                 *       A SYMBOL IS DEFINED WHICH INDICATES THIS LEVEL
0006                 *            0 - NO DEBUG
0007                 *            5 - ENTRY /EXIT SHORT DUMPS
0008                 *            10- THE ABOVE, OUTER LOOP SHORT DUMPS, ENTRY/EXIT
0009                 *                LONG DUMPS
0010                 *            15- ALL THE ABOVE  & INNER LOOP SHORT DUMPS
0011                 *
0012       000C      DEBUG  EQU  12
0013                 *       A VALUE OF 12 FOR DEBUG WILL GIVE THE FIRST 3 LEVELS
0014                 *       OF DEBUG INFORMATION
0015                 *
0016                        REF  SRTDMP,LNGDMP
0017                 * PROGRAM ENTRY POINT
0018                        DEF  ENTRY
0019 0000           ENTRY
0020                        ASMIF DEBUG              IF DEBUG=0, SKIP THIS BLOCK*
0021                              ASMIF DEBUG>5                               *
0022 0000 06A0                        BL LNGDMP      ENTRY POINT SHORT DUMP  *
     0002 0000
0023                               ASMELS                                     *
0024                                  BL SRTDMP      ENTRY POINT LONG DUMP   *
0025                               ASMEND                                     *
0026                        ASMEND                                 ************
0027 0004 C18B              MOV  R11,R6        (SAVE RETURN ADDRESS)
0028                 *
0029                 * <CODE>
0030                 *
0031                 * OUTER LOOP
0032 0006           LABEL1
0033 0006 0205              LI   R5,>100
     0008 0100
0034                 * INNER LOOP
0035 000A           LABEL2
0036 000A 0204              LI   R4,6
     000C 0006
0037                 *
0038                 * <CODE>
0039                 *
0040                        ASMIF DEBUG>=15
0041                              BL SRTDMP         INNER LOOP SHORT DUMP
0042                        ASMEND
0043 000E 0604              DEC  R4
0044 0010 15FC              JGT  LABEL2
0045                 *
0046                 * <CODE>
0047                 *
0048                        ASMIF DEBUG>=10
0049 0012 06A0                    BL   SRTDMP       OUTER LOOP SHORT DUMP
     0014 0000
0050                        ASMEND
0051 0016 0605              DEC  R5
0052 0018 15F6              JGT  LABEL1
0053                 *
0054                 * <CODE>
0055                 *
0056                 * EXIT POINT
```

```
              SDSMAC 947075 *E   00:01:36
                                                                PAGE 0002

 0057                    ASMIF DEBUG             IF DEBUG=0, SKIP THIS BLOCK*
 0058                          ASMIF DEBUG>5                               *
 0059 001A 06A0                      BL LNGDMP       EXIT LONG DUMP        *
      001C 0002'
 0060                          ASMELS                                      *
 0061                                BL SRTDMP       EXIT SHORT DUMP       *
 0062                          ASMEND                                      *
 0063                    ASMEND                                  **********
 0064 001E 0456          B     *R6               (RETURN THROUGH SAVED REGISTER
 0065                    END
NO ERRORS
```

6.4.9 DEFINE OPERATION. Only SDSMAC supports this directive. DFOP

Syntax definition:

[<label>] b . . . DFOPb . . . <symbol>,< operation>b . . . [<comment>]

DFOP defines a synonym for an operation. Use of the label field is optional. The operation field contains DFOP. The operand field contains a symbol which is the synonym for an operation, and the operation, which may be the mnemonic operation code of a machine instruction, a macro name, or the symbol of a previous DFOP or DXOP directive. The comment field is optional.

The following example shows a DFOP directive:

```
DFOP      LD,MOV
```

The directive in the example defines LD for a synonym for the MOV machine instruction. The LD symbol might be more meaningful where the source is a symbolic memory location and the destination is a workspace register. The machine code for the MOV instruction is assembled whenever either symbol appears in the operation field of a source statement. A single symbol may appear in more than one DFOP directive in the same assembly, and an operation symbol may appear in the label field of a DFOP directive. When an operation symbol appears as the defined symbol of a DFOP directive, the corresponding operation is not available unless it had appeared in the operand field of a previous DFOP directive. The effect of a group of DFOP directives is shown in the following example:

```
DFOP      HOLD, LWPI        HOLD DEFINED TO BE LWPI
DFOP      LWPI,SOMMAC       LWPI REDEFINED AS MACRO SOMMAC
 .
 .
 .
DFOP      SAVE,HOLD         SAVE DEFINED AS HOLD (LWPI)
DFOP      HOLD,BLWP         HOLD REDEFINED AS BLWP
 .
 .
 .
DFOP      LWPI,SAVE         LWPI RESTORED
```

The first pair of DFOP directives substitutes macro SOMMAC for the LWPI machine instruction, which may be specified by the symbol HOLD. The second pair of DFOP directives changes the symbol by which the LWPI machine instruction is specified to SAVE, and the symbol by which the BLWP instruction is specified to HOLD. The last DFOP directive restores the symbol LWPI to the LWPI machine instruction.

6.4.10 TRANSFER VECTOR. Only SDSMAC supports this pseudo-instruction. XVEC

Syntax definition:

<label>b . . . XVECb . . . <wp address>[,<subr address>]b . . . [<comment>]

The XVEC pseudo-instruction is a means of coding the transfer vector for a subroutine. XVEC places a set of assembler directives in the source code to provide a transfer vector for a BLWP instruction. XVEC also provides a WPNT directive to define the newly active workspace to the assembler. The label field contains the label of the resulting transfer vector. The operation field contains XVEC. The operand field contains the label (wp address) of the workspace that becomes active when the BLWP instruction is executed. Optionally, the wp address may be followed by a comma and the label (subr address) of the first instruction to be executed in the subroutine. When the second operand is omitted, the assembler assumes that the first instruction to be executed follows the transfer vector. The use of the comment field is optional.

Enter the XVEC pseudo-instruction as shown in the following example:

```
SUBRA    XVEC    WKSPA,ENTRYA
```

Transfer of control to a subroutine at location ENTRYA with a workspace at location WKSPA becoming the active workspace is coded as follows:

```
         BLWP    SUBRA
```

The resulting object code and assembler processing is the same as would result from the following directives:

```
SUBRA    DATA    WKSPA
         DATA    ENTRYA
         WPNT    WKSPA
```

Alternatively, the XVEC pseudo-instruction may be entered as follows:

```
SUBRA    XVEC    WKSPA
```

In this case, the executable code of the subroutine must immediately follow the XVEC pseudo-instruction. The resulting object code and assembler processing is the same as would result from the following directives:

```
SUBRA    DATA    WKSPA
         DATA    $+2
         WPNT    WKSPA
```

NOTE

No executable code that requires a different active workspace than that of the subroutine may be entered between the XVEC pseudo-instruction and the subroutine entry address.

6.4.11 SET MAXIMUM MACRO NESTING LEVEL. Only SDSMAC supports this directive.

Syntax definition:

[<label>]b. . . SETMNLb. . . <exp>b. . . [<comment>]

The SETMNL directive allows the programmer to change the maximum macro nesting stack level as required. SDSMAC maintains a count of the number of levels of macro nesting and declares an

error if this count exceeds the maximum number allowed. The default maximum is sixteen. The SETMNL directive may be used to set the allowed maximum to greater or less than sixteen.

6.4.12 SYMBOLIC ADDRESSING TECHNIQUES. SDSMAC processes symbolic memory addresses differently than the other assemblers so that the user may:

- Use the symbolic memory address of a workspace register to address the workspace register.
- Omit the @ character to identify a symbolic memory address.

When SDSMAC processes a symbol as an operand of a machine instruction, it compares the value of the symbol to the address of the current workspace. When the value is equal to the workspace address, or is greater by 15 or less, the symbol represents a workspace and SDSMAC assembles a workspace register address. Otherwise SDSMAC assembles a symbolic memory address. A WPNT directive or an LWPI instruction supplies the address of the current workspace to the assembler.

Without this capability, two symbols are frequently assigned to the same address. The following example illustrates this type of coding:

```
SUM     EQU   0              ASSIGN SUM FOR WORKSPACE REGISTER 0
QUAN    EQU   1              ASSIGN QUAN FOR WORKSPACE REGISTER 1
          .
          .
          .
WS1     DATA  0              WORKSPACE REGISTER 0
QUANT   DATA  0              WORKSPACE REGISTER 1
FIVE    DATA  5              WORKSPACE REGISTER 2
          .
          .
          .
        MOV   @FIVE,@QUANT   INITIALIZE QUANTITY
        BLWP  @SUB1          BRANCH TO SUBROUTINE
          .
          .
          .
SUB1    DATA  WS1            TRANSFER VECTOR
        DATA  ENT1           FOR SUBROUTINE
ENT1    A     QUAN,SUM       ADD QUAN TO SUM
```

The two initial EQU directives assign meaningful labels to be used as workspace register addresses in the subroutine. The labels of the DATA statements are required to access the same memory locations in the main program, when another workspace is active. The following code would produce the same object code when assembled on SDSMAC:

```
SUM     DATA  0              WORKSPACE REGISTER 0
QUAN    DATA  0              WORKSPACE REGISTER 1
FIVE    DATA  5              WORKSPACE REGISTER 2
          .
          .
          .
        MOV   FIVE,QUAN      INITIALIZE QUANTITY
        BLWP  SUB1           BRANCH TO SUBROUTINE
          .
          .
          .
```

```
SUB1    XVEC    SUM             TRANSFER VECTOR FOR SUBROUTINE
ENT1    A       QUAN,SUM        ADD QUAN TO SUM
```

The MOV instruction in the main program results in symbolic memory addresses for both operands. The BLWP instruction uses transfer vector SUB1, provided by the XVEC directive labelled SUB1. The XVEC directive also provides a WPNT directive that identifies SUM as the address of the current workspace. The A instruction uses the symbol QUAN (as used in the MOV instruction) but results in a workspace register address, because QUAN is now workspace register 1.

SDSMAC is compatible with the other assemblers, however, because the code of the first example would be correctly assembled on SDSMAC.

In assemblers other than SDSMAC, a @ character is required, to denote the "indexed" mode of addressing where the instruction is defined as having a generalized address as an operand. When using SDSMAC, the @ character is considered redundant if

- all symbols in the expression have been previously defined and the resulting values of the expression is greater than 15, or

- another @ character prefaces the expression.

The following notations for the MOV instruction in the previous example would generate the same object and result in an error-free assembly:

```
MOV     @FIVE, @QUAN

MOV     FIVE, QUAN

MOV     @@FIVE, @@QUAN
```

NOTE

When the @ is omitted from a symbolic expression, the symbol must be defined before its use. If the symbol is not first defined, a register reference is assumed. If later the symbol is defined as a memory reference, an 'OPERAND CONFLICT PASS1/PASS2' error is generated.

SECTION VII

MACRO LANGUAGE

7.1 GENERAL

The SDSMAC assembler supports a macro defining language used in programs. A macro definition is a set of source statements (machine instructions and assembler directives) specified by a macro call in a source program. When the assembler processes a macro call it substitutes the predefined source statements of the macro definition for the macro call source statement, and assembles the substituted statements as if they had been included in the source program. MACRO definitions may be placed in a MACRO library for use in a subsequent assembly. This section describes the macro language, the verbs used to define macros, and the MACRO library directives.

7.2 PROCESSING OF MACROS

Figure 7-1 illustrates the data paths between the basic assembler, the macro translator (consisting of the Statement Classify, Macro Define, and Macro Expander modules) and the Macro library. The Statement Classify module processes all source statements to detect macro language statements and macro calls, and ignoring non-macro language statements. A special macro language statement, $MACRO, identifies the beginning of a macro definition, and $END identifies the end of a macro definition. Statements that occur between these two statements constitute a macro definition, and are passed to the Macro Define module. The module writes them in the Macro library in an encoded form. The Macro Define module also supplies to the Statement Classify module the macro name.

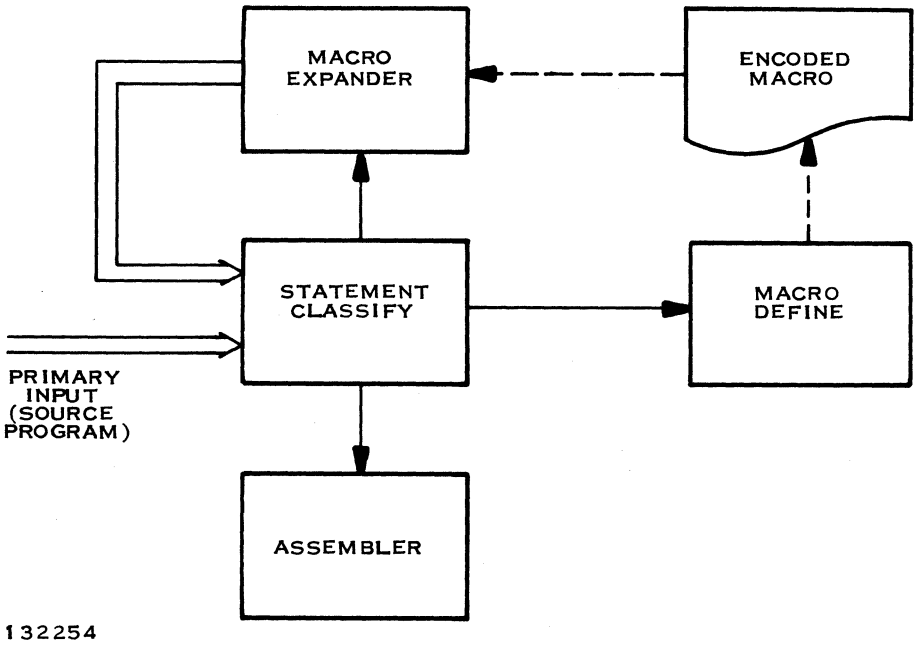


Figure 7-1. Macro Assembler Block Diagram

The Statement classify module recognizes a macro call by the macro name in the operation field. The statement classify module then passes the name to the macro expander module. The macro expander module assesses the desired macro definition. The macro call is expanded as specified in the macro definitions. The source statement that results from this expansion is used as input by the Statement Classify module.

During the expansion of a macro call, a macro language statement may call another macro, or a resulting source statement may be a macro call. A nesting of Macro's calls can occur in the expansion of one Macro call. The macro processor suspends processing of the current macro, processes the new macro, then resumes processing the original macro at the point of interruption. The macro translator allows a macro to be a recursive.

7.3 MACRO TRANSLATOR INTERFACE WITH THE ASSEMBLER

Expansion of a macro call may be varied according to the contents of the assembler symbol table (AST) and may result in alteration of the contents of the AST. The AST contains an entry for each symbol identified in the source program. The entry in the AST is divided into a number of components. The value of the symbol is stored as the value component (is a binary value used in computations). The segment component contains the location counter segment number of the symbol, and the attributes of this symbol are stored in the attribute component as a group of bits each of which represents an attribute of the symbol. The string component is null unless the macro translator places a string of characters in it. The length component contains the number of characters in the string component. An eight-bit user attribute field allows special attributes to be defined for a symbol. In this section, the symbol table entry components are referred to as *symbol components.*

Using keywords, a macro definition may access any component of any symbol in the AST. Symbols that are operands of the macro call may be used in the definition without any further declarations. Other symbols used in the Macro definitions must be explicitly declared before use.

A set of macro language statements beginning with a $MACRO statement and ending with a $END statement is a macro definition. The $MACRO statement includes a macro name that is used as the operation field. Macro definitions may appear anywhere in a program prior to macro calls that activate the definitions and may be unique to a program or shared by many programs.

The LIBIN directive makes it easy to incorporate a library of previously encoded macro definitions in every program. These definitions become a part of the source program but they are used only when a macro is called in the source program.

A macro definition need only be as sophisticated as its application requires. The macro definition simply redefine an instruction, supply one or more fixed operands for commonly used instructions, contain one or more calls for other macros, or call itself recursively. The statements in a macro definition may access AST symbol components to specify processing of a macro or alter the contents of the AST. To prevent the assembler from getting into an infinite loop, the maximum nesting level for macros is sixteen. However, the SETMNL directive may be used to change the established maximum as required.

7.4 MACRO LIBRARY

A MACRO library is a DX10 directory and each member file of the directory contains a MACRO definition. Two assembler directives, LIBIN and LIBOUT, identify MACRO libraries for input and output, respectively. In addition, a system MACRO library may be input via the assembler input parameters.

The purpose of a MACRO library is to reduce execution time and memory overhead associated with using MACROs. Execution time is reduced by encoding the MACRO definitions only once

and making them available for subsequent assembler runs. Memory requirements are reduced since MACRO definitions not under expansion reside only in the directory on disc.

7.5 MACRO LANGUAGE ELEMENTS

The elements of the macro language are labels, strings, constant operators, variables, variable qualifiers, keywords, and verbs. A macro definition consists of statements containing macro language verbs and model statements. A model statement can be constructed from some of the elements and results in an assembly language source statement. The elements of the macro language and model statements are explained fully in the following sections.

7.5.1 LABELS. A macro language label consists of one or two characters. The first must be an alphabetic character (A...Z) optionally followed by an alphanumeric character (A...Z, 0...9). Macro language labels are used to determine the sequence of processing of statements in a macro definition when the statements are not to be processed in order and have no significance in the actual assembly language. The following are examples of valid macro language labels:

```
L1  MA  C
```

7.5.2 STRINGS. The literal strings of the macro language consist of one or more characters enclosed in single quotes, and are identical to the character strings used in the assembly language.

An example is 'ONE'.

Another example is 'b' (a blank).

7.5.3 CONSTANTS AND OPERATORS. Constants for the macro language are defined the same as constants for the assembly language. The arithmetic operators of the assembly language applys also to the macro language. The logical operators and the relational operators of SDSMAC also apply to the macro language.

The macro language permits concatenation of macro variable components with literal strings, characters of model statements and other macro variables. Concatenation is indicated by writing character strings in juxtaposition with string mode references to MACRO variables.

7.5.4 VARIABLES. A macro definition may include variables, which are represented in the same manner as symbols in the assembler symbol table, with the restriction that they may be a maximum of two characters in length.

```
VA  P4  SC  F2  A  Z
```

NOTE

Macro variables are strictly local; they are available only to the macro which defines them. Access to symbols in the AST is through the symbol components.

7.5.4.1 Parameters. A parameter is a variable that is an operand of the expanded macro call and is declared in the $MACRO statement at the beginning of the macro definition. The sequence of parameters in the operand field of the $MACRO statement corresponds to the sequence of operands in the operand field of the macro instruction.

7.5.4.2 **Macro Symbol Table.** The macro translator maintains a macro symbol table (MST) similar to the symbol table of the assembler, and each entry consists of the string, value, length, and attributes of a variable or parameter. The Macro Expander module places parameters in the MST as it processes a macro call, and places variables in the MST as it processes the macro language statements that declare variables.

The string component contains a character string assigned to the macro variable or parameter by the macro expander. The value component contains the binary equivalent of the string component if the string component is an integer. The value component can also contain the value of the symbol if the string component is a symbol in the AST.

The length component contains the number of characters in the string component. The attribute components of the MST is similar to the attribute component of the AST entry in that it is a bit vector, the bits of which correspond to the attributes of the variable or parameter.

The Macro Expander comprehends the addressing modes of the assembler language. The value components contains a binary value which can be interpreted if the operand is a valid integer expression of any assembler addressing mode.

For example, the statement:

```
ADD $MACRO AU, AD
```

identifies a macro, ADD, having parameters AU and AD.

A macro call to activate that macro definition could be coded as follows:

```
ADD NUM, *3
```

The MST would now contain parameters AU and AD, and string component of parameter AU would be 'NUM'. The value component would be the value of the symbol NUM, and the attribute component would indicate that the parameter is supplied in a macro call. The length component would be 3. The string component of parameter AD would be '*3'. The value component would be 3 expressed as a binary number, and the length component would be 2. The attribute component would indicate that the parameter is an indirect workspace register address appearing in the macro call.

Another macro call for the same macro could be coded as follows:

```
ADD VAL(5), SUM
```

The components of the parameters AU and AD would now correspond to the operands of this instruction. The string component of parameter AU would be 'VAL(5)'. The value component would be 5 (the index register number), and the length component would be 6. The attribute component would indicate that parameter AU is an indexed memory address appearing in the macro call instruction.

The string component of parameter AD would be 'SUM', and the value component would be the value of SUM. The length component would be 3, and the attribute component would indicate that parameter AD appears in the macro call.

Each component of a macro variable may be accessed individually. Reference to a variable is made in either *binary mode* or *string mode.* In the binary mode, the referenced macro variable component is treated as a signed sixteen-bit integer. Binary mode access is made by writing the variable name and component. Thus, the binary mode value of the length component of AD would be the sixteen bit integer, 3. A reference to the string component of a macro variable in binary mode is, by definition, the sixteen-bit integer value of the ASCII representation of the first two characters of the string. The binary mode value of the string component of AD is > 5255, which is the ASCII representation for 'SU'.

String mode access of macro variable components is signified by enclosing the variable in colon characters (:); for example, :AD:.

NOTE

Colons are always used in pairs to enclose a variable name.

The string mode value of a component, other than the string component, is the decimal character string whose value is the binary value of the component. In the previous example, the string mode value of the length component of AD would be the character string '3'. If the value of SUM were > 28, then the string mode value of the value component of AD would be the character string '40', which is the decimal equivalent of > 28. Since the string component of a macro variable is a string, the string mode value of a string component is the entire string.

7.5.4.3 **Variable Qualifiers.** The components of a parameter or variable may be specified, using the specific names as shown in table 7-1. The variable name is followed by a period (.) and the single letter qualifier. The following examples show qualified variables:

AU.S String component of variable AU.
In the first example of the macro call for a macro 'ADD', AU.S equals the binary equivalent for 'NU', or > 4E55. If a colon (:) has indicated the string mode, the string component is 'NUM' (:AU.S: = 'NUM').

AU.A Attribute component of variable AU.
This component may be accessed by use of logical operators and keywords.

AU.V Value component of variable AU.
In the first example of the macro call for a macro 'ADD', this would be the value of the symbol 'NUM' in the AST.

AU.L Length component of variable AU.
In the first example fo the macro call for a macro 'ADD', AU.L = 3.

Table 7-1. Variable Qualifiers

Qualifier	Meaning
S	The string component of the variable
A	The attribute component of the variable.
V	The value component of the variable.
L	The length component of the variable

Except in an $ASG statement (described in a subsequent paragraph), an unqualified variable means the string component of the variable. In the two following examples, the concatenated strings are equivalent:

:CT.S: ♭ WAY	Variable CT qualified.
:CT: ♭ WAY	Variable CT unqualified.

When the string component of a variable is a symbol in the AST, the additional qualifiers of table 7-2 may be used to access the symbol components of that symbol. The symbol components of the parameters of macro instructions and the symbol value of an AST symbol are accessible directly. To access the other components of a symbol which has not been passed as a parameter in the macro definition, the symbol must be assigned as a string component of a macro variable and the symbol component qualifiers of table 7-2 applied to that variable. The following are examples of qualified variables that specify symbol componets of string components of variables:

B.SS	String component of symbol that is the string component of variable B. This is null unless a macro instruction has caused a string to be associated with by using a $ASG statement.
G2.SV	Value component of the symbol that is the string component of variable G2. If G2.S has been defined as 'MASK', a statement MASK EQU> FF has been encountered in the assembly language source when G2.SV = > FF. In string mode, :G2.SV: = '255'.
NO.SA	Attribute component of the symbol that is the string component of variable NO. This component may be accessed by use of logical operators and keywords, as described later.
V2.SL	Length component of the symbol that is the string component of macro variable V2. If a string has been assigned to the symbol which is V2.S, then V2.SL is the length of that string.
NV.SU	User attribute component of symbol that is the string component of variable NV. This component is zero except when a macro instruction has been issued to set bits in the component with a $ASG macro verb. This component is 8 bits long and may be used as desired.
LM.SG	Segment component of symbol that is the string component of variable LM.

Concatenation is especially useful when a previously defined string is augmented with additional characters. The string 'ONE' could be represented by a qualified variable such as CT.S. In that case, concatenation expressed as follows:

:CT.S: 'ƀWAY'

would provide the same result as writing

'ONE ƀWAY'

If the qualified variable CT.S represented 'TWO', the result of the concatenation in the example would be 'TWO WAY'. Strings and qualified variables may be concatenated as required and the variable need not be first. Components of variables that are represented by a binary value (e.g.,

CT.V and CT.L) are converted to their ASCII decimal equivalent before concatenation.

For example:

```
:CT.S:'bWAYb' :CT.L:
```

is expanded as

```
ONE WAY 3
```

since the length component of the variable CT is three.

Table 7-2. Variable Qualifiers for Symbol Components

Qualifier	Meaning
SS	String component of a symbol that is the string component of a variable.
SV	Value component of a symbol that is the string component of a variable.
SA	Attribute component of a symbol that is the string component of a variable.
SL	Length component of a symbol that is the string component of a variable.
SU	User attribute component of a symbol that is the string component of a variable.
SG	Segment component of a symbol that is the string component of a variable.

7.5.5 MODEL STATEMENTS. As mentioned earlier, a macro definition consists of statements that contain macro language verbs, and model statements. A model statement always results in an assembly language source statement and may consist only of an assembly language statement, or portions of an assembly language statement combined with string mode qualified variable components using the colon operator (:). In any case, the resulting source statement must be a legal assembler language statement or errors will result. The following examples show model statements:

```
MOVB  R6,R7
```

This model statement is itself an assembly language source statement that contains a machine instruction.

```
:P7.S:bbbSOCbbb:P2.S;,R8bbb:V4.S:
```

This model statement begins with the string component of variable P7. Three blanks, SOC, and three more blanks are concatenated to the string. The string component of variable P2 is concatenated to the result, to which, R8 and three blanks are concatenated. A final concatenation places the string component of variable V4 in the model statement. The result is an assembly language machine instruction having the label and comment fields and part of the operand field supplied as string components.

:MS.S: This model statement is the string component of variable MS. Preceeding statements in the macro definition must place a valid assembly language source statement in the string component to prevent assembly errors.

CAUTION

Conditional assembly directives may not appear as operations in a model statement. Comments supplied in model statement may not contain periods (.), since SDSMAC scans comments in the same way as model statements and improper use of punctuation may cause syntax errors.

7.5.6 SYMBOL ATTRIBUTE COMPONENT KEYWORDS. The macro language recognizes keywords to specify the attributes of assembler symbols and macro parameters. Each keyword represents a bit position within a word that contains all attributes of the symbol or parameter. A keyword may be used with a logical operator and the attribute component ot test or set a specific attribute of a symbol or parameter.

The keywords listed in table 7-3 may be used with a logical operator and the symbol attribute components (.SA) to test or set the corresponding attribute component in both the AST or MST. The following example shows an expression that uses a symbol attribute component keyword:

P5.SA&$STR This is the result of an AND operation between the attribute component of the symbol that is the string component of variable P5 and a bit vector corresponding to keyword $STR. The expression has a nonzero value when the contents of the string component of P5 is not null; otherwise, the expression has a value of 0.

Another example shows an expression that uses a symbol attribute keyword:

CT.SA++$REL This is the result of an OR operation between the attribute component of the symbol in the string component of variable CT and the bit corresponding to keyword $REL. The value of the expression is that of the attribute component showing the symbol as relocatable.

Table 7-3. Symbol Attribute Keywords

Keyword	Meaning
$REL	Symbol is relocatable.
$REF	Symbol is an operand of an REF directive.
$DEF	Symbol is an operand of a DEF directive.
$STR	Symbol has been assigned a component string.
$VAL	Symbol has been assigned a value.
$MAC	Symbol is defined as a macro name.
$UNDF	Symbol is not defined.

7.5.7 **PARAMETER ATTRIBUTE KEYWORDS.** The keywords listed in table 7-4 may be used with a logical operator and the macro symbol attribute component to test or set the corresponding attribute in the MST attribute component. These attribute keywords may be used to test or set attributes of both parameters and variables in the MST. The following examples show expressions that use parameter attribute component keywords:

P6.A&$PCALL	This is the result of an AND operation between the attribute component of variable P6 and the bit vector corresponding to keyword $PCALL. The expression has a nonzero value when variable P6 is a parameter supplied in a macro call. Otherwise the value of the expression is zero.
RA.A++$PSYM	This is the result of an OR operation between the attribute component of variable RA and the bit vector corresponding to keyword $PSYM. The value of the expression is that of the parameter attribute component showing the parameter as a symbolic memory address.

Table 7-4. Parameter Attribute Keywords

Keyword	Meaning
$PCALL	Parameter appears as a macro-instruction operand.
$POPL	Parameter is an operand list. The value component contains the number of operands in the list.
$PNDX	Parameter is an indexed memory address. The value component contains the index register number.
$PIND	Parameter is an indirect workspace register address.
$PATO	Parameter is an indirect autoincrement address.
$PSYM	Parameter is a symbolic memory address.

7.5.8 **VERBS.** The macro language supports eleven verbs that are used in macro language statements. Any statement in a macro definition that does not contain a macro language verb in the operation field is processed as a model statement. The verbs and the statements named after all verbs are described in the following paragraphs.

7.5.9 $MACRO.

Syntax definition:

<macro name>b . . . $MACROb . . . [<parm>] [,<parm>] . . . b . . . [<comment>]

The $MACRO statement must be the first statement of a macro definition, assigns a name to the macro and declares the parameters for the macro. The macro name consists of from one to six alphanumeric characters, the first of which must be alphabetic. Each < parm > is a parameter for the definition, as previously described in paragraph 7.5.4.1. The operand field may contain as many parameters as the size of the field allows, and must contain all parameters used in the macro definition.

The macro definition is used in the expansion of macro calls that have the macro name as an operation code. The syntax for a call is as follows:

$$[\langle\text{label}\rangle]\,\text{b}\ldots\langle\text{macro name}\rangle\text{b}\ldots\left[\left\{\begin{array}{l}\langle\text{operand}\rangle\\ \langle\text{operand list}\rangle\end{array}\right\}\right]\left[,\left\{\begin{array}{l}\langle\text{operand}\rangle\\ \langle\text{operand list}\rangle\end{array}\right\}\right]\ldots\text{b}\ldots[\langle\text{comment}\rangle]$$

When the label field contains a label, the label is assigned to the location of the first object or dummy object code of the expanded macro instruction. The macro name specifies the macro definition to be used. Each operand may be any expression or address type recognized by the assembler or a character string enclosed in quotes. Alternatively, an operand list may be used. An operand list is a group of operands enclosed in parentheses and separated by commas (when two or more operands are in the list) and is processed as a set after removal of the outer parentheses during macro expansion.

Operands (or operand lists) may be nested in parentheses in the macro call for use within macro definitions.

For example:

ONE $MACRO P1, P2

specifies 2 parameters.

A call such as

ONE PAR1, PAR2

will result in

'PAR1' being associated with P1 and 'PAR2' being associated with P2.

However, a call such as

ONE PAR1, (PAR21, PAR22)

will result in

'PAR1' being associated with P1 and 'PAR21, PAR22' being associated with P2.

Now if :P2: or :P2.S: is used as an operand in a model statement, it has the effect of being two operands (i.e., matching two parameters in the macro definition).

Processing of each macro call in a source program causes the Macro Expander to associate the first parameter in the $MACRO statement with the first operand or operand list on the macro call line and the second parameter with the second operand or operand list, etc. Each parameter receiving a value has the $PCALL attribute set. When the macro definition has more parameters specified than the number of operands in the macro call, the $PCALL attribute is not set for the excess parameters. The $PCALL attribute is also not set if an operand is null, i.e., the call line has two adjacent commas or an operand list of zero operands. Expansion of the macro can be conditioned on the number of operands by testing this attribute, $PCALL.

For example, a macro definition containing

```
AMAC $MACRO P1, P2, P3
```

when called by

```
AMAC AB1, AB2
```

sets $PCALL in parameters P1 and P2 but not for P3.

Similarly,

```
AMAC XY1,,XY3
```

causes $PCALL to be set for P1 and P3 but not for P2.

When the macro instruction has more operands than the number of parameters in the $MACRO statement, the excess operands are combined with the operand or operand list corresponding to the last parameter to form an operand list (or a longer operand list). For example, with the macro statement shown, the operands of the two macro calls in the following code would be assigned to the parameters in the same way:

```
ONE     EQU     9
TWO     EQU     43
THREE   EQU     86
FIX     $MACRO  P1,P2              MACRO FIX
          .
          .
          .
        FIX     ONE,TWO,THREE     MACRO-INSTRUCTION
        FIX     ONE,(TWO,THREE)   MACRO-INSTRUCTION
```

Parameter assignments:

P1.S	=	ONE	P2.S	=	TWO,THREE
P1.A	=	$PCALL	P2.A	=	$PCALL,$POPL
P1.L	=	3	P2.L	=	9
P1.V	=	9	P2.V	=	2 (number of operands in the list)

Another example of a parameter assignment in a macro statement is as follows:

```
A       EQU     7
B       EQU     15
C       DATA    17
D       DATA    63
E       EQU     95
F       EQU     47
G       EQU     58
H       EQU     101
I       EQU     119
PARM    $MACRO  P1,P2,P3,P4,P5,P6,P7,P8,P9
        .
        .
        .
        PARM    A,,B,(),C,(D),(E)(F),(G,(H,I)),*R7+
```

Parameter assignments:

P1.S	=	A	P2.S	=	(no string)
P1.A	=	$PCALL	P2.A	=	(zeroes)
P1.L	=	1	P2.L	=	0
P1.V	=	7	P2.V	=	0
P3.S	=	B	P4.S	=	(no string)
P3.A	=	$PCALL	P4.A	=	$POPL
P3.L	=	1	P4.L	=	0
P3.V	=	15	P4.V	=	0
P5.S	=	C	P6.S	=	D
P5.A	=	$PCALL	P6.A	=	$PCALL,$POPL
P5.L	=	1	P6.L	=	1
P5.V	=	0	P6.V	=	1
P7.S	=	(E) (F)	P8.S	=	G,(H,I)
P7.A	=	$PCALL,$PNDX	P8.A	=	$PCALL,$POPL
P7.L	=	6	P8.L	=	7
P7.V	=	47	P8.V	=	2

```
P9.S    =   *R7+
P9.A    =   $PCALL,$PATO
P9.L    =   4
P9.V    =   7
```

7.5.10 $VAR.

Syntax definition:

ƀ . . . $VARƀ . . . <var>[,<var>] . . . ƀ . . .[<comment>]

The $VAR statement declares the variables for a macro definition. The $VAR statement is required only if the macro definition contains one or more variables other than parameters. More than one $VAR statement may be included and each $VAR statement may declare more than one variable. Each <var> in the operand is a variable as previously described.

The following is an example of a $VAR statement:

```
$VAR      A,CT,V3      THREE VARIABLES FOR A MACRO
```

The example declares variables A, CT, and V3. A, CT, and V3 must not have been declared as parameters. The $VAR statement does not assign values to any components of the variables. $VAR statements may appear anywhere in the macro definition to which they apply, except that each variable must be declared before the first statement that uses the variable. It is logical to place $VAR statements immediately following the $MACRO statement.

7.5.11 $ASG

Syntax definition:

ƀ . . . $ASGƀ . . . { <expression> / <string> } ƀTOƀ var ƀ . . . [<comment>]

The $ASG statement assigns values to the components of a variable. Variables that are not parameters have no values for components until values are assigned using $ASG statements. Components previously assigned to parameters or to variables by $ASG statements may be assigned new values with $ASG statements.

The expression operand may be any expression valid to the assembler, and may contain binary mode variable references and the keywords in tables 7-3 and 7-4.

NOTE

The binary mode value of a string component or symbol string component used in an expression is the binary value of the first two characters of the string.

Thus, if GP.S has the string 'LAST', the value used for GP.S in an expression is the <string> hexadecimal number >4C41 which is the ASCII representation for LA.

A string may be one or more characters enclosed in single quotes or the concatenation of a literal with the string mode value of a qualified variable. The <VAR> may be either an unqualified variable or a qualified variable.

When the operands are both unqualified variables, all components are transfered to target variables. When the source variable is qualified or is a quoted string and the destination variable is unqualified, an error results. When the destination variable is qualified, only the specified component receives the corresponding component of the expression or string, with the exception that when a string is assigned to the string component of a variable or symbol, the length component of that variable or symbol is set to the number of characters in the assigned string. If the attribute component of the target variable is to be changed, only those attributes which can be tested using keywords are changed. Other attributes maintained by SDSMAC may or may not be changed, as appropriate.

NOTE

A qualified variable that specifies the length component is illegal as the target in a $ASG statement. Also, a qualified variable that specifies the attribute component or the value component of a macro variable which was declared to be a macro language label (for the purpose of a $GOTO) is illegal as the target in a $ASG statement.

The following examples show the use of the $ASG statement:

$ASG	P3 TO V3	Assign all the components of variable P3 to variable V3.
$ASG	:P3.S:'ES' TO P3.S	Concatenate string 'ES' to the string component of variable P3, and set the string component to the result. Also set the length component to a new value, 2 greater than the previous value.
$ASG	CT.A++$PSYM TO CT.A	Set the bit in the attribute component of variable CT to indicate the symbolic address attribute.

Variables P3, V3, and CT must have previously declared, either as parameters in a $MACRO statement or as variables in a $VAR statement.

The $ASG statement may be used to modify symbol components, as shown in the following examples. Assume that P3.V = 6 and P3.S = SUB.

$ASG	'TEN' TO G.S	Assigns 'TEN' as the string component of variable G. When 'TEN' is a label in the AST, this statement allows the use of symbol component qualifiers to modify the components of symbol TEN.
$ASG	P3.V TO G.SV	Sets the value component of the symbol in the string component of variable G to the value component of variable P3. In this case, the value component of TEN is set to 6.

$ASG 'A':P3.S: 'S' TO G.SS	Concatenates string 'A', the string component of variable P3 and string S and places the result in the string component of the symbol in the string component of variable G. Also sets the length component of the same symbol. Thus, the string component of TEN is ASUB5 and length component is 5.

7.5.12 $NAME.

Syntax definition:

<label>b. . .$NAMEb. . .[<comment>]

The $NAME statement associates a macro language label with a macro language statement. When a label is required for branching within a macro definition it must be provided by a $NAME statement. The $NAME statement performs no processing in the expansion of a macro instruction.

The following example shows a $NAME statement:

AB $NAME	BRANCH TO THIS POINT	A $GOTO statement with AB as an operand branches to this point.
$ASG	P3 TO V3	Expansion of the macro instruction continues with the $ASG statement.

7.5.13 $GOTO.

Syntax definition:

b. . .$GOTOb. . .<label>b. . .[<comment>]

The $GOTO statement branches within a macro definition, either to a $NAME statement or to an $END statement. The label is a macro language label of either type of statement.

The following example shows a $GOTO statement:

$GOTO AB	Branch to a $NAME statement having the label AB and execute the following statement, or to an $END statement having the label AB.

7.5.14 $EXIT.

Syntax definition:

b. . .$EXITb. . .[<comment>]

The $EXIT statement terminates processing of the macro expansion. The $EXIT statement has the same effect as a $GOTO statement with the label of the $END statement as the operand.

7.5.15 $CALL.

Syntax definition:

ƀ. . .$CALLƀ. . .<macro name>ƀ. . .[<comment>]

The $CALL statement initiates processing of the macro definition named in the operand field. The operands passed to the macro being expanded are mapped to the parameters of the macro specified in the $CALL statement. When the Macro Expander executes a $END statement or a $EXIT statement in the called macro, processing returns to the statement following the $CALL statement in the calling macro.

The following is an example of a $CALL statement:

$CALL CONV	Activates the macro definition CONV. The parameters of the calling macro are passed as the operands of the macro CONV.

7.5.16 $IF.

Syntax definition:

ƀ. . .$IFƀ. . .<expression>ƀ. . .[<comment>]

The $IF statement provides conditional processing in a macro definition. An $IF statement is followed by a block of macro language statements terminated by an $ELSE statement or an $ENDIF statement. When the $ELSE statement is used, the $ELSE statement is followed by another block of macro language statements terminated by an $ENDIF statement. When the expression in the $IF statement has a nonzero value, the block of statements following the $IF statement is processed. When the expression in the $IF statement has a zero value, the block of statements following the $IF statement is skipped. When the $ELSE statement is used, and the expression in the $IF statement has a nonzero value, the block of statements following the $ELSE statement and terminated by the $ENDIF statement is skipped. Thus, the condition of the $IF statement may determine whether or not a block of statements is processed, or which of two blocks of statements is processed. Furthermore, a block may consist of zero or more statements.

The expression may be any expression as defined for the $ASG statement and may include qualified variables and keywords. The expression defines the condition for the $IF statement.

NOTE

The expression is always performed in binary mode. Specifically, the relational operators (<, >, =, #=, etc.) operate only on the binary mode value of the macro variable. This has the effect that comparisons of two character strings may be done only on the initial two character positions.

The following examples show conditional processing in macro definition:

```
.
.
.
$IF     KY.SV
.
.       BLOCK A
.
$ELSE
.
.       BLOCK B
.
$ENDIF
$IF     T.A&$PCALL=0
.
.       BLOCK A
.
$ENDIF

$IF     T.L=5
.
.       BLOCK A
.
$ENDIF
```

Process the statements of Block A when the value component of the symbol in the string component of variable KY contains a nonzero value. Process the statements of Block B when the component contains zero. After processing either block of statements, continue processing at the statement following the $ENDIF statement.

Process the statements of Block A when the attribute component of parameter T indicates that parameter T was not supplied in the macro instruction. If parameter T was supplied, do not process the statements of Block A. Continue processing at the statement following the $ENDIF statements in either case.

Process the statement of Block A when the length component of variable T is equal to 5. If the length component of variable is not equal to 5, do not process the statements of Block A. Continue processing at the statement following the $ENDIF statement.

7.5.17 $ELSE

Syntax definition:

ƀ . . . $ELSEƀ . . . . [<comment>]

The $ELSE statement begins an alternate block to be processed if the proceding $IF expression was false.

7.5.18 $ENDIF

Syntax definition:

ƀ. . .$ENDIFƀ. . .[<comment>]

The $ENDIF statement terminates conditional processing initiated by an $IF statement in a macro definition. Examples of $ENDIF statements and their use are shown in a preceding paragraph.

7.5.19 $END

Syntax definition:

[<label>]ƀ. . .$ENDƀ. . .<macro name>ƀ. . .[<comment>]

The $END statement marks the end of the group of statements of the macro definition named in the operand. When executed, the $END statement terminates the processing of the macro definition. The label may be used in a $GOTO statement to terminate processing of the macro definition.

The following is an example of an $END statement:

```
$END FIX          Terminates the definition of macro FIX
```

7.6 ASSEMBLER DIRECTIVES TO SUPPORT MACRO LIBRARIES

Two directives have been added to support the use of libraries of macros in SDSMAC. These two directives are LIBOUT, which is used to build or add to a library of macro definitions, and LIBIN, which is used to "recall" a previously built macro library.

7.6.1 LIBOUT DIRECTIVE.

Format:

```
b. . .LIBOUTb. . .<LIBRARY-ACCESS-NAME>
```

The LIBOUT directive declares a MACRO library where MACRO definitions are written during an assembly. The library must have been previously created by a CFDIR (create file directory) utility command. MACRO definitions appearing in the assembler input stream following a LIBOUT directive are written to the specified library upon successful translation. MACRO definitions appearing prior to the first LIBOUT directive remain in memory and are not written to any library. Multiple LIBOUT directives may appear in a single assembly. Each successive output library supercedes its predecessor so that only one output library is in effect at a time, the same library specified on multiple LIBOUT directives. Furthermore, a library may be used for both input and output simultaneously. MACRO definitions are written to the library using the replace option which will redefine any MACRO with the same library name. Hence, a macro library may be maintained (updated) without difficulty.

In addition to MACRO definitions, a sub-directory of the MACRO library with the name DDFX contains the result of DXOP and DFOP directives and MACRO names which redefine an assembly language instruction, directive, or peudo-instruction appearing within the span of the current LIBOUT directive.

The MACRO definitions, DXOPs and DFOPs are written to the library completely replacing any prior definitions of the symbols on that MACRO library. For example, if a MACRO library contained a MACRO definition for the symbol 'LOCK' and a subsequent assembly encounters a 'DFOP LOCK, ABS' statement while a LIBOUT directive to that library is in effect, the MACRO library will result in containing information that 'LOCK' is another name for the instruction 'ABS'. The MACRO definition which existed on the library previously will have been deleted.

7.6.2 LIBIN DIRECTIVE.

Format:

```
b. . .LIBINb. . .<LIBRARY-ACCESS-NAME>
```

The LIBIN directive declares a MACRO library to be used in the current assembly. The library must have been previously created and must contain only MACRO definitions and DFOP and

DXOP directives previously encoded during another assembly (by use of the LIBOUT directive). Multiple LIBIN directives may appear in a single assembly. When the LIBIN directive is encountered the library directory is examined for any redefinition of assembler instructions and their existence flagged. No further use is made of the MACRO library until an undefined operation is encountered. At that time, the MACRO library is searched for a possible MACRO definition of the operation. In the case of multiple MACRO libraries, the search order is inverse to the order of presentation, i.e., the last MACRO library is searched first. The system MACRO library specified in the SCI XMA command is always searched last.

7.6.3 MACRO LIBRARY MANAGEMENT. The MACRO library may be listed, added to, deleted from, and replicated using a combination of utility commands provided by the operating system and the MACRO assembler LIBIN and LIBOUT directives.

To list or replicate a MACRO library, use the utility commands provided by the operating system.

To add to an existing MACRO library or change an existing MACRO definition, DFOP, or DXOP, use only the LIBOUT directive provided by the MACRO assembler. Do not use utility commands for copying files to copy a MACRO definition to another MACRO library.

To delete MACRO definitions, DFOPs, and DXOPs, use the utility commands provided by the operating system to delete files. In the following examples assume that a MACRO library with the name

. SYSTEM. MACROS

is present.

a. If the result of the DFOP

DFOP T, TEXT

is to be deleted, then use the delete file utility command to delete the file:

. SYSTEM. MACROS. DDFX. T

b. If the result of the DXOP

DXOP SVC,15

is to be deleted, use the delete file utility command to delete the following file in the same manner as above:

. SYSTEM. MACROS. DDFX. SVC

c. If a MACRO definition for 'CALL' is to be deleted, use the delete file utility command to delete the following file:

. SYSTEM. MACROS. CALL

d. If a MACRO definition is to be deleted which redefines an assembly language instruction, directive, or pseudo-instruction, then two files must be deleted. If the MACRO name were 'TEXT' then delete:

. SYSTEM. MACROS. TEXT
. SYSTEM. MACROS. DDFX. TEXT

If only one of these is deleted either an "invalid opcode" assembly error will result or the intended macro will not have been used.

7.7 MACRO EXAMPLES

Macros may simply substitute a machine instruction for a macro instruction, or they may include conditional processing, access the assembler symbol table, and employ recursion. Several examples of macro definitions are described in the following paragraphs.

7.7.1 MACRO GOSUB. Macro GOSUB is an example of a macro that substitutes a machine instruction for the macro instruction. The macro definition consists of three macro language statements, one of which is a model statement, as follows:

GOSUB	$MACRO	AS	Defines macro GOSUB and declares a parameter, AD.
	BL	:AD.S:	A model statement that results in a BL instruction with the string component of the parameter as operand.
	$END	GOSUB	Terminates macro GOSUB.

The syntax of the macro instruction for the GOSUB macro is defined as follows:

[<label>]ƀ. . .GOSUBƀ. . .<address>ƀ. . .[<comment>]

When a label is used, it is effectively the label of the resulting BL machine instruction. The address may be any address form that is valid for a BL instruction. When a comment is used, it applies to the macro instruction. For example, the following macro instruction is valid for the GOSUB macro:

```
GOSUB @SUBR
```

The statement in the example results in a machine instruction to branch and link to a subroutine at location SUBR, as follows:

```
BL @SUBR
```

Another example shows the macro instruction that could be used if the subroutine address were in workspace register 8 and had a label.

```
NEXIT   GOSUB   *R8
```

The resulting instruction would be:

```
NEXIT   BL      *R8
```

7.7.2 MACRO EXIT. Macro EXIT is an example of a macro that supplies an assembler directive the first time the macro is executed, and a machine instruction each successive time. The macro requires an EQU directive to be placed in the source program prior to calling the macro, and the definition consists of nine macro language statements, including two model statements. The definition is as follows:

EXIT	$MACRO		Defines macro EXIT with no parameters.
	$VAR	L	Defines variable L.
	$ASG	'F1' TO L.S	Assign F1 to the string component of variable L to allow access to symbol F1 in assembler symbol table.
	XOP	@TERM,15	Model statement – places an XOP machine instruction in source program.
	$IF	L.SV	If the value component of symbol F1 is a nonzero value, perform the next two statements and terminate the macro. Otherwise, terminate the macro.
TERM	BYTE	16	Model statement – places a byte directive referenced by the XOP instruction following the XOP instruction.
	$ASG	0 TO L.SV	Set the value component of symbol F1 to zero. Any further calls to macro EXIT will omit the preceding model statement and its statement.
	$ENDIF		Defines the end of conditional processing.
	$END		End of macro definition.
	.		
	.		
	.		
F1	EQU	1	Defines F1 with a value of 1. This is not part of the macro definition, but is a source statement. It must precede the first macro call for macro EXIT, and may precede the definition.

The syntax of the macro instruction for the EXIT macro is defined as follows:

[<label>] b. . .EXIT

When a lable is used it is effectively the label of the XOP machine instruction resulting from macro. The first time the macro is called, the following source statements are placed in program:

```
        XOP     @TERM,15
TERM    BYTE    16
```

Subsequent calls for the macro result in the following:

```
        XOP     @TERM,15
```

7.7.3 **MACRO ID.** Macro ID is an example of a macro having a default value. The macro supplies two DATA directives to the source program. The macro consists of nine macro language statements, four of which are model statements. The definition is as follows:

ID	$MACRO	WS,PC	Defines ID with parameters WS and PC.
	DATA	:WS.S:	Model statement – places a DATA directive with the string of the first parameter as the operand in the source program.
	$IF	PC.A&$PCALL	Tests for presence of parameter PC.
	DATA	:PC.S:,15	Model statement – places a DATA directive in the source program. The first operand is the string of the second parameter, and the second operand is 15. This statement is processed if the second parameter is present.
	$ELSE		Start of alternate portion of definition.
	DATA	START,15	Model statement – places a DATA directive in the source program. The first operand is label START, and the second operand is 15. This statement is processed if the second parameter is omitted.
START			Model statement – places a label START in the source program. This statement is processed if the second parameter is omitted.
	$ENDIF		End of conditional processing.
	$END		End of macro.

This macro could be used to place a three-word vector at the beginning of a program. The first word could be the workspace address, the second, the entry point, and the third, the value 15 to be placed in the SR register. The first operand of the macro instruction would be the workspace address, and the second operand would be the entry point. When the executable code immediately follows the vector, and the entry point is the first word of executable code, the second parameter may be omitted. The syntax definition of the macro instruction for macro ID is as follows:

<label>b. . .IDb. . .<address>[,<address>]b. . .[<comment>]

The label becomes the label of the three-word vector, and the addresses may be expressions or symbols.

The following is an example of a macro instruction for macro ID:

```
PROG1   ID      WORK1,BEGIN
```

The resulting source code would be:

```
PROG1   DATA    WORK1
        DATA    BEGIN,15
```

When the entry point immediately follows the macro instruction, the macro instruction could be coded as follows:

```
PROG2    ID          WORK2
```

This would result in the following source code:

```
PROG     DATA        WORK2

         DATA        START,15

START
```

This form of the macro instruction imposes two restrictions on the source program. The source program may not use the label START and may not call macro ID more than once. The user may prevent problems with labels supplied in macros by reserving certain characters for use in macro-generated labels. A macro definition may maintain a count of the number of times it is called, and use this count in each label generated by the macro.

7.7.4 MACRO UNIQUE

```
 0001                    IDT 'UNIQUE'
 0003          *         THIS EXAMPLE DEMONSTRATES A METHOD FOR CREATING UNIQUE
 0004          *         LABELS USING THE MACRO LANGUAGE. EACH CALL OF THE MACRO
 0005          *         GENERATES A UNIQUE LABEL OF THE FORM 'U,,xxx' WHERE'xxx'
 0006          *         IS A NUMBER
 0007          LABEL     $MACRO
 0008          *         DECLARE A VARIABLE TO USE IN THE MACRO
 0009                    $VAR L
 0010          *         ASSIGN THE CHARACTER STRING OF A SYMBOL THAT WILL HOLD
 0011          *             A COUNTER VALUE AND THE LAST LABEL GENERATED
 0012                    $ASG 'U;;;;;' TO L.S
 0013          *         INCREMENT THE SYMBOL VALUE OF 'U;;;;;' TO OBTAIN THE
 0014          *             LABEL VALUE
 0015                    $ASG L. SV+1 TO L.SV
 0016          *         CREATE THE LABEL AND SAVE IN THE SYMBOL STRING COMPONENT
 0017          *         GENERATE THE LABEL IN THE NEXT LABEL FIELD. NOTE THAT
 0018          *             MODEL STATEMENT STARTS IN COLUMN 1
 0019          U ; ; : L.SV:
 0020                    $END
 0021          *
 0022          *         NOW GENERATE SOME LABELS
 0023          *
 0024                    LABEL
*0001  0000    U;;1
 0025  0000    0000      DATA 0, 1
       0002    0001
 0026                    LABEL
*0001  0004    U;;2
 0027                    LABEL
*0001  0004    U;;3
 0028  0004    0004      DATA 4
 0029                    END
NO ERRORS
```

7.7.5 **MACRO GENCMT.** Macro GENCMT is an example showing how to implement both those comments which appear in the macro definition only, and those comments which appear in the expansion of the macro. When this macro is called, the statement in line six generates a comment.

```
0001                                    IDT 'GENCMT'
0002                          GENCMT    $MACRO
0003                                    $VAR V
0004                          *THIS IS A MACRO DEFINITION COMMENT
0005                                    $ASG '*' TO V.S
0006                          :V.S: THIS IS A MACRO EXPANSION COMMENT
0007                                    $END
0008                                    GENCMT
*0001                         *THIS IS A MACRO EXPANSION COMMENT
 0009    0000    0000                   DATA 0,1
         0002    0001
 0010                                   GENCMT
*0001                         *THIS IS A MACRO EXPANSION COMMENT
 0011                                   GENCMT
*0001                         *THIS IS A MACRO EXPANSION COMMENT
 0012    0004    0004         LABEL     DATA 4
 0013                                   END
```

7.7.6 **MACRO LOAD.**

```
0001                                    IDT 'LOAD'
0002                          *
0003                          * GENERALIZED LOAD IMMEDIATE MACRO
0004                          *
0005                          * THIS MACRO DEMONSTRATES USE OF THE MACRO
0006                          * SYMBOL ATTRIBUTES $PSYM, $PNDX, $PATO, $PIND.
0007                          *
0008                          * OPERANDS: D (DESTINATION) MAY BE REGISTER,
0009                          *                           INDIRECT, SYMBOLIC,
0010                          *                           OR AUTO-INC.
0011                          *           V (VALUE) SHOULD BE LITERAL VALUE.
0012                          *
0013                          *
0014                          * IF THE FIRST OPERAND IS NOT A REGISTER, IT
0015                          * WILL BE MOVED INTO THE SCRATCH REGISTER
0016                          * BEFORE PERFORMING THE LOAD. THE SCRATCH
0017                          * REGISTER IS ASSUMED TO BE R0.
0018                          *
0019                          *
0020                          * THIS SYMBOL DEFINITION OR'S TOGETHER ALL
0021                          * ADDRESSING MODES BUT 'REGISTER'.
0022                          *
0023              001E        COMPLX EQU  $PATO++$PSYM++$PNDX++$PIND
0024                          *
0025                          * THIS MACRO WILL MASK OUT THE REGULAR 'LI'
0026                          * INSTRUCTION, SO THE 'DFOP' FOR 'LI' IS
0027                          * USED TO DEFINE A SYNONYM FOR THE 'LI'
0028                          * INSTRUCTION.
0029                          *
```

```
0030                                    DFOP LI$,LI
0031                          LI        $MACRO D,V
0032                                    $IF D.A&COMPLX
0033                                      LI$    RO,:V:
0034                                      MOV  RO,:D:
0035                                    $ELSE
0036                                      LI$    :D:,:V:
0037                                    $ENDIF
0038                                    $END
0039    0000    0000      LOC           DATA 0
0040                                    LI   *R5,25
*0001   0002    0200                      LI$    R0,25
        0004    0019
*0002   0006    C540                      MOV  R0,*R5
0041                                    LI     R12,4
*0001   0008    020C                      LI$    R12,4
        000A    0004
0042                                    LI    12(R13),16
*0001   000C    0200                     LI$    R0,16
        000E    0010
*0002   0010    CB40                      MOV  R0,12(R13)
        0012    000C
0043                                    LI  @LOC,111
*0001   0014    0200                      LI$    R0,111
        0016    006F
*0002   0018    C800                      MOV  R0,@LOC
        001A    0000'
0044                          *
0045                          * NOTE THAT THE FOLLOWING CASE DOES NOT
0046                          * GENERATE THE DESIRED CODE. TO CORRECTLY
0047                          * DETECT MEMORY LOCATION REFERENCES, LABELS
0048                          * SHOULD HAVE '@' SIGNS PRECEEDING THEM.
0049                          *
0050                                    LI  LOC,111
*0001   001C    0200                      LI$    LOC,111
        001E    006F
********  REGISTER REQUIRED
0051                                    END
0001 ERRORS, LAST ERROR AT  0050
```

7.7.7 MACRO TABLE.

```
0001                                    IDT 'TABLE'
0002                          *
0003                          * THIS MACRO DEMONSTRATES RECURSIVE PROCESSING
0004                          *
0005                          * WHEN MORE OPERANDS ARE PASSED TO A MACRO
0006                          * THAN WERE INCLUDED IN THE DEFINITION, ALL THE
0007                          * SURPLUS OPERANDS ARE ASSIGNED (WITH THE
0008                          * COMMAS BETWEEN THEM) TO THE LAST PARAMETER.
0009                          * THIS IS A USEFUL FEATURE WHEN RECURSIVE PRO-
0010                          * CESSING IS NEEDED.
0011                          *
```

```
0012                        * THE EXPECTED OPERAND FOR THE 'OR' MACRO IS A
0013                        * LIST OF BIT PATTERNS 16 BITS IN WIDTH. THIS
0014                        * MACRO USES RECURSION TO 'OR' THE BITS
0015                        * TOGETHER. 'TEMP' IS A SYMBOL USED BY THE
0016                        * MACRO.
0017                        *
0018            0000        TEMP        EQU 0
0019                        OR          $MACRO A,B
0020                                    $VAR T
0021                                    $ASG 'TEMP' TO T.S
0022                                    $ASG A.V++T.SV TO T.SV
0023                                    $IF  B.A&$PCALL
0024                                        OR :B.S:
0025                                    $ELSE
0026                                        DATA :T.SV:
0027                                        $ASG 0 TO T.SV
0028                                    $ENDIF
0029                                    $END
0030                                    OR >100
*0001       0000        0100                DATA 256
 0031                                   OR 1,2,4,8
*0001                                       OR 2, 4, 8
*0001                                       OR 4, 8
*0001                                       OR 8
*0001       0002        000F                DATA 15
 0032                                   OR 1, 1, 2, 4, 8
*0001                                       OR 1, 2, 4, 8
*0001                                       OR 2, 4, 8
*0001                                       OR 4, 8
*0001                                       OR 8
*0001       0004        000F                DATA 15
 0033                                   OR >11, >1100
*0001                                       OR >1100
*0001       0006        1111                DATA 4369
 0034                                   END
NO ERRORS
```

7.7.8 MACRO LISTS.

```
0001                                    IDT 'LISTS'
0002                        *
0003                        * THE PREORD AND ENDORD MACROS DEMONSTRATE
0004                        * RECURSION AND LIST PROCESSING.
0005                        *
0006                        *
0007                        * INPUTS:    A PARENTHESIZED EXPRESSION OF
0008                        *            THE FOLLOWING FORM:
0009                        *
0010                        *                                   A,OP,C
0011                        *
```

```
0012                *               A= PARENTHESIZED EXPRESSION
0013                *               OP= OPERATION
0014                *                    (MULTIPLICATION IS REPRESENTED
0015                *                     AS A NULL PARAMETER, SIMILAR
0016                *                     TO ITS REPRESENTATION IN
0017                *                     ALGEBRAIC EXPRESSIONS)
0018                *               B= PARENTHESIZED EXPRESSION
0019                *
0020                * OUTPUTS:  UNPARENTHESIZED EXPRESSION IN
0021                *           PREORDER (PREORD), OR ENDORDER
0022                *           (ENDORD).
0023                ***************************
0024                * PREORDER MACRO DEFINITION
0025                *
0026                PREORD $MACRO A,OP,B
0027                       $VAR C VARIABLE TO HOLD '*' FOR COMMENTS.
0028                *
0029                * PRINT THE OPERATION
0030                *
0031                       $ASG '*' TO C.S
0032                       $IF OP.A&$PCALL=0
0033                            $ASG '*' TO OP.S
0034                       $ENDIF
0035                :C: :OP:
0036                *
0037                * PRINT THE FIRST OPERAND
0038                *
0039                       $IF A.A&$POPL
0040                            PREORD :A:
0041                       $ELSE
0042                :C: :A:
0043                       $ENDIF
0044                *
0045                * PRINT THE SECOND OPERAND
0046                *
0047                       $IF B.A&$POPL
0048                            PREORD :B:
0049                       $ELSE
0050                :C: :B:
0051                       $ENDIF
0052                       $END
0053                ***************************
0054                * ENDORDER MACRO DEFINITION
0055                *
0056                ENDORD $MACRO A,OP,B
0057                       $VAR C VARIABLE TO HOLD '*' FOR COMMENTS.
0058                       $ASG '*' TO C.S
0059                *
0060                * PRINT THE FIRST OPERAND
0061                *
0062                       $IF A.A&$POPL
0063                            ENDORD :A:
0064                       $ELSE
```

```
0065                    :C:  :A:
0066                              $ENDIF
0067                    *
0068                    * PRINT THE SECOND OPERAND
0069                    *
0070                              $IF B.A&$POPL
0071                                 ENDORD  :B:
0072                              $ELSE
0073                    :C:  :B:
0074                              $ENDIF
0075                    *
0076                    * PRINT THE OPERATION
0077                    *
0078                              $IF OP.A&$PCALL=0 THEN
0079                                 $ASG '*' TO OP.S
0080                              $ENDIF
0081                    :C:  :OP:
0082                              $END
0083                    *
0084                    * SAMPLE MACRO CALLS
0085                    *
0086                     PREORD A, /, B
*0001                   * /
*0002                   * A
*0003                   * B
0087                     ENDORD A, /, B
*0001                   * A
*0002                   * B
*0003                   * /
0088                     PREORD (A, +, B) , , (6, /, (2, –, B))
*0001                   * *
*0002                                PREORD A, +, B
*0001                   * +
*0002                   * A
*0003                   * B
*0003                                PREORD 6, /, (2, –, B)
*0001                   * /
*0002                   * 6
*0003                                PREORD 2, –, B
*0001                   * –
*0002                   * 2
*0003                   * B
0089                     ENDORD (A, +, B), , (6, /, (2, –, B))
*0001                                ENDORD A, +, B
*0001                   * A
*0002                   * B
*0003                   * +
*0002                                ENDORD 6, /, (2, –, B)
*0001                   * 6
*0002                                ENDORD 2, –, B
*0001                   * 2
*0002                   * B
*0003                   * –
```

```
*0003                    * /
*0003                    * *
 0090                     PREORD ((X, +, Y), /, (X, –, Y)), –, (1, /, Z)
*0001                    * –
*0002                                PREORD (X, +, Y), /, (X, –, Y)
*0001                    * /
*0002                                PREORD X, +, Y
*0001                    * +
*0002                    * X
*0003                    * Y
*0003                                PREORD X, –, Y
*0001                    * –
*0002                    * X
*0003                    * Y
*0003                                PREORD 1, /, Z
*0001                    * /
*0002                    * 1
*0003                    * Z
 0091                     ENDORD ( (X, +, Y), /, (X, –, Y) ), –, (1, /, Z)
*0001                                ENDORD (X, +, Y), /, (X, –, Y)
*0001                                ENDORD X, +, Y
*0001                    * X
*0002                    * Y
*0003                    * +
*0002                                ENDORD X, –, Y
*0001                    * X
*0002                    * Y
*0003                    * –
*0003                    * /
*0002                                ENDORD 1, /, Z
*0001                    * 1
*0002                    * Z
*0003                    * /
*0003                    * –
```

THE FOLLOWING SYMBOLS ARE UNDEFINED

A
B
X
Y
Z

NO ERRORS

SECTION VIII

RELOCATABILITY AND PROGRAM LINKING

8.1 INTRODUCTION

The assemblers for the Model 990 Computers and the TMS 9900 Microprocessor supply both absolute and relocatable object code that may be linked as required to form executable programs from separately assembled modules. This section contains guidelines to assist the user in taking full advantage of these capabilities.

8.2 RELOCATION CAPABILITY

Relocatable code includes information that allows a loader to place the code in any available area of memory. This allows the most efficient use of available memory, and is required for disk-resident programs executed under DX10. Absolute code must be loaded into a specified area of memory. Absolute code is appropriate for code that must be placed in dedicated areas of memory, and may be used for memory-resident programs executing under operating systems.

Object code generated by an assembly is a representation of machine language instructions, addresses, and data comprising the assembled program. The code may include absolute segments and program-relocatable segments. If SDSMAC, the Cross Assembler or TXMIRA is used, the code may include a data-relocatable segment and numerous common-relocatable segments. In assembly language source programs, symbolic references to locations within a relocatable segment are called relocatable addresses. These addresses are represented in the object code as displacements from the beginning of a specified segment. A program-relocatable address, for example, is a displacement into the program segment. At load time, all program-relocatable addresses are adjusted by a value equal to the load address. SDSMAC, the Cross Assembler, and TXMIRA support additional types of relocatability-data relocatability and common-relocatability. Data-relocatable addresses are represented by a displacement into the data segment. There may be several types of common-relocatable addresses in the same program, since distinct common segments may be relocated independently of each other. A subsequent section of this manual describes the representation of these relocatable addresses in the object code.

8.2.1 RELOCATABILITY OF SOURCE STATEMENT ELEMENTS. Elements of source statements are expressions, constants, symbols, and terms. Terms are absolute in all cases; the other elements may be either absolute or relocatable.

The relocatability of an expression is a function of the relocatability of the symbols and constants that make up the expression. An expression is relocatable when the number of relocatable symbols or constants added to the expression is one greater than the number of relocatable symbols or constants subtracted from the expression. (All other valid expressions are absolute.) When the first symbol or constant is unsigned, it is considered to be added to the expression. When a unary minus follows an addition operator in an expression, the effective operation is subtraction. When a unary minus follows a subtraction operator, the effective operation is addition. For example, when all symbols in the following expressions are relocatable, the expressions are relocatable:

```
LABEL+1

LABEL+TABLE+-INC

-LABEL+TABLE+INC
```

Decimal, hexadecimal, and character constants are absolute. Assembly-time constants defined by absolute expressions are absolute, and assembly-time constants defined by relocatable expressions are relocatable.

Any symbol that appears in the label field of a source statement other than an EQU directive is absolute when the statement is in an absolute block of the program. Any symbol that appears in the label field of a source statement other than an EQU directive is relocatable when the statement is in a relocatable block of the program.

The relocatability of expressions having logical and relational operators (SDSMAC only) follows similar rules to those for expressions containing only arithmetic operators. The result of a logical operation between a relocatable constant or symbol and an absolute constant or symbol is relocatable. A logical operation between two relocatable elements of an expression is invalid. Relational operators result in an absolute value, 0 or 1. The relation is the assembly-time relation and ignores the effect of relocation on relocatable values.

To summarize, a location is either absolute or relocatable. The location may contain either absolute or relocatable values. The example program in Appendix J includes absolute locations with relocatable contents and relocatable locations with absolute contents.

8.3 PROGRAM LINKING

Since the assembler includes directives that generate the information required to link program modules, it is not necessary to assemble an entire program in the same assembly. A long program may be divided into separately assembled modules to avoid a long assembly or to reduce the symbol table size. Also, modules common to several programs may be combined as required. A linking loader links the programs as it loads them, so that the loaded program functions as if it has been assembled in a single assembly. Alternatively, program modules may be linked by the Link Editor to form a linked object module that may be stored on a library and/or loaded as required. The following paragraphs define the linking information that must be included in a program module.

8.3.1 EXTERNAL REFERENCE DIRECTIVES. Each symbol from another program module must be placed in the operand field of an REF or SREF directive in the program module that requires the symbol. When the modules are to be linked by the linking loader, the IDT character string of each program module that defines one or more of these symbols must also be placed in the operand field of an REF directive within one of the program modules being linked. The first module may contain an REF directive that contains the IDT character strings of all modules to be linked. When the modules are to be linked by the Link Editor, IDT character strings need not be placed in REF directive operand fields.

8.3.2 EXTERNAL DEFINITION DIRECTIVE. Each symbol defined in a program module and required by one or more other program modules must be placed in the operand field of a DEF directive.

8.4 PROGRAM IDENTIFIER DIRECTIVE

Program modules that are to be linked by the Link Editor should include an IDT directive. The module names in the character strings of the IDT directives should be unique.

Program modules that are to be linked by the linking loader must meet the following requirements:

- Subsequent program modules after the first module must include an IDT directive.
- The first six characters of the IDT character string must be unique with respect to the other IDT character strings submitted to the loader during the loading operation.

8.5 LINKING PROGRAM MODULES

The linking loader builds a list of symbols from REF directives as it loads the program modules. The loader matches symbols from DEF directives to the symbols in the reference list. The loader also matches the first six characters of IDT character strings with symbols in the reference list.

When object code for several program modules is on the same cassette, and a program that requires only some of these modules is being loaded, the loader ignores those program modules whose IDT character strings do not appear in the reference list of the loader. This allows program modules from several cassettes to be loaded without requiring the user to locate the required modules on the cassettes. However, it requires that all referencing modules precede the modules they reference in the sequence in which the loader loads the modules.

The Link Editor matches symbols from REF directives and symbols from DEF directives in a similar manner within a program phase. The Editor follows linking commands to determine the modules to be linked, and does not match IDT character strings with REF directive operands. Refer to Sections 4.5.3 and 4.5.4 for linking commands generatable from the assembler.

SECTION IX

OPERATION OF THE MACRO ASSEMBLER

9.1 GENERAL

The 990 Macro Assembler executes under the DX10 operating system. The Macro Assembler has the following features:

- Assembles the 72 instructions of the instruction set for the Model 990/10 with map option.
- Supports 31 assembler directives, 11 in addition to those supported by other assemblers.
- Supports three pseudo-instructions, one in addition to those supported by other assemblers.
- Supports use of parentheses in expressions.
- Supports logical operators in expressions.
- Supports relational operators in expressions.
- Supports a logical division operator.
- Supports additional output options.
- Supports a powerful macro language.

The Macro Assembler is defined in detail in Section VII of this document.

9.2 OPERATING THE MACRO ASSEMBLER

The Macro Assembler is executed by the DX10 System Command Interpreter (SCI) and may run in either of two modes:

1. Background
2. Batch Background.

To execute the Macro Assembler in background mode, enter the SCI command XMA.

The XMA command prompts for the following parameters:

```
        SOURCE ACCESS NAME: <access name>
        OBJECT ACCESS NAME: <access name>
          LIST ACCESS NAME: <access name>
         ERROR ACCESS NAME: <access name>
                   OPTIONS: <keyword list>
MACRO LIBRARY PATHNAME: <directory access name>
```

SOURCE ACCESS NAME specifies the input file or device containing the assembly language code to be assembled. No default is allowed for this parameter.

OBJECT ACCESS NAME specifies the output file or device to which the object code is to be written. If this parameter is null, no object output is produced. This is useful for preliminary assemblies to check for errors; since the assembler produces no output, it operates faster.

LIST ACCESS NAME specifies the file or device to which the assembly listing is to be written. If DUMY is entered, no assembly listing is produced.

ERROR ACCESS NAME specifies the output file to which assembly errors are written. This file may be viewed by entering the SFC (Show File) SCI command. If the ERROR ACCESS NAME is null, or if it is the same as the listing file, then errors will be displayed on the terminal by the SBS (Show Background Status) SCI command. If the device DUMY is specified, no error listing is produced.

The error file contains a complete list of any source records which caused assembly errors along with the errors. If a condition is sensed which prevents the assembler from continuing, a message is written to the error file as to what has occurred. Then the user must enter the SBS (Show Background Status) SCI command to view the error messages output by the assembler. Table 9-1 contains a list of these abnormal completion messages and possible causes.

Table 9-1. Abnormal Completion Messages

Message	Cause and Recovery
I/O Errors	
SOURCE FILE I/O ERROR, CODE = XXXX OBJECT FILE I/O ERROR, CODE = XXXX LIST FILE I/O ERROR, CODE = XXXX TEMP FILE I/O ERROR, CODE = XXXX	The codes are defined in the *DX10 Operating System Production Operations Guide*, Manual Number 945250-9702.
Assembler Bugs	
ATTEMPT TO POP EMPTY STACK – SDSMAC BUG DIRECTIVE EXPECTED – SDSMAC BUG UNEXPECTED END OF PARSE – SDSMAC BUG ERROR MAPPING PARSE – SDSMAC BUG INVALID OPERATION ENCOUNTERED – SDSMAC NO OP CODE – SDSMAC BUG INVALID LISTING ERROR ENCOUNTERED SYMBOL TABLE ERROR MACRO EXPANSION ERROR BUG – INVALID SDSLIB COMMAND ID UNKNOWN ERROR PASSED, CODE = XXXX END ACTION TAKEN BY MACRO ASSEMBLER	Call a Texas Instruments representative.

OPTIONS specifies any (or all) of the following options:

XREF– prints a cross reference listing at the end of the listing file.

SYMT– includes a symbol table with the output object code. This option must be specified to allow fully symbolic debugging.

TUNLIST– Text statement unlist.

BUNLST– Byte statement unlist.

DUNLST– Data statement unlist.

MUNLST– Macro expansion unlist.

FUNL– Overrides unlist directives.

TEXT, BYTE, and DATA statements and Macro usage often expand to produce multiple lines of code. If these options are selected, the statements appear in the listing but the expansion does not. For example, the source statement TEXT 'ABCDEF' produces the listing:

```
41   TEXT 'ABCDEF'
42
43
44
45
46
```

With the TUNLST option specified, only the line

```
41 TEXT 'ABCDEF'
```

is produced in the listing.

NOLIST– Suppresses all listing output, except to the error file.

Any of the Option Key words may be abbreviated; for example, any of the following may be used for the TUNLST option:

```
T
TU
TUN
TUNL
TUNLS
TUNLST
```

To select more than one option, enter a list of keywords separated by commas. The keywords may appear in any order. To select all the options one could enter the line:

```
OPTIONS: X,S,T,B,D,M
```

The options specified for this parameter are in addition to any options specified by "OPTION" directives in the source.

MACRO LIBRARY PATHNAME specifies a directory containing macro definitions for this assembly. This pathname specification is equivalent to specifying the same pathname in a LIBIN directive, except that this pathname becomes the system macro library and is retained through the

stacked assemblies. This pathname is printed on the cover sheet of the first module only. If this parameter is not specified, no macro library is used.

9.2.1 **COMPLETION MESSAGES.** A completion message is displayed on the terminal at the first available time after the macro assembler has terminated. Table 9-2 contains these messages.

Table 9-2. Completion Messages

Message	Possible Causes and Recovery
MEMORY REQUIRED EXCEEDS SYSTEM CAPACITY	a) Program is too large – break into several assembly, modules, take out some of the macros or use the LIBIN capability, decrease the number of symbol definitions. b) A macro containing an infinite loop or infinite recursion is being expanded – check all macros. c) The assembler itself is in a loop infinitely allocating memory – call a TI representative.
MACRO ASSEMBLY COMPLETE,XXXX ERRORS,YYYY WARNINGS	
ERROR FILE ERROR	The error access name specified when using the XMA command can not be accessed. Verify that the file can be created and is not currently open for another program. If a null input was entered for this parameter, then there is an SCI problem.
TCA ERROR	The assembly was unable to access the parameters specified in the XMA command. There is an SCI problem.
ABNORMAL COMPLETION	A condition was sensed which caused the assembler to abort. Display the error file to get more information and use table 9-1 to understand its contents.
UNABLE TO LOAD OVERLAY	Macro assembler has been denied access to its overlay file. Check that global luno S10 is assigned to a program file.

9.2.2 **OPERATING THE ASSEMBLER IN BATCH MODE.** Operating the Macro Assembler in batch mode requires two steps:

1. Prepare the batch command stream.
2. Execute Batch using the XB command.

The Batch command stream for macro assembly is pictured in figure 9-1.

```
.DATA  .MYFILE
IDT  XXXX
XXXX
XXXX
XXXX
END
.EOD
XMA  S=.MYFILE, L=LP01
Q
```

Figure 9-1. Macro Assembly Stream

Any sequential media (cards, cassette, magnetic tape, or sequential file) may be used for the batch stream.

The parameters for records in a Macro Assembly batch stream are the following:

1. .DATA record. This record has the form:

   .DATA <file name>

   The file name must be the name of the sequential file to which the input source is to be copied.

2. .EOD record. This record has the form:

   .EOD

   No parameters are required. This card signifies the end of data to be copied.

   NOTE

   If the source file already exists, or is to come from a source other than the batch stream then the sequence:

   .DATA
   <Source>
   .EOD

   should be omitted from the batch stream.

3. XMA record. This record, in addition to specifying macro assembly, also supplies the parameters required by the Macro Assembler. Parameters are supplied in the following format:

   <keyword or keyword abbreviation> = value.

   For example, to specify a source file .MYFILE, the following characters may be used:

   SOURCE = .MYFILE

Keywords may be abbreviated. Any unambiguous intial segment is acceptable. For example:

S = MYDISC.MYFILE

means the same thing as:

SOURCE = MYDISC.MYFILE

But O = MYDISC.MYFILEO is not acceptable since O could mean OBJECT ACCESS NAME or OPTIONS.

When a keyword takes a list as input, the list should be enclosed in parenthesis:

OPTIONS = (X,T,U)

Each keyword string must be separated from other keyword strings by a comma. For example, the following record assembles a source file named .SOURCE, producing an object file .OBJECT, a listing file .LIST, and reporting errors to .ERR; the options selected are cross reference (XREF) and symbol table (SYMT); no macro library is to be used:

XMA S = .SOURCE, OB = .OBJECT, L = .LIST, E = .ERR, OP = (X,S)

The only required parameters are SOURCE and LISTING. Other parameters may take defaults as indicated in the paragraph on background processing except that the batch listing file replaces the terminal local file as a default output file.

When a card reader is used, use Macro Assembly Stream as shown in figure 9-2.

To execute in batch mode enter the SCI command XB. XB requires two parameters.

INPUT ACCESS NAME: <sequential device or sequential file name>

LISTING ACCESS NAME: <file or device name>

The INPUT ACCESS NAME specifies the batch stream source. The LISTING ACCESS NAME specifies a listing file or device.

Batch mode operation of SCI is defined in detail in the *DX10 Operating System Production Operation Guide*, manual number 946250-9702.

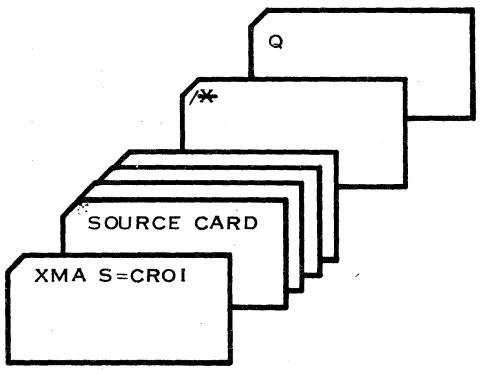


Figure 9-2. Macro Assembly Stream for Cards

When the macro assembler is executed in batch mode, the condition codes returned by the assembler may be checked. The synonym $$CC contains this condition code. The values returned are as follows:

0 -	no errors
4xxx-	assembly errors. The least significant three digits contain the error count.
C000 -	the assembly aborted.

For more information about condition codes, see *DX10 Operating System Release 3 Reference Manual, Volume V.*

SECTION X

ASSEMBLER OUTPUT

10.1 INTRODUCTION

All assemblers from Model 990 Computer and TMS 9900 Microprocessor print source listings. Optionally, the Cross Assembler and SDSMAC print a cross-reference listing and include a symbol table in the object file. Optionally, TXMIRA produces a sorted symbol table list with a facility for cross-reference. Cross assembler prints an object listing. All assemblers produce an object file. Optionally, the Cross Assembler prints an object file listing.

10.2 SOURCE LISTING

The source listings show the source statements and the resulting object code. The formats of the listings printed by all the assemblers are similar. A typical listing is shown with the example program in Appendix J.

SDSMAC produces a cover sheet as the first output in the listing. This cover page contains a table which provides a record of the files and devices used during the assembly process. An example of this output is as follows:

```
                    SDSMAC 3.2.0 78.274   11:26:51 MONDAY, OCT 17, 1977.

ACCESS NAMES TABLE                                              PAGE 0001

SOURCE ACCESS NAME =          .SUSAN.SRC.TEST1
OBJECT ACCESS NAME =
LISTING ACCESS NAME =         .SUSAN.LIST.TEST1
ERROR ACCESS NAME =
OPTIONS =                     XR, SY, TU, MU
MACRO LIBRARY PATHNAME =      .SDSMAC.MACRODEF


LINE     KEY          NAME

0001     LI           .SDSMAC.MACRODEF
                        =>. SDSMAC.MACRODEF
0001     L0           MACROS
                        =>.SDSMAC.MACRODEF
0002      A           DSC.SYSTEM.TABLES.DOR
                        =>DS01. SYSTEM.TABLES.DOR
0003     LI           .SDSMAC.MACRODEF
                        =>.SDSMAC.MACRODEF
```

The output has two sections:

- A listing of the parameters that were passed to the assembler via SCI.
- A list of access names encountered during the first pass of the assembly.

In the first section, any parameters which had no value are left blank. The fields in the second section are labeled as follows:

LINE - This field contains the record number in which the access name was encountered.

KEY - This field contains one of the following:

LI - indicating a LIBIN usage,

LO - indicating a LIBOUT usage,

one character - indicating a copy file to be given this character as a key.

NAME - This field contains two access names. The first name is an image of the name on the source record. The second name, appearing after the =>, is the result of synonym substitution on the first name.

Each page of the source listing has a title line at the top of the page. Any title supplied by a TITL directive is printed on this line, and a page number is printed to the right of the title area. The printer skips a line below the title line, and prints a line for each source statement listed. The line for each source statement contains a source statement number, a location counter value, object code assembled, and the source statement as entered. When a source statement results in more than one word of object code, the assembler prints the location counter value and object code on a separate line following the source statement for each additional word of object code. The source listing lines for a machine instruction source statement are shown in the following example:

```
0018    0156    C820    MOV     @INIT+3,@3
        0158    012B'
        015A    0003
```

The source statement number, 0018 in the example, is a four-digit decimal number. Source records are numbered in the order in which they are entered, whether they are listed or not. The TITL, LIST, UNL, and PAGE directives are not listed, and source records between a UNL directive and a LIST directive are not listed. The difference between source record numbers printed indicates how many source records are not listed.

The next field on a line of the listing contains the location counter value, a hexadecimal value. In the example, 0156 is the location counter value. No all directives affect the location counter, and those that do not affect the location counter leave this field blank. Specifically, of the directives that the assembler lists, the IDT, REF, DEF, DXOP, EQU, SREF, LOAD, and END directives leave the location counter field blank.

The third field normally contains a single blank. However, SDSMAC places a dash in this field when warning errors are detected.

The fourth field contains the hexadecimal representation of the object code placed in the location by the Assembler, C820 in the example. The apostrophe following the fourth field of the second line in the example indicates that the contents, 012B, is program-relocatable. A quote (") in this location would indicate that the location is data-relocatable, while a plus (+) would indicate that the label INIT is relocatable with respect to a common segment. All machine instructions and the BYTE, DATA, and TEXT directives use this field for object code. The EQU directive places the value corresponding to the label in the object code field.

In listings printed by PX9ASM, the fourth field may contain two or four hyphens (-) instead of hexadecimal digits. This occurs when a forward reference determines the values of these digits.

Later, when the forward reference is defined, the assembler prints an additional line in the listing following the statement that defines the forward reference. This line contains the location being resolved, two asterisks (**), and the contents. An error-free listing will include such a line for each location previously printed with hyphens as the contents. The listings printed by the other assemblers do not contain this type of information because all references are either resolved or identified as undefined before the listings are printed.

The fifth field contains the first 60 characters of source statement as supplied to the assembler. Spacing in this field is determined by the spacing in the source statement. The four fields of source statements will be aligned in the listing only when they are aligned in the same character positions in the source statements or when tab characters are used.

The machine instruction used in the example specifies the symbolic memory addressing mode for both operands. This causes the instruction to occupy three words of memory, and three lines of the listing. The object code corresponds to the operands in the order in which they appear in the source statement.

10.3 ERROR MESSAGES

The error codes and messages placed in the source listing by the various assemblers are described in the following paragraphs.

10.3.1 PX9ASM ERROR CODES. PX9ASM prints the following error message on the next lines of the listing when it detects an error:

```
**ERR N - STMT XXXX LAST ERR - STMT XXXX
```

N is an error code as shown in table 10-1. XXXX is a decimal statement number. The first statement number identifies the current error; the second statement number identifies the preceding error.

Error message for undefined symbols are printed at the end of the assembly. When a statement allows a forward reference, the reference is not undefined until PX9ASM recognizes an END statement without having recognized a statement defining the symbol. Error messages may be printed at any point, from the lines immediately following the statement in error to lines following the END statement.

The assembler can accommodate a minimum of 150 symbols in a 4K memory allocation. When the assembler is unable to continue because the area of memory available for symbols and forward references has been filled, the assembler prints the following message:

```
** ABORT **
```

The user may divide the program into two or more modules and assemble them separately. Considerations for properly linking these modules are described in Section VIII. Alternatively, the user may shorten the symbols in the program and reassemble. Since shorter symbols use less space in the symbol table, a symbol table of a given size may contain more shorter symbols.

Following the last statement or error message, the assembler prints undefined symbols, if there are any, one symbol per line. The undefined symbol may correspond to one of several error codes, or may be a symbol in a DEF directive that does not appear in the label field of a statement:

```
UNDEF     SYMBL
  LOC     XXXX
YYYYYY
```

Table 10-1. PX9ASM Error Codes

Code	Description
1	Undefined symbol. A symbol in the operand field of the statement corresponding to the error location does not appear in the label field of a source statement, or in the operand field of a REF directive.
2	Syntax error. The statement corresponding to the error location contains a syntax error.
3	Illegal external reference. The statement corresponding to the error location contains an external reference (and an arithmetic operator) in an expression or an external reference to be placed in a field smaller than 16 bits.
4	Truncation error. The statement corresponding to the error location contains a number that is too large or a character string that is too long. The number may be the result of evaluating an expression. Relocatability of a term or expression may be in error.
5	Multiply defined symbol. A symbol in the statement corresponding to the error location has been previously referenced or defined.
6	Unrecognizable operator. Contents of the operator field of the statement corresponding to the error location is not a mnemonic operation code, a directive, or a name defined as an extended operation.
7	Illegal forward reference. A symbol in the statement corresponding to the error location that should have been previously defined is not previously defined.
8	Illegal term. A term has an illegal value less than zero or greater than 15.

XXXX is the hexadecimal address of the location that referenced an undefined symbol; YYYYYY is the symbol name of an undefined symbol.

At the end of the listing is an error summary, as follows:

NNNN ERRORS

LAST ERROR - STMT XXXX

NNNN is the count of errors in the assembly. The second line identifies the last error detected in the assembly. The second lines of the error messages link the error messages so that the user may begin at the error summary message and readily locate all error messages. In an error-free assembly, the final message is:

0000 ERRORS ENCOUNTERED

10.3.2 **CROSS ASSEMBLER.** The Cross Assembler prints the following error message on successive lines of the listing when they detect errors:

NN *** error description ***

LAST ERROR ON STATEMENT XXXX

NN is the error code, and the error description is the brief description shown in table 10-2. The second line identifies the statement in which the previous error was detected.

At the end of the listing is an error summary, as follows:

NNNN ERRORS

LAST ERROR ON STATEMENT XXXX

NNNN is the count of errors in the assembly. The second line identifies the last error detected in the assembly. The second lines of the error messages link the error messages so that the user may begin at the error summary message and readily locate all error messages. In an error-free assembly, the final message is:

NO ERRORS IN THIS ASSEMBLY

10.3.3 **SDSMAC ERROR MESSAGES.** SDSMAC prints the following error message on successive lines of the listing when a error is detected:

*** error description

LAST ERROR ON STATEMENT XXXX

The error description is the brief description shown in table 10-3. The second line identifies the statement in which the previous error was detected.

At the end of the listing is an error summary, as follows:

NNNN ERRORS,LAST ERROR ON STATEMENT XXXX, YYYY WARNINGS

NNNN is the count of errors in the assembly. XXXX identifies the last error detected in the assembly. YYYY is the count of the warnings in the assembly. The second lines of the error messages link the error messages so that the user may begin at the error summary message and readily locate all error messages. In an error-free assembly, the final message is:

NO ERRORS, NO WARNINGS OR NO ERRORS, XXXX WARNINGS

10.3.4 **SDSMAC WARNING MESSAGES.** Several errors detected by SDSMAC (such as arithmetic overflow while evaluating expressions) are considered to be only warning errors. The programmer should examine the code generated when warning messages occur, since the results may or may not be the code expected. Warning error messages are wirtten only to the error file and not included in the listing. However, a dash is placed in column eleven of the listing where the warning error occurred. Warning messages do not include an indication of a previous warning or error.

The following program listing and error file demonstrate the output of error messages and warnings.

WARNING

The dash in column 11 is the only warning indication printed in the program listing. If the error file is assigned to a dummy device, corresponding warning messages will be lost.

```
0001                      IDT   'WARN'
0002             *
0003             *      THE NEXT LINE WILL GENERATE BOTH WARNING
0004             *           AND ERROR MESSAGES
0005             *
0006 0000-0004' MXLINK XVEC WP00,PC
     0002 7BC4
*********  UNDEFINED SYMBOL
0007 0000                 CSEG 'AB    '
0008 0000 1234            DATA >1234
0009 0002                 CEND
0010             *
0011 0004        WP00     BSS  32
0012             *
0013             *      THE NEXT LINE WILL GENERTE MULTIPLE WARNING MESSAGES
0014             *
0015 0024-0002   "GOGETT DATA >8001*2,>7000*7,>8001*8
     0026 1000
     0028 0008
0016             *
0017             *      THE NEXT LINE WILL GENERATE A SINGLE WARNING MESSAGE
0018             *
0019 002A-                DEND
0020             *
0021             *      THE NEXT LINE WILL GENERATE A SINGLE ERROR MESSAGE
0022             *
0023 002A 1000            JMP  @WP00
*********  DISPLACEMENT TOO BIG
LAST ERROR AT  0006
0024       0000'          END  MXLINK

THE FOLLOWING SYMBOLS ARE UNDEFINED

  PC
0002 ERRORS, LAST ERROR AT  0023,  0007 WARNINGS
0006 0000-0004' MXLINK XVEC WP00,PC
     0002 7BC4
*********  WORKSPACE ADDRESS NOT PREVIOUSLY DEFINED
*********  UNDEFINED SYMBOL
0015 0024-0002   "GOGETT DATA >8001*2,>7000*7,>8001*8
     0026 1000
     0028 0008
*********  VALUE TRUNCATED
*********  VALUE TRUNCATED
*********  VALUE TRUNCATED
*********  SYMBOL TRUNCATED
*********  INVALID CHARACTER IN SYMBOL- BLANK USED
0019 002A-                DEND
*********  'PEND' ASSUMED
0023 002A 1000            JMP  @WP00
*********  DISPLACEMENT TOO BIG
THE FOLLOWING SYMBOLS ARE UNDEFINED
  PC
0002 ERRORS, LAST ERROR AT  0023,  0007 WARNINGS
```

Table 10-2. Cross Assembler Error Messages

Error Messages	Explanation or Possible Cause
A MAXIMUM OF 64 COMMON BLOCKS ARE ALLOWED	
ASSEMBLER ERROR 01	Indicates an internal assembler error. Contact a TI representative.
ASSEMBLER ERROR 02	Same as above.
COMMON BLOCK NAME IS MULTIPLY DEFINED	The symbol appears as the name of more than one distinct common block.
CROSS REFERENCE TABLE FULL	
DIVIDING BY ZERO	
EXCESSIVE NUMBER OF OPERANDS	Too many operands were found.
INCORRECT PAIRING OF PARENTHESES	Verify that there is a ')' for every '('.
INCORRECT USE OF EXTERNAL REFERENCE	A REF'D or DEF'D symbol is not permitted in this context i.e., an expression cannot contain a REF'D symbol.
INCORRECT USE OF RELOCATABLE SYMBOLS	May indicate invalid use of a relocatable symbol in arithmetic.
INVALID IDT NAME	The symbol used on the 'IDT' card is not permitted as an 'IDT' name.
INVALID LABEL	The label may contain invalid characters or be too long.
INVALID OPCODE	The second field of the source record contained an entry that is not a defined instruction, directive, pseudo-op, or DXOP.
INVALID OPERAND IN COLUMN XX–	This message precedes the next seven errors. It identifies the card column containing the symbol in question.

Table 10-2. Cross Assembler Error Messages (Continued)

Error Messages	Explanation or Possible Cause
LABEL PREVIOUSLY DEFINED	The symbol appears more than once in the label field of the source.
LABEL TRUNCATED	The maximum length of a label is six characters.
MISSING OPERAND	On instructions having a fixed number of operands, too few appeared before encountering a blank. On instructions having a variable number of operands, a comma may have been encountered with no operand following it.
OPERATOR STACK OVERFLOW	The expression was too large in terms of operators.
REGISTER 0 IS INVALID INDEX REGISTER	
RELOCATABLE TABLE FULL	
SYMBOL TABLE FULL	There are too many symbols for this assembly. Break up the program if possible.
SYMBOL TRUNCATED	The maximum length for a symbol is six characters.
SYNTAX ERROR IN EXPRESSION	a) Unbalanced parentheses. b) Illegal symbols or operators. c) Invalid operations on relocatable symbols.
UNDEFINED SYMBOL	a) A symbol is used which did not appear in the label field of a source record. b) The use requires definition in the first pass and is undefined when the assembler first encounters it.
VALUE TRUNCATED	The result of expression evaluation was too large.

Table 10-3. SDSMAC Listing Errors

Error Message	Possible Causes
$MACRO invalid within macro definition.	a) The $END verb belonging to the previous macro was missing. b) A $MACRO verb was unintentionally included.
Absolute value required.	
Blank missing.	
'CEND' assumed.	A warning (Note 1).
Close (')') missing.	
Comma missing.	
Conditional assembly nesting error.	An if-then-else construct is in error. Conditions which could cause this are: a) Missing ASMEND'S b) Surplus ASMELS'S c) Surplus ASMEND'S
'DEND' assumed.	A warning.
Directory open error.	Check that any synonyms are valid and that no other processor is currently writing to the MACRO Library.
Directory read error.	An I/O error was encountered while trying to read a MACRO Library Directory. Verify that no other processor is currently writing to that MACRO Library.
Directory required.	The access name specified is not an existing directory. Verify that all synonyms are correct and that the MACRO Library does indeed exist; it can not be auto-created.
Directory write error.	Verify that no other processor is currently writing to that MACRO Library.
Displacement too big.	An instruction requiring an operand with a fixed upper limit was encountered which overflowed this limit. An example is the 'JMP' instruction, whose single operand must evaluate to within >7F words distance from the current program counter.
'DSEG' assumed.	This is a warning that the following two statements have the same result: CSEG '$DATA' DSEG

Table 10-3. SDSMAC Listing Errors (Continued)

Error Message	Possible Cause
Duplicate definition.	a) The symbol appears more than once in the label field of the source. b) The symbol appears as an operand of a REF statement as well as in the label field of the source. c) An attempt was made to define a macro variable or macro language label which was previously defined in the macro.
Error expanding call.	The symbol in the operand field of the $CALL statement is not a defined Macro.
Error on copy open.	The access name specified as the operand of copy directive can not be opened. Check that the synonyms are correct and that the file is not currently being written to by another processor.
Expression syntax error.	a) Unbalanced parentheses. b) Invalid operations on relocatable symbols.
Indirect (*) missing.	
Invalid $ASG variable.	a) An attempt was made to change the length component of a variable. b) An attempt was made to change the attribute component or the value component of a macro variable which was declared as a macro language variable.
Invalid character in symbol – blank used.	A warning (Note 1). The legal characters to be used in symbols under SDSMAC are A-Z, 0-9, ';', and '$'.
Invalid CRU or shift value.	A warning (Note 1).
Invalid directive in absolute code.	The directives PEND, DEND, CEND have no meaning in absolute code.
Invalid expression.	May indicate invalid use of a relocatable symbol in arithmetic.
Invalid macro variable.	The target variable specified on a $ASG or $GOTO verb is not a valid target variable.
Invalid model statement.	A macro symbol in a model statement must be followed with either a colon operator (:) or end-of-record.
Invalid opcode.	The second field of the source record contained an entry that is not a defined instruction, directive, pseudo-op, DXOP, DFOP, or Macro name.

Table 10-3. SDSMAC Listing Errors (Continued)

Error Message	Possible Cause
Invalid option.	A warning (Note 1). The only legal options are: XREF SYMT NOLIST MUNLST TUNLST BUNLST DUNLST FUNL (or suitable abbreviation).
Invalid relocation type.	Only PSEG relocatable or absolute symbols are allowed as the operand of an 'END' statement.
Invalid use of conditional assembly.	A conditional assembly directive may not appear as a model statement.
Invalid $ASG expression.	The expression is not present.
Invalid $ASG variable.	The target variable is not present or is not a symbol.
Invalid $IF expression.	The expression either is not present or does not evaluate to an integer value.
Label required.	$NAME statements must begin with a label of maximum length 2. $MACRO statements must begin with a label of maximum length 6.
Macro definition discarded due to errors.	An error was detected during the assembly of the macro definition. Use of the macro name in succeeding lines will cause error messages.
Macro expansion error.	Indicates an internal assembler error. Contact a TI representative.
MACRO Library read error.	A 'LIBIN' was in effect and the statement was a Macro in a specified MACRO Library, but an I/O error was encountered when reading it.
MACRO Library write error.	The current 'LIBOUT' Library could not be used at completion of a Macro definition. Check that the Macro is not currently being written by another processor.
Macro symbol truncated.	A warning (Note 1). The maximum length for a macro symbol is two characters. The following are legal macro symbols: A, A.S, B2.SV. The following are illegal macro symbols: CNT, CNT.A, PM2.SL.

Table 10-3. SDSMAC Listing Errors (Continued)

Error Message	Possible Cause
Max macro nesting stack depth overflow.	a) A macro calls itself recursively more than the allowed maximum number of times. b) More levels of macro calling have been used than the allowed maximum.
Memory exceeded.	The program counter overflowed the value >FFFF.
Model statement truncated.	A warning (Note 1). When expanded, the model statement exceeded 80 characters in length.

Operand conflict PASS1/PASS2.	The assembler defaults currently undefined symbols to register uses in the first pass if that symbol is used in an ambiguous way. If during the second pass it is discovered that the symbol was not a register use, this error will result. An example is:
	`BL SUB`
	`:`
	`SUB EQU $`
	If this has been coded as follows, no ambiguity would have existed due to the explicit "@" sign:
	`BL @SUB`
	`:`
	`SUB EQU $`

Operand missing.	On instructions having a fixed number of operands, too few appeared before encountering a blank. On instructions having a variable number of operands, such as 'DATA', a comma may have been encountered with no operand following it. An expression extending beyond the 60th column could cause this problem.
'PEND' assumed.	A warning.
REF'D symbol in expression.	Due to the object code format of the 990 computer, REF'D symbols may not appear within an expression.
Register required.	
String required.	
String truncated.	A warning (Note 1). Check the syntax for the directive in question to determine the maximum length for the string.

Table 10-3. SDSMAC Listing Errors (Continued)

Error Message	Possible Cause
Symbol truncated.	A warning (Note 1). The maximum length for a symbol is six characters.
Symbol required.	
Symbol used in both REF and DEF.	This is a conflicting, duplicate definition.
Syntax error.	
'TO' missing.	'TO' is a required part of the syntax for the $ASG Macro verb.
Undefined macro variable.	The target variable specified on a $ASG or $GOTO verb is undefined.
Undefined symbol.	a) A symbol is used which did not appear in the label field of a source record. b) The use requires definition in the first pass and is undefined when the assembler first encounters it.
Valid op code required.	The defining symbol (i.e., the second operand) is not a valid instruction or directive.
Value truncated.	A warning (Note 1). Overflow is checked after every operation in an arithmetic expression. This may result in several truncations in one expression.
Workspace address not previously defined.	The operand field must have been previously defined. Note that the WPNT directive (or implied WPNT) is ignored. Any previous WPNT is also ignored from this point on.
$IF – $ELSE – $ENDIF construct in error.	Possible errors are: a) Surplus $ELSEs b) Surplus $ENDIFs c) Missing $ENDIFs.
$MACRO invalid within Macro definition.	

NOTE 1

Warnings are defined by a dash (–) in column 11 of the assembled program listing.

10.3.5 TXMIRA ERROR MESSAGES. The TXMIRA assembler processes fatal errors and non-fatal errors. The fatal errors cause the run to abort with the appropriate error message printed on the LOG. The error messages are shown in table 10-4.

The nonfatal errors are shown in table 10-5 and do not cause the run to abort. An error message is printed following the statement containing the error. The format of the printout is as follows:

*****SYNTAX ERROR - RCD nnnn

where nnnn is the source record number

If there are any undefined symbols in an assembly, the undefined symbols are listed at the end of the run under the following heading:

THE FOLLOWING SYMBOLS ARE UNDEFINED:

NOTE: If listing is not selected, than all error messages go to default listing device.

Table 10-4. TXMIRA Fatal Errors

SYMBOL TABLE OVERFLOW
CANT GET COMMON
CANT GET MEMORY
nn* - ILLEGAL PATHNAME
nn* - I/O ERROR - [L] LISTING
[O] OBJECT
[S] SOURCE
NO END CARD FOUND

*nn is a system returned error code.

Table 10-5. TXMIRA Nonfatal Errors

***** SYNTAX ERROR - RCD nnnn†
***** ILLEGAL EXTERNAL REF. RCD nnnn†
***** VALUE TRUNCATION - RCD nnnn†
***** MULTIPLY DEFINED SYM. - RCD nnnn†
***** INVALID OPERATOR - RCD nnnn†
***** ILLEGAL FORWARD REF. - RCD nnnn†
***** ILLEGAL TERM - RCD nnnn†
***** ILLEGAL REGISTER - RCD nnnn†
***** SYMBOL TRUNCATION - RCD nnnn†
***** UNDEFINED SYMBOL - RCD nnnn†
***** COMMON TABLE OVERFLOW - RCD nnnn†
***** PEND ASSUMED - RCD nnnn†
***** PEND ASSUMED - RCD nnnn†
***** CEND ASSUMED - RCD nnnn†

†nnnn is the source record in which the error was detected.

10.4 CROSS REFERENCE LISTING

The Cross Assembler and SDSMAC each print an optional cross reference listing following the source listing. The format of the listing is shown in figure 10-1. In the left column, the assembler prints each symbol defined or referenced in the assembly. In the second column, the attributes of the symbol are indicated as a list of single characters. The characters that appear in the second column, and their meanings, are listed in table 10-6. The third column contains a four-digit hexadecimal number, the value assigned to the symbol. The number of the statement that defines the symbol appears in the fourth column, unless the symbol is undefined. For undefined symbols, the fourth column contains UNDF. The right column contains a list of the numbers of statements that reference the symbol, or the words NOT REFERENCED, as applicable. For SDSMAC these fields are left blank if the symbol is undefined or never used.

```
                        CROSS REFERENCE

LABEL           VALUE     DEFN     REFERENCES
ADDT            01A8'     325      314
ADSR            01A0'     316      342     343     348     349
GT              0006      997      NOT REFERENCED
```

Figure 10-1. Cross Reference Listing Format

Table 10-6. Symbol Attributes

Character	Meaning	Applicability: Cross Assembler	Applicability: SDSMAC	Applicability: TXMIRA
A	Absolute			X
R	External Reference (REF)	X	X	X
D	External Definition (DEF)	X	X	X
X	Extended Operation (XOP)	X	X	X
U	Undefined	X	X	X
O	Defined Operation (DFOP)		X	
M	Macro name		X	
S	Secondary Reference (SREF)	X	X	
L	Force Load (LOAD)	X	X	

10.5 OBJECT CODE

The assemblers produce object code that may be linked to other object code modules or programs and loaded into the Model 990 computer, or may be loaded into the computer directly. References to the "loader" apply to a link editor, linking loader, or loader depending on the assembler being used. Object code consists of records containing up to 71 ASCII characters each. The format, described in the next paragraph, permits correction using a keyboard device. Re-assembly to correct errors is unnecessary. An example of output code is shown in figure 10-2.

10.5.1 OBJECT CODE FORMAT. The object record consists of a number of tag characters, each followed by one to three fields as defined in table 10-7. The first character of a record is the first tag character, which tells the loader which field or fields follows the tag. The next tag character follows the end of the field or fields associated with the preceding tag character. When the assembler has no more data for the record, the assembler writes the tag character 7 followed by the check sum field, and the tag character F, which requires no fields. The assembler then fills the rest of the record with blanks and a sequence number, and begins a new record with the appropriate tag character.

```
00000SAMPROG 90040C0000A0020BC06DB00029004 2C0020A0024BC81BC002A7F219F
A0028B0241B0000BCB41B0002B0380A00CAC0052C00A2B02E0C0032B0200B0F0F7F1DEF
A00D6BC0A0C00CAB04C3BC160C00CCBC1A0C00D0BC072B0281B3A00A00ECB02217F151F
A00EEB0900B06C1A00EAB1102A00F2B0543B11F8B2C20C0032BC101B0B44BE0447F18EF
A0100BDD66B0003B0282C00A2B11EDB03407F832F
200CE0010C          7FCABF
:
(A)132255
```

Figure 10-2. Object Code Example

Tag character 0 is followed by two fields. Field 1 contains the number of bytes of program-relocatable code, and field 2 contains the program identifier assigned to the program by an IDT 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 program-relocatable code to determine the load bias for the next module or program. SDSMAC, TXMIRA, and the Cross Assembler place a single tag character 0 at the beginning of each program. PX9ASM is unable to determine the value for Field 1 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.

The tag character M, used only when data or common segments are defined in the program (SDSMAC or TXMIRA), is followed by three fields. Field 1 contains the length, in bytes, of data- or common-relocatable code, Field 2 contains the data or common segment identifier, and Field 3 contains a "common number". The identifier is a six-character field containing the name $DATA$\hat{b}$ for data segments and $BLANK for blank common segments. If a named common segment appears in the program, an M tag will appear in the object code with an identifier field corresponding to the operand in the defining CSEG directive(s). Field 3 of the M tag consists of a four-character hexadecimal number defining a unique common number to be used by other tags which reference or initialize data of that particular segment. For data segments, this common number is always zero. For common segments (including blank common), the common numbers are assigned in increasing order beginning at 1 and ending with the number of different common segments. The maximum number of common segments that a program may contain is 125.

Tag characters 1 and 2 are used with entry addresses. Tag character 1 is used when the entry address is absolute. Tag character 2 is used when the entry address is relocatable. Field 1 contains the entry address in hexadecimal. One of these tags may appear at the end of the object code file. The associated field is used by the loader to determine the entry point at which execution starts when the loading is complete.

Tag characters 3, 4 and X are used for external references. Tag character 3 is used when the last appearance of the symbol in Field 2 of the tag is in program-relocatable code. Tag character 4 is used when the last appearance of the symbol is in absolute code. The X tag is used when the last

appearance of the symbol in Field 2 is in data- or common-relocatable code (SDSMAC or TXMIRA). Field 3 of the X tag gives the common numbers. Field 1 of the tag characters contains the location of the last appearance of the symbol. The symbol in Field 2 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 Field 1 of the tag character contains absolute zero, no location in the program requires the address that corresponds to the reference. When Field 1 of the tag character contains a location, the address corresponding to the reference is placed by the loader in the location specified and the location's previous value is used to point to the next location or, if it contains absolute zero, to discontinue linking.

Table 10-7. 990 Object Tags

Tag	Field 1	Field 2	Field 3	Note
0	PSEG Length	Program ID (8)		1
1	Absolute Address	–	–	2
2	P-R Address	–	–	2
3	P-R Address of Chain	Symbol (6)	–	6
4	Absolute Address of Chain	Symbol (6)	–	6
5	P-R Address	Symbol (6)	–	5
6	Absolute Value	Symbol (6)		5
7	Value	–	–	*
8	Any Value	–	–	**
9	Absolute Address	–	–	3
A	P-R Address	–	–	3
B	Absolute Value	–	–	4
C	P-R Address	–	–	4
D	Absolute Address	–	–	***
F	Unused	–	–	****
G	P-R Address	Symbol (6)	–	7
H	Absolute Value	Symbol (6)	–	7
I	P-R Address	Program ID (8)		
J	D-R/C-R Address	Symbol (6)	Common #	7
M	DSEG Length	$ Data	0000	1
M	Blank Common Length	$ Blank	0001	1
M	CSEG Length	Common Name (6)	Common #	1
N	C-R Address			4
P	C-R Address	Common Name #	–	3
S	D-R Address	–	–	3
T	D-R Address	–	–	4
U	0000	Symbol (6)	–	8
V	P-R Address of Chain	Symbol (6)		9

Table 10-7. 990 Object Tags (Continued)

Tag	Field 1	Field 2	Field 3	Note
X	D-R/C-R Address of Chain			6
W	D-R/C-R Address	Symbol (6)	Common #	5

Note:

1	Module Definition
2	Entry Point Definition
3	Load Address
4	Data
5	External Definitions
6	External References
7	Symbol Definitions
8	Force External Link
9	Secondary External Reference
*	Checksum
**	Ignore Checksum
***	Load Bias
****	End of Record

Note: PX9ASM or TXMIRA supports only tags 0 through F.

Figure 10-3 illustrates the chain of the external reference EXTR. The object code contains the following tag and fields:

4C00EEXTR

At location C00E, the address C00A points to the preceding appearance of the reference. The chain includes both absolute and relocatable addresses and consists of absolute addresses C00E, C00A, C006, and C002, relocatable addresses 029E, 029A, and 0298, absolute addresses B00E, B00A, B006, and B002, and relocatable addresses 0290 and 028E. Each location points to the preceding appearance, except for location 028E, which contains zero. The zero identifies location 028E as the first appearance of EXTR, the end of the chain.

```
0229                        *
0230                        *       DEMONSTRATE EXTERNAL REFERENCE LINKING
0231                        *
0232                              REF  EXTR
0233  028C                        RORG
0234  028C  C820                 MOV  @EXTR,@EXTR
      028E  0000
      0290  028E'
0235  0292  28E0                 XOR  @EXTR,3
      0294  0290'
0236  B000                        AORG >B000
0237  B000  3220                 LDCR @EXTR,8
      B002  0294'
0238  B004  0420                 BLWP @EXTR
      B006  B002
0239  B008  0223                 AI   3,EXTR
      B00A  B006
0240  B00C  38A0                 MPY  @EXTR,2
      B00E  B00A
0241  0296                        RORG
0242  0296  C820                 MOV  @EXTR,@EXTR
      0298  B00E
      029A  0298'
0243  029C  28E0                 XOR  @EXTR,3
      029E  029A'
0244  C000                        AORG >C000
0245  C000  3220                 LDCR @EXTR,8
      C002  029E'
0246  C004  0420                 BLWP @EXTR
      C006  C002
0247  C008  0223                 AI   3,EXTR
      C00A  C006
0248  C00C  38A0                 MPY  @EXTR,2
      C00E  C00A
(A)132256
```

Figure 10-3. External Reference Example

Tag characters 5, 6, and W are used for external definitions. Tag character 5 is used when the location is program-relocatable. Tag character 6 is used when the location is absolute. Tag character W is used when the location is data- or common-relocatable (SDSMAC or TXMIRA). The fields are used by the loader to provide the desired linking to the external definition. Field 2 contains the symbol of the external definition. Field 3 of tag character W contains the common number.

Tag character 7 precedes the checksum, which is an error detection word. The checksum is formed as the record is being written. It is the two'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.

Tag characers 9, A, S, and P 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 program-relocatable. Tag character S is used when the load address is data-relocatable; and tag character P is used when the load address is common-relocatable (SDSMAC or TXMIRA). Field 1 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. Field 2 of tag character P contains the common number.

Tag characters B, C, T, and N are used with data words. Tag character B is used when the data is absolute, e.g., an instruction word or a word that contains text characters or absolute constants. Tag character C is used for a word that contains a program-relocatable address. Tag character T is used for a word that contains a data-relocatable address and tag character N is used for a word that contains a common-relocatable address (SDSMAC or TXMIRA). Field 1 contains the data word. The loader places the data word in the memory location specified in the preceding load address field or in the memory location that follows the preceding data word. Field 2 of tag character N contains the common number.

Tag characters G, H, and J are used when the symbol table option is specified with SDSMAC or the Cross Assembler. Tag character G is used when the location or value of the symbol is program-relocatable, and tag character H is used when the location or value of the symbol is absolute. Tag character J is used when the location or value of the symbol is data- or common-relocatable (SDSMAC and TXMIRA). Field 1 contains the location or value of the symbol, and Field 2 contains the symbol to which the location is assigned. Field 3 of tag character J contains the common number.

Tag character U is generated by the LOAD directive. The symbol specified is treated as if it were the value specified in an INCLUDE command to the loader. Field 1 contains zeros. Field 2 contains the symbol for which the loader will search for a definition. Refer to the LOAD directive for further information.

Tag character V specifies a program-relocatable address for a secondary external reference. Field 1 contains the location of the last appearance of the symbol. Field 2 contains the symbol.

Tag character 8 is used to ignore the checksum. Field 1 contains the checksum to be ignored.

Tag character D is used to specify a load bias. Field 1 contains the absolute address which will be used by the loader to relocate the symbols when loaded. The link editor does not accept the D tag. Tag character D is described in detail in a subsequent paragraph.

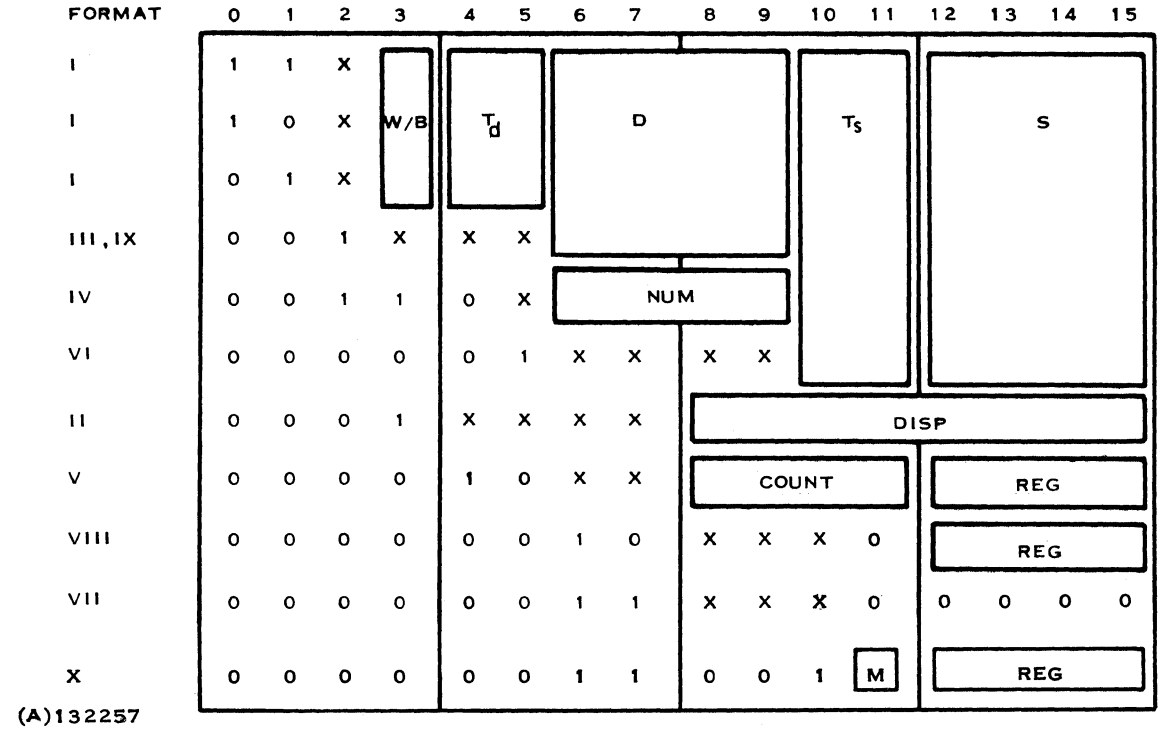
Tag character F indicates the end of record. It may be followed by blanks.

The last record of an object module has a colon (:) in the first character position of the record, followed by blanks or a time and data identifying stamp.

10.5.2 MACHINE LANGUAGE FORMAT. Some of the data words preceded by tag character B represent machine instructions. Comparing the source listing with the object code fields identifies the data words that represent machine instructions. Figure 10-4 shows the manner in which the bits of the machine instructions relate to the operands in the source statements for each format of machine instructions.

10.5.3 SYMBOL TABLE. When the SYMT option is specified (SDSMAC and Cross Assembler only), the symbol table is included in the object code file. One entry, using tag character G or H as appropriate, is supplied for each symbol defined in the assembly.

10.5.4 OBJECT CODE LISTING. When the OBJ option is specified (Cross Assembler), the assembler prints the object code following the source code listing. When the cross reference listing is also specified, the object listing follows the cross reference listing. The object code shown in figure 10-2 is shown in the object code listing format in figure 10-5. Notice that blanks have been inserted for clarity, and a sequence number included at the right.



X — IS A BIT OF THE OPERATION CODE THAT IS EITHER 0 OR 1 ACCORDING TO THE SPECIFIC INSTRUCTION IN THE FORMAT

W/B — IS A BIT OF THE OPERATION CODE THAT IS 0 IN INSTRUCTIONS THAT OPERATE ON WORDS, AND 1 IN INSTRUCTIONS THAT OPERATE ON BYTES

T_D — IS A PAIR OF BITS THAT SPECIFY THE ADDRESSING MODE OF THE DESTINATION OPERAND, AS FOLLOWS

- 00 = WORKSPACE REGISTER ADDRESSING
- 01 = WORKSPACE REGISTER INDIRECT ADDRESSING
- 10 = SYMBOLIC MEMORY ADDRESSING WHEN D = 0
- 10 = INDEXED MEMORY ADDRESSING WHEN D ≠ 0
- 11 = WORKSPACE REGISTER INDIRECT AUTOINCREMENT ADDRESSING

D — IS THE WORKSPACE REGISTER FOR THE DESTINATION OPERAND

T_S — IS A PAIR OF BITS THAT SPECIFY THE ADDRESSING MODE OF THE SOURCE OPERAND AS SHOWN FOR T_D

S — IS THE WORKSPACE REGISTER FOR THE SOURCE OPERAND

NUM — IS THE NUMBER OF BITS TO BE TRANSFERRED

DISP — IS A TWO S COMPLEMENT NUMBER THAT REPRESENTS A DISPLACEMENT

REG — IS A WORKSPACE REGISTER ADDRESS

COUNT — IS A SHIFT COUNT

M — IS A MAP REGISTER FILE NUMBER (0 OR 1)

Figure 10-4. Machine Instruction Formats

```
                                OBJECT FILE LISTING
0 0000SAMPROG   9 0040  C 0000  A 0020  B C06D  B 0002  9 0042  C 0020  A 0024  B C81B  C 002A  7 F219  F          0001
A 0028  B 0241  B 0000  B CB41  B 0002  B 0380  A 00CA  C 0052  C 00A2  B 02E0  C 0032  B 0200  B 0F0F  7 F1DE  F 0002
A 00D6  B C0A0  C 00CA  B 04C3  B C160  C 00CC  B C1A0  C 00D0  B C072  B 0281  B 3A00  A 00EC  B 0221  7 F151  F 0003
A 00EE  B 0900  B 06C1  A 00EA  B 1102  A 00F2  B 0453  B 11F8  B 2C20  C 0032  B C101  B 0B44  B E044  7 F18E  F 0004
A 0100  B DD66  B 0003  B 0282  C 00A2  B 11ED  B 0340  7 F832  F                                                0005
2 00CE  0 010C        7 FCAB  F                                                                                  0006
```

Figure 10-5. Object Code Listing Format

10.5.5 PROCEDURES FOR CHANGING OBJECT CODE. To correct the object code without reassembling a program, change the object code by changing or adding one or more records. One additional tag character is recognized by the loader to permit specifying a load point. The additional tag character, D, may be used in object records changed or added manually.

Tag character D is followed by a load bias (offset) value. The loader uses this value instead of the load bias computed by the loader itself. The loader adds the load bias to all relocatable entry addresses, external references, external definitions, load addresses, and data. The effect of the D tag character is to specify the area of memory into which the loader loads the program. The tag character D and the associated field must be placed ahead of the object code generated by the assembler.

Correction of the object code may require only changing a character or a word in an object code record. The user may duplicate the record up to the character or word in error, replace the incorrect data with the correct data, and duplicate the remainder of the record up to the 7 tag character. Because the changes the user has made will cause a checksum error when the checksum is verified as the record is loaded, the user must change the 7 tag character to 8.

When more extensive changes are required, the user may write an additional object code record or records. Begin each record with a tag character 9, A, S, or P, followed by an absolute load address or a relocatable load address. This may be an address into which an existing object code record places a different value. The new value on the new record will override the other value when the new record follows the other record in the loading sequence. Follow the load address with a tag character B, C, T_S, or N and an absolute data word or a relocatable data word. Additional data words preceded by appropriate tag characters may follow. When additional data is to be placed at a non-sequential address, write another load address tag character followed by the load address and data words preceded by tag characters. When the record is full, or all changes have been written, write tag character F to end the record.

When additional memory locations are loaded as a result of changes, the user must change Field 1 of tag character 0 which contains the number of bytes of relocatable code. For example, when the object file written by the assembler contained 1000_{16} bytes of relocatable code, and the user has added 8 bytes in a new object record, additional memory locations will be loaded. The user must find the 0 tag character in the object code file and change the value following the tag character from 1000 to 1008; he must also change the 7 tag character to 8 in that record.

When added records place corrected data in locations previously loaded, the added records must follow the incorrect records. The loader processes the records as they are read from the object medium, and the last record that affects a given memory location determines the contents of that location at execution time.

The object code records that contain the external definition fields, the external reference fields, the entry address field, and the final program start field must follow all other object records. An additional field or record may be added to include reference to a program identifier. The tag character is 4, and the hexadecimal field contains zeros. The second field contains the first six characters of the IDT character string. External definitions may be added using tag character 5 or 6 followed by the relocatable or absolute address, respectively. The second field contains the defined symbol, filled to the right with blanks when the symbol contains less than six characters.

APPENDIX A

CHARACTER SET

APPENDIX A

CHARACTER SET

All of the 990 assemblers recognize the ASCII characters listed in table A-1. The table includes both the ASCII code for each character, represented as a hexadecimal value and as a decimal value, and the corresponding Hollerith code. The assemblers also recognize the five special characters shown in table A-2. The Macro Assembler, SDSMAC, will accept the characters shown in table A-3 if they occur within quoted strings or in comment fields.

The device service routine for the card reader accepts (and stores in the calling program's buffer) all the characters shown in tables A-1, A-2, and A-3, as well as the special characters shown in table A-4. Although not accepted by the 990 assemblers, other programs may recognize the characters shown in table A-4 and perform appropriate action.

Table A-1. Character Set

Hexadecimal Value	Decimal Value	Character	Hollerith Code
20	32	Space	Blank
21	33	!	12-8-7
22	34	"	8-7
23	35	#	8-3
24	36	$	11-8-3
25	37	%	0-8-4
26	38	&	12
27	39	'	8-5
28	40	(	12-8-5
29	41	)	11-8-5
2A	42	*	11-8-4
2B	43	+	12-8-6
2C	44	,	0-8-3
2D	45	-	11
2E	46	.	12-8-3
2F	47	/	0-1
30	48	0	0
31	49	1	1
32	50	2	2
33	51	3	3
34	52	4	4
35	53	5	5
36	54	6	6
37	55	7	7
38	56	8	8
39	57	9	9
3A	58	:	8-2
3B	59	;	11-8-6
3C	60	<	12-8-4

Table A-1. Character Set (Continued)

Hexadecimal Value	Decimal Value	Character	Hollerith Code
3D	61	=	8-6
3E	62	>	0-8-6
3F	63	?	0-8-7
40	64	@	8-4
41	65	A	12-1
42	66	B	12-2
43	67	C	12-3
44	68	D	12-4
45	69	E	12-5
46	70	F	12-6
47	71	G	12-7
48	72	H	12-8
49	73	I	12-9
4A	74	J	11-1
4B	75	K	11-2
4C	76	L	11-3
4D	77	M	11-4
4E	78	N	11-5
4F	79	O	11-6
50	80	P	11-7
51	81	Q	11-8
52	82	R	11-9
53	83	S	0-2
54	84	T	0-3
55	85	U	0-4
56	86	V	0-5
57	87	W	0-6
58	88	X	0-7
59	89	Y	0-8
5A	90	Z	0-9

Table A-2. Additional Special Characters

Hexadecimal Value	Decimal Value	Character	Hollerith Code	Model 29 Keypunch Character
5B	91	[	12-2-8	¢
5C	92	\	0-2-8	0-2-8
5D	93	]	11-2-8	!
5E	94	^	11-7-8	¬(logical NOT)
5F	95	–	0-5-8	– (underscore)

Table A-3. Additional Special Characters Recognized by SDSMAC

Hexadecimal Value	Decimal Value	Character	Hollerith Code
60	96	\	8-1
61	97	a	12-0-1
62	98	b	12-0-2
63	99	c	12-0-3
64	100	d	12-0-4
65	101	e	12-0-5
66	102	f	12-0-6
67	103	g	12-0-7
68	104	h	12-0-8
69	105	i	12-0-9
6A	106	j	12-11-1
6B	107	k	12-11-2
6C	108	l	12-11-3
6D	109	m	12-11-4
6E	110	n	12-11-5
6F	111	o	12-11-6
70	112	p	12-11-7
71	113	q	12-11-8
72	114	r	12-11-9
73	115	s	11-0-2
74	116	t	11-0-3
75	117	u	11-0-4
76	118	v	11-0-5
77	119	w	11-0-6
78	120	x	11-0-7
79	121	y	11-0-8
7A	122	z	11-0-9
7B	123	{	12-0
7C	124	|	12-11
7D	125	}	11-0
7E	126	~	11-0-1

Table A-4. Additional Characters Recognized by the Card Reader Device Service Routine

Hexadecimal Value	Decimal Value	Character	Hollerith Code
00	0	NUL	12-0-9-8-1
01	1	SOH	12-9-1
02	2	STX	12-9-2
03	3	ETX	12-9-3
04	4	EOT	9-7
05	5	ENQ	0-9-8-5
06	6	ACK	0-9-8-6
07	7	BEL	0-9-8-7
08	8	BS	11-9-6
09	9	HT	12-9-5
0A	10	LF	0-9-5

Table A-4. Additional Characters Recognized by the Card Reader Device Service Routine (Continued)

Hexadecimal Value	Decimal Value	Character	Hollerith Code
0B	11	VT	12-9-8-3
0C	12	FF	12-9-8-4
0D	13	CR	12-9-8-5
0E	14	SO	12-9-8-6
0F	15	SI	12-9-8-7
10	16	DLE	12-11-9-8-1
11	17	DC1	11-9-1
12	18	DC2	11-9-2
13	19	DC3	11-9-3
14	20	DC4	9-8-4
15	21	NAK	9-8-5
16	22	SYN	9-2
17	23	ETB	0-9-6
18	24	CAN	11-9-8
19	25	EM	11-9-8-1
1A	26	SUB	9-8-7
1B	27	ESC	0-9-7
1C	28	FS	11-9-8-4
1D	29	GS	11-9-8-5
1E	30	RS	11-9-8-6
1F	31	US	11-9-8-7
7F	127	DEL	12-9-7

APPENDIX B

INSTRUCTION TABLES

APPENDIX B

INSTRUCTION TABLES

The source formats for the machine instructions are summarized in eight tables. Refer to Section III for descriptions of the machine instructions. Arithmetic instructions are listed in table B-1, and branch instructions are listed in table B-2. Table B-3 lists compare instructions and table B-4 lists control and CRU instructions. Load and move instructions are listed in table B-5, and logical instructions are listed in table B-6. Workspace register shift instructions are listed in table B-7, and the extended operation instruction is listed in table B-8. Long distance addressing instructions are listed in table B-9.

The pseudo-instructions are listed in table B-10.

The following symbols are used in tables B-1 through B-10.

G, G1, G2 - A general address in one of the five modes described in Section III.

R - A workspace register address.

S - A symbolic memory address (a label or an expression that contains a label or $)

E - An expression, with the additional limitation that the expression must not contain a symbol that is not previously defined.

I - An immediate value, which is an expression.

T - A term.

M - Memory map file, 0 or 1.

(,) - The contents of the address within parentheses.

→ - "replaces"

: - "is compared to"

The following example shows the use of the symbols in the source format column:

```
XOR   G,R
```

The source format entry means that the mnemonic operation code XOR requires a general address and a workspace register address separated by a comma. In the effect column, the symbols are used as in the following example:

(G) XOR (R) → (R)

This means that the result of an exclusive OR of the contents of the general address with the contents of the workspace register replaces the contents of the workspace register. In the status bits test column, the symbols are used as in the following example:

(R) : 0

This means that the result placed in the workspace register is compared to zero and the status bits contain the result of this comparison.

Table B-1. Arithmetic Instructions

Instruction	Format	Effect	Opcode	Status Bits Affected	Status Bits Test	Format Number
Add words	A G1, G2	(G1)+(G2)→(G2)	A000	0 - 4	(G2) :0	I
Add bytes	AB G1, G2	(G1)+(G2)→(G2)	B000	0 - 5	(G2) :0	I
Absolute value	ABS G	Absolute (G)→(G)	0740	0 - 2, 4	Note 1	VI
Add immediate	AI R, I	(R)+I→(R)	0220	0 - 4	(R) :0	VIII
Decrement	DEC G	(G)-1→(G)	0600	0 - 4	(G) :0	VI
Decrement by 2	DECT G	(G)-2→(G)	0640	0 - 4	(G) :0	VI
Divide	DIV G,R	Note 2	3C00	4	Note 3	IX
Increment	INC G	(G)+1→(G)	0580	0 - 4	(G) :0	VI
Increment by 2	INCT G	(G)+2→(G)	05C0	0 - 4	(G) :0	VI
Multiply	MPY G,R	Note 4	3800	None		IX
Negate	NEG G	-(G)→(G)	0500	0 - 2, 4	(G) :0	VI
Subtract	S G1, G2	(G2)-(G1)→(G2)	6000	0 - 4	(G2) :0	I
Subtract Bytes	SB G1, G2	(G2)-(G1)→(G2)	7000	0 - 5	(G2) :0	I

NOTES:

1. The original value of G is compared to zero.

2. The contents of register R and the next consecutive register (32-bit magnitude) are divided by G (16-bit magnitude). The quotient (16-bit magnitude) is placed in R and the remainder is placed R+1. If R=15, the remainder is placed in the location immediately following the workspace.

3. If the divisor is less than or equal to the left half of the dividend, the divide instruction is aborted and overflow status bit (bit 4) is set.

4. (G) is multiplied by (R). The result (32-bit magnitude) is placed in R and R+1. R contains the most significant half of the result. If R=15, the least significant half of the result if placed in the location immediately following the workspace.

Table B-2. Branch Instructions

Instruction	Format	Effect	Necessary Status	Opcode	Format Number
Branch	B G	G → (PC)	Unconditional	0440	VI
Branch and Link	BL G	G → (PC) (PC) → (R11)	Unconditional	0680	VI
Branch and Link WP	BLWP G	Note 1	Unconditional	0400	VI
Jump If Equal	JEQ S	S → (PC)	Bit 2 = 1	1300	II
Jump If High or Equal	JHE S	S → (PC)	Bit 0 or Bit 2 = 1	1400	II
Jump If Greater Than	JGT S	S → (PC)	Bit 1 = 1	1500	II
Jump If Logical High	JH S	S → (PC)	Bit 0 = 1 and Bit 2 = 0	1B00	II
Jump If Logical Low	JL S	S → (PC)	Bit 0 = 0 and Bit 2 = 0	1A00	II
Jump If Less or Equal	JLE S	S → (PC)	Bit 1 = 0 and Bit 2 = 1	1200	II
Jump If Less Than	JLT S	S → (PC)	Bit 1 = 0 and Bit 2 = 0	1100	II
Unconditional Jump	JMP S	S → (PC)	Unconditional	1000	II
Jump If No Carry	JNC S	S → (PC)	Bit 3 = 0	1700	II
Jump If Not Equal	JNE S	S → (PC)	Bit 2 = 0	1600	II
Jump If No Overflow	JNO S	S → (PC)	Bit 4 = 0	1900	II
Jump If Odd Parity	JOP S	S → (PC)	Bit 5 = 1	1C00	II
Jump On Carry	JOC S	S → (PC)	Bit 3 = 1	1800	II
Return WP	RTWP	Note 2	Unconditional	0380	VII
Execute	X G	Note 3	Unconditional	0480	VI

NOTES:

1. BLWP is explained in detail in paragraph 8.2. It can be summarized as follows:

   (G) → (WP)
   (G + 2) → (PC)
   (original WP) → (R13)
   (old PC) → (R14)
   (ST) → (R15)

2. RTWP is explained in detail in paragraph 8.2. It can be summarized as follows:

   (R13) → (WP)
   (R14) → (PC)
   (R15) → (ST)

3. An instruction at address G is executed as if it were located in memory where the Execute instruction resides. Observe that if the instruction executed is not a single word instruction, the word following the Execute instruction is used (i.e., if symbolic memory addressing or indexed addressing is required, the symbol value must be in the word following the Execute instruction). The Execute instruction does not affect the status bits but the instruction executed will set the status bits appropriately.

Table B-3. Compare Instructions

Instruction	Format	Opcode	Status Bits Affected	Status Bits Test	Format Number
Compare Words	C G1, G2	8000	0-2	(G1) :(G2)	I
Compare Bytes	CB G1, G2	9000	0-2, 5	(G1) :(G2)	I
Compare Immediate	CI R, I	0280	0-2	(R) :I	VIII
Compare Ones Corresponding	COC G, R	2000	2	Note 1	III
Compare Zeros Corresponding	CZC G, R	2400	2	Note 2	III

NOTES:

General: Compare instructions have no effect other than setting status bits. Note that the two's complement representation negative numbers are logically greater than positive numbers, and that negative numbers of small magnitude are logically greater than negative numbers of larger magnitude.

1. The bits in the destination operand that correspond to bits equal to one in the source operand are compared to one. If the corresponding bits are equal to one, status bit 2 is set to 1. Otherwise the status bit is set to 0.

2. The bits in the destination operand that correspond to bits equal to one in the source operand are compared to zero. If the corresponding bits are equal to zero, status bit 2 is set to 1. Otherwise the status bit is set to 0.

Table B-4. Control and CRU Instructions

Instruction	Format	Effect	Opcode	Status Bits Affected	Status Bits Test	Format Number
Clock Off (Note 12)	CKOF	Note 1	03C0	None		VII
Clock On (Note 12)	CKON	Note 2	03A0	None		VII
Load Communication Register	LDCR G, T	Note 3	3000	0 - 2, 5	(G) :0	IV
Idle (Note 12)	IDLE	Note 4	0340	None		VII
Load ROM and Execute (Note 12)	LREX	Note 6	03E0	None		VII
Reset I/O (Note 12)	RSET	Note 5	0360	0 - 5	Note 7	VII
Set Bit to One	SBO E	Note 8	1D00	None		II
Set Bit to Zero	SBZ E	Note 9	1E00	None		II
Store Communication Register	STCR G, T	Note 10	3400	0 - 2, 5	(G) :0	IV
Test Bit	TB E		1F00	2	Note 11	II

NOTES:

1. Disables 120 Hz clock.

2. Enables 120 Hz clock. If interrupt level 5 is enabled, an interrupt occurs every 8.33 ms. Interrupt address is 14_{16}.

3. Transfers consecutive data bits from the byte address specified by G to the CRU. The number of bits to be transferred is specified by T. The CRU address is the contents of R12 of the current workspace. The least significant bit of the byte address specified by G is placed in the CRU bit addressed by R12. See illustration, Memory CRU Transfer (Note 9).

4. Places the computer in the idle state. An interrupt or start signal causes the computer to resume execution at the instruction following the IDLE instruction.

5. Disables all interrupts. Resets all directly connected I/O devices.

6. Places contents of $FFFC_{16}$ into WP register, and contents of $FFFE_{16}$ into PC.

7. Sets bits 0 - 5 to zero.

8. Sets CRU bit at address in R12 + E to one.

9. Sets CRU bit at address in R12 + E to zero.

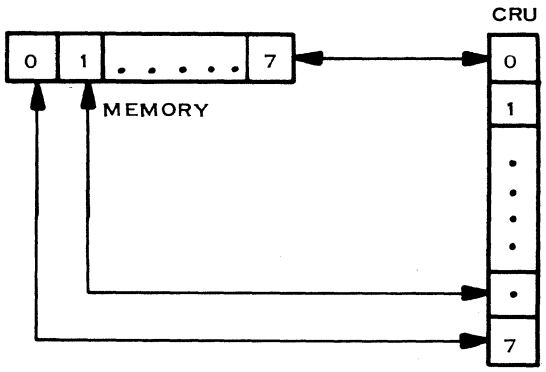
Table B-4. Control and CRU Instructions (Continued)

NOTES

10. Transfers consecutive data bits from the CRU to the byte address specified by G. The number of bits transferred is specified by T. The CRU address is the contents of R12 of the current workspace. The CRU bit addressed by R12 is placed in the least significant bit of the byte addressed by G. See Memory - CRU Transfer illustration.

11. Tests CRU bit at address in R12 + E. Set status bit 2 to the value of the CRU bit.

12. Does not apply to TMS 9900.



(A)128444

Memory - CRU Transfer

Table B-5. Load and Move Instructions

Instruction	Format	Effect	Opcode	Status Bits Affected	Status Bits Test	Format Format
Load Immediate	LI R, I	I → (R)	0200	0 - 2	I:0	VIII
Load Interrupt Mask	L IMI I	Note 1	0300	None		VIII
Load Memory Map File	LMF R,M	Notes 4, 5	0320	None		X
Load Workspace Pointer	LWPI I	I → (WP)	02E0	None		VIII
Move Words	MOV G1, G2	(G1) → (G2)	C000	0 - 2	(G2) :0	I
Move Bytes	MOVB G1, G2	(G1) → (G2)	D000	0 - 2, 5	(G2) :0	I
Store Status	STST R	(ST) → (R)	02C0	None		VIII
Store WP	STWP R	(WP) → (R)	02A0	None		VIII
Swap Bytes	SWPB G	Note 3	06C0	None		VI

NOTES:

1. Places the least-significant 4 bits of the immediate value I in the interrupt mask.
2. Loads the 256 words of the ROM program into the first 256 words of memory. Places the contents of the memory pair at address 0 into WP and PC and starts execution.
3. Interchanges bits 0 - 7 with bits 8 - 15 of word at address specified by G.
4. Place the contents of a six-word area at the address in R into memory map file M.
5. 990/10 with mapping only.

Table B-6. Logical Instructions

Instruction	Format	Effect	Opcode	Status Bits Affected	Status Bits Test	Format Number
AND Immediate	ANDI R, I	(R) AND I → (R)	0240	0 - 2	(R) :0	VIII
Clear	CLR G	0 → (G)	04C0	None		VI
Invert Bits	INV G	Note 1	0540	0 - 2	(G) :0	VI
OR Immediate	ORI R, I	(R) OR I → (R)	0260	0 - 2	(R) :0	VIII
Set to Ones	SETO G	>FFFF → (G)	0700	None		VI
Set Ones Corresponding	SOC G1, G2	Note 2	E000	0 - 2	(G2) :0	I
Set Ones Corresponding Bytes	SOCB G1, G2	Note 2	F000	0 - 2, 5	(G2) :0	I
Set Zeros Corresponding	SZC G1, G2	Note 3	4000	0 - 2	(G2) :0	I
Set Zeros Corresponding Bytes	SZCB G1, G2	Note 3	5000	0 - 2, 5	(G2) :0	I
Exclusive OR	XOR G, R	(G) XOR (R) → (R)	2800	0 - 2	(R) :0	III

NOTES:

1. Places one's complement of contents of location G in location G.

2. Sets bits to one in G2 that correspond to bits equal to one in G1. (G1) OR (G2) → (G2).

```
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0  G1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0  G2
-------------------------------
1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 0  G2 (result)
```

3. Sets bits to zero in G2 that correspond to bits equal to one in G1. (INV (G1)) AND (G2) → (G2).

```
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0  G1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0  G2
-------------------------------
0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0  G2 (result)
```

Table B-7. Workspace Register Shift Instructions

Instruction	Format	Value Placed in Vacated Bit Position on Each Shift	Opcode	Format Number
Shift Right Arithmetic	SRA R, C	Original value of leftmost bit	0800	V
Shift Right Logical	SRL R, C	Logical zero	0900	V
Shift Left Arithmetic	SLA R, C	Logical zero (Note 1)	0A00	V
Shift Right Circular	SRC R, C	Rightmost bit moves to leftmost bit	0B00	V

NOTES:

General: If C is zero, the 4 least-significant bits of R0 contain the shift value. If the 4 least-significant bits of R0 equal 0, shift 16 positions. Otherwise shift C positions. The value of the last bit shifted out of the register is placed in status bit 3. The shifted value is compared to zero-setting status bits 0 - 2.

1. If the sign of the value in R changes during shift, sets status bit 4.

Table B-8. Extended Operation Instruction

Instruction	Format	Effect	Opcode	Status Bits Affected	Status Bits Test	Format Number
Extended Operation	XOP G, T	Note 1	2C00	6	Note 2	IX

NOTES:

1. T specifies the extended operation, 0 - 15, to be executed.

2. Sets status bit 6 to one when extended operation is software implemented, and to zero when extended operation is hardware implemented.

Table B-9. Long Distance Addressing Instructions (990/10 with mapping only)

Instruction	Format	Effect	Opcode	Status Bits Affected	Status Bits Test	Format Number
Long Distance Source	LDS G	Note 1	0780	None		VI
Long Distance	LDD G	Note 2	07C0	None		VI

NOTES:

1. Places the contents of a six-word area of memory at G into memory map file 2, to use for source address of following instruction.

2. Places the contents of a six-word area of memory at G into memory map file 2, to use for destination address of following instruction.

Table B-10. Pseudo-Instructions

Instruction	Equivalent Instruction	Opcode
NOP	JMP $ + 2	1000
RT	B *11	045B
XVEC	DATA, DATA, WPNT	N.A. (Note 1)

Note: 1. Applies to SDSMAC only.

APPENDIX C

PROGRAM ORGANIZATION

APPENDIX C

PROGRAM ORGANIZATION

C.1 PROGRAM AREAS

There are three types of areas in a program for the Model 990 Computer. These are the procedure, the workspace, and the data areas. The procedure area contains the computer instructions. The workspace area contains program linkage, high activity data, and addresses. As many workspaces as convenient may be allocated for a program. Data areas may be allocated as required.

The three previously described hardware registers - WP, PC, and ST - control program execution. The workspace pointer contains the address of the first word of a 16-word area of memory called the workspace. Note that the program workspace may be changed by changing the contents of the WP register. The PC contains the address of the next instruction to execute. The status register contains condition bits set by instructions already performed and the interrupt level mask. These three registers then, completely control and define the context of a program.

The general environment of the 990 Computer is shown in figure C-1. This arrangement of workspace, procedure, and data is the simplest approach to 990 programming. However, though many application programs may be written in this manner, a more segregated approach, with possibly several workspaces, data areas, and connected simple procedures, would provide increased flexibility and applicability.

Most of the programs execute in the environment provided by a resident executive. The areas may be combined in a single task, or the workspace and data areas may be combined in the data division of the task. The procedure area becomes the procedure division of the task in that case. Some of the available executives support writing a procedure division to be used with several data divisions to form tasks that perform the same functions on several sets of data. Refer to the user's guide of the appropriate executive for information about the environment it provides for user programs.

C.2 PROCEDURE

A procedure is the main body of a program and contains computer instructions. It is the action part of a program. Procedures could be coded to solve an equation, run a motor, determine status of a process, or condition a set of data that is to be processed by another procedure. Procedures in the Model 990 Computer may have workspaces and data as an integral part of the coding or may use workspaces and data passed from another procedure.

C.3 WORKSPACE

The Model 990 Computer uses workspaces that may be anywhere in memory and that consist of sixteen consecutive memory words. A context switch due to an interrupt, an XOP instruction, or a BLWP instruction changes the active workspace. A return from the subroutine provided for either of these context switches using an RTWP instruction restores the original workspace. Execution of an LWPI instruction makes a specified workspace active without changing the PC contents. When the data division is separate from the procedure division, any workspace that contains data that is unique to the task represented by the data division should be a part of the data division.

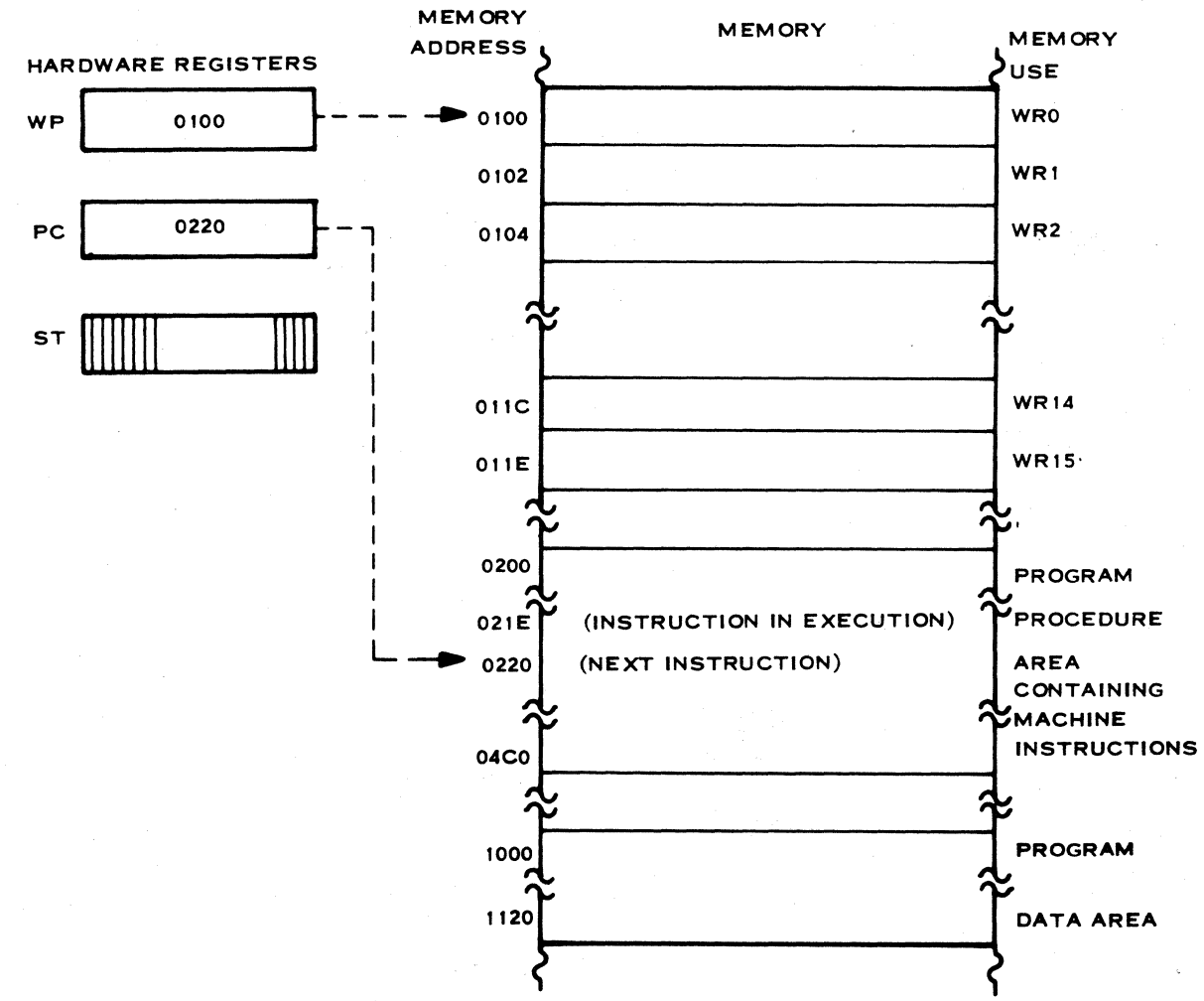


Figure C-1. Model 990 Computer Programming Environment

C.4 DATA

Data for a procedure may appear in many forms. In assembly language, there are three directives available to the programmer to initialize data within a program module. These directives are:

- DATA - Initializes one or more consecutive words of memory to specific values that are input on this statement.
- BYTE - Initializes one or more consecutive bytes of memory as does the DATA statement, except that bytes are initialized.
- TEXT - Initializes a textual string of characters in consecutive bytes of memory. The characters are represented in USASCII code.

Also, data input from the data terminal or device attached to the CRU or TILINE is available to procedures in the 990 Computer.

The available executives for the Model 990 Computer support the user programs by executing supervisor calls to perform input and output operations, data conversions, and other functions. The user provides data in required formats for supervisor call blocks that define the supervisor call, and for other data blocks as appropriate. The assembler directives described previously may be used to provide this data. Details of the data requirements for interface with an executive are described in the user's guide for the executive.

APPENDIX D

HEXADECIMAL INSTRUCTION TABLE

APPENDIX D

HEXADECIMAL INSTRUCTION TABLE

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Paragraph
0200	LI	Load Immediate	VIII	3.60
0220	AI	Add Immediate	VIII	3.12
0240	ANDI	AND Immediate	VIII	3.70
0260	ORI	OR Immediate	VIII	3.71
0280	CI	Compare Immediate	VIII	3.45
02A0	STWP	Store Workspace Pointer	VIII	3.68
02C0	STST	Store Status	VIII	3.67
02E0	LWPI	Load Workspace Pointer Immediate	VIII	3.62
0300	LIMI	Load Interrupt Mask Immediate	VIII	3.61
0320	LMF (Note 1)	Load Memory Map File	X	3.63
0340	IDLE (Note 2)	Computer Idle	VII	3.50
0360	RSET (Note 2)	Computer Reset	VII	3.49
0380	RTWP	Return From Interrupt Subroutine	VII	3.27
03A0	CKON (Note 2)	Clock On	VII	3.52
03C0	CKOF (Note 2)	Clock Off	VII	3.51
03E0	LREX (Note 2)	Load ROM and Execute	VII	3.53
0400	BLWP	Branch And Load Workspace Pointer	VI	3.26
0440	B	Branch	VI	3.24
0480	X	Execute	VI	3.41
04C0	CLR	Clear Operand	VI	3.74
0500	NEG	Negate	VI	3.22
0540	INV	Invert	VI	3.73
0580	INC	Increment By One	VI	3.17
05C0	INCT	Increment By Two	VI	3.18
0600	DEC	Decrement By One	VI	3.14
0640	DECT	Decrement By Two	VI	3.20
0680	BL	Branch and Link	VI	3.25
06C0	SWPB	Swap Bytes	VI	3.66
0700	SETO	Set Ones	VI	3.75

Notes 1. 990/10 with mapping only.
2. Does not apply to TMS 9900

HEXADECIMAL INSTRUCTION TABLE (Continued)

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Paragraph
0740	ABS	Absolute Value	VI	3.21
0780	LDS (Note 1)	Long Distance Source	VI	3.87
07C0	LDD (Note 1)	Long Distance Destination	VI	3.88
0800	SRA	Shift Right Arithmetic	V	3.81
0900	SRL	Shift Right Logical	V	3.83
0A00	SLA	Shift Left Arithmetic	V	3.82
0B00	SRC	Shift Right Circular	V	3.84
1000	JMP	Jump Unconditional	II	3.28
1100	JLT	Jump Less Than	II	3.34
1200	JLE	Jump Low Or Equal	II	3.32
1300	JEQ	Jump Equal	II	3.35
1400	JHE	Jump High Or Equal	II	3.31
1500	JGT	Jump Greater Than	II	3.33
1600	JNE	Jump Not Equal	II	3.36
1700	JNC	Jump No Carry	II	3.38
1800	JOC	Jump On Carry	II	3.37
1900	JNO	Jump No Overflow	II	3.39
1A00	JL	Jump Low	II	3.30
1B00	JH	Jump High	II	3.29
1C00	JOP	Jump Odd Parity	II	3.40
1D00	SBO	Set Bit To One	II	3.54
1E00	SBZ	Set Bit To Zero	II	3.55
1F00	TB	Test Bit	II	3.56
2000	COC	Compare Ones Corresponding	III	3.46
2400	CZC	Compare Zeros Corresponding	III	3.47
2800	XOR	Exclusive OR	III	3.72
2C00	XOP	Extended Operation	IX	3.85
3000	LDCR	Load Communication Register	IV	3.57
3400	STCR	Store Communication Register	IV	3.58
3800	MPY	Multiply	IX	3.15
3C00	DIV	Divide	IX	3.16

Note 1. 990/10 with mapping only.

HEXADECIMAL INSTRUCTION TABLE (Continued)

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Paragraph
4000	SZC	Set Zeros Corresponding, Word	I	3.78
5000	SZCB	Set Zeros Corresponding, Byte	I	3.79
6000	S	Subtract Word	I	3.13
7000	SB	Subtract Byte	I	3.14
8000	C	Compare Words	I	3.43
9000	CB	Compare Bytes	I	3.44
A000	A	Add Words	I	3.10
B000	AB	Add Bytes	I	3.11
C000	MOV	Move Word	I	3.64
D000	MOVB	Move Byte	I	3.65
E000	SOC	Set Ones Corresponding, Word	I	3.76
F000	SOCB	Set Ones Corresponding, Byte	I	3.77

APPENDIX E

ALPHABETICAL INSTRUCTION TABLE

APPENDIX E

ALPHABETICAL INSTRUCTION TABLE

Mnemonic Operation Code	Hexadecimal Operation Code	Name	Format	Paragraph
A	A000	Add Words	I	3.10
AB	B000	Add Bytes	I	3.11
ABS	0740	Absolute Value	VI	3.21
AI	0220	Add Immediate	VIII	3.12
ANDI	0240	AND Immediate	VIII	3.70
B	0440	Branch	VI	3.24
BL	0680	Branch and Link	VI	3.25
BLWP	0400	Branch and Load Workspace Pointer	VI	3.26
C	8000	Compare Words	I	3.43
CB	9000	Compare Bytes	I	3.44
CI	0280	Compare Immediate	VIII	3.45
CKOF (Note 2)	03C0	Clock Off	VII	3.51
CKON (Note 2)	03A0	Clock On	VII	3.52
CLR	04C0	Clear Operand	VI	3.74
COC	2000	Compare Ones Corresponding	III	3.46
CZC	2400	Compare Zeros Corresponding	III	3.47
DEC	0600	Decrement By One	VI	3.19
DECT	0640	Decrement By Two	VI	3.20
DIV	3C00	Divide	IX	3.16
IDLE (Note 2)	0340	Computer Idle	VII	3.50
INC	0580	Increment By One	VI	3.17
INCT	05C0	Increment By Two	VI	3.18
INV	0540	Invert	VI	3.23
JEQ	1300	Jump Equal	II	3.35
JGT	1500	Jump Greater Than	II	3.33
JH	1B00	Jump High	II	3.29
JHE	1400	Jump High Or Equal	II	3.31
JL	1A00	Jump Low	II	3.30
JLE	1200	Jump Low Or Equal	II	3.32
JLT	1100	Jump Less Than	II	3.34
JMP	1000	Jump Unconditional	II	3.28

Notes 1. 990/10 with mapping only.
2. Does not apply to TMS 9900.

ALPHABETICAL INSTRUCTION Table (Continued)

Mnemonic Operation Code	Hexadecimal Operation Code	Name	Format	Paragraph
JNC	1700	Jump No Carry	II	3.38
JNE	1600	Jump Not Equal	II	3.36
JNO	1900	Jump No Overflow	II	3.39
JOC	1800	Jump On Carry	II	3.37
JOP	1C00	Jump Odd Parity	II	3.40
LDCR	3000	Load Communication Register	IV	3.57
LDD (Note 1)	07C0	Long Distance Destination	VI	3.88
LDS (Note 1)	0780	Long Distance Source	VI	3.87
LI	0200	Load Immediate	VIII	3.60
LIMI	0300	Load Interrupt Mask Immediate	VIII	3.61
LMF (Note 1)	0320	Load Memory Map File	X	3.63
LREX (Note 2)	03E0	Load or Restart Execution	VII	3.53
LWPI	02E0	Load Workspace Pointer Immediate	VIII	3.62
MOV	C000	Move Word	I	3.64
MOVB	D000	Move Byte	I	3.65
MPY	3800	Multiply	IX	3.15
NEG	0500	Negate	VI	3.22
ORI	0260	OR Immediate	VIII	3.71
RSET (Note 2)	0360	Computer Reset	VII	3.49
RTWP	0380	Return From Interrupt Subroutine	VII	3.27
S	6000	Subtract Word	I	3.13
SB	7000	Subtract Byte	I	3.14
SBO	1D00	Set Bit To One	II	3.54
SBZ	1E00	Set Bit To Zero	II	3.55
SETO	0700	Set Ones	VI	3.75
SLA	0A00	Shift Left Arithmetic	V	3.82
SOC	E000	Set Ones Corresponding, Word	I	3.76
SOCB	F000	Set Ones Corresponding, Byte	I	3.77

Notes 1. 990/10 with mapping option only.
2. Does not apply to TMS 9900.

ALPHABETICAL INSTRUCTION TABLE (Continued)

Mnemonic Operation Code	Hexadecimal Operation Code	Name	Format	Paragraph
SRA	0800	Shift Right Arithmetic	V	3.81
SRC	0B00	Shift Right Circular	V	3.84
SRL	0900	Shift Right Logical	V	3.83
STCR	3400	Store Communication Register	IV	3.58
STST	02C0	Store Status	VIII	3.67
STWP	02A0	Store Workspace Pointer	VIII	3.68
SWPB	06C0	Swap Bytes	VI	3.66
SZC	4000	Set Zeros Corresponding, Word	I	3.78
SZCB	5000	Set Zeros Corresponding, Byte	I	3.79
TB	1F00	Test Bit	II	3.56
X	0480	Execute	VI	3.41
XOP	2C00	Extended Operation	IX	3.85
XOR	2800	Exclusive OR	III	3.72

APPENDIX F

ASSEMBLER DIRECTIVE TABLE

APPENDIX F

ASSEMBLER DIRECTIVE TABLE

The assembler directives for the Model 990 Assembly Language are listed in table F-1. All directives may include a comment field following the operand field. Those directives that do not require an operand field may have a comment field following the operator field. Those directives that have optional operand fields (RORG and END) may have comment fields only when they have operand fields.

The following symbols and conventions are used in defining the syntax of assembler directives:

- Angle brackets (< >) enclose items supplied by the user
- Brackets ([ ]) enclose optional items
- An ellipsis (. . .) indicates that the preceding item may be repeated
- Braces ({ }) enclose two or more items of which one must be chosen.

The following words are used in defining the items used in assembler directives:

- symbol -
- label - a symbol used in the label field
- string - a character string of a length defined for each directive
- expr - an expression
- wd expr - well-defined expression
- term
- operation - mnemonic operation code, macro name, or previously defined operation or extended operation

Table F-1. Assembler Directives

Directive	Syntax	Force Word Boundary	Note	Applicability
Output Options	OPTION <keyword>[,<keyword>] . . .	NA	5	Cross & SDSMAC
Page Title	[<label>] TITL <string>	NA		All
Program Identifier	[<label>] IDT <string>	NA		All
Copy Source File	[<label>] COPY <file name>	NA		SDSMAC
External Definition	[<label>] DEF <symbol>[,<symbol>] . . .	NA		All
External Reference	[<label>] REF <symbol>[,<symbol>] . . .	NA		All
Secondary Reference	[<label>] SREF <symbol>[,<symbol>] . . .	NA		SDSMAC, TXMIRA
Force Load	[<label>] LOAD<symbol>[,<symbol>] . . .	NA		SDSMAC, TXMIRA
Absolute Origin	[<label>] AORG <wd expr>	No		All
Relocatable Origin	[<label>] RORG [<expr>]	No	3	All
Dummy Origin	[<label>] DORG <expr>	No		All
Workspace Pointer	[<label>] WPNT <label>	NA		SDSMAC
Block Starting with Symbol	[<label>] BSS <wd expr>	No		All
Block Ending with Symbol	[<label>] BES <wd expr>	No		All
Initialize Word	[<label>] DATA <expr>[, <expr>] . . .	Yes		All
Initialize Text	[<label>] TEXT [-] <string>	No	2	All
Define Extended Operation	[<label>] DXOP <symbol>, <term>	NA		All
Define Operation	[<label>] DFOP <symbol>, <operation>	NA		SDSMAC
Define Assembly-Time Constant	<label> EQU <expr>	NA	3	All
Word Boundary	[<label>] EVEN	Yes		All
No Source List	[<label>] UNL	NA		All
List Source	[<label>] LIST	NA		All
Page Eject	[<label>] PAGE	NA		All
Initialize Byte	[<label>] BYTE <wd expr>[,<wd expr>] . . .	No		All
Program End	[<label>] END [<symbol>]	NA	4	All
Program Segment	[<label>] PSEG	Yes		Cross & SDSMAC, TXMIRA
Program Segment End	[<label>] PEND	Yes		Cross & SDSMAC, TXMIRA
Data Segment	[<label>] DSEG	Yes		Cross & SDSMAC, TXMIRA
Data Segment End	[<label>] DEND	Yes		Cross & SDSMAC, TXMIRA
Common Segment	[<label>] CSEG [<string>]	Yes		Cross & SDSMAC, TXMIRA
Common Segment	[<label>] CEND	Yes		Cross & SDSMAC, TXMIRA
END	[<label>] END [Symbol]	NA	4	All
Assemble if	[<label>] ASMIF <expr>	NA	3	SDSMAC
Assemble else	[<label>] ASMELS	NA		SDSMAC
Assemble end	[<label>] ASMEND	NA		SDSMAC

NOTES

1. The expression must be relocatable.
2. The minus sign causes the assembler to negate the rightmost character.
3. Symbols in expressions must have been previously defined.
4. Symbol must have been previously defined.
5. Keywords are XREF, OBJ, SYMT, NOLIST, TUNLST, DUNLST, BUNLST, and MUNLST.

APPENDIX G

MACRO LANGUAGE TABLE

APPENDIX G

MACRO LANGUAGE TABLE

The syntax of the statements that contain the Macro Language verbs is shown in the following table.

Statement	Syntax
Macro	<macro name>b. . .$MACROb. . .[<parm>] . . .b. . .[<comment>]
Variable	b. . .$VARb. . .<var>[,<var>] . . .b. . .[<comment>]
Assign	b. . .$ASGb. . . { <expression> / <string> } bTOb <var>b. . . [<comment>]
Name	<label>b. . .$NAMEb. . .[<comment>]
Go to	b. . .$GOTOb. . .<label>b. . .[<comment>]
Exit	b. . .$EXITb. . .[<comment>]
Call	b. . .$CALLb. . .<macro name>b. . .[<comment>]
If	b. . .$IFb. . .<expression>b. . .[<comment>]
Else	b. . .$ELSEb. . .[<comment>]
End if	b. . .$ENDIFb. . .[<comment>]
End	<label>b. . .$ENDb. . .<macro name>b. . .[<comment>]

MACRO Variable Components

Qualifier	Meaning
S	The string component of the variable.
A	The attribute component of the variable.
V	The value component of the variable.
L	The length component of the variable.

Symbol Components

Qualifier	Meaning
SS	String component of a symbol that is the string component of a variable.
SV	Value component of a symbol that is the string component of a variable.
SA	Attribute component of a symbol that is the string component of a variable.
SL	Length component of a symbol that is the string component of a variable.
SU	User attribute component of symbol that is the string component of a variable.
SG	Segment component of symbol that is the string component of a variable.

APPENDIX H

CRU INTERFACE EXAMPLE

APPENDIX H

CRU INTERFACE EXAMPLE

H.1 GENERAL

This appendix contains an example of programming for a CRU device to aid the user in programming any CRU device which the executive does not support. A medium-speed line printer having the characteristics listed in table H-1 is used as an example device, although this device is supported by the executives.

Table H-1. Medium-Speed Line Printer Characteristics

Function	Description
Print line length	80 characters maximum
Paper width	Variable, up to 9-1/2 inches, sprocket fed
Character format	5x7 dot matrix, 10 characters per inch (horiz) 6 lines per inch (vertical)
Printer speed	60 lines per minute for 80 character lines or 150 lines per minute for 20 character lines
Printer input buffer	80 characters
Buffer data rate	75,000 characters per second (8-bit characters supplied in parallel) maximum

H.2 SOFTWARE INTERFACE REQUIREMENTS

The control characters recognized by the line printer, and the control and response signals for the printer are listed in table H-2. An arbitrary CRU signal arrangement shown in figure H-1 has been selected for this example.

H.2.1 ASSEMBLY LANGUAGE INSTRUCTIONS

The available assembly language instructions that may be used to cause data transfers between the CRU and the printer are:

- SBO - Set Bit to Logic One
- SBZ - Set Bit to Logic Zero
- LDCR - Load Communications Register
- TB - Test Bit

The instructions are described and examples of their use are shown in Section III.

Table H-2. Printer Control and Response Signals

Signal	Definition	Hexadecimal Value
Control Characters		
LF	Line Feed	$0A_{16}$
CR	Carriage Return	$0D_{16}$
TOF	Top of Form	$0C_{16}$
PS	Printer Strobe	11_{16}
PP	Printer Prime	FF_{16}
PD	Printer De-select	13_{16}
Discrete Signals		
PL	Paper Low ←	
PSD	Printer Selected ←	
PF	Printer Fault ←	
BSY	Printer Busy ←	
IM	Interrupt Mask →	
IR	Interrupt Reset →	
ACK	Acknowledge ←	

* ← Signal from printer
→ Signal to printer

H.2.2 SOFTWARE ROUTINES REQUIRED

To properly operate the medium-speed line printer, software routines must provide initialization, character transfer, and end-of-data reporting. The following paragraphs define these operations and provide specific programming examples.

H.2.2.1 INITIALIZATION. Initialization should occur when power is applied to the system. A generalized approach to initialization with specific printer initialization follows:

```
AORG    0
DATA    PWRONW      INITIALIZE POWER ON
DATA    PWRONP      INTERRUPT VECTOR
 .
 .                   OTHER VECTORS
 .
```

CRU BITS	CRU OUTPUTS	PRINTER OUTPUTS (CRU INPUTS)
0	DATA BIT 0 (LSB)	BUSY
1	DATA BIT 1	FAULT
2	DATA BIT 2	SELECTED
3	DATA BIT 3	NOT USED
4	DATA BIT 4	NOT USED
5	DATA BIT 5	NOT USED
6	DATA BIT 6	NOT USED
7	DATA BIT 7 (MSB)	NOT USED
8	STROBE	NOT USED
9	PRIME	NOT USED
10	NOT USED	NOT USED
11	NOT USED	NOT USED
12	NOT USED	NOT USED
13	NOT USED	NOT USED
14	INTERRUPT MASK	NOT USED
15	INTERRUPT RESET	ACKNOWLEDGE –

(A)128706

Figure H-1. CRU Bit Assignments

```
                RORG                    REMAINDER OF PROGRAM RELOCATABLE
PWRONP          EQU     $               POWER ON INITIALIZATION
                  .
                  .                     OTHER INITIALIZATIONS
                  .
PRBASE          EQU     >120            CONNECTED TO MODULE SELECT 9
PRIME           EQU     9               RESET PRINTER
STROBE          EQU     8               TAKE DATA
MASK            EQU     14              INTERRUPT MASK
INT             EQU     15              INTERRUPT RESET
BUSY            EQU     0               PRINTER BUSY
*
* PRINTER INITIALIZATION
*
                LI      12,PRBASE       LOAD CRU BASE ADDRESS
                SBZ     PRIME           SET PRIME TO ZERO
                SBZ     STROBE          SET STROBE TO ZERO
                SBZ     MASK            MASK INTERRUPTS
                  .
                  .
                  .
```

H.2.2.2 CHARACTER TRANSFER. Character transfer can be accomplished as follows by the use of a subroutine call. The assumptions for the subroutine are:

- Workspace register 8 (WR8) contains the address of the data to be printed.
- Workspace register 9 (WR9) is used for temporary storage.
- Workspace register 10 (WR10) contains the number of characters to transfer.
- Workspace register 12 (WR12) contains the CRU base address.

The following subroutine is one method of transferring characters to the printer:

```
PRINTR      EQU     $           PRINT SUBROUTINE
*
* SET UP INTERRUPTS
*
            SBZ     INT         RESET INTERRUPT
            LIMI    15          ENABLE LEVEL 15
            SBO     MASK        ENABLE INTERRUPTS
*
* TEST FOR PRINTER BUSY, PRINT IF NOT
* BUSY, WAIT FOR ANY INTERRUPT IF BUSY
* AND RETRY TEST
*
TSTBSY      TB      BUSY        TEST BUSY BIT
            JEQ     PRINT       IF NOT BUSY
            IDLE                WAIT IF BUSY
            JMP     TSTBSY      RETRY TEST
* CHARACTER PRINT SUBROUTINE
*
PRINT       EQU     $           START
            MOVB    *8+,9       WR9 CONTAINS PRINT CHAR
            INV     9           INVERT BITS (FALSE DATA)
            LDCR    9,8         OUTPUT TO PRINTER
            SBO     STROBE      PULSE STROBE LINE
            SBZ     STROBE      ABOUT 1.5 MICROSECONDS
            DEC     10          DECREMENT CHARACTER COUNT
            JEQ     EXIT        EXIT IF COMPLETE
            JMP     TSTBSY      GO FOR NEXT CHARACTER
*
* EXIT CODE
*
EXIT        SBZ     MASK        MASK INTERRUPT
            RTWP                RETURN TO CALLER
             .
             .
             .
```

H.2.2.3 END-OF-DATA REPORTING. End-of-data reporting in the example subroutine is the exit code, which is executed when the character count in WR10 reaches zero. The code masks the interrupt and returns control to the calling routine.

H.2.2.4 INTERRUPT ROUTINE. In the character transfer subroutine, the CPU enters an idle state when the printer is busy. The occurrence of any enabled interrupt signal causes the CPU to resume processing. The printer is assumed to be connected at interrupt level 15, and all levels are

enabled following execution of the LIMI 15 instruction. The following interrupt routine resets the printer interrupt and returns control to the instruction following the interrupted instruction, the JMP TSTBSY instruction, in this case:

```
            AORG    >3C         INTERRUPT LEVEL 15 VECTOR
            DATA    PRIWP       WORKSPACE ADDRESS
            DATA    PRIPC       PROGRAM ADDRESS
              .
              .
              .
PRIWP       RORG    $-24        SET WORKSPACE ADDRESS RELATIVE TO
            DATA    PRBASE      CRU BASE ADDRESS
            RORG    $+6         RESERVE REMAINDER OF WORKSPACE
PRIPC       EQU     $           INTERRUPT ROUTINE
            SBZ     INT         RESET INTERRUPT
            RTWP
```

H.2.3 PROGRAMMING NOTES

A specific device to be programmed might require more sophisticated routines. These are intended to show possible ways of programming input and output for a CRU device. Error tests may be included to transfer control to error recovery routines when errors are detected. Error recovery routines may simply indicate the occurrence of an error or may correct or overcome the error.

APPENDIX I

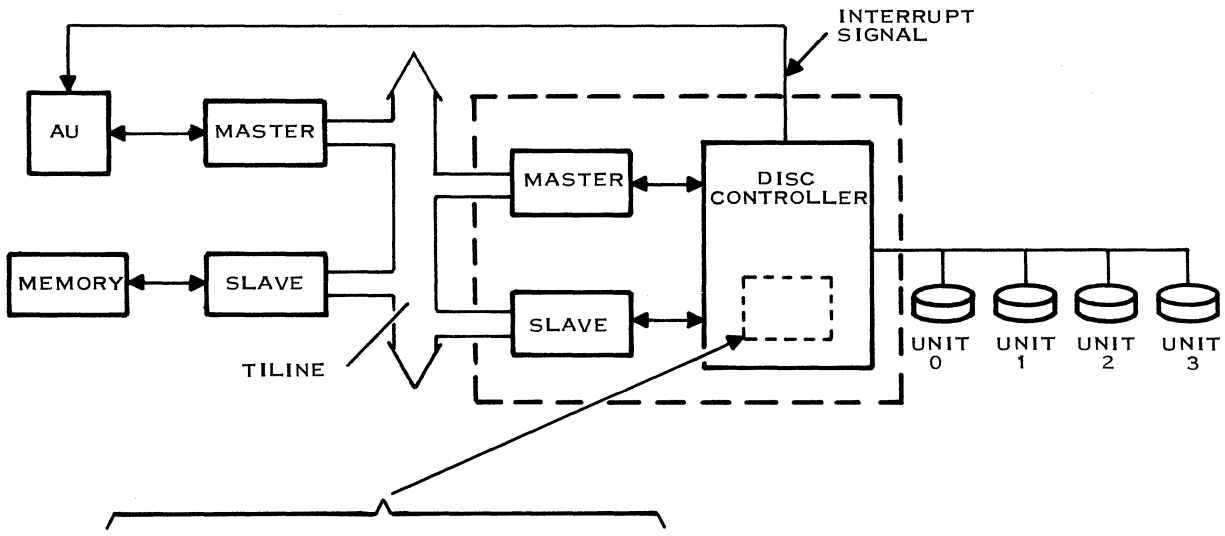
TILINE INTERFACE EXAMPLE

APPENDIX I

TILINE INTERFACE EXAMPLE

I.1 INTRODUCTION

This appendix contains an example of programming for a TILINE device to aid the user in programming any TILINE device which the executive does not support. Figure I-1 shows a typical TILINE device, a disc controller, which is used for this example. Actual input/output for the disc is provided by the executive. This example is only intended to illustrate the principles involved in TILINE input/output programming.



CONTROL REGISTERS

0 7 8 15	TYPICAL TILINE ADDRESS	CORRESPONDING ADDRESS USED BY PROGRAM
DISC STATUS	FFC00	F800
COMMAND / FORMAT	FFC01	F802
SECTOR / SURFACE	FFC02	F804
CYLINDER ADDRESS	FFC03	F806
WORD COUNT	FFC04	F808
MEMORY ADDRESS	FFC05	F80A
EXT. MA / SELECTION	FFC06	F80C
CONTROLLER STATUS	FFC07	F80E

BUSY/ACTIVATE

Figure I-1. TILINE Device Controller Example

I.2 PERIPHERAL CONTROLLER APPLICATION

Controllers for peripheral devices connected to the TILINE have a master interface for transferring data to and from memory. They also have a slave interface for receiving command information from the AU and for sending status information to the AU. A simplified block diagram for a disc controller is shown in figure I-1. Typical use of control registers accessed by the slave interface is also shown in figure I-1. It is assumed that address $F800_{16}$ is assigned to the controller. This corresponds to TILINE address $FFC00_{16}$. A program would operate the disc controller by moving the appropriate parameters into the control registers and setting the activate bit as follows:

```
PARAMS    DATA    >017F       COMMAND,FORMAT
          DATA    >0000       SECTOR, SURFACE
          DATA    0           CYLINDER ADDRESS
          DATA    >1000       WORD COUNT
          DATA    BUFF        MEMORY ADDRESS
          DATA    >0201       EXT. MA, SELECTION
*
* TEST FOR DISC CONTROLLER BUSY
*
          LI      7,>7FFF     WR7 = BUSY TEST MASK
          COC     @>F80E,7    TEST FOR BUSY
          JNE     BUSY        IF NOT
*
* TRANSFER DISC PARAMETER LIST FROM MEMORY TO
* DISC CONTROLLER
*
          LI      8,PARAMS    WR8 = PARAMETER LIST ADDRESS
          LI      9,>F802     WR9 = DISC CONTROLLER ADDRESS
FILL      MOV     *8+,*9+     MOVE LIST WORD TO CONTROLLER
          CI      9,>F80E     STOP WHEN WR9 = CONT. STATUS ADDR.
          JNE     FILL
          INV     7           WR7 = >8000
          SOC     7,*9        SET ACTIVATE BIT
```

The disc controller performs the action requested in the command register. During the time the controller is active, the busy flag is on. When the operation is complete the busy flag is turned off and an interrupt signal is generated. The interrupt can be connected to an external interrupt input line, as determined by the system designer.

APPENDIX J

EXAMPLE PROGRAM

APPENDIX J

EXAMPLE PROGRAM

J.1 CREATING A SOURCE PROGRAM

Once the source statements have been written, they must be prepared for input to the assembler using a medium that the computer is able to read. The statements may be keypunched on punched cards or written on a cassette using the 733 ASR Data Terminal in the off-line mode. The 733 ASR Data Terminal may also be used with the Terminal Source Editor (PX9EDT) to write source statements on a cassette.

The Terminal Source Editor executes under the Prototyping System Executive, PX990. Refer to the *Prototyping System Operation Guide* for information on loading and executing PX990, and on activating PX9EDT. When PX9EDT is ready to receive commands, perform the following steps:

1. Enter the following command:

   I 0

   The Terminal should perform a line feed and carriage return to all entry of source statements.

2. Enter each source statement, terminating the statement with a carriage return. Entering a tab character (CTRL I) following each field of the statement aligns the fields of the statements for better readability.

3. When the source program is long enough to fill the buffer of PX9EDT, enter a Keep command to write some or all of the statements onto the cassette, leaving room in the buffer for more statements. A Keep command to write the first 50 statements in the buffer is entered as follows:

   K 50

4. When the entire source program has been entered, enter the following command to write the buffer contents on the cassette and terminate PX9EDT:

   Q 0

J.2 ASSEMBLING A SOURCE PROGRAM

Once the source program is in machine-readable form, assembly of the code is the next step. The procedure for assembling a program depends upon the assembler to be used and on the operating system under which the assembler executes. The generalized procedure is as follows:

1. When the operating system is not in execution, load, initialize, and start execution of the operating system.

2. Place the source program in the appropriate device and place the device in ready. Assign the devices and files required using Job Control Language statements or Operator Communication statements or commands.

3. Place the device on which the listing is to be printed in ready.

4. Execute the assembler. The assembler should read the source program, print the listing, and write the object code on the output medium.

The one-pass assembler (PX9ASM) executes under PX990. It requires a source input device, an object output device, and a printing device. Refer to the *Prototyping System Operation Guide* for information about loading the executive, assigning the devices, and loading and executing PX9ASM.

The multi-pass assembler (SDSMAC) executes under DX10. It requires a source input device, a printing device, a scratch file on disc, and an object file on disc. Refer to the *DX10 Operating System Programmer's Guide* for information about loading and executing DX10, assigning the devices, and loading and executing SDSMAC.

The Cross Assembler executes on the user's computer, a System/370 for example. The assembler requires a device or data set for intermediate storage, a device or data set for source input, a device or data set for object output, and a device or data set for listings. Refer to the documentation of the system on which the cross assembler executes for information about assigning data sets or devices and executing the cross assembler.

J.3 EXAMPLE PROGRAM

The source program shown in figure J-1 is an example of a source program written on coding forms from which the source programs are prepared. Figure J-2 shows the listing produced by the one-pass assembler. The message printed as the program is executed is shown in figure J-3. The program executes on a 990 with a 733 ASR and it assumes that the CRU base address for the 733 ASR is 000_{16}.

TEXAS INSTRUMENTS
INCORPORATED

MODEL 990/TMS 9900 ASSEMBLY LANGUAGE CODING FORM

LABEL (1–6)	OPER (8–11)	OPERAND (13–25)	COMMENTS (31–60)
	TITL	'HELLØ PRØGRAM'	
* THIS	PRØGRAM PRINTS 'HELLØ!'		ØN THE TELEPRINTER.
DTR	EQU	9	DATA TERMINAL READY.
WRQ	EQU	11	WRITE REQUEST.
RTS	EQU	10	REQUEST TØ SEND.
ASRID	EQU	10	ASR733/33 ID.
DSR	EQU	14	DATA SET READY
*			
HELLØ			
	LWPI	REGS	INITIALIZE WØRKSPACE PØINTER.
	LIMI	0	DISABLE INTERRUPTS
	LI	12,0	INITIALIZE CRU BASE.
	LI	2,TABLE	LOAD TABLE ADDRESS.
	SBØ	DTR	
LØØP	SBØ	RTS	

PROGRAM	PROGRAMMED BY	CHARGE	PAGE	OF

(A)128625 (1/4)

Figure J-1. Example Program (Sheet 1 of 4)

TEXAS INSTRUMENTS
INCORPORATED

MODEL 990/TMS 9900 ASSEMBLY LANGUAGE CODING FORM

LABEL (1–6)	OPER (8–11)	OPERAND (13–25)	COMMENTS (31–60)
	MØVB	*2+,8	GET A CHARACTER.
	JLT	LAST	LAST CHARACTER?
	BL	@PUTC	NO, PUT IT ØUT
	JMP	LØØP	
LAST			
	BL	@PUTC	PUT IT ØUT.
			STØP.
	IDLE		
*			
* ØUTPUT RØUTINE.			
PUTC			
	TB	ASRID	CHECK ASR ID.
	JEQ	ØUT	GØ FØR TTY.

PROGRAM	PROGRAMMED BY	CHARGE	PAGE	OF

(A)128625 (2/4)

Figure J-1. Example Program (Sheet 2 of 4)

TEXAS INSTRUMENTS
INCORPORATED

MODEL 990/TMS 9900 ASSEMBLY LANGUAGE CODING FORM

LABEL (1–6)	OPER (8–11)	OPERAND (13–25)	COMMENTS (31–60)
	MOV	11,5 SAVE	RETURN.
	BL	@ØUT SEND	CHARACTER.
	SBZ	RTS TIMING FØR ASR733.	
	BL	@ØUT	
	BL	@ØUT	
	BL	@ØUT	
	SBØ	RTS	
	B	*5 RETURN TØ CALLER.	
*			
ØUT			
ØNLINE	TB	DSR ASR ØN LINE?	
	JNE	ØNLINE WAIT UNTIL IT IS.	
	LDCR	8,8 ØUTPUT CHARACTER.	

PROGRAM	PROGRAMMED BY	CHARGE	PAGE	OF

(A)128625 (3/4)

Figure J-1. Example Program (Sheet 3 of 4)

TEXAS INSTRUMENTS
INCORPORATED

MODEL 990/TMS 9900 ASSEMBLY LANGUAGE CODING FORM

LABEL (1–6)	OPER (8–11)	OPERAND (13–25)	COMMENTS (31–60)
WAIT	TB	WRQ WAIT	ØN IT.
	JNE	WAIT	
	SBØ	WRQ	
	B	*11 RETURN TØ CALLER.	
*			
* MESSAGE TABLE.			
TABLE			
	TEXT	'HELLØ'	
	BYTE	'!'+>80 SET PARITY BIT.	
	EVEN		
REGS	BSS	32 WØRKSPACE AREA.	
	END	HELLØ	

PROGRAM	PROGRAMMED BY	CHARGE	PAGE	OF

(A)128625 (4/4)

Figure J-1. Example Program (Sheet 4 of 4)

```
0001                    *THIS PROGRAM PRINTS 'HELLO!' ON THE TELEPRINTER.
0002          0009      DTR     EQU  9         DATA TERMINAL READY.
0003          000B      WRQ     EQU  11        WRITE REQUEST.
0004          000A      RTS     EQU  10        REQUEST TO SEND.
0005          000A      ASRID   EQU  10        ASR733/33 ID.
0006          000E      DSR     EQU  14        DATA SET READY.
0007                    *
0008                    HELLO
0009  0000    02E0              LWPI REGS      INITIALIZE WORKSPACE POINTER.
      0002    ----
0010  0004    0300              LIMI 0         DISABLE INTERRUPTS.
      0006    0000
0011  0008    020C              LI   12,0      INITIALIZE CRU BASE.
      000A    0000
0012  000C    0202              LI   2,TABLE LOAD TABLE ADDRESS.
      000E    ----
0013  0010    1D09              SBO  DTR
0014  0012    1D0A    LOOP      SBO  RTS
0015  0014    D232              MOVB *2+,8     GET A CHARACTER.
0016  0016    11--              JLT  LAST      LAST CHARACTER?
0017  0018    06A0              BL   @PUTC     NO, PUT IT OUT.
      001A    ----
0018  001C    10FA              JMP  LOOP
0019                    LAST
      0016**1103
0020  001E    06A0              BL   @PUTC     PUT IT OUT.
      0020    ----
0021  0022    0340              IDLE           STOP.
0022                    *
0023                    *OUTPUT ROUTINE.
0024                    *
0025                    PUTC
      001A**0024'
      0020**0024'
0026  0024    1F0A              TB   ASRID     CHECK ASR ID.
0027  0026    13--              JEQ  OUT       GO FOR TTY.
0028  0028    C14B              MOV  11,5      SAVE RETURN.
0029  002A    06A0              BL   @OUT      SEND CHARACTER.
      002C    ----
0030  002E    1E0A              SBZ  RTS         TIMING FOR ASR733.
0031  0030    06A0              BL   @OUT
      0032    ----
0032  0034    06A0              BL   @OUT
      0036    ----
0033  0038    06A0              BL   @OUT
      003A    ----
0034  003C    1D0A              SBO  RTS
0035  003E    0455              B    *5        RETURN TO CALLER.
0036                    *
0037                    OUT
      0026**130C
      002C**0040'
      0032**0040'
      0036**0040'
      003A**0040'
```

Figure J-2. Assembly Listing of Example Program (Sheet 1 of 2)

```
0038   0040   1F0E          TB    DSR     ASR ONLINE?
0039   0042   16FE          JNE   $-2     WAIT TILL IT IS.
0040   0044   3208          LDCR  8,8     OUTPUT CHARACTER.
0041   0046   1F0B          TB    WR0     WAIT ON IT.
0042   0048   16FE          JNE   $-2
0043   004A   1D0B          SBO   WR0
0044   004C   045B          B     *11     RETURN TO CALLER.
0045                 *
0046                 *MESSAGE TABLE.
0047                 TABLE
       000E**004E'
0048   004E     48          TEXT  'HELLO'
0049   0053     A1          BYTE  '!'+>80  SET PARITY BIT.
0050                         EVEN
0051   0054           REGS  BSS   32      WORKSPACE AREA.
       0002**0054'
0052                         END   HELLO

0000 ERRORS

ASM/TERM? T

(A)132259
```

Figure J-2. Assembly Listing of Example Program (Sheet 2 of 2)

```
HELLO!
```

Figure J-3. Example Program Message

APPENDIX K

NUMERICAL TABLES

Table K-1. Hexadecimal Arithmetic

ADDITION TABLE

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10
2	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
3	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12
4	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
5	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14
6	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15
7	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16
8	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17
9	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18
A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19
B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A
C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

MULTIPLICATION TABLE

1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E
3	06	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D
4	08	0C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0A	0F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Table K-2. Table of Powers of 16_{10}

16^n	n	16^{-n}
1	0	0.10000 00000 00000 00000 x 10
16	1	0.62500 00000 00000 00000 x 10^{-1}
256	2	0.39062 50000 00000 00000 x 10^{-2}
4 096	3	0.24414 06250 00000 00000 x 10^{-3}
65 536	4	0.15258 78906 25000 00000 x 10^{-4}
1 048 576	5	0.95367 43164 06250 00000 x 10^{-6}
16 777 216	6	0.59604 64477 53906 25000 x 10^{-7}
268 435 456	7	0.37252 90298 46191 40625 x 10^{-8}
4 294 967 296	8	0.23283 06436 53869 62891 x 10^{-9}
68 719 476 736	9	0.14551 91522 83668 51807 x 10^{-10}
1 099 511 627 776	10	0.90949 47017 72928 23792 x 10^{-12}
17 592 186 044 416	11	0.56843 41886 08080 14870 x 10^{-13}
281 474 976 710 656	12	0.35527 13678 80050 09294 x 10^{-14}
4 503 599 627 370 496	13	0.22204 46049 25031 30808 x 10^{-15}
72 057 594 037 927 936	14	0.13877 78780 78144 56755 x 10^{-16}
1 152 921 504 606 846 976	15	0.86736 17379 88403 54721 x 10^{-18}

Table K-3. Table of Powers of 10_{16}

10^n	n	10^{-n}
1	0	1.0000 0000 0000 0000
A	1	0.1999 9999 9999 999A
64	2	0.28F5 C28F 5C28 F5C3 x 16^{-1}
3E8	3	0.4189 374B C6A7 EF9E x 16^{-2}
2710	4	0.68DB 8BAC 710C B296 x 16^{-3}
86A0	5	0.A7C5 AC47 1B47 8423 x 16^{-4}
4240	6	0.10C6 F7A0 B5ED 8D37 x 16^{-4}
9680	7	0.1AD7 F29A BCAF 4858 x 16^{-5}
E100	8	0.2AF3 1DC4 6118 73BF x 16^{-6}
CA00	9	0.44B8 2FA0 9B5A 52CC x 16^{-7}
E400	10	0.6DF3 7F67 5EF6 EADF x 16^{-8}
E800	11	0.AFEB FF0B CB24 AAFF x 16^{-9}
1000	12	0.1197 9981 2DEA 1119 x 16^{-9}
A000	13	0.1C25 C268 4976 81C2 x 16^{-10}
4000	14	0.2D09 370D 4257 3604 x 16^{-11}
8000	15	0.480E BE7B 9D58 566D x 16^{-12}
0000	16	0.734A CA5F 6226 F0AE x 16^{-13}
0000	17	0.B877 AA32 36A4 B449 x 16^{-14}
0000	18	0.1272 5DD1 D243 ABA1 x 16^{-14}
0000	19	0.1D83 C94F B6D2 AC35 x 16^{-15}

Table K-4. Table of Powers of Two

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5

Table K-5. Hexadecimal–Decimal Integer Conversion Table

The table appearing on the following pages provides a means for direct conversion of decimal integers in the range of 0 to 4095 and for hexadecimal integers in the range of 0 to FFF.

To convert numbers above those ranges, add table values to the figures below:

Hexadecimal	Decimal	Hexadecimal	Decimal
01 000	4 096	20 000	131 072
02 000	8 192	30 000	196 608
03 000	12 288	40 000	262 144
04 000	16 384	50 000	327 680
05 000	20 480	60 000	393 216
06 000	24 576	70 000	458 752
07 000	28 672	80 000	524 288
08 000	32 768	90 000	589 824
09 000	36 864	A0 000	655 360
0A 000	40 960	B0 000	720 896
0B 000	45 056	C0 000	786 432
0C 000	49 152	D0 000	851 968
0D 000	53 248	E0 000	917 504
0E 000	57 344	F0 000	983 040
0F 000	61 440	100 000	1 048 576
10 000	65 536	200 000	2 097 152
11 000	69 632	300 000	3 145 728
12 000	73 728	400 000	4 194 304
13 000	77 824	500 000	5 242 880
14 000	81 920	600 000	6 291 456
15 000	86 016	700 000	7 340 032
16 000	90 112	800 000	8 388 608
17 000	94 208	900 000	9 437 184
18 000	98 304	A00 000	10 485 760
19 000	102 400	B00 000	11 534 336
1A 000	106 496	C00 000	12 582 912
1B 000	110 592	D00 000	13 631 488
1C 000	114 688	E00 000	14 680 064
1D 000	118 784	F00 000	15 728 640
1E 000	122 880	1 000 000	16 777 216
1F 000	126 976	2 000 000	33 554 432

Table K-5. Hexadecimal–Decimal Integer Conversion Table (Cont.)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
010	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
020	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
030	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
040	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
050	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
060	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
070	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
080	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
090	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0A0	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0B0	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0C0	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0D0	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0E0	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0F0	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255
100	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
110	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
120	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
130	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
140	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
150	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
160	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
170	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
180	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
190	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A0	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B0	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C0	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D0	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E0	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F0	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
200	0512	0513	0514	0515	0516	0517	0518	0519	0529	0521	0522	0523	0524	0525	0526	0527
210	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
220	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
230	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
240	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
250	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
260	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
270	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
280	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
290	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2A0	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2B0	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2C0	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2D0	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2E0	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2F0	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767

Table K-5. Hexadecimal–Decimal Integer Conversion Table (Cont.)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
300	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
310	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
320	0800	0801	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
330	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
340	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
350	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
360	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
370	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
380	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
390	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3A0	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3B0	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3C0	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3D0	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3E0	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3F0	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023
400	1024	1025	0126	0127	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
410	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
420	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
430	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
440	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
450	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
460	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
470	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
480	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
490	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4A0	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B0	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C0	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4D0	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E0	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4F0	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
500	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1291	1293	1294	1295
510	1296	1297	1298	1299	1399	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
520	1312	1313	1314	1315	1316	1317	1318	1319	1329	1321	1322	1323	1324	1325	1326	1327
530	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
540	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1367	1358	1359
550	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
560	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
570	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
580	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1429	1421	1422	1423
590	1324	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5A0	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
3B0	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5C0	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5D0	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5E0	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5F0	1520	1521	1522	1523	1524	1515	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535

Table K-5. Hexadecimal–Decimal Integer Conversion Table (Cont.)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
600	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
610	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
620	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
630	1584	1585	1586	1587	1588	1589	1590	1591	1592	1592	1594	1595	1596	1597	1598	1599
640	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
650	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
660	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
670	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
680	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
690	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6A0	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B0	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	17231	1724	1725	1726	1727
6C0	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D0	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E0	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F0	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791
700	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	8102	1803	1804	1805	1806	1807
710	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
720	1824	1825	1826	1827	1818	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
730	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
740	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
750	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
760	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1909	1902	1903
770	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
780	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
790	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A0	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B0	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C0	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D0	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7E0	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F0	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047
800	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
810	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
820	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
830	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
840	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
850	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
860	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
870	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
880	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
890	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A0	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B0	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C0	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D0	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E0	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F0	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303

Table K-5 Hexadecimal–Decimal Integer Conversion Table (Cont.)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
900	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
910	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
920	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
930	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
940	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
950	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	3496	2397	2398	2399
960	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
970	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
980	2432	2433	2434	24351	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
990	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A0	2464	2465	2466	2467	2468	2469	2479	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B0	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C0	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D0	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E0	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F0	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559
A00	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A10	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A20	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A30	2608	2609	2610	2611	2612	2613	2614	2615	2626	2617	2618	2619	2620	2621	2622	2623
A40	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A50	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A60	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A70	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A80	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A90	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA0	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB0	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC0	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767
AD0	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE0	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF0	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B00	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B10	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B20	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B30	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B40	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895
B50	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B60	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B70	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B80	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B90	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA0	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB0	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC0	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD0	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE0	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF0	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071

Table K-5. Hexadecimal–Decimal Integer Conversion Table (Cont.)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C00	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C10	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C20	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C30	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C40	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C50	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C60	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C70	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C80	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C90	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA0	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB0	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC0	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD0	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE0	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF0	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327
D00	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D10	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D20	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D30	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D40	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D50	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D60	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D70	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D80	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D90	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA0	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB0	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC0	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD0	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE0	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF0	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E00	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E10	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E20	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E30	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E40	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E50	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E60	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E70	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E80	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E90	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA0	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EB0	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775

Table K-5. Hexadecimal–Decimal Integer Conversion Table (Cont.)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
EC0	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED0	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE0	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF0	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F00	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F10	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F20	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F30	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F40	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F50	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F60	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F70	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F80	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F90	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA0	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB0	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC0	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD0	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE0	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF0	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

Table K-6. Hexadecimal–Decimal Fraction Conversion Table

Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal
.00 00 00 00	.00000 00000	.40 00 00 00	.25000 00000	.80 00 00 00	.50000 00000	.C0 00 00 00	.75000 00000
.01 00 00 00	.00390 62500	.41 00 00 00	.25390 62500	.81 00 00 00	.50390 62500	.C1 00 00 00	.75390 62500
.02 00 00 00	.00781 25000	.42 00 00 00	.25781 25000	.82 00 00 00	.50781 25000	.C2 00 00 00	.75781 25000
.03 00 00 00	.01171 87500	.43 00 00 00	.26171 87500	.83 00 00 00	.51171 87500	.C3 00 00 00	.76171 87500
.04 00 00 00	.01562 50000	.44 00 00 00	.26562 50000	.84 00 00 00	.51562 50000	.C4 00 00 00	.76562 50000
.05 00 00 00	.01953 12500	.45 00 00 00	.26953 12500	.85 00 00 00	.51953 12500	.C5 00 00 00	.76953 12500
.06 00 00 00	.02343 75000	.46 00 00 00	.27343 75000	.86 00 00 00	.52343 75000	.C6 00 00 00	.77343 75000
.07 00 00 00	.02734 37500	.47 00 00 00	.27734 37500	.87 00 00 00	.52734 37500	.C7 00 00 00	.77734 37500
.08 00 00 00	.03125 00000	.48 00 00 00	.28125 00000	.88 00 00 00	.53125 00000	.C8 00 00 00	.78125 00000
.09 00 00 00	.03515 62500	.49 00 00 00	.28515 62500	.89 00 00 00	.53515 62500	.C9 00 00 00	.78515 62500
.0A 00 00 00	.03906 25000	.4A 00 00 00	.28906 25000	.8A 00 00 00	.53906 25000	.CA 00 00 00	.78906 25000
.0B 00 00 00	.04296 87500	.4B 00 00 00	.29296 87500	.8B 00 00 00	.54296 87500	.CB 00 00 00	.79296 87500
.0C 00 00 00	.04687 50000	.4C 00 00 00	.29687 50000	.8C 00 00 00	.54687 50000	.CC 00 00 00	.79687 50000
.0D 00 00 00	.05078 12500	.4D 00 00 00	.30078 12500	.8D 00 00 00	.55078 12500	.CD 00 00 00	.80078 12500
.0E 00 00 00	.05468 75000	.4E 00 00 00	.30468 75000	.8E 00 00 00	.55468 75000	.CE 00 00 00	.80468 75000
.0F 00 00 00	.05859 37500	.4F 00 00 00	.30859 37500	.8F 00 00 00	.55859 37500	.CF 00 00 00	.80859 37500
.10 00 00 00	.06250 00000	.50 00 00 00	.31250 00000	.90 00 00 00	.56250 00000	.D0 00 00 00	.81250 00000
.11 00 00 00	.06640 62500	.51 00 00 00	.31640 62500	.91 00 00 00	.56640 62500	.D1 00 00 00	.81640 62500
.12 00 00 00	.07031 25000	.52 00 00 00	.32031 25000	.92 00 00 00	.57031 25000	.D2 00 00 00	.82031 25000
.13 00 00 00	.07421 87500	.53 00 00 00	.32421 87500	.93 00 00 00	.57421 87500	.D3 00 00 00	.82421 87500
.14 00 00 00	.07812 50000	.54 00 00 00	.32812 50000	.94 00 00 00	.57812 50000	.D4 00 00 00	.82812 50000
.15 00 00 00	.08203 12500	.55 00 00 00	.33203 12500	.95 00 00 00	.58203 12500	.D5 00 00 00	.83203 12500
.16 00 00 00	.08593 75000	.56 00 00 00	.33593 75000	.96 00 00 00	.58593 75000	.D6 00 00 00	.83593 75000
.17 00 00 00	.08984 37500	.57 00 00 00	.33984 37500	.97 00 00 00	.58984 37500	.D7 00 00 00	.83984 37500
.18 00 00 00	.09375 00000	.58 00 00 00	.34375 00000	.98 00 00 00	.59375 00000	.D8 00 00 00	.84375 00000
.19 00 00 00	.09765 62500	.59 00 00 00	.34765 62500	.99 00 00 00	.59765 62500	.D9 00 00 00	.84765 62500
.1A 00 00 00	.10156 25000	.5A 00 00 00	.35156 25000	.9A 00 00 00	.60156 25000	.DA 00 00 00	.85156 25000
.1B 00 00 00	.10546 87500	.5B 00 00 00	.35546 87500	.9B 00 00 00	.60546 87500	.DB 00 00 00	.85546 87500
.1C 00 00 00	.10937 50000	.5C 00 00 00	.35937 50000	.9C 00 00 00	.60937 50000	.DC 00 00 00	.85937 50000
.1D 00 00 00	.11328 12500	.5D 00 00 00	.36328 12500	.9D 00 00 00	.61328 12500	.DD 00 00 00	.86328 12500
.1E 00 00 00	.11718 75000	.5E 00 00 00	.36718 75000	.9E 00 00 00	.61718 75000	.DE 00 00 00	.86718 75000
.1F 00 00 00	.12109 37500	.5F 00 00 00	.37109 37500	.9F 00 00 00	.62109 37500	.DF 00 00 00	.87109 37500
.20 00 00 00	.12500 00000	.60 00 00 00	.37500 00000	.A0 00 00 00	.62500 00000	.E0 00 00 00	.87500 00000
.21 00 00 00	.12890 62500	.61 00 00 00	.37890 62500	.A1 00 00 00	.62890 62500	.E1 00 00 00	.87890 62500
.22 00 00 00	.13281 25000	.62 00 00 00	.38281 25000	.A2 00 00 00	.63281 25000	.E2 00 00 00	.88281 25000
.23 00 00 00	.13671 87500	.63 00 00 00	.38671 87500	.A3 00 00 00	.63671 87500	.E3 00 00 00	.88671 87500
.24 00 00 00	.14062 50000	.64 00 00 00	.39062 50000	.A4 00 00 00	.64062 50000	.E4 00 00 00	.89062 50000
.25 00 00 00	.14453 12500	.65 00 00 00	.39453 12500	.A5 00 00 00	.64453 12500	.E5 00 00 00	.89453 12500
.26 00 00 00	.14843 75000	.66 00 00 00	.39843 75000	.A6 00 00 00	.64843 75000	.E6 00 00 00	.89843 75000
.27 00 00 00	.15234 37500	.67 00 00 00	.40234 37500	.A7 00 00 00	.65234 37500	.E7 00 00 00	.90234 37500
.28 00 00 00	.15625 00000	.68 00 00 00	.40625 00000	.A8 00 00 00	.65625 00000	.E8 00 00 00	.90625 00000
.29 00 00 00	.16015 62500	.69 00 00 00	.41015 62500	.A9 00 00 00	.66015 62500	.E9 00 00 00	.91015 62500
.2A 00 00 00	.16406 25000	.6A 00 00 00	.41406 25000	.AA 00 00 00	.66406 25000	.EA 00 00 00	.91406 25000
.2B 00 00 00	.16796 87500	.6B 00 00 00	.41796 87500	.AB 00 00 00	.66796 87500	.EB 00 00 00	.91796 87500
.2C 00 00 00	.17187 50000	.6C 00 00 00	.42187 50000	.AC 00 00 00	.67187 50000	.EC 00 00 00	.92187 50000
.2D 00 00 00	.17578 12500	.6D 00 00 00	.42578 12500	.AD 00 00 00	.67578 12500	.ED 00 00 00	.92578 12500
.2E 00 00 00	.17968 75000	.6E 00 00 00	.42968 75000	.AE 00 00 00	.67968 75000	.EE 00 00 00	.92968 75000
.2F 00 00 00	.18359 37500	.6F 00 00 00	.43359 37500	.AF 00 00 00	.68359 37500	.EF 00 00 00	.93359 37500
.30 00 00 00	.18750 00000	.70 00 00 00	.43750 00000	.B0 00 00 00	.68750 00000	.F0 00 00 00	.93750 00000
.31 00 00 00	.19140 62500	.71 00 00 00	.44140 62500	.B1 00 00 00	.69140 62500	.F1 00 00 00	.94140 62500
.32 00 00 00	.19531 25000	.72 00 00 00	.44531 25000	.B2 00 00 00	.69531 25000	.F2 00 00 00	.94531 25000
.33 00 00 00	.19921 87500	.73 00 00 00	.44921 87500	.B3 00 00 00	.69921 87500	.F3 00 00 00	.94921 87500
.34 00 00 00	.20312 50000	.74 00 00 00	.45312 50000	.B4 00 00 00	.70312 50000	.F4 00 00 00	.95312 50000
.35 00 00 00	.20703 12500	.75 00 00 00	.45703 12500	.B5 00 00 00	.70703 12500	.F5 00 00 00	.95703 12500
.36 00 00 00	.21093 75000	.76 00 00 00	.46093 75000	.B6 00 00 00	.71093 75000	.F6 00 00 00	.96093 75000
.37 00 00 00	.21484 37500	.77 00 00 00	.46484 37500	.B7 00 00 00	.71484 37500	.F7 00 00 00	.96484 37500
.38 00 00 00	.21875 00000	.78 00 00 00	.46875 00000	.B8 00 00 00	.71875 00000	.F8 00 00 00	.96875 00000
.39 00 00 00	.22265 62500	.79 00 00 00	.47265 62500	.B9 00 00 00	.72265 62500	.F9 00 00 00	.97265 62500
.3A 00 00 00	.22656 25000	.7A 00 00 00	.47656 25000	.BA 00 00 00	.72656 25000	.FA 00 00 00	.97656 25000
.3B 00 00 00	.23046 87500	.7B 00 00 00	.48046 87500	.BB 00 00 00	.73046 87500	.FB 00 00 00	.98046 87500
.3C 00 00 00	.23437 50000	.7C 00 00 00	.48437 50000	.BC 00 00 00	.73437 50000	.FC 00 00 00	.98437 50000
.3D 00 00 00	.23828 12500	.7D 00 00 00	.48828 12500	.BD 00 00 00	.73828 12500	.FD 00 00 00	.98828 12500
.3E 00 00 00	.24218 75000	.7E 00 00 00	.49218 75000	.BE 00 00 00	.74218 75000	.FE 00 00 00	.99218 75000
.3F 00 00 00	.24609 37500	.7F 00 00 00	.49609 37500	.BF 00 00 00	.74609 37500	.FF 00 00 00	.99609 37500

Table K-6. Hexadecimal–Decimal Fraction Conversion Table (Cont.)

Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal
.00 00 00 00	.00000 00000	.00 40 00 00	.00097 65625	.00 80 00 00	.00195 31250	.00 C0 00 00	.00292 96875
.00 01 00 00	.00001 52587	.00 41 00 00	.00099 18212	.00 81 00 00	.00196 83837	.00 C1 00 00	.00294 49462
.00 02 00 00	.00003 05175	.00 42 00 00	.00100 70800	.00 82 00 00	.00198 36425	.00 C2 00 00	.00296 02050
.00 03 00 00	.00004 57763	.00 43 00 00	.00102 23388	.00 83 00 00	.00199 89013	.00 C3 00 00	.00297 54638
.00 04 00 00	.00006 10351	.00 44 00 00	.00103 75976	.00 84 00 00	.00201 41601	.00 C4 00 00	.00299 07226
.00 05 00 00	.00007 62939	.00 45 00 00	.00105 28564	.00 85 00 00	.00202 94189	.00 C5 00 00	.00300 59814
.00 06 00 00	.00009 15527	.00 46 00 00	.00106 81152	.00 86 00 00	.00204 46777	.00 C6 00 00	.00302 12402
.00 07 00 00	.00010 68115	.00 47 00 00	.00108 33740	.00 87 00 00	.00205 99365	.00 C7 00 00	.00303 64990
.00 08 00 00	.00012 20703	.00 48 00 00	.00109 86328	.00 88 00 00	.00207 51953	.00 C8 00 00	.00305 17578
.00 09 00 00	.00013 73291	.00 49 00 00	.00111 38916	.00 89 00 00	.00209 04541	.00 C9 00 00	.00306 70166
.00 0A 00 00	.00015 25878	.00 4A 00 00	.00112 91503	.00 8A 00 00	.00210 57128	.00 CA 00 00	.00308 22753
.00 0B 00 00	.00016 78466	.00 4B 00 00	.00114 44091	.00 8B 00 00	.00212 09716	.00 CB 00 00	.00309 75341
.00 0C 00 00	.00018 31054	.00 4C 00 00	.00115 96679	.00 8C 00 00	.00213 62304	.00 CC 00 00	.00311 27929
.00 0D 00 00	.00019 83642	.00 4D 00 00	.00117 49267	.00 8D 00 00	.00215 14892	.00 CD 00 00	.00312 80517
.00 0E 00 00	.00021 36230	.00 4E 00 00	.00119 01855	.00 8E 00 00	.00216 67480	.00 CE 00 00	.00314 33105
.00 0F 00 00	.00022 88818	.00 4F 00 00	.00120 54443	.00 8F 00 00	.00218 20068	.00 CF 00 00	.00315 85693
.00 10 00 00	.00024 41406	.00 50 00 00	.00122 07031	.00 90 00 00	.00219 72656	.00 D0 00 00	.00317 38281
.00 11 00 00	.00025 93994	.00 51 00 00	.00123 59619	.00 91 00 00	.00221 25244	.00 D1 00 00	.00318 90869
.00 12 00 00	.00027 46582	.00 52 00 00	.00125 12207	.00 92 00 00	.00222 77832	.00 D2 00 00	.00320 43457
.00 13 00 00	.00028 99169	.00 53 00 00	.00126 64794	.00 93 00 00	.00224 30419	.00 D3 00 00	.00321 96044
.00 14 00 00	.00030 51757	.00 54 00 00	.00128 17382	.00 94 00 00	.00225 83007	.00 D4 00 00	.00323 48632
.00 15 00 00	.00032 04345	.00 55 00 00	.00129 69970	.00 95 00 00	.00227 35595	.00 D5 00 00	.00325 01220
.00 16 00 00	.00033 56933	.00 56 00 00	.00131 22558	.00 96 00 00	.00228 88183	.00 D6 00 00	.00326 53808
.00 17 00 00	.00035 09521	.00 57 00 00	.00132 75146	.00 97 00 00	.00230 40771	.00 D7 00 00	.00328 06396
.00 18 00 00	.00036 62109	.00 58 00 00	.00134 27734	.00 98 00 00	.00231 93359	.00 D8 00 00	.00329 58984
.00 19 00 00	.00038 14697	.00 59 00 00	.00135 80322	.00 99 00 00	.00233 45947	.00 D9 00 00	.00331 11572
.00 1A 00 00	.00039 67285	.00 5A 00 00	.00137 32910	.00 9A 00 00	.00234 98535	.00 DA 00 00	.00332 64160
.00 1B 00 00	.00041 19873	.00 5B 00 00	.00138 85498	.00 9B 00 00	.00236 51123	.00 DB 00 00	.00334 16748
.00 1C 00 00	.00042 72460	.00 5C 00 00	.00140 38085	.00 9C 00 00	.00238 03710	.00 DC 00 00	.00335 69335
.00 1D 00 00	.00044 25048	.00 5D 00 00	.00141 90673	.00 9D 00 00	.00239 56298	.00 DD 00 00	.00337 21923
.00 1E 00 00	.00045 77636	.00 5E 00 00	.00143 43261	.00 9E 00 00	.00241 08886	.00 DE 00 00	.00338 74511
.00 1F 00 00	.00047 30224	.00 5F 00 00	.00144 95849	.00 9F 00 00	.00242 61474	.00 DF 00 00	.00340 27099
.00 20 00 00	.00048 82812	.00 60 00 00	.00146 48437	.00 A0 00 00	.00244 14062	.00 E0 00 00	.00341 79687
.00 21 00 00	.00050 35400	.00 61 00 00	.00148 01025	.00 A1 00 00	.00245 66650	.00 E1 00 00	.00343 32275
.00 22 00 00	.00051 87988	.00 62 00 00	.00149 53613	.00 A2 00 00	.00247 19238	.00 E2 00 00	.00344 84863
.00 23 00 00	.00053 40576	.00 63 00 00	.00151 06201	.00 A3 00 00	.00248 71826	.00 E3 00 00	.00346 37451
.00 24 00 00	.00054 93164	.00 64 00 00	.00152 58789	.00 A4 00 00	.00250 24414	.00 E4 00 00	.00347 90039
.00 25 00 00	.00056 45751	.00 65 00 00	.00154 11376	.00 A5 00 00	.00251 77001	.00 E5 00 00	.00349 42626
.00 26 00 00	.00057 98339	.00 66 00 00	.00155 63964	.00 A6 00 00	.00253 29589	.00 E6 00 00	.00350 95214
.00 27 00 00	.00059 50927	.00 67 00 00	.00157 16552	.00 A7 00 00	.00254 82177	.00 E7 00 00	.00352 47802
.00 28 00 00	.00061 03515	.00 68 00 00	.00158 69140	.00 A8 00 00	.00256 34765	.00 E8 00 00	.00354 00390
.00 29 00 00	.00062 56103	.00 69 00 00	.00160 21728	.00 A9 00 00	.00257 87353	.00 E9 00 00	.00355 52978
.00 2A 00 00	.00064 08691	.00 6A 00 00	.00161 74316	.00 AA 00 00	.00259 39941	.00 EA 00 00	.00357 05566
.00 2B 00 00	.00065 61279	.00 6B 00 00	.00163 26904	.00 AB 00 00	.00260 92529	.00 EB 00 00	.00358 58154
.00 2C 00 00	.00067 13867	.00 6C 00 00	.00164 79492	.00 AC 00 00	.00262 45117	.00 EC 00 00	.00360 10742
.00 2D 00 00	.00068 66455	.00 6D 00 00	.00166 32080	.00 AD 00 00	.00263 97705	.00 ED 00 00	.00361 63330
.00 2E 00 00	.00070 19042	.00 6E 00 00	.00167 84667	.00 AE 00 00	.00265 50292	.00 EE 00 00	.00363 15917
.00 2F 00 00	.00071 71630	.00 6F 00 00	.00169 37255	.00 AF 00 00	.00267 02880	.00 EF 00 00	.00364 68505
.00 30 00 00	.00073 24218	.00 70 00 00	.00170 89843	.00 B0 00 00	.00268 55468	.00 F0 00 00	.00366 21093
.00 31 00 00	.00074 76806	.00 71 00 00	.00172 42421	.00 B1 00 00	.00270 08056	.00 F1 00 00	.00367 73681
.00 32 00 00	.00076 29394	.00 72 00 00	.00173 95019	.00 B2 00 00	.00271 60644	.00 F2 00 00	.00369 26269
.00 33 00 00	.00077 81982	.00 73 00 00	.00175 47607	.00 B3 00 00	.00273 13232	.00 F3 00 00	.00370 78857
.00 34 00 00	.00079 34570	.00 74 00 00	.00177 00195	.00 B4 00 00	.00274 65820	.00 F4 00 00	.00372 31445
.00 35 00 00	.00080 87158	.00 75 00 00	.00178 52783	.00 B5 00 00	.00276 18408	.00 F5 00 00	.00373 84033
.00 36 00 00	.00082 39746	.00 76 00 00	.00180 05371	.00 B6 00 00	.00277 70996	.00 F6 00 00	.00375 36621
.00 37 00 00	.00083 92333	.00 77 00 00	.00181 57958	.00 B7 00 00	.00279 23583	.00 F7 00 00	.00376 89208
.00 38 00 00	.00085 44921	.00 78 00 00	.00183 10546	.00 B8 00 00	.00280 76171	.00 F8 00 00	.00378 41796
.00 39 00 00	.00086 97509	.00 79 00 00	.00184 63134	.00 B9 00 00	.00282 28759	.00 F9 00 00	.00379 94384
.00 3A 00 00	.00088 50097	.00 7A 00 00	.00186 15722	.00 BA 00 00	.00283 81347	.00 FA 00 00	.00381 46972
.00 38 00 00	.00090 02685	.00 7B 00 00	.00187 68310	.00 BB 00 00	.00285 33935	.00 FB 00 00	.00382 99560
.00 3C 00 00	.00091 55273	.00 7C 00 00	.00189 20898	.00 BC 00 00	.00286 86523	.00 FC 00 00	.00384 52148
.00 3D 00 00	.00093 07861	.00 7D 00 00	.00190 73486	.00 BD 00 00	.00288 39111	.00 FD 00 00	.00386 04736
.00 3E 00 00	.00094 60449	.00 7E 00 00	.00192 26074	.00 BE 00 00	.00289 91699	.00 FE 00 00	.00387 57324
.00 3F 00 00	.00096 13037	.00 7F 00 00	.00193 78662	.00 BF 00 00	.00291 44287	.00 FF 00 00	.00389 09912

Table K-6. Hexadecimal–Decimal Fraction Conversion Table (Cont.)

Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal
.00 00 00 00	.00000 00000	.00 00 40 00	.00000 38146	.00 00 80 00	.00000 76293	.00 00 C0 00	.00001 14440
.00 00 01 00	.00000 00596	.00 00 41 00	.00000 38743	.00 00 81 00	.00000 76889	.00 00 C1 00	.00001 15036
.00 00 02 00	.00000 01192	.00 00 42 00	.00000 39339	.00 00 82 00	.00000 77486	.00 00 C2 00	.00001 15633
.00 00 03 00	.00000 01788	.00 00 43 00	.00000 39935	.00 00 83 00	.00000 78082	.00 00 C3 00	.00001 16229
.00 00 04 00	.00000 02384	.00 00 44 00	.00000 40531	.00 00 84 00	.00000 78678	.00 00 C4 00	.00001 16825
.00 00 05 00	.00000 02980	.00 00 45 00	.00000 41127	.00 00 85 00	.00000 79274	.00 00 C5 00	.00001 17421
.00 00 06 00	.00000 03576	.00 00 46 00	.00000 41723	.00 00 86 00	.00000 79870	.00 00 C6 00	.00001 18017
.00 00 07 00	.00000 04172	.00 00 47 00	.00000 42319	.00 00 87 00	.00000 80466	.00 00 C7 00	.00001 18613
.00 00 08 00	.00000 04768	.00 00 48 00	.00000 42915	.00 00 88 00	.00000 81062	.00 00 C8 00	.00001 19209
.00 00 09 00	.00000 05364	.00 00 49 00	.00000 43511	.00 00 89 00	.00000 81658	.00 00 C9 00	.00001 19805
.00 00 0A 00	.00000 05960	.00 00 4A 00	.00000 44107	.00 00 8A 00	.00000 82254	.00 00 CA 00	.00001 20401
.00 00 0B 00	.00000 06556	.00 00 4B 00	.00000 44703	.00 00 8B 00	.00000 82850	.00 00 CB 00	.00001 20997
.00 00 0C 00	.00000 07152	.00 00 4C 00	.00000 45299	.00 00 8C 00	.00000 83446	.00 00 CC 00	.00001 21593
.00 00 0D 00	.00000 07748	.00 00 4D 00	.00000 45895	.00 00 8D 00	.00000 84042	.00 00 CD 00	.00001 22189
.00 00 0E 00	.00000 08344	.00 00 4E 00	.00000 46491	.00 00 8E 00	.00000 84638	.00 00 CE 00	.00001 22785
.00 00 0F 00	.00000 08940	.00 00 4F 00	.00000 47087	.00 00 8F 00	.00000 85234	.00 00 CF 00	.00001 23381
.00 00 10 00	.00000 09536	.00 00 50 00	.00000 47683	.00 00 90 00	.00000 85830	.00 00 D0 00	.00001 23977
.00 00 11 00	.00000 10132	.00 00 51 00	.00000 48279	.00 00 91 00	.00000 86426	.00 00 D1 00	.00001 24573
.00 00 12 00	.00000 10728	.00 00 52 00	.00000 48875	.00 00 92 00	.00000 87022	.00 00 D2 00	.00001 25169
.00 00 13 00	.00000 11324	.00 00 53 00	.00000 49471	.00 00 93 00	.00000 87618	.00 00 D3 00	.00001 25765
.00 00 14 00	.00000 11920	.00 00 54 00	.00000 50067	.00 00 94 00	.00000 88214	.00 00 D4 00	.00001 26361
.00 00 15 00	.00000 12516	.00 00 55 00	.00000 50663	.00 00 95 00	.00000 88810	.00 00 D5 00	.00001 26957
.00 00 16 00	.00000 13113	.00 00 56 00	.00000 51259	.00 00 96 00	.00000 89406	.00 00 D6 00	.00001 27553
.00 00 17 00	.00000 13709	.00 00 57 00	.00000 51856	.00 00 97 00	.00000 90003	.00 00 D7 00	.00001 28149
.00 00 18 00	.00000 14305	.00 00 58 00	.00000 52452	.00 00 98 00	.00000 90599	.00 00 D8 00	.00001 28746
.00 00 19 00	.00000 14901	.00 00 59 00	.00000 53048	.00 00 99 00	.00000 91195	.00 00 D9 00	.00001 29342
.00 00 1A 00	.00000 15497	.00 00 5A 00	.00000 53644	.00 00 9A 00	.00000 91791	.00 00 DA 00	.00001 29938
.00 00 1B 00	.00000 16093	.00 00 5B 00	.00000 54240	.00 00 9B 00	.00000 92387	.00 00 DB 00	.00001 30534
.00 00 1C 00	.00000 16689	.00 00 5C 00	.00000 54836	.00 00 9C 00	.00000 92983	.00 00 DC 00	.00001 31130
.00 00 1D 00	.00000 17285	.00 00 5D 00	.00000 55432	.00 00 9D 00	.00000 93579	.00 00 DD 00	.00001 31726
.00 00 1E 00	.00000 17881	.00 00 5E 00	.00000 56028	.00 00 9E 00	.00000 94175	.00 00 DE 00	.00001 32322
.00 00 1F 00	.00000 18477	.00 00 5F 00	.00000 56624	.00 00 9F 00	.00000 94771	.00 00 DF 00	.00001 32918
.00 00 20 00	.00000 19073	.00 00 60 00	.00000 57220	.00 00 A0 00	.00000 95367	.00 00 E0 00	.00001 33514
.00 00 21 00	.00000 19669	.00 00 61 00	.00000 57816	.00 00 A1 00	.00000 95963	.00 00 E1 00	.00001 34110
.00 00 22 00	.00000 20265	.00 00 62 00	.00000 58412	.00 00 A2 00	.00000 96559	.00 00 E2 00	.00001 34706
.00 00 23 00	.00000 20861	.00 00 63 00	.00000 59008	.00 00 A3 00	.00000 97155	.00 00 E3 00	.00001 35302
.00 00 24 00	.00000 21457	.00 00 64 00	.00000 59604	.00 00 A4 00	.00000 97751	.00 00 E4 00	.00001 35898
.00 00 25 00	.00000 22053	.00 00 65 00	.00000 60200	.00 00 A5 00	.00000 98347	.00 00 E5 00	.00001 36494
.00 00 26 00	.00000 22649	.00 00 66 00	.00000 60796	.00 00 A6 00	.00000 98943	.00 00 E6 00	.00001 37090
.00 00 27 00	.00000 23245	.00 00 67 00	.00000 61392	.00 00 A7 00	.00000 99539	.00 00 E7 00	.00001 37686
.00 00 28 00	.00000 23841	.00 00 68 00	.00000 61988	.00 00 A8 00	.00001 00135	.00 00 E8 00	.00001 38282
.00 00 29 00	.00000 24437	.00 00 69 00	.00000 62584	.00 00 A9 00	.00001 00731	.00 00 E9 00	.00001 38878
.00 00 2A 00	.00000 25033	.00 00 6A 00	.00000 63180	.00 00 AA 00	.00001 01327	.00 00 EA 00	.00001 39474
.00 00 2B 00	.00000 25629	.00 00 6B 00	.00000 63776	.00 00 AB 00	.00001 01923	.00 00 EB 00	.00001 40070
.00 00 2C 00	.00000 26226	.00 00 6C 00	.00000 64373	.00 00 AC 00	.00001 02519	.00 00 EC 00	.00001 40666
.00 00 2D 00	.00000 26822	.00 00 6D 00	.00000 64969	.00 00 AD 00	.00001 03116	.00 00 ED 00	.00001 41263
.00 00 2E 00	.00000 27418	.00 00 6E 00	.00000 65565	.00 00 AE 00	.00001 03712	.00 00 EE 00	.00001 41859
.00 00 2F 00	.00000 28014	.00 00 6F 00	.00000 61661	.00 00 AF 00	.00001 04308	.00 00 EF 00	.00001 42455
.00 00 30 00	.00000 28610	.00 00 70 00	.00000 66757	.00 00 B0 00	.00001 04904	.00 00 F0 00	.00001 43051
.00 00 31 00	.00000 29206	.00 00 71 00	.00000 67353	.00 00 B1 00	.00001 05500	.00 00 F1 00	.00001 43647
.00 00 32 00	.00000 29802	.00 00 72 00	.00000 67949	.00 00 B2 00	.00001 06096	.00 00 F2 00	.00001 44243
.00 00 33 00	.00000 30398	.00 00 73 00	.00000 68545	.00 00 B3 00	.00001 06692	.00 00 F3 00	.00001 44839
.00 00 34 00	.00000 30994	.00 00 74 00	.00000 69141	.00 00 B4 00	.00001 07228	.00 00 F4 00	.00001 45435
.00 00 35 00	.00000 31590	.00 00 75 00	.00000 69737	.00 00 B5 00	.00001 07884	.00 00 F5 00	.00001 46031
.00 00 36 00	.00000 32186	.00 00 76 00	.00000 70333	.00 00 B6 00	.00001 08480	.00 00 F6 00	.00001 46627
.00 00 37 00	.00000 32782	.00 00 77 00	.00000 70929	.00 00 B7 00	.00001 09076	.00 00 F7 00	.00001 47223
.00 00 38 00	.00000 33378	.00 00 78 00	.00000 71525	.00 00 B8 00	.00001 09672	.00 00 F8 00	.00001 47819
.00 00 39 00	.00000 33974	.00 00 79 00	.00000 75121	.00 00 B9 00	.00001 10268	.00 00 F9 00	.00001 48415
.00 00 3A 00	.00000 34570	.00 00 7A 00	.00000 72717	.00 00 BA 00	.00001 10864	.00 00 FA 00	.00001 49011
.00 00 3B 00	.00000 35166	.00 00 7B 00	.00000 73313	.00 00 BB 00	.00001 11460	.00 00 FB 00	.00001 49607
.00 00 3C 00	.00000 35762	.00 00 7C 00	.00000 73909	.00 00 BC 00	.00001 12056	.00 00 FC 00	.00001 50203
.00 00 3D 00	.00000 36358	.00 00 7D 00	.00000 74505	.00 00 BD 00	.00001 12652	.00 00 ED 00	.00001 50799
.00 00 3E 00	.00000 36954	.00 00 7E 00	.00000 75101	.00 00 BE 00	.00001 13248	.00 00 FE 00	.00001 51395
.00 00 3F 00	.00000 37550	.00 00 7F 00	.00000 75697	.00 00 BF 00	.00001 13844	.00 00 FF 00	.00001 51991

Table K-6. Hexadecimal–Decimal Fraction Conversion Table (Cont.)

Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal
.00 00 00 00	.00000 00000	.00 00 00 40	.00000 00149	.00 00 00 80	.00000 00298	.00 00 00 C0	.00000 00447
.00 00 00 01	.00000 00002	.00 00 00 41	.00000 00151	.00 00 00 81	.00000 00300	.00 00 00 C1	.00000 00449
.00 00 00 02	.00000 00004	.00 00 00 42	.00000 00153	.00 00 00 82	.00000 00302	.00 00 00 C2	.00000 00451
.00 00 00 03	.00000 00006	.00 00 00 43	.00000 00155	.00 00 00 83	.00000 00305	.00 00 00 C3	.00000 00454
.00 00 00 04	.00000 00009	.00 00 00 44	.00000 00158	.00 00 00 84	.00000 00307	.00 00 00 C4	.00000 00456
.00 00 00 05	.00000 00011	.00 00 00 45	.00000 00160	.00 00 00 85	.00000 00309	.00 00 00 C5	.00000 00458
.00 00 00 06	.00000 00013	.00 00 00 46	.00000 00162	.00 00 00 86	.00000 00311	.00 00 00 C6	.00000 00461
.00 00 00 07	.00000 00016	.00 00 00 47	.00000 00165	.00 00 00 87	.00000 00314	.00 00 00 C7	.00000 00463
.00 00 00 08	.00000 00018	.00 00 00 48	.00000 00167	.00 00 00 88	.00000 00316	.00 00 00 C8	.00000 00465
.00 00 00 09	.00000 00020	.00 00 00 49	.00000 00169	.00 00 00 89	.00000 00318	.00 00 00 C9	.00000 00467
.00 00 00 0A	.00000 00023	.00 00 00 4A	.00000 00172	.00 00 00 8A	.00000 00321	.00 00 00 CA	.00000 00470
.00 00 00 0B	.00000 00025	.00 00 00 4B	.00000 00174	.00 00 00 8B	.00000 00323	.00 00 00 CB	.00000 00472
.00 00 00 0C	.00000 00027	.00 00 00 4C	.00000 00176	.00 00 00 8C	.00000 00325	.00 00 00 CC	.00000 00474
.00 00 00 0D	.00000 00030	.00 00 00 4D	.00000 00179	.00 00 00 8D	.00000 00328	.00 00 00 CD	.00000 00477
.00 00 00 0E	.00000 00032	.00 00 00 4E	.00000 00181	.00 00 00 8E	.00000 00330	.00 00 00 CE	.00000 00479
.00 00 00 0F	.00000 00034	.00 00 00 4F	.00000 00183	.00 00 00 8F	.00000 00332	.00 00 00 CF	.00000 00481
.00 00 00 10	.00000 00037	.00 00 00 50	.00000 00186	.00 00 00 90	.00000 00335	.00 00 00 D0	.00000 00484
.00 00 00 11	.00000 00039	.00 00 00 51	.00000 00188	.00 00 00 91	.00000 00337	.00 00 00 D1	.00000 00486
.00 00 00 12	.00000 00041	.00 00 00 52	.00000 00190	.00 00 00 92	.00000 00339	.00 00 00 D2	.00000 00488
.00 00 00 13	.00000 00044	.00 00 00 53	.00000 00193	.00 00 00 93	.00000 00342	.00 00 00 D3	.00000 00491
.00 00 00 14	.00000 00046	.00 00 00 54	.00000 00195	.00 00 00 94	.00000 00344	.00 00 00 D4	.00000 00493
.00 00 00 15	.00000 00048	.00 00 00 55	.00000 00197	.00 00 00 95	.00000 00346	.00 00 00 D5	.00000 00495
.00 00 00 16	.00000 00051	.00 00 00 56	.00000 00200	.00 00 00 96	.00000 00349	.00 00 00 D6	.00000 00498
.00 00 00 17	.00000 00053	.00 00 00 57	.00000 00202	.00 00 00 97	.00000 00351	.00 00 00 D7	.00000 00500
.00 00 00 18	.00000 00055	.00 00 00 58	.00000 00204	.00 00 00 98	.00000 00353	.00 00 00 D8	.00000 00502
.00 00 00 19	.00000 00058	.00 00 00 59	.00000 00207	.00 00 00 99	.00000 00356	.00 00 00 D9	.00000 00505
.00 00 00 1A	.00000 00060	.00 00 00 5A	.00000 00209	.00 00 00 9A	.00000 00358	.00 00 00 DA	.00000 00507
.00 00 00 1B	.00000 00062	.00 00 00 5B	.00000 00211	.00 00 00 9B	.00000 00360	.00 00 00 DB	.00000 00509
.00 00 00 1C	.00000 00065	.00 00 00 5C	.00000 00214	.00 00 00 9C	.00000 00363	.00 00 00 DC	.00000 00512
.00 00 00 1D	.00000 00067	.00 00 00 5D	.00000 00216	.00 00 00 9D	.00000 00365	.00 00 00 DD	.00000 00514
.00 00 00 1E	.00000 00069	.00 00 00 5E	.00000 00218	.00 00 00 9E	.00000 00367	.00 00 00 DE	.00000 00516
.00 00 00 1F	.00000 00072	.00 00 00 5F	.00000 00221	.00 00 00 9F	.00000 00370	.00 00 00 DF	.00000 00519
.00 00 00 20	.00000 00074	.00 00 00 60	.00000 00223	.00 00 00 A0	.00000 00372	.00 00 00 E0	.00000 00521
.00 00 00 21	.00000 00076	.00 00 00 61	.00000 00225	.00 00 00 A1	.00000 00374	.00 00 00 E1	.00000 00523
.00 00 00 22	.00000 00079	.00 00 00 62	.00000 00228	.00 00 00 A2	.00000 00377	.00 00 00 E2	.00000 00526
.00 00 00 23	.00000 00081	.00 00 00 63	.00000 00230	.00 00 00 A3	.00000 00379	.00 00 00 E3	.00000 00528
.00 00 00 24	.00000 00083	.00 00 00 64	.00000 00232	.00 00 00 A4	.00000 00381	.00 00 00 E4	.00000 00530
.00 00 00 25	.00000 00086	.00 00 00 65	.00000 00235	.00 00 00 A5	.00000 00384	.00 00 00 E5	.00000 00533
.00 00 00 26	.00000 00088	.00 00 00 66	.00000 00237	.00 00 00 A6	.00000 00386	.00 00 00 E6	.00000 00535
.00 00 00 27	.00000 00090	.00 00 00 67	.00000 00239	.00 00 00 A7	.00000 00388	.00 00 00 E7	.00000 00537
.00 00 00 28	.00000 00093	.00 00 00 68	.00000 00242	.00 00 00 A8	.00000 00391	.00 00 00 E8	.00000 00540
.00 00 00 29	.00000 00095	.00 00 00 69	.00000 00244	.00 00 00 A9	.00000 00393	.00 00 00 E9	.00000 00542
.00 00 00 2A	.00000 00097	.00 00 00 6A	.00000 00246	.00 00 00 AA	.00000 00395	.00 00 00 EA	.00000 00544
.00 00 00 2B	.00000 00100	.00 00 00 6B	.00000 00249	.00 00 00 AB	.00000 00398	.00 00 00 EB	.00000 00547
.00 00 00 2C	.00000 00102	.00 00 00 6C	.00000 00251	.00 00 00 AC	.00000 00400	.00 00 00 EC	.00000 00549
.00 00 00 2D	.00000 00104	.00 00 00 6D	.00000 00253	.00 00 00 AD	.00000 00402	.00 00 00 ED	.00000 00551
.00 00 00 2E	.00000 00107	.00 00 00 6E	.00000 00256	.00 00 00 AE	.00000 00405	.00 00 00 EE	.00000 00554
.00 00 00 2F	.00000 00109	.00 00 00 6F	.00000 00258	.00 00 00 AF	.00000 00407	.00 00 00 EF	.00000 00556
.00 00 00 30	.00000 00111	.00 00 00 70	.00000 00260	.00 00 00 B0	.00000 00409	.00 00 00 F0	.00000 00558
.00 00 00 31	.00000 00114	.00 00 00 71	.00000 00263	.00 00 00 B1	.00000 00412	.00 00 00 F1	.00000 00561
.00 00 00 32	.00000 00116	.00 00 00 72	.00000 00265	.00 00 00 B2	.00000 00414	.00 00 00 F2	.00000 00563
.00 00 00 33	.00000 00118	.00 00 00 73	.00000 00267	.00 00 00 B3	.00000 00416	.00 00 00 F3	.00000 00565
.00 00 00 34	.00000 00121	.00 00 00 74	.00000 00270	.00 00 00 B4	.00000 00419	.00 00 00 F4	.00000 00568
.00 00 00 35	.00000 00123	.00 00 00 75	.00000 00272	.00 00 00 B5	.00000 00421	.00 00 00 F5	.00000 00570
.00 00 00 36	.00000 00125	.00 00 00 76	.00000 00274	.00 00 00 B6	.00000 00423	.00 00 00 F6	.00000 00572
.00 00 00 37	.00000 00128	.00 00 00 77	.00000 00277	.00 00 00 B7	.00000 00426	.00 00 00 F7	.00000 00575
.00 00 00 38	.00000 00130	.00 00 00 78	.00000 00279	.00 00 00 B8	.00000 00428	.00 00 00 F8	.00000 00577
.00 00 00 39	.00000 00132	.00 00 00 79	.00000 00281	.00 00 00 B9	.00000 00430	.00 00 00 F9	.00000 00579
.00 00 00 3A	.00000 00135	.00 00 00 7A	.00000 00284	.00 00 00 BA	.00000 00433	.00 00 00 FA	.00000 00582
.00 00 00 3B	.00000 00137	.00 00 00 7B	.00000 00286	.00 00 00 BB	.00000 00435	.00 00 00 FB	.00000 00584
.00 00 00 3C	.00000 00139	.00 00 00 7C	.00000 00288	.00 00 00 BC	.00000 00437	.00 00 00 FC	.00000 00586
.00 00 00 3D	.00000 00142	.00 00 00 7D	.00000 00291	.00 00 00 BD	.00000 00440	.00 00 00 FD	.00000 00589
.00 00 00 3E	.00000 00144	.00 00 00 7E	.00000 00293	.00 00 00 BE	.00000 00442	.00 00 00 FE	.00000 00591
.00 00 00 3F	.00000 00146	.00 00 00 7F	.00000 00295	.00 00 00 BF	.00000 00444	.00 00 00 FF	.00000 00593

Table K-7. Common Mathematical Constants

Constant	Decimal Value			Hexadecimal Value	
π	3.14159	26535	89793	3.243F	6A89
π^{-1}	0.31830	98861	83790	0.517C	C1B7
$\sqrt{\pi}$	1.77245	38509	05516	1.C5BF	891C
$\ln\pi$	1.14472	98858	49400	1.250D	048F
e	2.71828	18284	59045	2.B7E1	5163
e^{-1}	0.36787	94411	71442	0.5E2D	58D9
$\sqrt{e}$	1.64872	12707	00128	1.A612	98E2
$\log_{10}e$	0.43429	44819	03252	0.6F2D	EC55
$\log_{2}e$	1.44269	50408	88963	1.7154	7653
γ	0.57721	56649	01533	0.93C4	67E4
$\ln\gamma$	−0.54953	93129	81645	−0.8CAE	9BC1
$\sqrt{2}$	1.41421	35623	73095	1.6A09	E668
ln2	0.69314	71805	59945	0.B172	17F8
$\log_{10}2$	0.30102	99956	63981	0.4D10	4D42
$\sqrt{10}$	3.16227	76601	68379	3.298B	075C
ln 10	2.30258	40929	94046	2.4D76	3777

APPENDIX L

TMS 9940 PROGRAMMING CONSIDERATIONS

APPENDIX L

TMS 9940 PROGRAMMING CONSIDERATIONS

L.1 TMS 9940 DESCRIPTION

The TMS 9940 is a single chip, 16-bit microcomputer containing a CPU, memory (RAM and EPROM/ROM), and extensive I/O. The instruction set of the TMS 9940 is a subset of the TMS 9900 microprocessor instruction set except for three additional instructions, two instructions which facilitate manipulation of binary coded decimal (BCD) data (DCA and DCS), and a single word load interrupt mask (LIIM) instruction. Compatibility with the TMS 9900 instruction set enhances the TMS 9940 microcomputer to equivalent capabilities of minicomputers. Program and data memory is implemented on the microcomputer chip consisting of 128 bytes of RAM and 2048 bytes of EPROM/ROM. The TMS 9940 implements four levels of interrupts including an internal decrementer which can be programmed as a timer or an event counter. Extra features of the TMS 9940 include program definable input/output pin configuration and a multiprocessor system interface. All members of the TMS 9900 family of peripheral circuits are compatible with the TMS 9940.

The following paragraphs describe the memory organization particular to the TMS 9940, the machine registers, and the additional instructions implemented by the TMS 9940 microcomputer. The extra features of the TMS 9940 are programmed using this instruction set. Refer to the *TMS 9940 16-Bit Microcomputer Data Manual* for description and details of the extra features mentioned above.

L.2 TMS 9940 MEMORY MAP

The TMS 9940 memory map is shown in figure L-1. The 2K X 8 EPROM/ROM is assigned memory addresses 0000_{16} through $07FF_{16}$, and the 128 X 8 RAM is assigned memory addresses 8300_{16} through $837F_{16}$.

The first eight words in the EPROM/ROM (addresses 0000_{16} through $000F_{16}$) are used for the interrupt vectors (only four interrupts are defined), and 24 words (addresses 0050_{16} through $007F_{16}$) are used for the extended operation (XOP) instruction trap vectors (only XOP 4 through XOP 15 are defined). The remaining memory is available for programs, data, and workspace registers. If desired, any of the special areas may also be used as general EPROM/ROM memory.

L.3 TMS 9940 MACHINE REGISTERS

The machine registers (program counter, workspace pointer, and status register) are identical to the TMS 9900 microprocessor with the following exceptions:

The workspace register files may not be overlapping as the workspace register pointer (WP) is only 11 bits wide. During instruction execution, the processor addresses any register in the workspace by concatenating the 11-bit WP value (most significant) with the specified register number (least significant) to form a memory address, and initiates a memory request for the word. Thus workspace register files always begin on a 32 bytes memory address boundary as illustrated in figure L-1.

The status register of the TMS 9940 microcomputer does not implement the extended operation bit (ST 6) and status bit 7 is implemented as a digit carry bit used by the added DCA and DCS instructions. Since there are only four interrupt levels, only the least two significant bits of the status registers are needed for the interrupt mask. Table L-1 defines how each status bit is defined for all TMS 9940 instructions. Notice that the DCA and DCS instructions effect the carry, overflow, and digit carry status bits; the digit carry status bit is also effected by the add, subtract, increment, and decrement instructions.

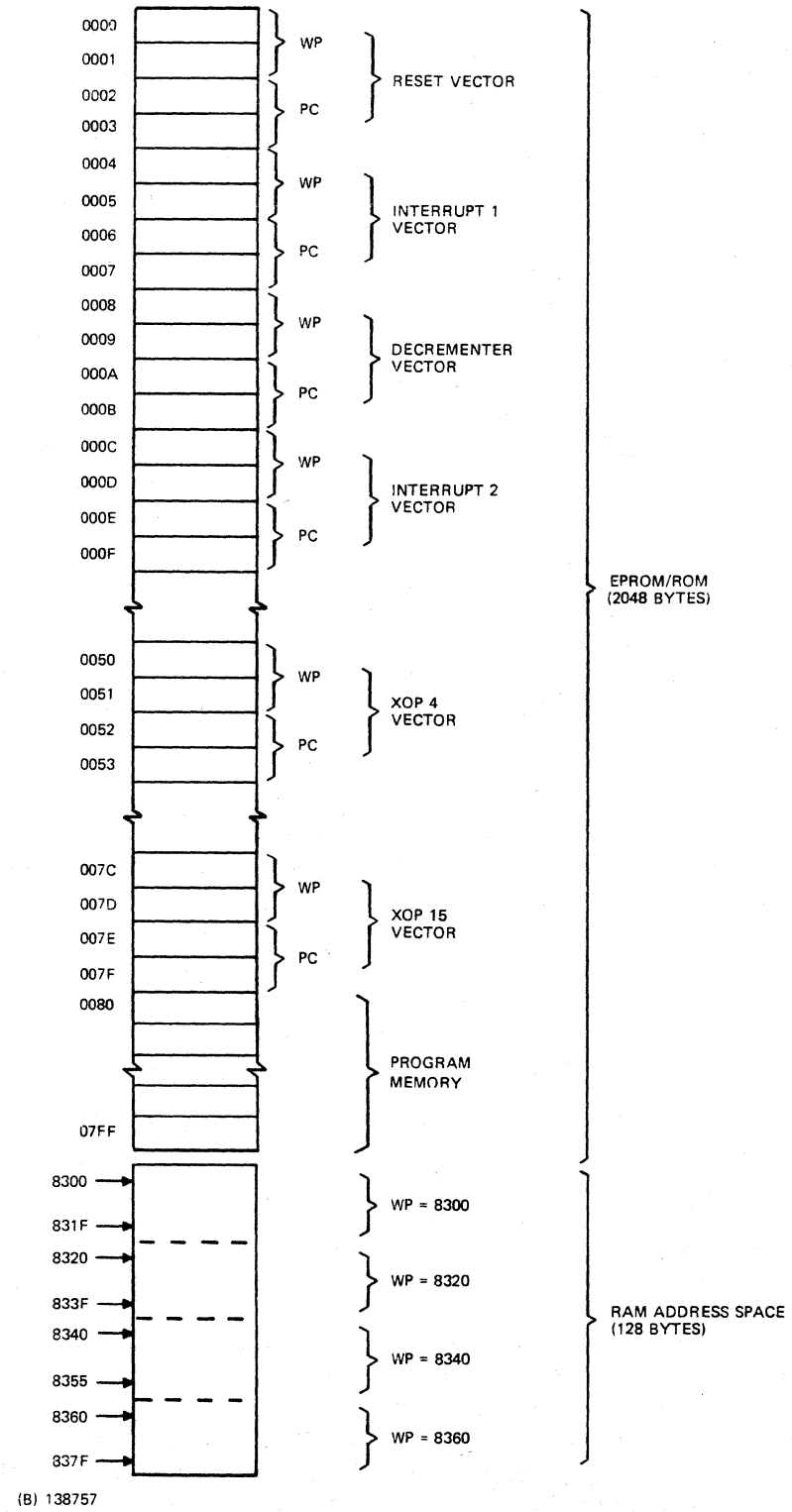


Figure L-1. TMS 9940 Memory Map

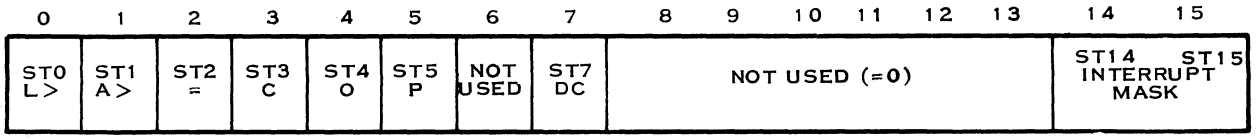


Table L-1. Effect of TMS 9940 on Status Register

BIT	NAME	INSTRUCTION	CONDITION TO SET BIT TO 1
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 OP and MSB of {IOP – (W)} = 1
		ABS	If (SA) ≠ 0
		All Others	If result ≠ 0
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 OP and MSB of IOP–(W) = 1
		ABS	If MSB(SA) = 0 and (SA) ≠ 0
		All Others	If MSB of result = 0 and result ≠ 0
ST2	EQUAL	C,CB	If (SA) = (DA)
		CI	If (W) = IOP
		COC	If (SA) and (DA) = 0
		CZC	If (SA) and (DA) = 0
		TB	If CRUIN = 1
		ABS	If (SA) = 0
		All Others	If result = 0
ST3	CARRY	A,AB,ABS,AI,DEC, DECT,INC,INCT, NEG,S,SB	If CARRY OUT = 1
		DCA	If most significant digit was BCD corrected
		DCS	If most significant digit was not BCD corrected
		SLA,SRA,SRC,SRL	If last bit shifted out = 1
ST4	OVERFLOW	A,AB	If MSB(SA) = MSB(DA) and MSB of result ≠ MSB(DA)
		AI	If MSB(W) = MSB of IOP & 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 = 1
		SAL	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
		ABS,NEG	If (SA) = 8000_{16}
ST5	PARITY	CB MOVB	If (SA) has odd number of 1's
		LDCR,STCR	If 1 ≤C ≤8 and (SA) has odd number of 1's
		AB,SB,SOCB,SZCB, DCA,DCS	If result has odd number of 1's

Table L-1. Effect of TMS 9940 on Status Register (Continued)

BIT	NAME	INSTRUCTION	CONDITION TO SET BIT TO 1
ST7	DIGIT CARRY	A,ABS,AI,DEC, DECT,INC,INCT NEG,S	If carry out of least significant BCD Digit of most significant byte = 1
		AB,DCA,DCS,SB	If carry out of least significant BCD Digit = 1
ST14-ST15	INTERRUPT MASK	LIIM	If corresponding bit of S is 1
		LIMI	If corresponding bit of IOP is 1
		RTWP	If corresponding bit of WR 15 is 1

L.4 TMS 9940 INSTRUCTION SET

The instruction set of the TMS 9940 microcomputer is identical to that of the TMS 9900 with the following exceptions:

- Instructions deleted:

  RSET
  LREX
  CKON
  CKOF

- Instructions added:

  LIIM
  DCA
  DCS

Each additional instruction is described in the following paragraphs in the same syntax conventions used throughout the Assembly Language Programmer's Guide.

NOTE

The additional instructions of the TMS 9940 are implemented using instruction operation codes $2C00_{16}$ through $2CFF_{16}$ which correspond to the TMS 9900 instructions XOP 0 through XOP 4. In the TMS 9940, instructions XOP 0 through XOP 4 may not be used.

L.4.1 LOAD INTERRUPT MASK LIIM.

Syntax definition:

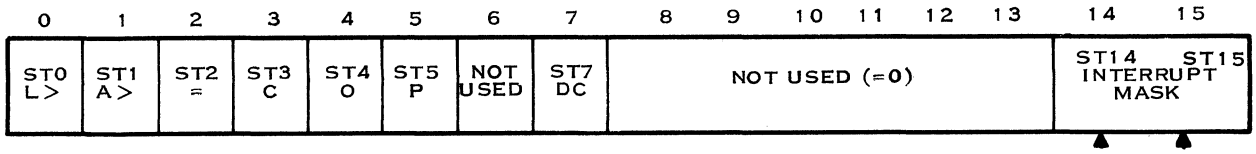
[<label>]b...LIIMb...<iop>b...[<comment>]

Example:

```
LABEL  LIIM  1          MASK LEVEL 2 AND 3
```

Definition: Place the low order two bits (bits 14 and 15) of the instruction in the least significant two bits of the status register. The remaining bits of the status register (bits 0 through 13) are not affected.

Status bits affected: Interrupt mask.



Execution results: The two least significant bits of the instruction code are placed into the interrupt mask, the two least significant bits of the ST register.

Application notes: Use the LIIM instruction to initialize the interrupt mask for a particular level of interrupt to be accepted. For example, the instruction

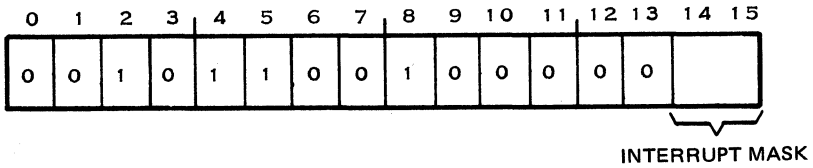
```
LIIM 2
```

sets the interrupt mask to level 2 and enables interrupts at levels 0, 1, and 2.

Op Code: 2C80

Addressing mode: Format VIII

Format:



L.4.2. DECIMAL CORRECT ADDITION DCA.

Syntax defintion:

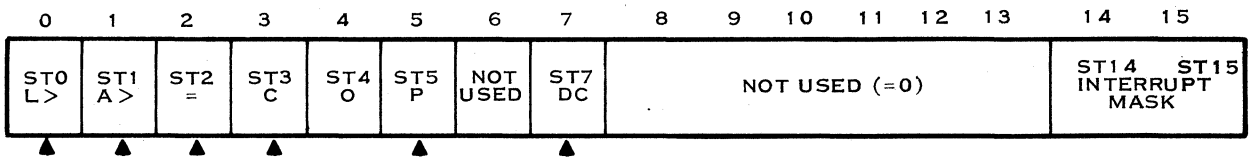
[<label>] ƀ...DCAƀ...<ga_S>ƀ...[<comment>]

Example:

```
LABEL   DCA  @RESULT  DECIMAL CORRECT FOR BCD ADDITION
```

Definition: The byte specified by the source operand is corrected to form two BCD digits as shown in table L-2. The result is then compared to zero and the status register set to indicate the result of the comparison. The carry bit is set if the most significant digit was BCD corrected. If the result has an odd number of ones, the parity bit is set. The digit carry bit is set if carry out of least significant BCD digit was a one.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, parity, and digit carry.



Execution results: See table L-2.

Application notes: Use the DCA instruction to aid in BCD addition. For example, if the high order bytes of R0 and R1 contain BCD (R0=0400, R1=0900), then the code to perform a BCD add with the result in R1 is:

```
AB    R0,R1   ADD BYTE
              (INTERMEDIATE RESULT IN R1 IS NOW 0D00)
DCA   R1      DECIMAL CORRECT FOR BCD ADDITION
              (RESULT IN R1 IS 1300)
```

Op code: 2C00

Addressing mode: Format VI

Format:

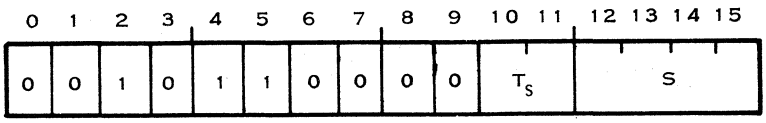
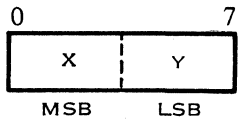


Table L-2. Function of DCA Instruction (Assume the Specified Byte Contains Two BCD Digits, X and Y)



Byte Before Execution				Byte After DCA			
Status Bits		BCD Digits		Status Bits		BCD Digits	
C	DC	X	Y	C	DC	X	Y
0	0	X<10	Y<10	0	0	X	Y
0	1	X<10	Y<10	0	0	X	Y+6
0	0	X<9	Y≥10	0	1	X+1	Y+6
1	0	X<10	Y<10	1	0	X+6	Y
1	1	X<10	Y<10	1	0	X+6	Y+6
1	0	X<10	Y≥10	1	1	X+7	Y+6
0	0	X≥10	Y<10	1	0	X+6	Y
0	1	X≥10	Y<10	1	0	X+6	Y+6
0	0	X≥9	Y≥10	1	1	X+7	Y+6

L.4.3 DECIMAL CORRECT SUBTRACTION DCS.

Syntax defintion:

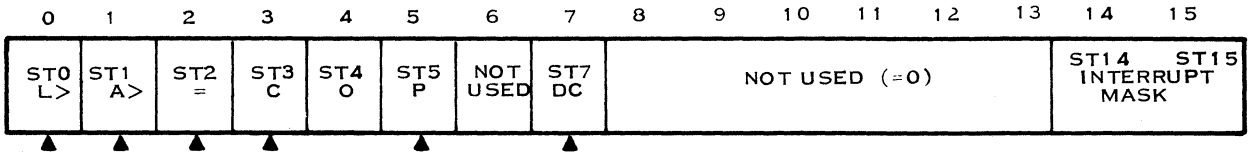
[<label>]ƀ...DCSƀ...<ga_S>ƀ...[<comment>]

Example:

LABEL DCS @RESULT DECIMAL CORRECT FOR BCD SUBTRACTION

Definition: The byte specified by the source operand is corrected to form two BCD digits as shown in table L-2. The result is then compared to zero and the status register set to indicate the result of the comparison. The carry bit is set if the most significant digit was not BCD corrected. If the result has an odd number of ones, the parity bit is set. The digit carry bit is set if carry out of least significant BCD digit was a one.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, parity, and digit carry.



Execution results: See table L-3.

Application notes: Use the DCS instruction to aid in BCD subtraction. For example, if the high order bytes of R0 and R1 contain BCD (R0=0400, R1=1300), then the code to perform a BCD subtract with the result in R1 is:

```
SB    R0,R1    SUBTRACT BYTE
               (INTERMEDIATE RESULT IN R1 IS NOW 0F00)
DCS   R1       DECIMAL CORRECT FOR BCD SUBTRACTION
               (RESULT IN R1 IS 0900)
```

Op Code: 2C40

Addressing mode: Format VI

Format:

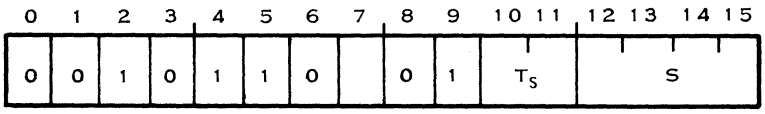
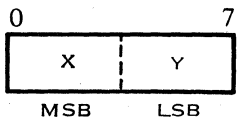


Table L-3. Function of DCS Instruction (Assume the Specified Byte Contains Two BCD Digits)



Byte Before Execution				Byte After DCS			
Status Bits		BCD Digits		Status Bits		BCD Digits	
C	DC	X	Y	C	DC	X	Y
0	0	X	Y	0	1	X+10	Y+10
0	1	X	Y	0	0	X+10	Y
1	0	X	Y	1	1	X	Y+10
1	1	X	Y	1	0	X	Y

ALPHABETICAL INDEX

ALPHABETICAL INDEX

INTRODUCTION

The following index lists key words and concepts from the subject material of the manual together with the area(s) in the manual that supply major coverage of the listed concept. The numbers along the right side of the listing reference the following manual areas:

- Sections - References to Sections of the manual appear as "Section x" with the symbol x representing any numeric quantity.
- Appendixes - References to Appendixes of the manual appear as "Appendix y" with the symbol y representing any capital letter.
- Paragraphs - References to paragraphs of the manual appear as a series of alphanumeric or numeric characters punctuated with decimal points. Only the first character of the string may be a letter; all subsequent characters are numbers. The first character refers to the section or appendix of the manual in which the paragraph is found.
- Tables - References to tables in the manual are represented by the capital letter T followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the table). The second character is followed by a dash (-) and a number:

  Tx-yy

- Figures - References to figures in the manual are represented by the capital letter F followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the figure). The second character is followed by a dash (-) and a number:

  Fx-yy

- Other entries in the Index - References to other entries in the index are preceded by the word "See" followed by the referenced entry.

USER'S RESPONSE SHEET

Manual Title: Model 990 Computer TMS 9900 Microprocessor Assembly Language Programmer's Guide (943441-9701)

Manual Date: 15 October 1978 **Date of This Letter:** _____

User's Name: _____ **Telephone:** _____

Company: _____ **Office/Department:** _____

Street Address: _____

City/State/Zip Code: _____

Please list any discrepancy found in this manual by page, paragraph, figure, or table number in the following space. If there are any other suggestions that you wish to make, feel free to include them. Thank you.

Location in Manual	Comment/Suggestion
_____	_____
	_____
	_____
	_____
_____	_____
	_____
	_____
	_____
_____	_____
	_____
	_____
	_____

CUT ALONG LINE

NO POSTAGE NECESSARY IF MAILED IN U.S.A.
FOLD ON TWO LINES (LOCATED ON REVERSE SIDE), TAPE AND MAIL

FOLD

FIRST CLASS
PERMIT NO. 7284
DALLAS, TEXAS

BUSINESS REPLY MAIL
NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY

TEXAS INSTRUMENTS INCORPORATED
DIGITAL SYSTEMS DIVISION

P.O. BOX 2909 · AUSTIN, TEXAS 78769

ATTN: TECHNICAL PUBLICATIONS
MS 2146

FOLD

TEXAS INSTRUMENTS
INCORPORATED
DIGITAL SYSTEMS DIVISION
POST OFFICE BOX 2909 AUSTIN, TEXAS 78769

12.90