CHAPTER 6

Instruction Set

6◂

SOFTWARE FEATURES OF THE 9900

In order to understand the operation of the 9900 instructions, the basic software features of the 9900 must be understood. These features include the processor-memory interrelationships, the available addressing modes, the terminology and formats used in the 9900 assembly language, and the interrupt and subroutine procedures used by the 9900.

Processor Registers and System Memory

There are three registers in the 9900 that are of interest to the programmer; their functions are illustrated in *Figure 6-1:*

Program Counter—This register contains the address of the instruction to be executed by the 9900. This instruction address can point to or locate an instruction anywhere in system memory, though instructions normally are not placed in the first 64 words of memory. These locations are reserved for interrupt and extended operation transfer vectors.

Workspace Pointer—This register contains the address of the first word of a group of 16 consecutive words of memory called a workspace. The workspace can be located anywhere in memory that is not already dedicated to transfer vector or program storage. These 16 workspace words are called workspace registers 0 through 15, and are treated by the 9900 processor as data registers much as other processors treat on-chip data registers for high access storage requirements.

▸6 *Status Register*—The status register stores the summary of the results of processor operations, including such information as the arithmetic or logical relation of the result to some reference data, whether or not the result can be completely contained in a 16-bit data word, and the parity of the result. The last bits of the status register contain the system interrupt mask which determines which interrupts will be responded to.

These three 16-bit registers completely define the current state of the processor: what part of the overall program is being executed, where the general purpose workspace is located in memory, and what the current status of operations and the interrupt system is. This information completely defines the current program environment or context of the system. A change in the program counter contents and workspace register contents switches the program environment or context to a new part of program memory with a new workspace area. Performing such a context switch or change in program environment is a very efficient method of handling subroutine jumps to subprograms that require the use of a majority of the workspace registers.

Program Counter

Figure 6-1 illustrates the use of the three processor registers. The program counter is the pointer which locates the instruction to be executed. All instructions require one or more 16-bit words and are always located at *even* addresses. Multiple word instructions include one 16-bit operation word and one or two 16-bit operand addresses. Two of the processors in the 9900 family (TMS9900, SBP9900) employ a 16-bit data bus and receive the instructions 16 bits at a time. The other processors (TMS9980A/81, TMS9985, TMS9940) use an 8-bit data bus and require extra memory cycles to fetch instructions. In both cases the even and odd bytes are located at even and odd addresses respectively as illustrated in *Figure 6-2.* In addition, data may be stored as 16-bit words located at even addresses or as 8-bit bytes at either even or odd addresses.

Workspace

The workspace is a set of 16 contiguous words of memory, the first of which is located by the workspace pointer. The individual 16-bit words, called workspace registers, are located at even addresses (see *Figure 6-1*). All of the registers are available for use as general registers; however, some instructions make use of certain registers as illustrated in *Figure 6-3.* Care should be exercised when using these registers for data or addresses not related to their special functions.

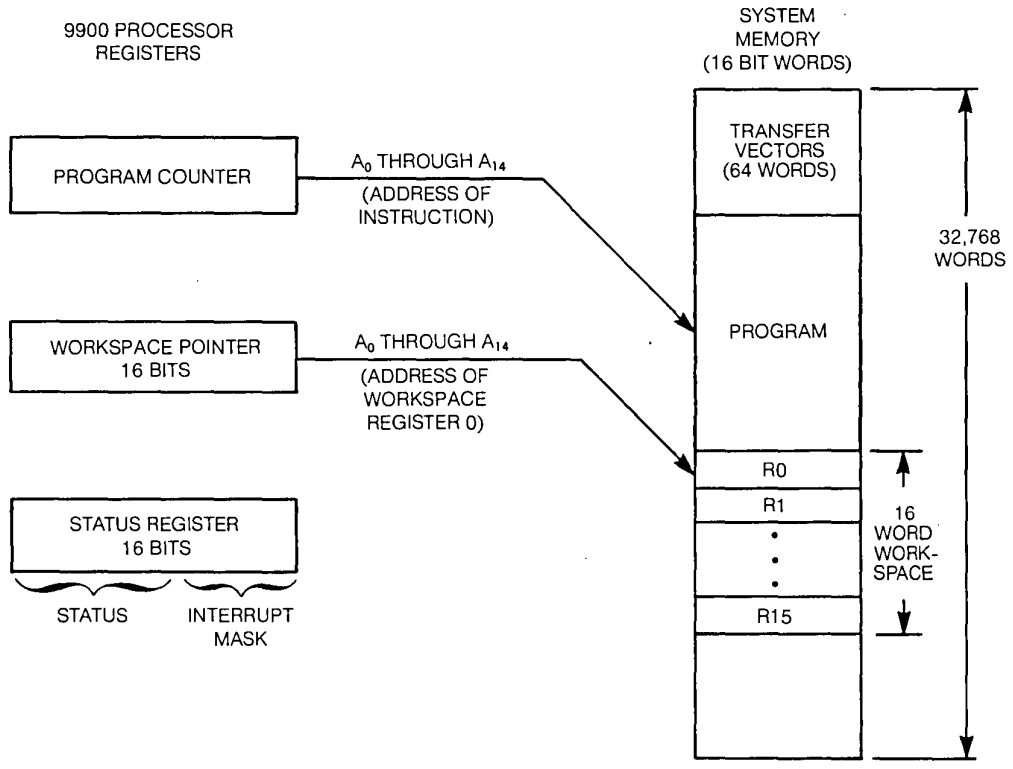


Figure 6-1. 9900 System Memory and Processor Registers.

Status Register

The status register contents for the 9900 are defined in *Figure 6-4.* The 9900 interrupt mask is a 4-bit code, allowing the specification of 16 levels of interrupt. Interrupt levels equal to or less than the mask value will be acknowledged and responded to by the 9900. The 9940 status register is similar, except the interrupt mask occupies bits 14 and 15 of the status register, providing for four interrupt levels in the 9940.

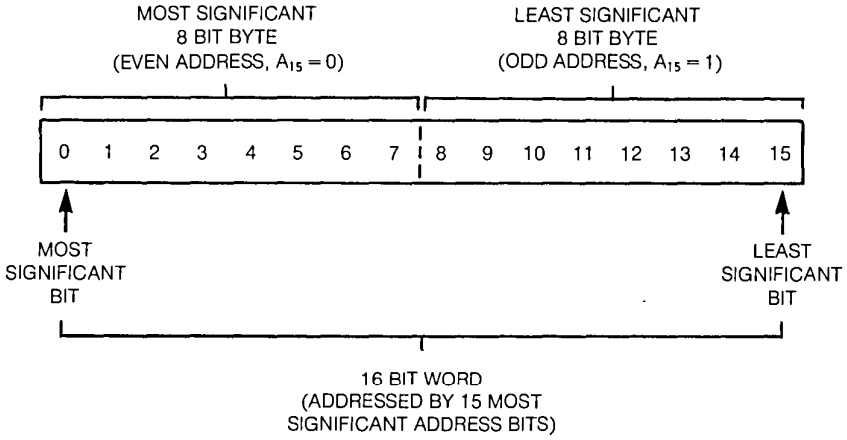


Figure 6-2. Word and Byte Definition.

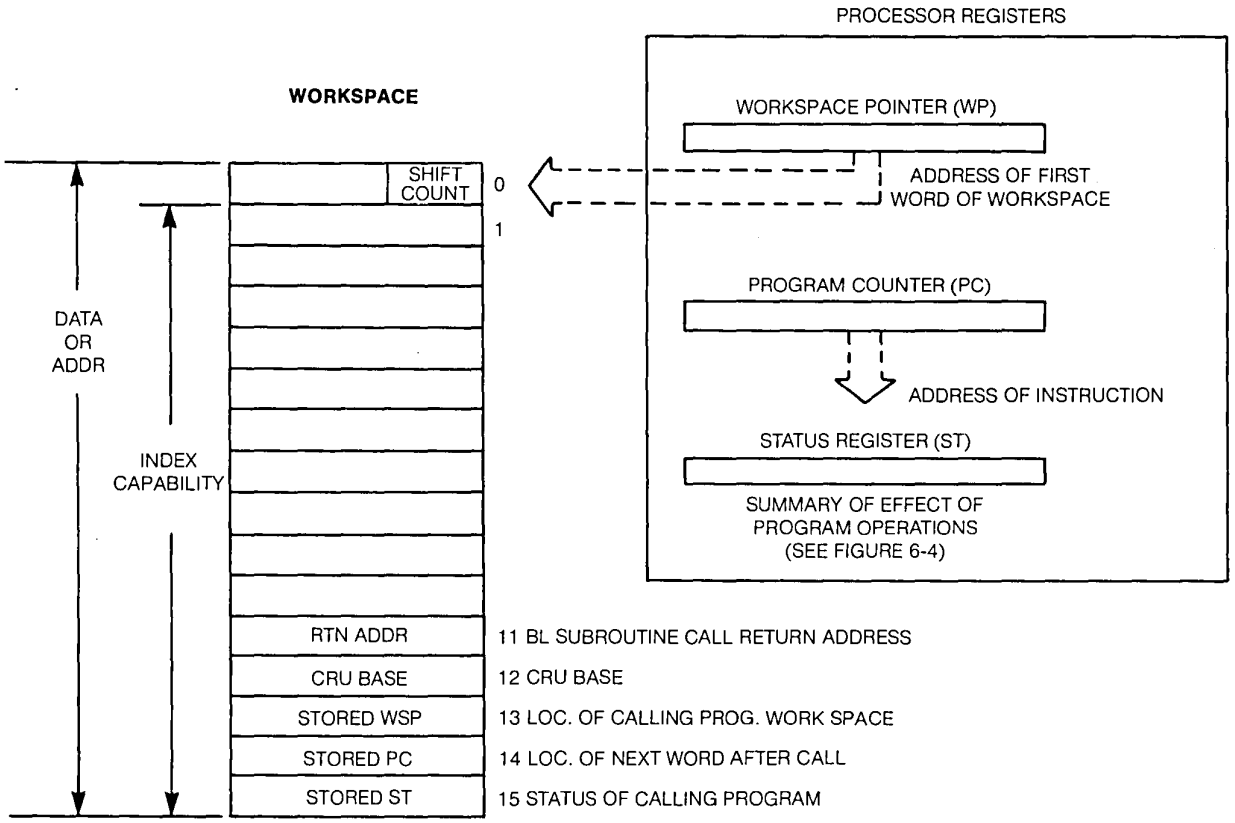
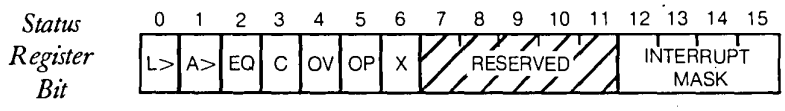


Figure 6-3. Workspace Register Utilization.



0 LGT — *Logical Greater Than*—set in a comparison of an unsigned number with a smaller unsigned number.

1 AGT — *Arithmetic Greater Than*—set when one signed number is compared with another that is less positive (nearer to −32,768).

2 EQ — *Equal*—set when the two words or two bytes being compared are equal.

3 C — *Carry*—set by carry out of most significant bit of a word or byte in a shift or arithmetic operation.

4 OV — *Overflow*—set when the result of an arithmetic; operation is too large or too small to be correctly represented in 2's complement form. OV is set in addition if the most significant bit of the two operands are equal and the most significant bit of the sum is different from the destination operand most significant bit. OV is set in subtraction if the most significant bits of the operands are not equal and the most significant bit of the result is different from the most significant bit of the destination operand. In single operand instructions affecting OV, the OV is set if the most significant bit of the operand is changed by the instruction.

5 OP — *Odd Parity*—set when there is an odd number of bits set to one in the result.

6 X — *Extended Operation*—set when the PC and WP registers have been to set to values of the transfer vector words during the execution of an extended operation.

7-11 Reserved for special Model 990/10 computer applications.

12-15 *Interrupt Mask*—All interrupts of level equal to or less than mask value are enabled.

Figure 6-4. 9900 Status Register Contents

Addressing Modes

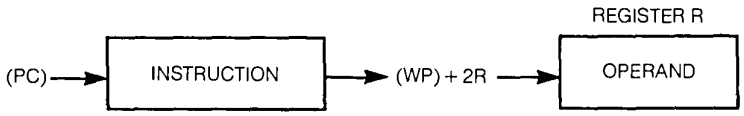
The 9900 supports five general purpose addressing modes or methods of specifying the location of a memory word:

Workspace Register Addressing

The data or address to be used by the instruction is contained in the workspace register number specified in the operand field of the instruction. For example, if the programmer wishes to decrement the contents of workspace register 2, the format of the decrement instruction would be:

```
DEC     2
```

The memory address of the word to be used by the instruction is computed as follows:



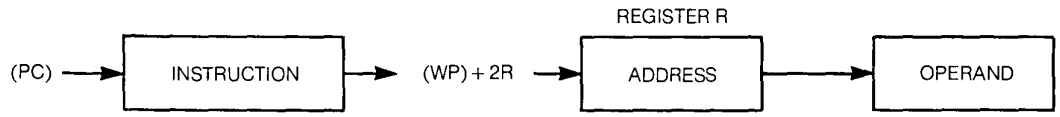
This type of addressing is used to access the often used data contained in the workspace.

Workspace Register Indirect Addressing

The address of the data to be used by the instruction is contained in the workspace register specified in the operand field (the workspace register number is preceded by an asterisk). This type of addressing is used to establish data counters so the programmer can sequence through data stored in successive locations in memory. If register 3 contains the address of the data word to be used, the following instruction would be used ▸6 to clear (CLR) that data word:

```
CLR     *3
```

In this instruction the contents of register 3 would not be changed, but the data word addressed by the contents of register 3 would be cleared (set to all zeroes — 000_{16}). The word address is computed as follows for this type of addressing:

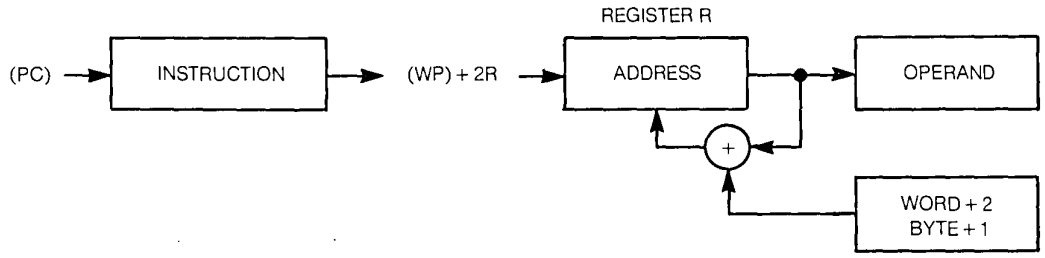


Workspace Register Indirect Addressing With Autoincrement—

This addressing mode locates the data word in the same way that workspace register indirect addressing does, with the added feature of incrementing the contents of the address register after the instruction has been completed. The address in the register is incremented by one if a byte operation is performed and by two if a word operation is performed. Thus, to set up a true data counter to clear a group of successive words in memory whose address will be contained in register 3, the following instruction would be used:

```
CLR     *3+
```

where the asterisk (*) indicates the workspace register indirect addressing feature and the plus (+) indicates the autoincrementing feature. With this type of addressing, the following computations occur:



Symbolic or Direct Addressing

The address of the memory word is contained in the operand field of the instruction and is contained in program memory (ROM) in the word immediately following the operation code word for the instruction. For example, to clear the memory word at address 1000_{16}, the following format would be used:

```
CLR     @>1000
```

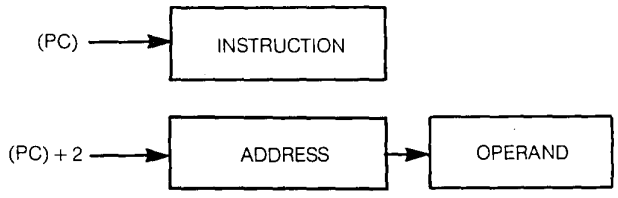
where the at sign (@) indicates direct addressing and the greater than (>) sign indicates a base 16 (hexadecimal) constant. Alternatively, the data word to be cleared could be named with a symbolic name such as COUNT and then the instruction would be:

```
CLR     @COUNT
```

and if COUNT is later equated to 1000_{16}, this instruction would clear the data word at address 1000_{16}. The instruction would occupy two words of program memory:

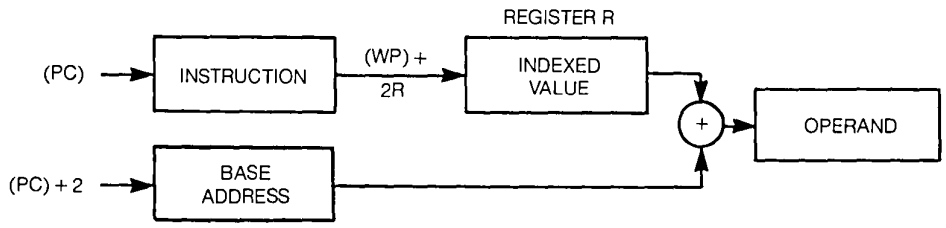
(PC)	$04C0_{16}$	Operation Code for Clear
(PC) + 2	1000_{16}	Address of Data

The address of the memory word is thus contained in the instruction itself and is located by the program counter. Since this address is part of the instruction, it cannot be modified by the program. As a result, this type of addressing is used for program variables that occupy a single memory word such as program counters, data masks, and so on. The address computations for direct addressing are as follows:



Indexed Addressing

Indexed addressing is a combination of symbolic and register indirect addressing. It provides for address modification since part of the address is contained in the workspace register used as an index register. Registers 1 through 15 can be used as index registers. The memory word address is obtained by adding the contents of the index register specified to the constant contained in the instruction:



Thus, to locate the data word whose address is two words down from the address contained in register 5, and to clear this memory word, the following instruction is used:

```
CLR     @4(5)
```

This instruction will cause the processor to add 4 to the contents of register 5 to generate the desired address. Alternatively, a symbolic name could be used for the instruction constant:

▸6

```
CLR     @DISP(5)
```

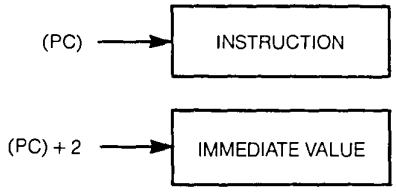
with the value for the symbol DISP defined elsewhere in the assembly language program.

Special Addressing Modes

Three additional types of special purpose addressing are used by the 9900.

Immediate Addressing

Immediate addressing instructions contain the data to be used as a part of the instruction. In these instructions the first word is the instruction operation code and the second word of the instruction is the data to be used:



Program Counter Relative Addressing

Conditional branch or jump instructions use a form of program counter relative addressing. In such instructions the address of the instruction to be branched to is relative to the location of the branch instruction. The instruction includes a signed displacement with a value between -128 and $+127$. The branch address is the value of the program counter plus two plus twice the displacement. For example, if LOOP is the label at location 10_{16} and the instruction:

```
JMP     LOOP
```

is at location 18_{16}, the displacement in the instruction machine code generated by the assembler will be -5 or FB_{16}. This value is obtained by adding two to the current program counter:

$$18_{16} + 2 = 1A_{16}$$

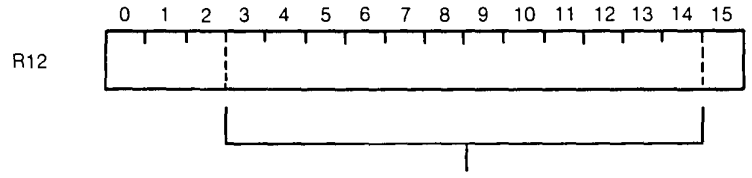
and subtracting from this result the location of LOOP:

$$1A_{16} - 10_{16} = A_{16} = 10 \text{ decimal.}$$

The displacement of 5 is one-half this value of 10 and it is negative since LOOP is 5 words prior to the $18_{16} + 2$ location.

CRU Addressing

CRU addressing uses the number contained in bits 3 through 14 of register 12 to form a hardware base address:



CRU Hardware Base address = Contents of R12 divided by 2

Thus if R12 contains 0400_{16} (the software base address), bits 3 through 14 will be 0200_{16}. This hardware base address is used to indicate the starting CRU bit address for multiple bit CRU transfer instructions (STCR and LDCR). It is added to the displacement contained in single bit CRU instructions (TB, SBO, SBZ) to form the CRU bit address for these instructions. For example, to set CRU bit 208 to a one, with register 12 containing 400_{16}, the following CRU instruction would be used:

```
SBO     8
```

so that the CRU bit address is $200_{16} + 8. = 208_{16}$.

ASSEMBLY LANGUAGE PROGRAMMING INFORMATION †

In order to understand the instruction descriptions and applications the assembly language nomenclature must be understood. Assembly language is a readily understood language in which the 9900 instructions can be written. The machine code that results from the assembly of programs written in this language is called object code. Such object code may be absolute or relocatable, depending on the assembly language coding. Relocatable code is that which can be loaded into any block of memory desired, without reassembling or without changing program operation. Such code has its address information relative to the first instruction of the assembly language program so that once a loader program specifies the location of this first instruction, the address of all instructions are adjusted to be consistent with this location. Absolute code contains absolute addresses which cannot be changed by the loader or any operation other than reassembling the program. Generally, relocatable code is preferable since it allows the program modules to be located anywhere in memory of the final system.

Assembly Language Formats

The general assembly language source statements consist of four fields as follows:

LABEL MNEMONIC OPERANDS COMMENT

The first three fields must occur within the first 60 character positions of the source record. At least one blank must be inserted between fields.

Label Field

·6 The label consists of from one to six characters, beginning with an alphabetic character in character position one of the source record. The label field is terminated by at least one blank. When the assembler encounters a label in an instruction it assigns the current value of the location counter to the label symbol. This is the value associated with the label symbol and is the address of the instruction in memory. If a label is not used, character position 1 may be a blank, or an asterisk.

Mnemonic or Opcode Field

This field contains the mnemonic code of one of the instructions, one of the assembly language directives, or a symbol representing one of the program defined operations. This field begins after the last blank following the label field. Examples of instruction mnemonics include A for addition and MOV for data movement. The mnemonic field is required since it identifies which operation is to be performed.

Operands Field

The operands specify the memory locations of the data to be used by the instruction. This field begins following the last blank that follows the mnemonic field. The memory locations can be specified by using constants, symbols, or expressions, to describe one of several addressing modes available. These are summarized in *Figure 6-5.*

†Excerpts from Model 990 computer TMS 9900 Microprocessor Assembly Language Programmer's Guide.

Type of Addressing	*Operand Format*	*Memory Location Specified*	*MOV Instruction Example Coding*		*Result*	*T_d or T_s Field Code*
Workspace Register	n	Workspace Register n Rn	MOV	3,5	R3 ⟶ R5	00
Workspace Register Indirect	*n	Address given by the contents of workspace register n M(Rn)	MOV	*3,*5	M(R3) ⟶ M(R5)	01
Workspace Register Indirect, Autoincrement	*n+	As in register Indirect; address register Rn is incremented after the operation (by one for byte operations, by two for word operations)	MOV	*3+,*5+	M(R3) ⟶ M(R5) R3+2 ⟶ R3 R5+2 ⟶ R5	11
Symbolic Memory	@exp	Address is given by value of exp. M(exp)	MOV	@ONE, @10	M(ONE) ⟶ M(10)	10
Indexed Memory	@exp(n)	Address is the sum of the contents of Rn and the value of exp M(Rn+exp)	MOV	@2(3), @DP(5)	M(R3+2) ⟶ M(R5+DP)	10

Notes:

n is the number of the workspace register: $0 \leq n \leq 15$; n may not be 0 for indexed addressing.

exp is a symbol, number, or expression

The T_d and T_s fields are two bit portions of the instruction machine code. There are also S and D four bit fields, which are filled in with the four bit code for n. n is 0 for symbolic or direct addressing.

Figure 6-5. Addressing Modes

Comments Field

Comments can be entered after the last blank that follows the operands field. If the first character position of the source statement contains an asterisk (*), the entire source statement is a comment. Comments are listed in the source portion of the assembler listing, but have no affect on the object code.

Terms and Symbols

Symbols are used in the label field, the operator field, and the operand field. A symbol is a string of alphanumeric characters, beginning with an alphabetic character.

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

Constants

Constants can be decimal integers (written as a string of numerals) in the range of $-32{,}768$ to $+65{,}535$. For example:

257

Constants can also be hexadecimal integers (a string of hexadecimal digits preceded by >). For example:

▸6 >09AF

ASCII character constants can be used by enclosing the desired character string in single quotes. For example:

'DX' = 4458_{16} 'R' + 0052_{16}

Throughout this book the subscript 16 is used to denote base 16 numbers. For example, the hexadecimal number 09AF will be written $09AF_{16}$.

Symbols

Symbols must begin with an alphabetic character and contain no blanks. Only the first six characters of a symbol are processed by the assembler.

The assembler predefines the dollar sign ($) to represent the current location in the program.

A given symbol can be used as a label only once, since it is the symbolic name of the address of the instruction. Symbols defined with the DXOP directive are used in the OPCODE field. Any symbol in the OPERANDS field must have been used as a label or defined by a REF directive.

Expressions

Expressions are used in the OPERANDS fields of assembly language statements. An expression is a term or a series of terms separated by the following arithmetic operations:

+ addition

− subtraction

* multiplication

/ division

The operator precedence is +, −, *, / (left to right).

The expression must not contain any imbedded blanks or extended operation defined (DXOP directive defined) symbols. Unary minus (a minus sign in front of a number or symbol) is performed first and then the expression is evaluated from left to right. An example of the use of the unary minus in an expression is:

LABEL+TABLE+(−INC)

which has the effect of the expression:

LABEL+TABLE−INC

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 expressions. All other expressions are absolute. The expression given earlier would be relocatable if the three symbols in the expression are all relocatable. 6◄

The following are examples of valid expressions.

BLUE+1

2*16+RED

440/2−RED

Survey of the 9900 Instruction Set

The 9900 instructions can be grouped into the following general categories: data transfer, arithmetic, comparison, logical, shift, branch, and CRU input/output operations. The list of all instructions and their effect on status bits is given in *Figure 6-6.*

Mnemonic	L>	A>	EQ	C	OV	OP	X
A	X	X	X	X	X	-	-
AB	X	X	X	X	X	X	-
ABS	X	X	X	X	X	-	-
AI	X	X	X	X	X	-	-
ANDI	X	X	X	-	-	-	-
B	-	-	-	-	-	-	-
BL	-	-	-	-	-	-	-
BLWP	-	-	-	-	-	-	-
C	X	X	X	-	-	-	-
CB	X	X	X	-	-	X	-
CI	X	X	X	-	-	-	-
CKOF	-	-	-	-	-	-	-
CKON	-	-	-	-	-	-	-
CLR	-	-	-	-	-	-	-
COC	-	-	X	-	-	-	-
CZC	-	-	X	-	-	-	-
DEC	X	X	X	X	X	-	-
DECT	X	X	X	X	X	-	-
LDCR	X	X	X	-	-	1	-
LI	X	X	X	-	-	-	-
LIMI	-	-	-	-	-	-	-
LREX	-	-	-	-	-	-	-
LWPI	-	-	-	-	-	-	-
MOV	X	X	X	-	-	-	-
MOVB	X	X	X	-	-	X	-
MPY	-	-	-	-	-	-	-
NEG	X	X	X	X	X	-	-
ORI	X	X	X	-	-	-	-
RSET	-	-	-	-	-	-	-
RTWP	X	X	X	X	X	X	X
S	X	X	X	X	X	-	-
SB	X	X	X	X	X	X	-
SBO	-	-	-	-	-	-	-

Mnemonic	L>	A>	EQ	C	OV	OP	X
DIV	-	-	-	-	X	-	-
IDLE	-	-	-	-	-	-	-
INC	X	X	X	X	X	-	-
INCT	X	X	X	X	X	-	-
INV	X	X	X	-	-	-	-
JEQ	-	-	-	-	-	-	-
JGT	-	-	-	-	-	-	-
JH	-	-	-	-	-	-	-
JHE	-	-	-	-	-	-	-
JL	-	-	-	-	-	-	-
JLE	-	-	-	-	-	-	-
JLT	-	-	-	-	-	-	-
JMP	-	-	-	-	-	-	-
JNC	-	-	-	-	-	-	-
JNE	-	-	-	-	-	-	-
JNO	-	-	-	-	-	-	-
JOC	-	-	-	-	-	-	-
JOP	-	-	-	-	-	-	-
SBZ	-	-	-	-	-	-	-
SETO	-	-	-	-	-	-	-
SLA	X	X	X	X	X	-	-
SOC	X	X	X	-	-	-	-
SOCB	X	X	X	-	-	X	-
SRA	X	X	X	X	-	-	-
SRC	X	X	X	X	-	-	-
SRL	X	X	X	X	-	-	-
STCR	X	X	X	-	-	1	-
STST	-	-	-	-	-	-	-
STWP	-	-	-	-	-	-	-
SWPB	-	-	-	-	-	-	-
SZC	X	X	X	-	-	-	-
SZCB	X	X	X	-	-	X	-
TB	-	-	X	-	-	-	-
X	2	2	2	2	2	2	2
XOP	2	2	2	2	2	2	2
XOR	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.

Figure 6-6. Status Bits Affected by Instructions

Data Transfer Instructions

Load— used to initialize processor or workspace registers to a desired value.

Move— used to move words or bytes from one memory location to another.

Store— used to store the status or workspace pointer registers in a workspace register.

Arithmetic Instructions

Addition and Subtraction—perform addition or subtraction of signed or unsigned binary words or bytes stored in memory.

Negate and Absolute Value—changes the sign or takes the absolute value of data words in memory.

Increment and Decrement—Adds or subtracts 1 or 2 from the specified data words in memory.

Multiply—Performs unsigned integer multiplication of a word in memory with a workspace register word to form a 32 bit product stored in two successive workspace register locations.

Divide—Divides a 32 bit unsigned integer dividend (contained in two successive workspace registers) by a memory word with the 16 bit quotient and 16 bit remainder stored in place of the dividend.

Compare Instructions

These instructions provide for masked or unmasked comparison of one memory word or byte to another or a workspace register word to a 16 bit constant.

6

Logical Instructions

OR and AND—masked or unmasked OR and AND operations on corresponding bits of two memory words. A workspace register word can be ORed or ANDed with a 16 bit constant.

Complement and Clear — The bits of a selected memory word can be complemented, or cleared or set to ones.

Exclusive OR—A workspace register word can be exclusive ORed with another memory word on a bit by bit basis.

Set Bits Corresponding—Set bits to one (SOC) or to zero (SZC) whose positions correspond to one positions in a reference word.

Shift Instructions

A workspace register can be shifted arithmetically or logically to the right. The registers can be shifted to the left (filling in vacated positions with zeroes) or circulated to the right. The shifts and circulates can be from 1 to 16 bit positions.

Branch Instructions

The branch instructions and the JMP (jump) instruction unconditionally branch to different parts of the program memory. If a branch occurs, the PC register will be changed to the value specified by the operand of the branch instruction. In subroutine branching the old value of the PC is saved when the branch occurs and then is restored when the return instruction is executed. The conditional jump instructions test certain status bits to determine if jump is to occur. When a jump is made the PC is loaded with the sum of its previous value and a displacement value specified in the operand portion of the instruction.

Control/CRU Instructions

These instructions provide for transferring data to and from the communications register input/output unit (CRU) using the CRUIN, CRUOUT and CRUCLK pins of the 9900.

Instruction Descriptions

The information provided for each instruction in the next section of this chapter is as follows:

▸6

Name of the instruction.

Mnemonic for the instruction.

Assembly language and machine code formats.

Description of the operation of the instruction.

Effect of the instruction on the Status Bits.

Examples.

Applications.

The format descriptions and examples are written without the label or comment fields for simplicity. Labels and comments fields can be used in any instruction if desired.

Each instruction involves one or two operand fields which are written with the following symbols:

G—Any addressing mode is permitted except I (Immediate).

R—Workspace register addressing.

exp—A symbol or expression used to indicate a location.

value—a value to be used in immediate addressing.

cnt—A count value for shifts and CRU instructions.

CRU—CRU (Communications Register Unit) bit addressing.

The instruction operation is described in written and equation form. In the equation form, an arrow(⟶) is used to indicate a transfer of data and a colon (:) is used to indicate a comparison. In comparisons, the operands are not changed. In transfers, the source operand (indicated with the subscript s) is not changed while the destination operand (indicated with the subscript d) is changed. For operands specified by the symbol G, the M(G) nomenclature is used to denote the memory word specified by G. MB(G) is used to denote the memory byte specified by G. Thus, transferring the memory word contents addressed by G_s to the memory word location specified by G_d and comparing the source (G_s) data to zero during the transfer, can be described as:

$M(G_s) \longrightarrow M(G_d)$

$M(G_s):0$

6◀

which is the operation performed by the MOV instruction:

MOV G_s,G_d

A specific example of this instruction could be:

MOV @ONE,3

which moves the contents of the memory word addressed by the value of the symbol ONE to the contents of workspace register 3:

M(ONE)⟶R3

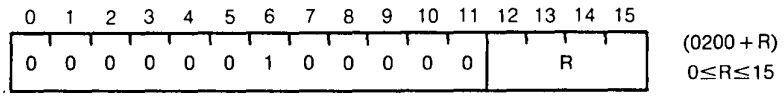
M(ONE) : 0

DATA TRANSFER INSTRUCTIONS

The MOV instructions are used to transfer data from one part of the system to another part. The LOAD instructions are used to initialize registers to desired values. The STORE instructions provide for saving the status register (ST) or the workspace pointer (WP) in a specified workspace register.

LOAD IMMEDIATE — LI

Format: **LI R,value**



Operation: The 16 bit data value in the word immediately following the instruction is loaded into the specified workspace register R.

value ⟶ R
immediate operand: 0

Affect on Status: **LGT,AGT,EQ**

Examples: **LI 7,5** 5 ⟶ R7

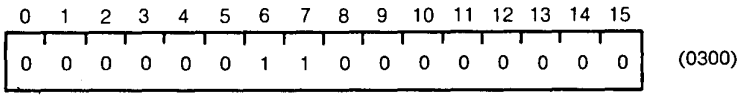
LI 8,>FF $00FF_{16}$ ⟶ R8

Applications: The LI instruction is used to initialize a workspace register with a program constant such as a counter value or data mask.

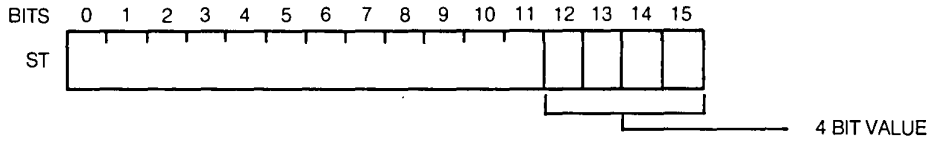
▸6

LOAD INTERRUPT MASK IMMEDIATE — LIMI

Format: **LIMI value**



Operation The low order 4 bit value (bits 12-15) in the word immediately following the instruction is loaded into the interrupt mask portion of the status register:



Affect on Status: Interrupt mask code only

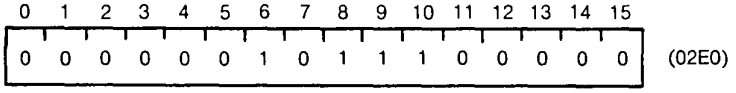
Example: **LIMI 5**

Enables interrupt levels 0 through 5

Application: The LIMI instruction is used to initialize the interrupt mask to control which system interrupts will be recognized.

LOAD WORKSPACE POINTER IMMEDIATE — LWPI

Format: **LWPI value**



Operation: The 16 bit value contained in the word immediately following the instruction is loaded into the workspace pointer (WP):

value ⟶ WP

Affect on Status: None

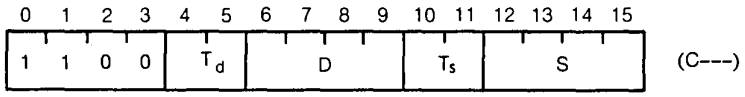
Example: **LWPI >0500**

Causes 0500_{16} to be loaded into the WP.

Application: LWPI is used to establish the workspace memory area for a section of the program.

MOVE WORD — MOV

Format: **MOV G_s,G_d**



Operation: The word in the location specified by G_s is transferred to the location specified by G_d, without affecting the data stored in the G_s location. During the transfer, the word (G_s data) is compared to 0 with the result of the comparison stored in the status register:

$M(G_s) \longrightarrow M(G_d)$
$M(G_s):0$

Status Bits Affected: **LGT, AGT, and EQ**

Examples:

MOV	**R1,R3**	R1 ⟶ R3, R1:0
MOV	***R1,R3**	M(R1) ⟶ R3, M(R1):0
MOV	**@ONES,*1**	M(ONES) ⟶ M(R1), M(ONES):0
MOV	**@2(5),3**	M(R5 + 2) ⟶ R3, M(R5 + 2):0
MOV	***R1 +,*R2 +**	M(R1) ⟶ M(R2), M(R1):0, (R1) + 2 ⟶ R1, (R2) + 2 ⟶ R2

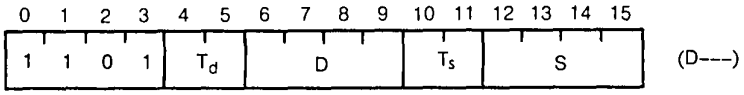
Application: MOV is used to transfer data from one part of the system to another part.

MOVB

MOVE BYTE

MOVB

Format: **MOVB G_s,G_d**



Operation: The Byte addressed by G_s is transferred to the byte location specified by G_d. If G is workspace register addressing, the most significant byte is selected. Otherwise, even addresses select the most significant byte; odd addresses select the least significant byte. During the transfer, the source byte is compared to zero and the results of the comparison are stored in the status register.

$MB(G_s) \longrightarrow MB(G_d)$
$MB(G_s):0$

Status Bits Affected: **LGT, AGT, EQ, OP**

Examples:
```
MOVB  @>1C14,3
MOVB  *8,4
```

These instructions would have the following example affects:

Memory Location	*Contents Initially*	*Contents After Transfer*
1C14	**2016**	**2016**
R3	**542B**	**202B**
R8	**2123**	**2123**
2123	**1040**	**1040**
R4	**0A0C**	**400C**

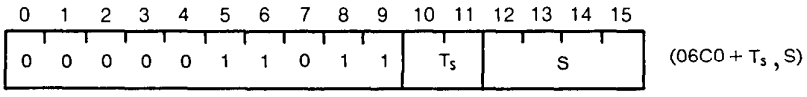
The underlined data are the bytes selected.

Application: MOVB is used to transfer 8 bit bytes from one byte location to another.

SWAP BYTES

SWPB

Format: **SWPB G**



Operation: The most significant byte and the least significant bytes of the word at the memory location specified by G are exchanged.

Affect on Status: None

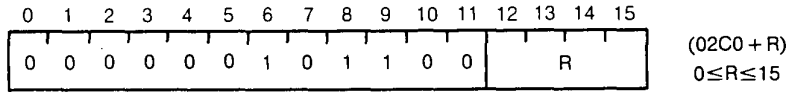
		Before	After
Example: **SWPB 3**	R3 Contents:	F302	02F3

Application: Used to interchange bytes if needed for subsequent byte operations.

STORE STATUS

STST

Format: **STST R**



Operation: The contents of the status register are stored in the workspace register specified:

ST ⟶ R

Affect on Status: None

Example: **STST 3** ST is transferred to R3

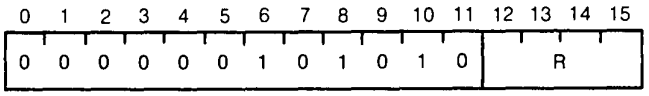
Application: STST is used to save the status for later reference.

STWP

Store Workspace Pointer

STWP

Format: **STWP R**



(02A0 + R)
0≤R≤15

Operation: The contents of the workspace pointer are stored in the workspace register specified:

WP ⟶ R

Affect on Status: None

Example: **STWP 3** WP is transferred into R3

Appliation: STWP is used to save the workspace pointer for later reference.

▸6

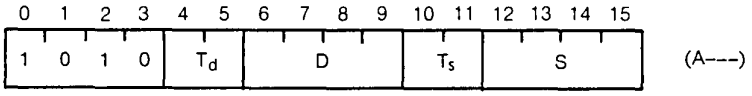
ARITHMETIC INSTRUCTIONS

These instructions perform the following basic arithmetic operations: addition (byte or word), subtraction (byte or word), multiplication, division, negation, and absolute value. More complicated mathematical functions must be developed using these basic operations. The basic instruction set will be adequate for many system requirements.

ADD WORDS

A

Format: **A** G_s,G_d



Operation: The data located at the address specified by G_s is added to the data located at the address specified by G_d. The resulting sum is placed in the G_d location and is compared to zero:

$M(G_s) + M(G_d) \longrightarrow M(G_d)$
$M(G_s) + M(G_d)$:0

Status Bits Affected: **LGT, AGT, EQ, C, OV**

Examples:

A	**5,@TABLE**	R5 + M(TABLE) ⟶ M(TABLE)
A	**3,*2**	R3 + M(R2) ⟶ M(R2)

with the sums compared to 0 in each case. Binary addition affects on status bits can be understood by studying the following examples:

M(Gs)	*M(Gd)*	*Sum*	*LGT*	*AGT***	*EQ*	*C*	*OV**
1000	**0001**	**1001**	**1**	**1**	**0**	**0**	**0**
F000	**1000**	**0000**	**0**	**0**	**1**	**1**	**0**
F000	**8000**	**7000**	**1**	**1**	**0**	**1**	**1**
4000	**4000**	**8000**	**1**	**0**	**0**	**0**	**1**

*OV (overflow) is set if the most significant bit of the sum is different from the most significant bit of $M(G_d)$ and the most significant bit of both operands are equal.

**AGT (arithmetic greater than) is set if the most significant bit of the sum is zero and if EQ (equal) is 0.

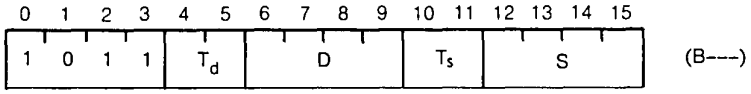
Application: Binary addition is the basic arithmetic operation required to generate many mathematical functions. This instruction can be used to develop programs to do multiword addition, decimal addition, code conversion, and so on.

AB

ADD BYTES

AB

Format: **AB** G_s,G_d



Operation: The source byte addressed by G_s is added to the destination byte addressed by G_d and the sum byte is placed in the G_d byte location. Recall that even addresses select the most significant byte and odd addresses select the least significant byte. The sum byte is compared to 0.

$MB(G_s) + MB(G_d) \longrightarrow MB(G_d)$
$MB(G_s) + MB(G_d):0$

Status Bits Affected: **LGT, AGT, EQ, C, OV, OP**

Example:

AB	**3,*4+**	R3 + MB(R4) ⟶ MB(R4),	R4 + 2 ⟶ R4
AB	**@TAB,5**	MB(TAB) + R5 ⟶	R5

To see how the AB works, the following example should be studied:

AB @>2120,@>2123

Memory Location	*Data Before Addition*	*Data After Addition*
2120	**F320**	**F320**
2123	**2106**	**21F9**

The underlined entries are the addressed and changed bytes.

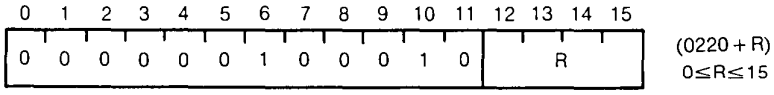
▸6

Application: AB is one of the byte operations available on the 9900. These can be useful when dealing with subsystems or data that use 8 bit units, such as ASCII codes.

ADD IMMEDIATE

AI

Format: **AI R,Value**



Operation: The 16 bit value contained in the word immediately following the instruction is added to the contents of the workspace register specified.

R + Value ⟶ R, R + Value:0

Status Bits Affected: **LGT, AGT, EQ, C, OV**

Example: **AI 6,>C**

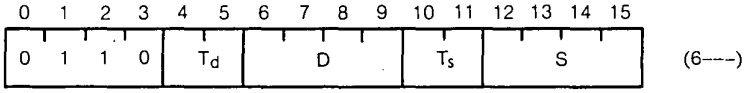
Adds C_{16} to the contents of workspace register 6. If R6 contains 1000_{16}, then the instruction will change its contents to $100C_{16}$, and the LGT and AGT status bits will be set.

Application: This instruction is used to add a constant to a workspace register. Such an operation is useful for adding a constant displacement to an address contained in the workspace register.

SUBTRACT WORDS

S

Format: **S G_s,G_d**



Operation: The source 16 bit data (location specified by G_s) is subtracted from the destination data (location specified by G_d) with the result placed in the destination location G_d. The result is compared to 0.

$M(G_d) - M(G_s) \longrightarrow M(G_d)$
$M(G_d) - M(G_s)$:0

Status Bits Affected: **LGT, AGT, EQ, C, OV**

Examples: **S @OLDVAL,@NEWVAL**
would yield the following example results:

Memory Location	*Before Subtraction Contents*	*After Subtraction Contents*
OLDVAL	**1225**	**1225**
NEWVAL	**8223**	**6FFE (8223-1225)**

All status bits affected would be set to 1 except equal which would be reset to 0.

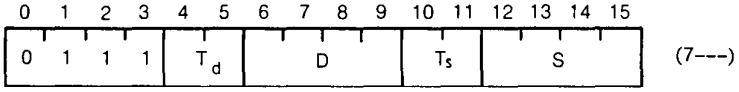
Application: Provides 16 bit binary subtraction.

SB

SUBTRACT BYTES

SB

Format: **SB** G_sG_d



Operation: The source byte addressed by G_s is subtracted from the destination byte addressed by G_d with the result placed in byte location G_d. The result is compared to 0. Even addresses select the most significant byte and odd addresses select the least significant byte. If workspace register addressing is used, the most significant byte of the register is used.

$MB(G_d) - MB(G_s) \longrightarrow MB(G_d)$
$MB(G_d) - MB(G_s):0$

Status Bits Affected: **LGT, AGT, C, EQ, OV, OP**

Format: **SB** ***6+,1**

$R1 - MB(R6) \longrightarrow R1$
$R1 - MB(R6):0$
$R6 + 1 \longrightarrow R6$

This operation would have the following example result:

Memory Location	*Contents Before Instruction*	*Contents After Instruction*
R6	**121D**	**121E**
121D	**3123**	**4123**
R1	**1344**	**F044**

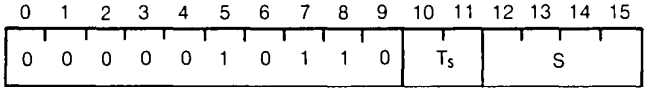
▶6 The underlined entries indicated the addressed and changed bytes. The LGT (logical greater than) status bit would be set to 1 while the other status bits affected would be 0.

Application: SB provides byte subtraction when 8 bit operations are required by the system.

INCREMENT

INC

Format: **INC G**



Operation: The data located at the address indicated by G is incremented and the result is placed in the G location and compared to 0.

M(G) + 1 ⟶ M(G)
M(G) + 1 : 0

Status Bits Affected: **LGT, AGT, EQ, C, OV**

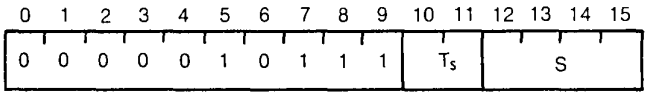
Examples: **INC @TABL** M(TABL) + 1 ⟶ M(TABL)
INC 1 (R1) + 1 ⟶ R1

Application: INC is used to increment byte addresses and to increment byte counters. Autoincrementing addressing on byte instructions automatically includes this operation.

INCREMENT BY TWO

INCT

Format: **INCT G**



Operation: Two is added to the data at the location specified by G and the result is stored at the G location and is compared to 0:

M(G) + 2 ⟶ M(G)
M(G) + 2 : 0

Status Bits Affected: **LGT, AGT, EQ, C, OV**

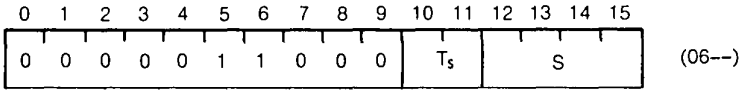
Example: **INCT 5** (R5) + 2 ⟶ R5

Application: This can be used to increment word addresses, though autoincrementing on word instructions does this automatically.

6◀

DECREMENT — DEC

Format: **DEC G**



Operation: One is subtracted from the data at the location specified by G, the result is stored at that location and is compared to 0:

M(G) – 1 ⟶ M(G)
M(G) – 1 : 0

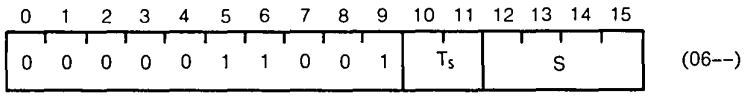
Status Bits Affected: **LGT, AGT, EQ, C, OV**

Example: **DEC @TABL** M(TABL) – 1 ⟶ M(TABL)

Application: This instruction is most often used to decrement byte counters or to work through byte addresses in descending order.

DECREMENT BY TWO — DECT

Format: **DECT G**



▶6 *Operation:* Two is subtracted from the data at the location specified by G and the result is stored at that location and is compared to 0:

M(G) – 2 ⟶ M(G)
M(G) – 2 : 0

Status Bits Affected: **LGT, AGT, EQ, C, OV**

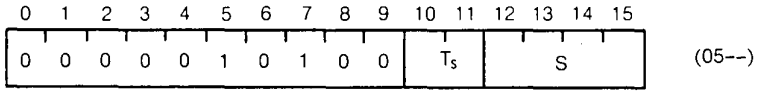
Example: **DECT 3** (R3) – 2 ⟶ R3

Application: This instruction is used to decrement word counters and to work through word addresses in descending order.

NEGATE

NEG

Format: **NEG G**



Operation: The data at the address specified by G is replaced by its two's complement. The result is compared to 0:

$-M(G) \longrightarrow M(G)$
$-M(G) : 0$

Status Bits Affected: **LGT, AGT, EQ, C, OV** (OV set only when operand = 8000_{16})

Example: **NEG 5** $-(R5) \longrightarrow R5$

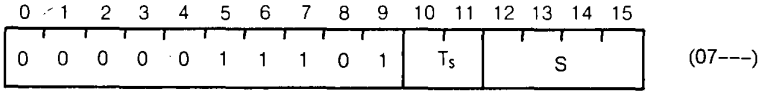
If R5 contained $A342_{16}$, this instruction would cause the R5 contents to changed to $5CBE_{16}$ and will cause the LGT and AGT status bits to be set to 1.

Application: NEG is used to form the 2's complement of 16 bit numbers.

ABSOLUTE VALUE

ABS

Format: **ABS G**



Operation: The data at the address specified by G is compared to 0. Then the absolute value of this data is placed in the G location:

$M(G) : 0$
$|M(G)| \longrightarrow M(G)$

Status Bits Affected: **LGT, AGT, EQ, OV** (OV set only when operand = 8000_{16})

Example: **ABS @LIST(7)** $|M(R7 + LIST)| \longrightarrow M(R7 + LIST)$

If the data at R7 + LIST is $FF3C_{16}$, it will be changed to $00C4_{16}$ and LGT will be set to 1.

Application: This instruction is used to test the data in location G and then replace the data by its absolute value. This could be used for unsigned arithmetic algorithms such as multiplication.

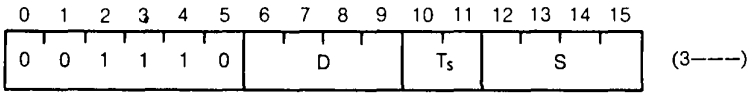
6◀

MPY

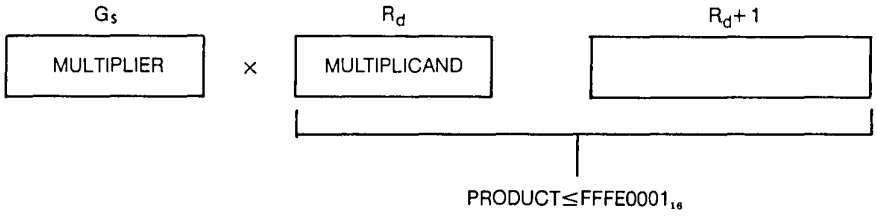
MULTIPLY

MPY

Format: **MPY** G_s,R_d



Operation: The 16 bit data at the address designated by G_s is multiplied by the 16 bit data contained in the specified workspace register R. The unsigned binary product (32 bits) is placed in workspace registers R and R+1:



Affect on Status: None

Example: **MPY @NEW,5**

If the data at location NEW is 0005_{16} and R5 contains 0012_{16}, this instruction will cause R5 to contain 0000_{16} and R6 to contain $005A_{16}$.

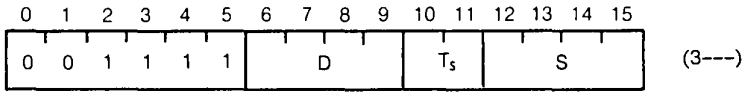
Application: MPY can be used to perform 16 bit by 16 bit binary multiplication. Several such 32 bit subproducts can be combined in such a way to perform multiplication involving larger multipliers and multiplicands such as a 32 bit by 32 bit multiplication.

▸6

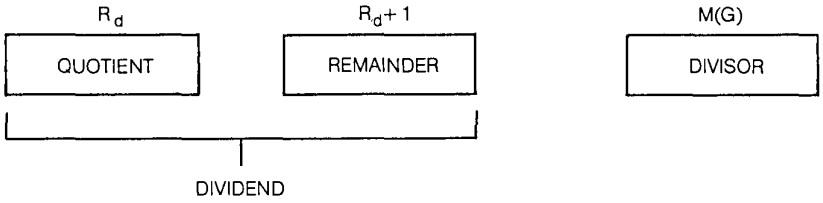
DIVIDE

DIV

Format: **DIV G_s,R_d**



Operation: The 32 bit number contained in workspace registers R_d and $R_d + 1$ is divided by the 16 bit data contained at the address specified by G_s. The workspace register R_d then contains the quotient and workspace $R_d + 1$ contains the 16 bit remainder. The division will occur only if the divisor at G is greater than the data contained in R_d:



Affect on Status: Overflow (OV) is set if the divisor is less than the data contained in R_d. If OV is set, R_d and $R_d + 1$ are not changed.

Example: **DIV @LOC,2**

If R2 contains 0 and R3 contains $000D_{16}$ and the data at address LOC is 0005_{16}, this instruction will cause R2 to contain 0002_{16} and R3 to contain 0003_{16}. OV would be 0.

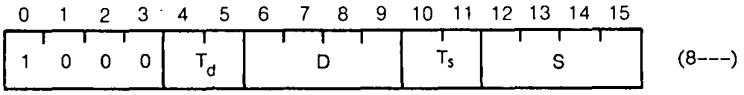
Application: DIV provides basic binary division of a 32 bit number by a 16 bit number.

COMPARISON INSTRUCTIONS

These instructions are used to test words or bytes by comparing them with a reference constant or with another word or byte. Such operations are used in certain types of division algorithms, number conversion, and in recognition of input command or limit conditions.

COMPARE WORDS

Format: **C** **G_s,G_d**



Operation: The 2's complement 16 bit data addressed by G_s is compared to the 2's complement 16 bit data addressed by G_d. The contents of both locations remain unchanged.

$M(G_s) : M(G_d)$

Status Bits Affected: **LGT, AGT, EQ**

Example: **C** **@T1,2**

This instruction has the following example results:

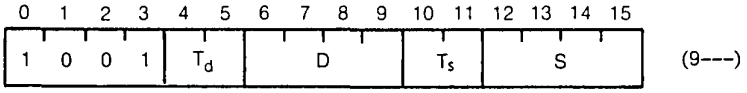
Data at Location T1	*Data in R2*	*Results of Comparison*		
		LGT	*AGT*	*EQ*
FFFF	**0000**	**1**	**0**	**0**
7FFF	**0000**	**1**	**1**	**0**
8000	**0000**	**1**	**0**	**0**
8000	**7FFF**	**1**	**0**	**0**
7FFF	**7FFF**	**0**	**0**	**1**
7FFF	**8000**	**0**	**1**	**0**

Application: The need to compare two words occurs in such system functions as division, number conversion, and pattern recognition.

COMPARE BYTES

CB

Format: **CB G_s,G_d**



Operation: The 2's complement 8 bit byte addressed by G_s is compared to the 2's complement 8 bit byte addressed by G_d:

$MB(G_s) : MB(G_d)$

Status Bits Affected: **LGT, AGT,EQ,OP**

OP (odd parity) is based on the number of bits in the source byte.

Example: **CB 1,*2**

with the typical results of (assuming R2 addresses an odd byte):

		Results of Comparison			
R1 data	*M(R2) Data*	*LGT*	*AGT*	*EQ*	*OP*
FFFF	**FF00**	**1**	**0**	**0**	**0**
7F00	**FF00**	**1**	**1**	**0**	**1**
8000	**FF00**	**1**	**0**	**0**	**1**
8000	**FF7F**	**1**	**0**	**0**	**1**
7F00	**007F**	**0**	**0**	**1**	**1**

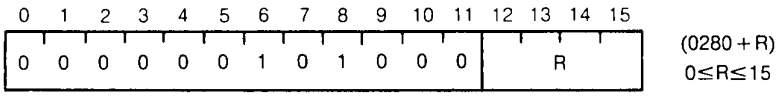
The underlined entries indicate the byte addressed.

Application: In cases where 8 bit operations are required, CB provides a means of performing byte comparisons for special conversion and recognition problems.

COMPARE IMMEDIATE

CI

Format: **CI R,Value**



Operation: CI compares the specified workspace register contents to the value contained in the word immediately following the instruction:

R : Value

Status Bits Affected: **LGT, AGT, EQ**

Example: **CI 9,>F330**

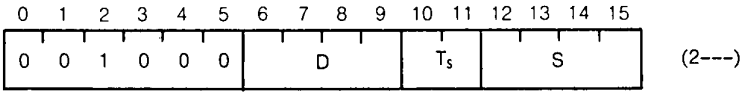
If R9 contains 2183_{16}, the equal (EQ) and logical greater than (LGT) bits will be 0 and arithmetic greater than (AGT) will be set to 1.

Application: CI is used to test data to see if system or program limits have been met or exceeded or to recognize command words.

COMPARE ONES CORRESPONDING

COC

Format: **COC G_s,R**



▶6

Operation: The data in the location addressed by G_s act as a mask for the bits to be tested in workspace register R. That is, only the bit position that contain ones in the G_s data will be checked in R. Then, if R contains ones in all the bit positions selected by the G_s data, the equal (EQ) status bit will be set to 1.

Status Bits Affected: **EQ**

Example: **COC @TESTBIT, 8**

If R8 contains $E306_{16}$ and location TESTBIT contains $C102_{16}$,

TESTBIT Mask = 1100 0001 0000 0010
R8 = 1110 0011 0000 0110

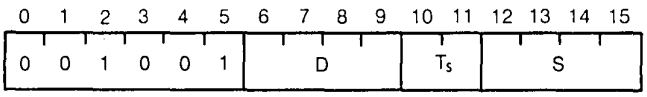
equal (EQ) would be set to 1 since everywhere the test mask data contains a 1 (underlined positions), R8 also contains a 1.

Application: COC is used to selectively test groups of bits to check the status of certain sub-systems or to examine certain aspects of data words.

COMPARE ZEROES CORRESPONDING

CZC

Format: **CZC G_s,R**



Operation: The data located in the address specified by G_s act as a mask for the bits to be tested in the specified workspace register R. That is, only the bit positions that contain ones in the G_s data are the bit positions to be checked in R. Then if R contains zeroes in all the selected bit positions, the equal (EQ) status bit will be set to 1.

Status Bits Affected: **EQ**

Examples: **CZC @TESTBIT,8**

If the TESTBIT location contains the value $C102_{16}$ and the R8 location contains 2301_{16},

TESTBIT Data = 1100 0001 0000 0010
R8 = 0010 0011 0000 0001
X

the equal status bit would be reset to zero since not all the bits of R8 (note the X position) are zero in the positions that the TESTBIT data contains ones.

Application: Similar to the COC instruction.

6◀

ANDI

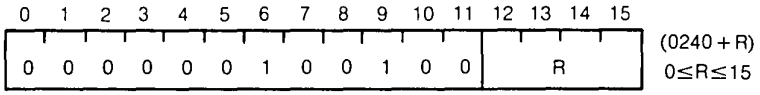
LOGIC INSTRUCTIONS

The logic instructions allow the processor to perform boolean logic for the system. Since AND, OR, INVERT, and Exclusive OR (XOR) are available, any boolean function can be performed on system data.

AND IMMEDIATE

ANDI

Format: **ANDI R,Value**



Operation: The bits of the specified workspace register R are logically ANDed with the corresponding bits of the 16 bit binary constant value contained in the word immediately following the instruction. The 16 bit result is compared to zero and is placed in the register R:

R AND Value ⟶ R
R AND Value : 0

Recall that the AND operation results in 1 only if *both* inputs are 1.

Status Bits Affected: **LGT, AGT, EQ**

Example: **ANDI 0,>6D03**

If workspace register 0 contains $D2AB_{16}$, then (D2AB) AND (6D03) is 4003_{16}:

Value =	0110	1101	0000	0011
R0 =	1101	0010	1010	1011
R0 AND Value =	0100	0000	0000	$0011 = 4003_{16}$

▸6

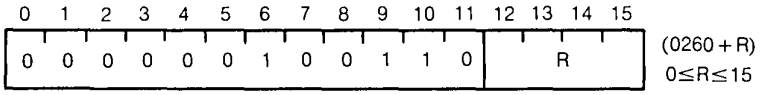
This value is placed in R0. The LGT and AGT status bits are set to 1.

Application: ANDI is used to zero all bits that are not of interest and leave the selected bits (those with ones in Value) unchanged. This can be used to test single bits or isolate portions of the word, such as a four bit group.

OR IMMEDIATE

ORI

Format: **ORI R,Value**



Operation: The bits of the specified workspace register R are ORed with the corresponding bits of the 16 bit binary constant contained in the word immediately following instruction. The 16 bit result is placed in R and is compared to zero:

R OR Value ⟶ R
R OR Value : 0

Recall that the OR operation results in a 1 if *either* of the inputs is a 1.

Status Bits Affected: **LGT, AGT, EQ**

Example: **ORI 5,>6D03**

If R5 contained $D2AB_{16}$, then R5 will be changed to $FFAB_{16}$:

R5 =	1101	0010	1010	1011
Value =	0110	1101	0000	0011
	1111	1111	1010	1011 = $FFAB_{16}$ = R5 OR Value

with LGT being set to 1.

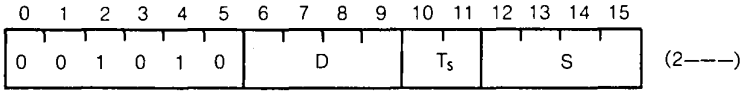
Application: Used to implement the OR logic in the system.

6◀

XOR

Exclusive OR

Format: **XOR** G_sR_d



Operation: The exclusive OR is performed between corresponding bits of the data addressed by G_s and the contents of workspace register R_d. The result is placed in workspace register R_d and is compared to 0:

$M(G_s)$ XOR $R_d \longrightarrow R_d$
$M(G_s)$ XOR R_d : 0

Status Bits Affected: **LGT, AGT, EQ**

Example: **XOR @CHANGE,2**

If location CHANGE contains $6D03_{16}$ and R2 contains $D2AA_{16}$, R2 will be changed to $BFA9_{16}$:

CHANGE Data =	0110	1101	0000	0011
R2 =	1101	0010	1010	1010
M(CHANGE) XOR R2 =	1011	1111	1010	1001 = $BFA9_{16}$

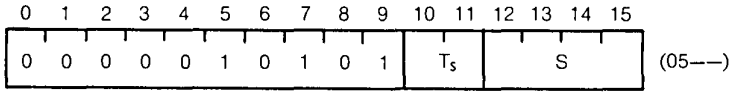
and the LGT status bit will be set to 1. Note that the exclusive OR operation will result in a 1 if *only one* of the inputs is a 1.

Application: XOR is used to implement the exclusive OR logic for the system.

INV

Invert

Format: **INV G**



Operation: The bits of the data addressed by G are replaced by their complement. The result is compared to 0 and is stored at the G location:

$\overline{M(G)} \longrightarrow M(G)$
$\overline{M(G)}$: 0

Status Bits Affected: **LGT, AGT, EQ**

Example: **INV 11**

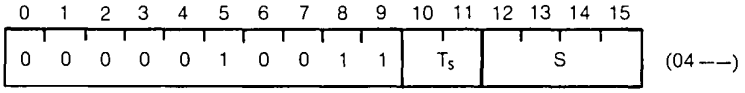
If R11 contains $00FF_{16}$, the instruction would change the contents to $FF00_{16}$, causing the LGT status bit to set to 1.

Application: INV is used to form the 1's complement of 16 bit binary numbers, or to invert system data.

CLR

CLEAR

Format: **CLR G**



Operation: 0000_{16} is placed in the memory location specified by G.

$0000_{16} \longrightarrow$ M(G)

Affect on Status: None

Example: **CLR *11**

would clear the contents of the location addressed by the contents of R11, that is:

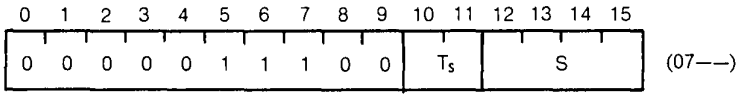
$0000_{16} \longrightarrow$ M(R11)

Application: CLR is used to set problem arguments to 0 and to initialize memory locations to zero during system start-up operations.

SETO

SET TO ONE

Format: **SETO G**



Operation: $FFFF_{16}$ is placed in the memory location specified by G: $FFFF_{16} \longrightarrow$ M(G)

Affect on Status: None

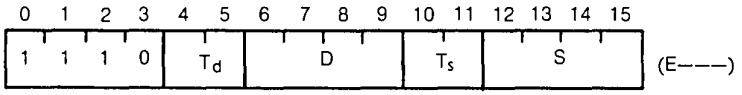
Example: **SETO 11**

would cause all bits of R11 to be 1.

Application: Similar to CLR

SET ONES CORRESPONDING — SOC

Format: **SOC G_s,G_d**



Operation: This instruction performs the OR operation between corresponding bits of the data addressed by G_s and the data addressed by G_d. The result is compared to 0 and is placed in the G_d location:

$M(G_s)$ OR $M(G_d) \longrightarrow M(G_d)$
$M(G_s)$ OR $M(G_d) : 0$

Status Bits Affected: **LGT, AGT, EQ**

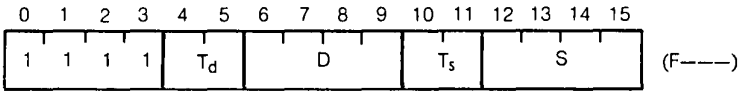
Example: **SOC 3,@NEW**

If location NEW contains $AAAA_{16}$ and R_3 contains $FF00_{16}$, the contents at location NEW will be changed to $FFAA_{16}$ and the LGT status bit will be set to 1.

Application: Provides the OR function between any two words in memory.

SET ONES CORRESPONDING, BYTE — SOCB

Format: **SOCB G_s,G_d**



▶6

Operation: The logical OR is performed between corresponding bits of the byte addressed by G_s and the byte addressed by G_d with the result compared to 0 and placed in location G_d:

$MB(G_s)$ OR $MB(G_d) \longrightarrow MB(G_d)$
$MB(G_s)$ OR $MB(G_d) : 0$

Status Bits Affected: **LGT, AGT, EQ, OP**

Example: **SOCB 5,8**

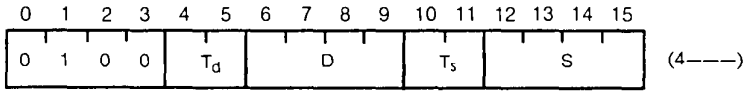
If R5 contains $F013_{16}$ and R8 contains $AA24_{16}$, the most significant byte of R8 will be changed to FA_{16} so that R8 will contain $FA24_{16}$ and the LGT status bit will be set to 1.

Application: The SOCB provides the logical OR function on system bytes.

SET TO ZEROES CORRESPONDING

SZC

Format: **SZC** G_s,G_d



Operation: The data addressed by G_s forms a mask for this operation. The bits in the destination data (addressed by G_d) that correspond to the one bits of the source data (addressed by G_s) are cleared. The result is compared to zero and is stored in the G_d location.

$\overline{M(G_s)}$ AND $M(G_d) \longrightarrow M(G_d)$
$\overline{M(G_s)}$ AND $M(G_d) : 0$

Status Bits Affected: **LGT, AGT, EQ**

Example: **SZC 5,3**

If R5 contains $6D03_{16}$ and R3 contains $D2AA_{16}$, this instruction will cause the R3 contents to change to $92A8_{16}$:

R5 (Mask) = 0110 1101 0000 0011
R3 = 1101 0010 1010 1010
Result = 1001 0010 1010 1000 = $92A8_{16}$

with the LGT status bit set. The underlined entries indicate which bits are to be cleared.

Application: SZC allows the programmer to selectively clear bits of data words. For example, when an interrupt has been serviced, the interrupt request bit can be cleared by using the SZC instruction.

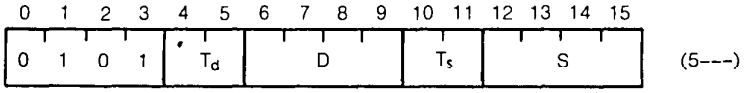
6◂

SZCB

SET TO ZEROES CORRESPONDING, BYTES

SZCB

Format: **SZCB G_sG_d**



Operation: The byte addressed by G_s will provide a mask for clearing certain bits of the byte addressed by G_d. The bits in the G_d byte that will be cleared are the bits that are one in the G_s byte. The result is compared to zero and is placed in the G_d byte:

$$\overline{MB(G_s)} \text{ AND } MB(G_d) \longrightarrow MB(G_d)$$
$$\overline{MB(G_s)} \text{ AND } MB(G_d) : 0$$

Status Bits Affected: **LGT, AGT, EQ, OP**

Example: **SZCB @BITS,@TEST**

If location BITS is an *odd* address which locates the data $18F0_{16}$, and location TEST contains an *even* address which locates the data $AA24_{16}$, the instruction will clear the first four bits of TEST data changing it to $0A24_{16}$.

Application: Provides selective clearing of bits of system bytes.

▶6

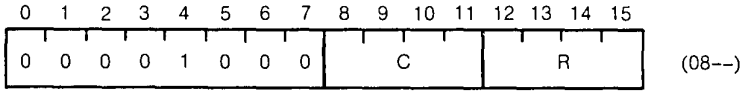
SHIFT INSTRUCTIONS

These instructions are used to perform simple binary multiplication and division on words in memory and to rearrange the location of bits in the word in order to examine a given bit with the carry (C) status bit.

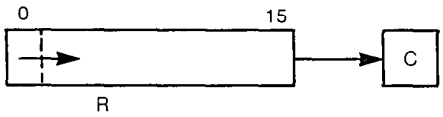
SHIFT RIGHT ARITHMETIC

SRA

Format: **SRA R,Cnt**



Operation: The contents of the specified workspace register R are shifted right Cnt times, filling the vacated bit position with the sign (most significant bit) bit: The shifted number is compared to zero:



Status Bits Affected: **LGT, AGT, EQ, C**

Number of Shifts: Cnt (number contained in the instruction from 0 to 15) specifies the number of bits shifted unless Cnt is zero in which case the shift count is taken from the four least significant bits of workspace register 0. If both Cnt and these four bits are 0, a 16 bit position shift is performed.

6◄

Example:
```
SRA   5,2     Shift R5 2 bit positions right
SRA   7,0
```

If R0 least four bits contain 6_{16}, then the second instruction will cause register 7 to be shifted 6 bit positions (Cnt in that instruction is 0):

If R7 Before Shift = 1011 1010 1010 1010 = $BAAA_{16}$
R7 After Shift = 1111 1110 1110 1010 = $FEEA_{16}$
If R5 Before Shift = 0101 0101 0101 0101 = 5555_{16}
R5 After Shift = 0001 0101 0101 0101 = 1555_{16}
After the R7 shift the LGT would be set, and Carry = 1
After the R5 shift LGT and AGT would be set and Carry = 0

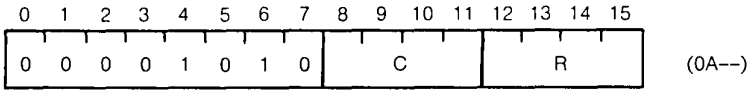
Application: SRA provides binary division by 2^{Cnt}.

SLA

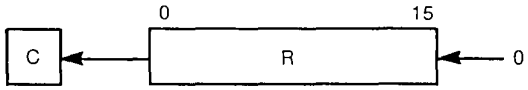
SHIFT LEFT ARITHMETIC

SLA

Format: **SLA R,Cnt**



Operation: The contents of workspace register R are shifted left Cnt times (or if Cnt = 0, the number of times specified by the least four bits of R0) filling the vacated positions with zeroes. The carry contains the value of the last bit shifted out to the left and the shifted number is compared to zero:



Status Bits Affected: **LGT, AGT, EQ, C, OV**

Example: **SLA 10,5**

If workspace register 10 contains 1357_{16} the instruction would change its contents to $6AE0_{16}$, causing the arithmetic greater than (AGT), logical greater than (LGT), and overflow (OV) bits to set. Carry would be zero, the value of the last bit shifted.

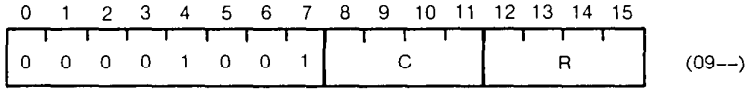
Application: SLA performs binary multiplication by 2^{Cnt}

▸6

SHIFT RIGHT LOGICAL

SRL

Format: **SRL R,Cnt**



Operation: The contents of the workspace register specified by R are shifted right Cnt times (or if Cnt = 0, the number of times specified by the least four bits or R0) filling in the vacated positions with zeroes. The carry contains the value of the last bit shifted out to the right and the shifted number is compared to zero:

Status Bits Affected: **LGT, AGT, EQ, C**

Example: **SRL 0,3**

If R0 contained $FFEF_{16}$, the contents would become $1FFD_{16}$ with the AGT, LGT, and C bits set to 1:

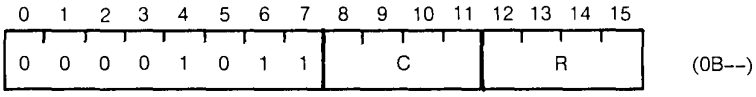
R0 Before Shift = 1111 1111 1110 1111 = $FFEF_{16}$
R0 After Shift = 0001 1111 1111 1101 = $1FFD_{16}$

Application: Performs binary division by 2^{Cnt}

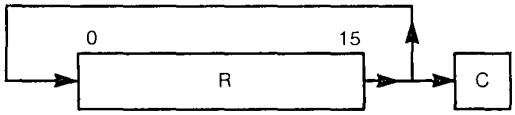
SHIFT RIGHT CIRCULAR

SRC

Format: **SRC R,Cnt**



Operation: On each shift the bit shifted out of bit 15 is shifted back into bit 0. Carry contains the value of the last bit shifted and the shifted number is compared to 0. The number of shifts to be performed is the number Cnt, or if Cnt = 0, the number contained in the least significant four bits of R0:



Status Bits Affected: **LGT, AGT, EQ, C**

Example: **SRC 2,7**

If R2 initially contains $FFEF_{16}$, then after the shift it will contain $DFFF_{16}$ with LGT and C set to 1.

R2 Before Shift = 1111 1111 1110 1111 = $FFEF_{16}$
R2 After Shift = 1101 1111 1111 1111 = $DFFF_{16}$

Application: SRC can be used to examine a certain bit in the data word, change the location of 4 bit groups, or swap bytes.

B

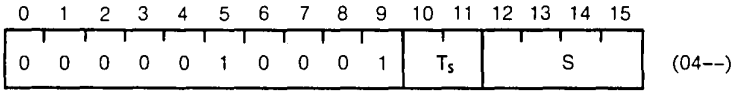
UNCONDITIONAL BRANCH INSTRUCTIONS

These instructions give the programmer the capability of choosing to perform the next instruction in sequence or to go to some other part of the memory to get the next instruction to be executed. The branch can be a subroutine type of branch, in which case the programmer can return to the point from which the branch occurred.

BRANCH

B

Format: **B** G_s



Operation: The G_s address is placed in the program counter, causing the next instruction to be obtained from the location specified by G_s.

Affect on Status: None

Example: **B** ***3**

If R3 contains $21CC_{16}$, then the next instruction will be obtained from location $21CC_{16}$.

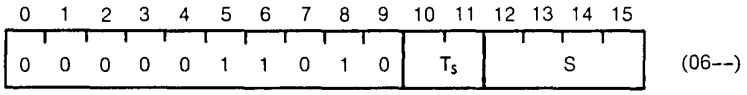
Application: This instruction is used to jump to another part of the program when the current task has been completed.

▶6

BRANCH AND LINK

BL

Format: **BL** G_s



Operation: The source address G_s is placed in the program counter and the address of the instruction following the BL instruction is saved in workspace register 11.

$G_s \longrightarrow PC$
(Old PC) $\longrightarrow$ R11

Affect on Status: None

Example: **BL @TRAN**

Assume the BL instruction is located at 3200_{16} and the value assigned to TRAN is 2000_{16}. PC will be loaded with the value 2000_{16} (TRAN) and R11 will be loaded with the value 3202_{16} (old PC value).

Application: This is a shared workspace subroutine jump. Both the main program and the subroutine use the same workspace registers. To get back to the main program at the branch point, the following branch instruction can be used at the end of the subroutine:

```
B       *11
```

which causes the R11 contents (old PC value) to be loaded into the program counter.

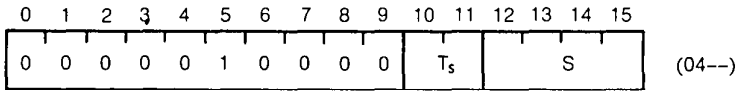
6

BLWP

Branch and Load Workspace Pointer

BLWP

Format: **BLWP** G_s



Operation: The word specified by the source G_s is loaded into the workspace pointer (WP) and the next word in memory ($G_s + 2$) is loaded into the program counter (PC) to cause the branch. The old workspace pointer is stored in the new workspace register 13, the old PC value is stored in the new workspace register 14, and the status register is stored in new workspace register 15:

$M(G_s) \longrightarrow WP$
$M(G_s + 2) \longrightarrow PC$
(Old WP) $\longrightarrow$ New R13
(Old PC) $\longrightarrow$ New R14
(Old ST) $\longrightarrow$ New R15

Affect on Status: None

Example: **BLWP *3**

Assuming that R3 contains 2100_{16} and location 2100_{16} contains 0500_{16} and location 2102_{16} contains 0100_{16}, this instruction causes WP to be loaded with 0500_{16} and PC to be loaded with 0100_{16}. Then, location $051A_{16}$ will be loaded with the old WP value, the old PC value will be saved in location $051C_{16}$, and the status (ST) will be saved in location $051E_{16}$. The next instruction will be taken from address 0100_{16} and the subroutine workspace will begin at 0500_{16} (R0). BLWP and XOP do not test IREQ at the end of instruction execution.

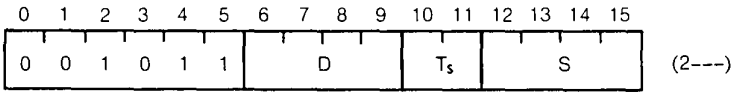
▸6

Application: This is a context switch subroutine jump with the transfer vector location specified by G_s. It uses a new workspace to save the old values of WP, PC, and ST (in the last three registers). The advantage of this subroutine jump over the BL jump is that the subroutine gets its own workspace and the main program workspace contents are not disturbed by subroutine operations.

EXTENDED OPERATION

XOP

Format: **XOP** G_s,n



Operation: n specifies which extended operation transfer vector is to be used in the context switch branch from XOP to the corresponding subprogram. The effective address G_s is placed in R11 of the subprogram workspace in order to pass an argument or data location to the subprogram:

M(n × 4 + 0040_{16}) ⟶ WP
M(n × 4 + 0042_{16}) ⟶ PC
(Old WP) ⟶ New R13
(Old PC) ⟶ New R14
(Old ST) ⟶ New R15
G_s ⟶ New R11

Affect on Status: Extended Operation (X) bit is set.

Example: **XOP** ***1,2**

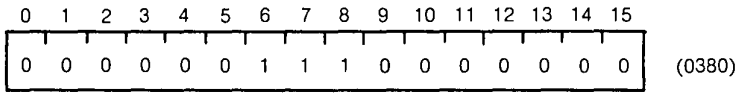
Assume R1 contains 0750_{16}. WP is loaded with the word at address 48_{16} (first part of transfer vector for extended operation 2) and PC is loaded with the word at address $4A_{16}$. If location 48_{16} contains 0200_{16}, this will be the address of R0 of the subprogram workspace. Thus, location 0236_{16} (new R11) will be loaded with 0750_{16} (contents of R1 in main program), location $023A_{16}$ (new R13) will be loaded with the old WP value, location $023C_{16}$ will be loaded with the old PC value, and location $023E_{16}$ (new R15) will be loaded with the old status value:

M(48_{16}) ⟶ WP
M($4A_{16}$) ⟶ PC
(Old WP) ⟶ M($023A_{16}$) New R13
(Old PC) ⟶ M($023C_{16}$) New R14
(Old ST) ⟶ M($023E_{16}$) New R15
0750_{16} ⟶ M(0236_{16}) New R11

Application: This can be used to define a subprogram that can be called by a single instruction. As a result, the programmer can define special purpose instructions to augment the standard 9900 instruction set.

RETURN WITH WORKSPACE POINTER — RTWP

Format: **RTWP**



Operation: This is a return from a context switch subroutine. It occurs by restoring the WP, PC, and ST register contents by transferring the contents of subroutine workspace registers R13, R14, and R15, into the WP, PC, and ST registers, respectively.

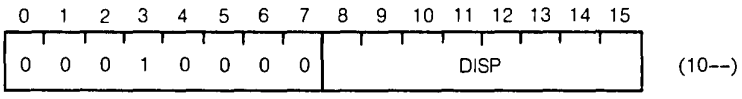
R13 ⟶ WP
R14 ⟶ PC
R15 ⟶ ST

Status Bits Affected: **All (ST receives the contents of R15)**

Application: This is used to return from subprograms that were reached by a transfer vector operation such as an interrupt, extended operation, or BLWP instruction.

UNCONDITIONAL JUMP — JMP

Format: **JMP EXP**



▸6 *Operation:* The signed displacement defined by EXP is added to the current contents of the program counter to generate the new value of the program counter. The location jumped to must be within −128 to +127 words of the present location.

Affect on Status: None

Example: **JMP THERE**

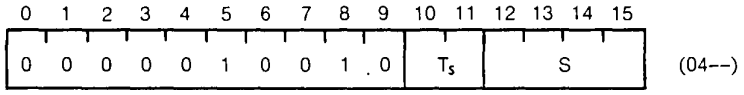
If this instruction is located at 0018_{16} and THERE is the label of the instruction located at 0010_{16}, then the Exp value placed in the object code would be FB (for −5). Since the Assembler makes this computation, the programmer only needs to place the appropriate label or expression in the operand field of the instruction.

Application: If the subprogram to be jumped to is within 128 words of the JMP instruction location, the unconditional JMP is preferred over the unconditional branch since only one memory word (and one memory reference) is required for the JMP while two memory words and two memory cycles are required for the B instruction. Thus, the JMP instruction can be implemented faster and with less memory cost than can the B instruction.

EXECUTE

X

Format: **X** G_s



Operation: The instruction located at the address specified by G_s is executed.

Status Bits Affected: **Depends on the instruction executed**

Example: **X *11**

If R11 contains 2000_{16} and location 2000_{16} contains the instruction for CLR 2 then this execute instruction would clear the contents of register 2 to zero.

Application: X is useful when the instruction to be executed is dependent on a variable factor.

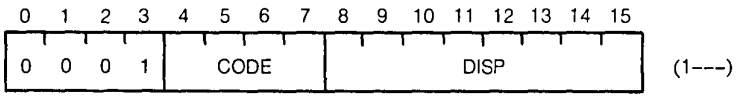
6◂

CONDITIONAL JUMP INSTRUCTIONS

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

These instructions perform a branching operation only if certain status bits meet the conditions required by the jump. These instructions allow decision making to be incorporated into the program. The conditional jump instruction mnemonics are summarized in *Table 6-1* along with the status bit conditions that are tested by these instructions.

Format: **Mnemonic Exp**



Operation: If the condition indicated by the branch mnemonic is true, the jump will occur using relative addressing as was used in the unconditional JMP instruction. That is, the Exp defines a displacement that is added to the current value of the program counter to determine the location of the next instruction, which must be within 128 words of the jump instruction.

Effect on Status Bits: None

Example:

```
C       R1, R2
JNE     LOOP
```

The first instruction compares the contents of registers one and two. If they are not equal, EQ = 0 and the JNE instruction causes the branch to LOOP to be taken. If R1 and R2 are equal, EQ = 1 and the branch is not taken.

Table 6-1. Status Bits Tested by Instructions

Mnemonic	*L>*	*A>*	*EQ*	*C*	*OV*	*OP*	*Jump if:*	*CODE**
JH	X	–	X	–	–	–	L> • $\overline{EQ}$ = 1	B
JL	X	–	X	–	–	–	L> + EQ = 0	A
JHE	X	–	X	–	–	–	L> + EQ = 1	4
JLE	X	–	X	–	–	–	$\overline{L>}$ + EQ = 1	2
JGT	–	X	–	–	–	–	A> = 1	5
JLT	–	X	X	–	–	–	A> + EQ = 0	1
JEQ	–	–	X	–	–	–	EQ = 1	3
JNE	–	–	X	–	–	–	EQ = 0	6
JOC	–	–	–	X	–	–	C = 1	8
JNC	–	–	–	X	–	–	C = 0	7
JNO	–	–	–	–	X	–	OV = 0	9
JOP	–	–	–	–	–	X	OP = 1	C

Note: In the Jump if column, a logical equation is shown in which • means the AND operation, + means the OR operation, and a line over a term means negation or inversion.

*CODE is entered in the CODE field of the OPCODE to generate the machine code for the instruction.

Application: Most algorithms and programs with loop counters require these instructions to decide which sequence of instructions to do next.

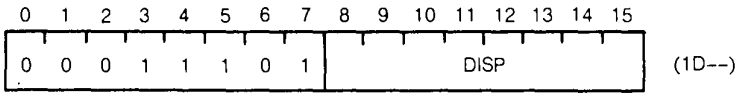
CRU INSTRUCTIONS

The communications register unit (CRU) performs single and multiple bit programmed input/output for the microcomputer. All input consists of reading CRU line logic levels into memory, and all 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 4096 output lines that may be individually selected by a 12 bit address which is located in bits 3 through 14 of workspace register 12. This address is the hardware base address for all CRU communications.

SET BIT TO LOGIC ONE

SBO

Format: **SBO disp**



Operation: The CRU bit at disp plus the hardware base address is set to one. The hardware base address is bits 3 through 14 of workspace register 12. The value disp is a signed displacement.

1 ⟶ Bit (disp + base address)

Affect on Status: None

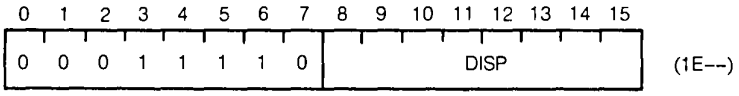
Example: **SBO 15**

If R12 contains a software base address of 0200_{16} so that the hardware base address is 0100_{16} (the hardware base address is one-half the value of the contents of R12 excluding bits 0, 1 and 2), the above instruction would set CRU line $010F_{16}$ to a 1.

Application: Output a one on a single bit CRU line.

SET BIT TO LOGIC ZERO — SBZ

Format: **SBZ disp**



Operation: The CRU bit at disp plus the base address is reset to zero. The hardware base address is bits 3 through 14 of workspace register 12. The value disp is a signed displacement.

0 ⟶ Bit (disp + hardware base address)

Affect on Status: None

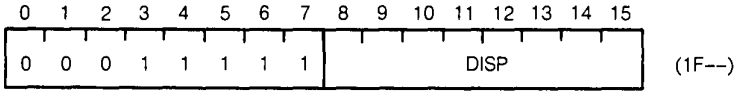
Example: **SBZ 2**

If R12 contains 0000_{16}, the hardware base address is 0 so that the instruction would reset CRU line 0002_{16} to zero.

Application: Output a zero on a single bit CRU line.

TEST BIT — TB

Format: **TB disp**



▸6

Operation: The CRU bit at disp plus the base address is read by setting the value of the equal (EQ) status bit to the value of the bit on the CRU line. The hardware base address is bits 3 through 14 of workspace register 12. The value disp is a signed displacement.

Bit (disp + hardware base address) ⟶ EQ

Status Bits Affected: **EQ**

Example: **TB 4**

If R12 contains 0140_{16}, the hardware base address is $A0_{16}$ (which is one-half of 0140_{16}):

R12 Contents = 0000 0001 0100 0000

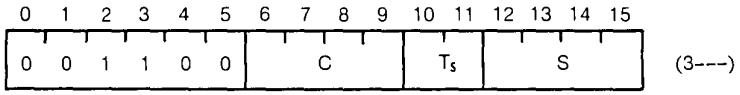
Note that the underlined hardware base address is $0A0_{16}$. Equal (EQ) would be made equal to the logic level on CRU line $0A0_{16} + 4 =$ CRU line $0A4_{16}$.

Application: Input the CRU bit selected.

LOAD CRU

LDCR

Format: **LDCR G_s,Cnt**



Operation: Cnt specifies the number of bits to be transferred from the data located at the address specified by G_s, with the first bit transferred from the least significant bit of this data, the next bit from the next least significant bit and so on. If Cnt = 0, the number of bits transferred is 16. If the number of bits to be transferred is one to eight, the source address is a byte address. If the number of bits to be transferred is 9 to 16, the source address is a word address. The source data is compared to zero before the transfer. The destination of the first bit is the CRU line specified by the hardware base address, the second bit is transferred to the CRU line specified by the hardware base address + 1, and so on.

Status Bits Affected: **LGT, AGT, EQ**
OP (odd parity) with transfer of 8 or less bits.

Example: **LDCR @TOM,8**

Since 8 bits are transferred, TOM is a byte address. If TOM is an even number, the most significant byte is addressed. If R12 contains 0080_{16}, the hardware base address is 0040_{16} which is the CRU line that will receive the first bit transferred. 0041_{16} will be the address of the next bit transferred, and so on to the last (8th) bit transferred to CRU line 0047_{16}. This transfer is shown in *Figure 6-7.*

6◀

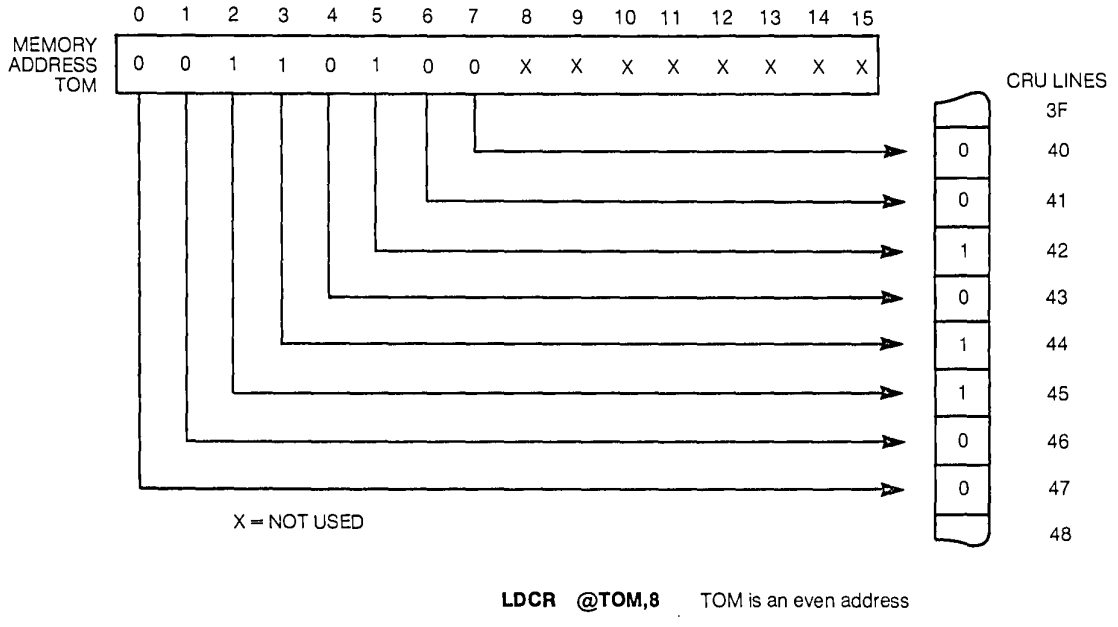
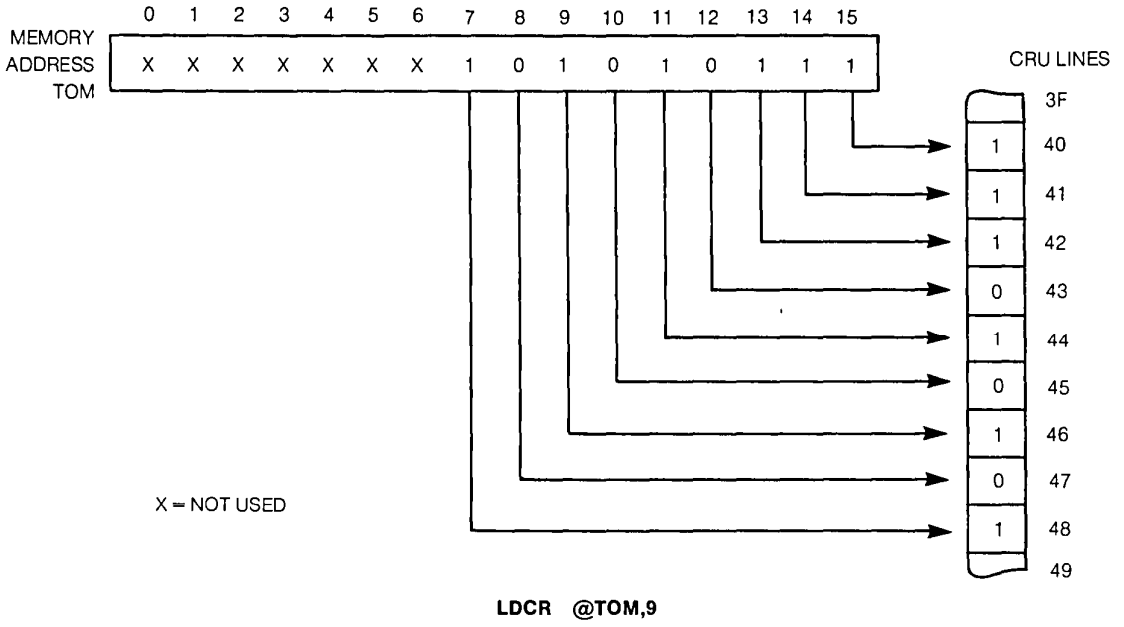


Figure 6-7. LDCR byte transfer

LDCR

Application: The LDCR provides a number of bits (from 1 to 16) to be transferred from a memory word or byte to successive CRU lines, starting at the hardware base address line; the transfer begins with the least significant bit of the source field and continues to successively more significant bits. A further example of word versus byte transfers is given in *Figure 6-8,* in which a 9 bit (word addressed source) transfer is shown.



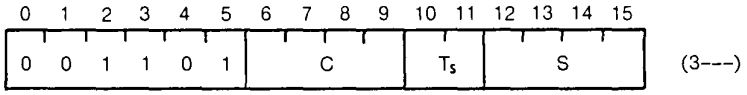
▸6

Figure 6-8. LDCR Word transfer

STORE CRU

STCR

Format: **STCR G_s,Cnt**



Operation: Cnt specifies the number of bits to be transferred from successive CRU lines (starting at the hardware base address) to the location specified by G_s, beginning with the least significant bit position and transferring successive bits to successively more significant bits. If the number of bits transferred is 8 or less, G_s is a byte address. Otherwise, G_s is a word address. If Cnt = 0, 16 bits are transferred. The bits transferred are compared to zero. If the transfer does not fill the entire memory word, the unfilled bits are reset to zero.

Status Bits Affected: **LGT, AGT, EQ**
OP for transfers of 8 bits or less

Example: **STCR 2,7**

Since 7 bits are to be transferred this is a byte transfer so that the bits will be transferred to the most significant byte of R2. *Figure 6-9* illustrates this transfer assuming that R12 contains 90_{16} so that the hardware base address is 48_{16} for the first bit to be transferred.

Note: Bits 8-15 are unchanged if transfer is less than 8 bits.

6◂

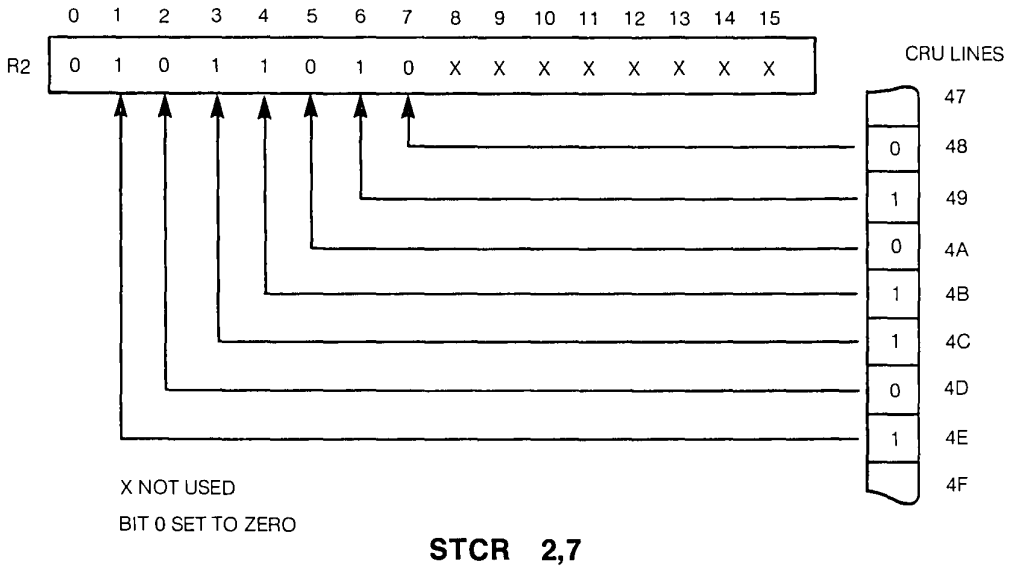


Figure 6-9. STCR Example

CONTROL INSTRUCTIONS

The control instructions are primarily applicable to the Model 990 Computer. These instructions are RSET (Reset), IDLE, CKOF (Clock off), CKON (Clock on), LREX (restart). The Model 990/10 also supports the long distance addressing instructions: LDS (Load long distance source) and LDD (Long distance destination). The use of these instructions are covered in the appropriate Model 990 computer programmer's manuals.

The control instructions have an affect on the 9900 signals on the address lines during the CRU Clock as shown below:

Instruction	A_0	A_1	A_2	OP CODE
LREX	H	H	H	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 (03E0)
CKOF	H	H	L	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 (03C0)
CKON	H	L	H	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 (03A0)
RSET	L	H	H	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 0 0 0 0 0 1 1 0 1 1 0 0 0 0 0 (0360)
IDLE	L	H	L	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 (0340)
CRU	L	L	L	

The IDLE instruction puts the 9900 in the idle condition and causes a CRUCLK output every six clock cycles to indicate this state. The processor can be removed from the idle state by 1) a $\overline{\text{RESET}}$ signal, 2) any interrupt that is enabled, or 3) a $\overline{\text{LOAD}}$ signal.

For the 9900 the above instructions are referred to as external instructions, since external hardware can be designed to respond to these signals. The address signals A_0, A_1, and A_2 can be decoded and the instructions used to control external hardware.

SPECIAL FEATURES OF THE 9940

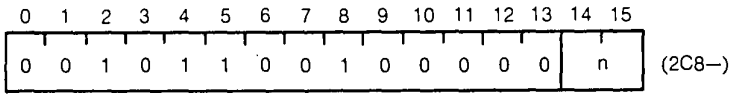
The 9940 instruction set includes the instructions already presented. Two of these instructions are slightly different for the 9940. These are the extended operation and the load interrupt mask immediate instructions. There are two new arithmetic instructions that provide for binary coded decimal (BCD) addition and subtraction. The 9940 uses extended operations 0 through 3 to generate the load interrupt mask and the decimal arithmetic instructions. Thus, the 9940 extended operations 4 through 15 are available to the programmer.

LOAD IMMEDIATE INTERRUPT MASK

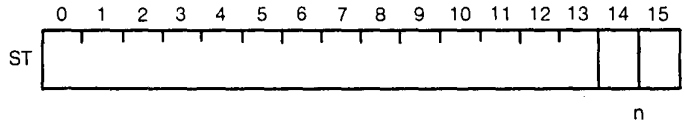
LIIM

Format: **LIIM n**

$0 \leq n \leq 3$



Operation: The interrupt mask bits 14 and 15 of the status register are loaded with n. Subsequent to this instruction, interrupt levels greater than n will be ignored by the processor, and interrupts of level n or less will be responded to by the processor.



Status Bits Affected: **Interrupt Mask (Bits 14 and 15)**

6◀

Example: **LIIM 2**

This operation will load the interrupt mask with 2, that is bit 14 would be set to a 1 and bit 15 would be reset to zero. This would disable interrupts of level 3, but would enable other interrupt levels.

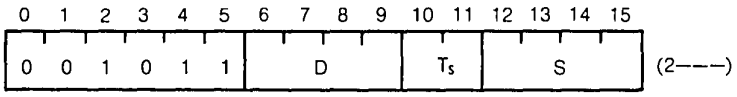
Application: This instruction is used to control the 9940 interrupt system.

XOP

Extended Operation

XOP

Format: **XOP** **G_s,n**



Operation: n specifies the extended operation transfer vector to be used in the context switch to the extended operation subprogram. The TMS9940 restricts the range of n ($4 \leq n \leq 15$) so that there are only 12 XOP's available. This is because the first four are used by the processor to implement the LIIM, DCA, and DCS instructions. The transfer vector procedure for the programmer-defined extended operations is:

$M(40_{16} + 4xn) \longrightarrow$ (WP)
$M(42_{16} + 4xn) \longrightarrow$ (PC)
$G_s \longrightarrow$ (New WR11)
(Old WP) $\longrightarrow$ (New WR13)
(Old PC) $\longrightarrow$ (New WR14)
(Old ST) $\longrightarrow$ (New WR15)

Status Bits Affected: **None**

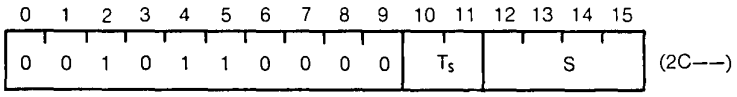
Example and Applications: **XOP** ***1,4**

This instruction will cause an extended operation 4 to occur with the new workspace register 11 containing the address found in workspace register 1. The new WP value ▸6 will be obtained from $40_{16} + 4 \times 4 = 50_{16}$ and the new PC value will be obtained from 52_{16}.

Decimal Correct Addition

DCA

Format: **DCA** G_s



Operation: The byte addressed by G_s is corrected according to the table given in *Figure 6-10.* This operation is a processor defined extended operation with n = 0 so that the sequence of events described under the XOP discussion will occur in executing this instruction.

Status Bits Affected: **LGT, AGT, EQ, C, P, and DC (Digit Carry).**

Example: **DCA *10**

This instruction would cause the byte addressed by the contents of the current workspace register 10 to be decimal adjusted in accordance with the truth table of *Figure 6-10.*

Application: This instruction is used immediately after the binary addition of two bytes (AB instruction) to correct any decimal digits outside the BCD code range of 0000_2 through 1001_2. It also keeps decimal addition accurate by responding to digit carries. For example, if 8_{16} is added to 8_{16} in BCD addition, 16_{16} should be generated. However, if this operation is performed with binary addition, 10_{16} results:

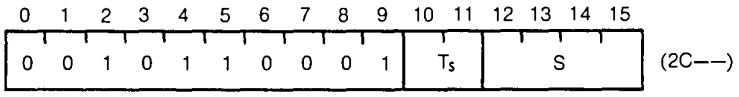
```
    0 0 0 0     1 0 0 0
+   0 0 0 0     1 0 0 0
-------------------------
    0 0 0 1     0 0 0 0      Digit Carry = 1
```

The DCA detects the digit carry and adds 0110_2 to the least significant digit to get the correct 16_{16}.

Decimal Correct Subtraction

DCS

Format: **DCS** G_s



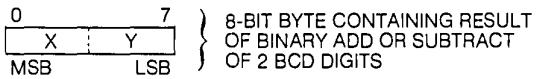
Operation: The byte addressed by G_s is corrected according to the table given in *Figure 6-10.* This instruction is a processor defined extended operation with n = 1, so that the sequence of events described under extended operation will occur in executing this instruction.

Status Bits Affected: **LGT, AGT, EQ, C, P, and DC**

Example: **DCS 3**

This instruction would cause the most significant byte of register 3 to be corrected in accordance with the truth table of *Figure 6-10.*

Application: As in the DCA instruction, this instruction extends the 9940 capability to include decimal subtraction. The programmer first performs binary subtraction on bytes (the SB instruction) and then immediately performs the DCS operation on the result byte to correct the result so that it is within the BCD code range 0000_2 through 1001_2.



BYTE BEFORE EXECUTION				BYTE AFTER DCA				BYTE AFTER DCS			
C	X	DC	Y	C	X	DC	Y	C	X	DC	Y
0	X<10	0	Y<10	0	X	0	Y	—	—	—	—
0	X<10	1	Y<10	0	X	0	Y+6	—	—	—	—
0	X<9	0	Y≥10	0	X+1	1	Y+6	—	—	—	—
1	X<10	0	Y<10	1	X+6	0	Y	—	—	—	—
1	X<10	1	Y<10	1	X+6	0	Y16	—	—	—	—
1	X<10	0	Y≥10	1	X+7	1	Y+6	—	—	—	—
0	X≥10	0	Y<10	1	X+6	0	Y	—	—	—	—
0	Z≥10	1	Y<10	1	X+6	0	Y+6	—	—	—	—
0	X≥9	0	Y≥10	1	X+7	1	Y+6	—	—	—	—
0	X	0	Y	—	—	—	—	0	X+10	1	Y+10
0	X	1	Y	—	—	—	—	0	X+10	0	Y
1	X	0	Y	—	—	—	—	1	X	1	Y+10
1	X	1	Y	—	—	—	—	1	X	0	Y

Figure 6-10. Result of DCA and DCS Instructions of the 9940.

CHAPTER 7

Program Development: Software Commands—Descriptions and Formats

INTRODUCTION

The purpose of this chapter is to provide reference data for the various software development systems available for the 9900 family of microprocessors and microcomputers.

Table 7-1 lists the sections in the chapter. One or more cards are made for those sections marked with a bullet. The section on Assembly Language programming describes the basic format for coding instructions and assembler directives. It is a general topic, applicable to all of the programming systems.

Explanation of the terms, mnemonics instruction execution rules, etc. can be found in Chapters, 4, 5, and 6.

The complete TM 990/402 Line-by-Line Assembler User's Guide is included because this EPROM resident software is used in Chapter 9. It should serve as an illustration of the need for some form of an assembler in writing even the simplest programs. Contrast the programming efforts of Chapter 3 with the programming efforts for the extended applications of Chapter 9, and you will appreciate the power of this LBL assembler.

Reference material for the other programming systems is in the form of lists of commands and their syntax. These pages are not stand-alone documents. Software documentation is supplied with each of the programming systems and is required for full explanations of the commands and their use. Experienced designers always need assistance in recalling exact command mnemonics and their formats. Thus, this chapter supports you in any programming environment by appropriate reminders.

Table 7-1

·7

Assembly language programming and assembler directives
- 9900 Reference Data
  TM 990/402 Line-by-Line Assembler
- TIBUG Monitor
- TM 990/302 Software Development board
- TXDS Commands for the FS 990 PDS
- AMPL Reference data
- POWER BASIC Commands
- Cross Support reference data
  - Assembler
  - Simulator
  - Utilities

Assembly Language Programming: Formats and Directives

7◀

ASSEMBLY LANGUAGE PROGRAMMING

An assembly language is a computer oriented language for writing programs. The TMS9900 recognizes instructions in the form of 16 bit (or longer) binary numbers, called instruction or operation codes (Opcodes). Programs could be written directly in these binary codes, but it is a tedious effort, requiring frequent reference to code tables. It is simpler to use names for the instructions, and write the programs as a sequence of these easily recognizable names (called mnemonics). Then, once the program is written in mnemonic or assembly language form, it can be converted to the corresponding binary coded form (machine language form). The assembler programs described here indicate parts of PX9ASM, TXMIRA and SDSMAC, which operate on cassette, floppy disc, and moving head disc systems respectively. Several other assemblers are available from TI which provide fewer features, but operate with much smaller memory requirements.

Assembly Language Application

The assembly language programming and program verification through simulation or execution are the main elements involved in developing microprocessor programs. The overall program development effort consists of the following steps:

- Define the problem.
- Flowchart the solution to the problem.
- Write the assembly language program for the flowchart.
- Execute the Assembler to generate the machine code.
- Correct any format errors indicated by the Assembler.
- Execute the corrected machine code program on a TMS9900 computer or on a Simulator to verify program operation.

This program development sequence is defined in flowchart form in *Figure 7-1.*

▶7 ### Assembly Language Formats

The general assembly language source statements consists of four fields as follows:

LABEL MNEMONIC OPERANDS COMMENT

The first three fields must occur within the first 60 character positions of the source record. At least one blank must be inserted between fields.

Label Field

The label consists of from one to six characters, beginning with an alphabetic character in character position one of the source record. The label field is terminated by at least one blank. When the assembler encounters a label in an instruction it assigns the current value of the location counter to the label symbol. This is the value associated with the label symbol and is the address of the instruction in memory. If a label is not used, character position 1 must be a blank.

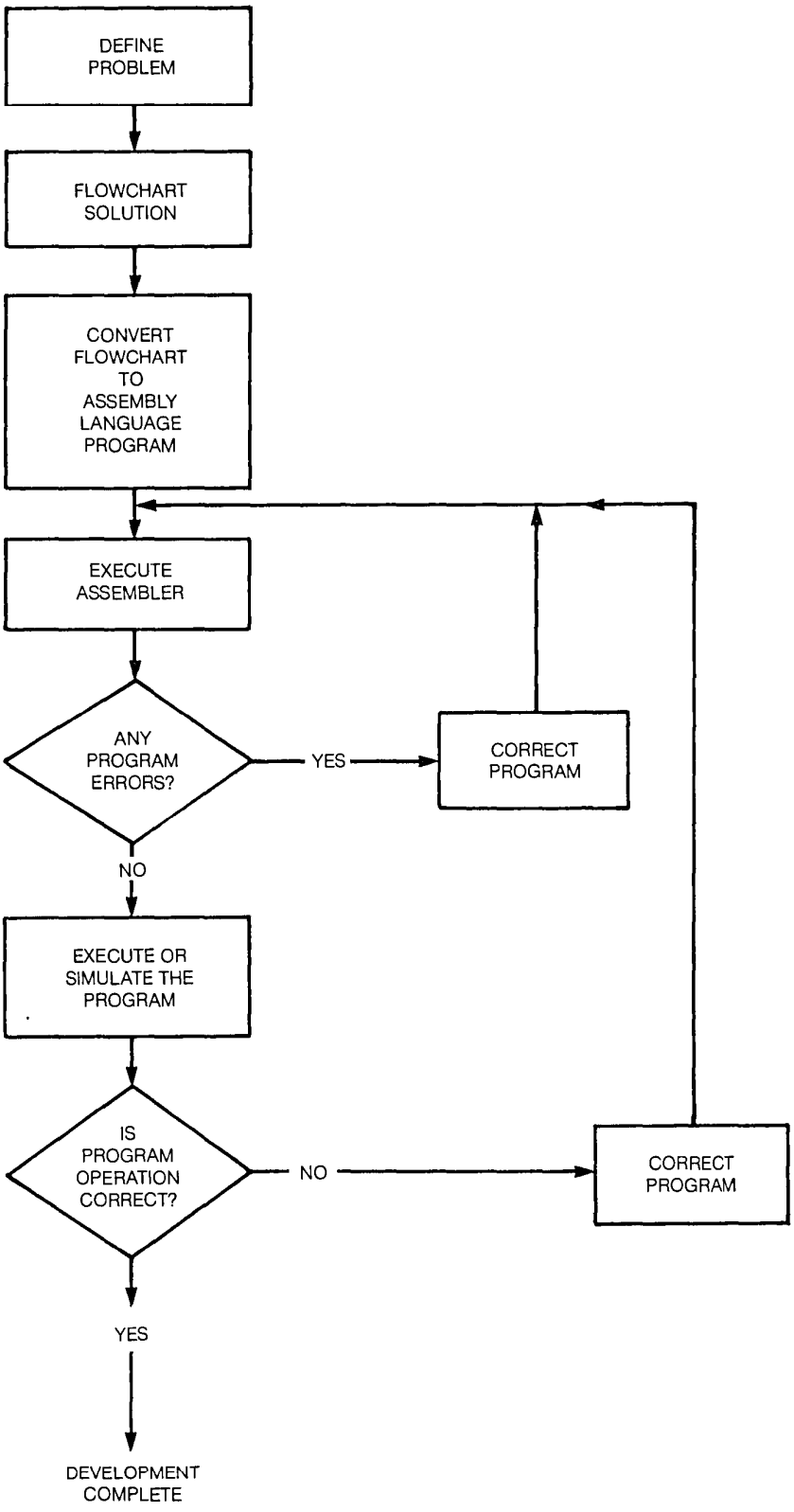


Figure 7-1. Program Development Flowchart

Mnemonic or Opcode Field

This field contains the mnemonic code of one of the instructions, one of the assembly language directives, or a symbol representing one of the program defined operations. This field begins after the last blank following the label field. Examples of instruction mnemonics include A for addition and MOV for data movement. The mnemonic field is required since it identifies which operation is to be performed.

Operands Field

The operands specify the memory locations of the data to be used by the instruction. This field begins following the last blank that follows the mnemonic field. The memory locations can be specified by using constants, symbols, or expressions, to describe one of several addressing modes available.

Comment Field

Comments can be entered after the last blank that follows the operands field. If the first character position of the source statement contains an asterisk (*), the entire source statement is a comment. Comments are listed in the source portion of the Assembler listing, but have no affect on the object code.

Terms and Symbols

Symbols are used in the label field, the operator field, and the operand field. A symbol is a string of alphanumeric characters, beginning with an alphabetic character.
Terms are used in the operand fields of instructions and assembler directives. A term is a decimal or hexadecimal constant, an absolute assembly-time constant, or a label having an absolute value. Expressions can also be used in the operand fields of instructions and assembler directives.

·7

Constants

Constants can be decimal integers (written as a string of numerals) in the range of −32,768 to +65,535. For example:

257

Constants can also be hexadecimal integers (a string of hexadecimal digits preceded by >). For example:

>09AF

ASCII character constants can be used by enclosing the desired character string in single quotes. For example:

'DX'

Throughout this book the subscript 16 is used to denote base 16 numbers. For example, the hexadecimal number 09AF is written $09AF_{16}$.

Symbols

Symbols must begin with an alphabetic character and contain no blanks. Only the first six characters of a symbol are processed by the Assembler.

The Assembler predefines the dollar sign ($) to represent the current location in the program. The symbols R0 through R15 are used to represent workspace registers 0 through 15, respectively.

A given symbol can be used as a label only once, since it is the symbolic name of the address of the instruction. Symbols defined with the DXOP directive are used in the OPCODE field. Any symbol in the OPERANDS field must have been used as a label or defined by a REF directive.

Expressions

Expressions are used in the OPERANDS fields of assembly language statements. An expression is a constant, a symbol, or a series of constants and symbols separated by the following arithmetic operators:

+ addition
− subtraction
* multiplication
/ division

Unary minus is performed first and then the expression is evaluated from left to right. A unary minus is a minus sign (negation) in front of a number or a symbol.

The expression must not contain any imbedded blanks or extended operation defined (DXOP directive) symbols.

The multiplication and division operations must be used on absolute code symbols. The result of evaluating the expression up to the multiplication or division operator must be an absolute value. There must not be more than one more relocatable symbol added to an expression than are subtracted from it.

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 32 and the value of symbol RED.
440/2−RED	220 minus the value of symbol RED.

ASSEMBLER DIRECTIVES

General Information

The assembler directives are used to assign values to program symbolic names, address locations, and data. There are directives to set up linkage between program modules and to control output format, titles, and listings.
The assembler directives take the general form of:

LABEL DIRECTIVE EXPRESSION COMMENT

The LABEL field begins in column one and extends to the first blank. It is optional on all directives except the EQU directive which requires a label. There is no label in the OPTION directive. When no label is present, the first character position in the field must be a blank. When a label is used (except in an EQU directive) the label is assigned the current value of the location counter.

The two required directives are:

IDT	Assign a name to the program
END	Terminate assembly

The most commonly used optional directives are:

EQU	Assign a value to a label or a data name.
RORG	Relocatable Origin
BYTE	Assign values to successive bytes of memory
DATA	Assign 16 bit values to successive memory words
TEXT	Assign ASCII values to successive bytes of memory

Other directives include:

AORG	Absolute (non-relocatable) Origin
DORG	Dummy Origin
BSS	Define bytes of storage beginning with symbol
BES	Define bytes of storage space ending with symbol
DXOP	Define an extended operation
NOP	No operation Pseudo-instruction
RT	Return from subroutine Pseudo-instruction
PAGE	Skip to new page before continuing listing
TITL	Define title for page headings
LIST	Allows listing of source statements
UNL	Prevents listing of source statements
OPTION	Selects output option to be used
DEF	Define symbol for external reference
REF	Reference to an external source

Required Directives

Two directives must be supplied to identify the beginning and end of the assembly language program. The IDT directive must be the first statement and the END directive must be the last statement in the assembly language program.

Program Identifier — IDT

This directive assigns a name to the program and must precede any directive that generates object code. The basic format is:

```
IDT  'Name'
```

The name is the program name consisting of up to 8 characters. As an example, if a program is to be named Convert, the basic directive would be:

```
IDT  'CONVERT'
```

The name is printed only when the directive is printed in the source listing.

Program End — END

This directive terminates the assembly. Any source statement following this directive is ignored. The basic format is:

```
END
```

Initialization Directives

These directives are used to establish values for program symbols and constants.

Define Assembly-Time Constant — EQU

Equate is used to assign values to program symbols. The symbol to be defined is placed in the label field and the value or expression is placed in the Expression field:

```
Symbol  EQU  Expression
```

The symbol can represent an address or a program parameter. This directive allows the program to be written in general symbolic form. The equate directive is used to set up the symbol values for a specific program application.

The following are examples of the use of the Equate directive:

```
TIME    EQU   HOURS+5
N       EQU   8
VAR     EQU   >8000
```

BYTE
DATA
TEXT

Initialize Memory

These directives provide for initialization of successive 8 bit bytes of memory with numerical data (BYTE directive) or with ASCII character codes (TEXT directive). The DATA directive provides for the initialization of successive 16 bit words with numerical data.

The formats are the same for all three directives:

```
Directive   Expression-list
```

The Label and Comment are optional. The expression or value list contains the data entries for the 8 bit bytes (BYTE directive), or the 16 bit words (DATA directive), or a character string enclosed in quotes (TEXT directive).

Examples of the use and effects of these directives are shown in *Figure 7-2.*

Program Location Directives

These directives affect the location counter by causing the instructions to be located in specified areas of memory.

AORG
RORG
DORG

Origin Directives

▸7

These directives set the address of the next instruction to the value listed in the expression field of the directive:

```
Directive   Expression
```

The expression field is required on all except the RORG directive. It is a value or an expression (containing only previously defined symbols). This value is the address of the next instruction and is the value that is assigned to the label (if any) and to the location counter. The AORG and DORG expressions must result in an absolute value and contain no character constants.

Example Directives:

```
KONS    BYTE  >10, -1, 'A', 'B', N + 3
WD1     DATA  >01FF, 3200, -'AF', 8, N+>1000
MSG1    TEXT  'EXAMPLE'
```

AFFECTS ON MEMORY LOCATION	MEMORY DATA: DIRECTIVE ENTRY	RESULTING DATA (BINARY FORM)				RESULTING DATA (HEXADECIMAL)
KONS	>10,-1	0001	0000	1111	1111	1 OFF
KONS+2	'A','B'	0100	0001	0100	0010	4142
KNOS+4	N+3	0000	1011	X	X	0B--
.	.	.	.	.	.	.
.	.	.	.	.	.	.
WD1	>01FF	0000	0001	1111	1111	01FF
WD1+2	3200	0000	1100	1000	0000	0C80
WD1+4	-'AF'	1011	1110	1011	1010	BEBA
WD1+6	8	0000	0000	0000	1000	0008
WD1+8	N+>1000	0001	0000	0000	1000	1008
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
MSG1	'EX'	0100	0101	0101	1000	4558
MSG1+2	'AM'	0100	0001	0101	1101	414D
MSG1+4	'PL'	0101	0000	0100	1100	504C
MSG1+6	'E'	0100	0101	X	X	4E--

XX (--) is original unaltered data in this location. N is assumed to be previously defined as 8.

Figure 7-2. Initialization Directive Examples

The AORG directive causes this value to be absolute and fixed. For example:

```
AORG >1000+X
```

If X has been previously defined to have an absolute value of 6, the next instruction would be unalterably located at the address 1006_{16}. If a label had been included, it would have been assigned this same value.

The RORG directive causes this value to be relative or relocatable so that subsequent operations by the assembler or simulator can relocate the block of instructions to any desired area of memory. Thus, a relocatable block of instructions occupying memory locations 1000_{16} to 1020_{16} could be moved by subsequent simulator (or other software) operations to locations 2000_{16} to 2020_{16}. An example RORG statement is:

```
SEG1    RORG >1000
```

This directive would cause SEG1 and the value of the location counter (address of the next instruction) to be set to 1000_{16}. This and all subsequent locations are relocatable.

```
SEG2    RORG
```

This directive would cause subsequent instructions to be at relocatable addresses. SEG2 and the address of the next instruction would be set to the value of the location counter.

The DORG directive causes the instructions to be listed but the assembler does not generate object code that can be passed on to simulators or other subsystems. However, symbols defined in the dummy section would then be legitimate symbols for use in the AORG or RORG program sections. For example:

```
DORG 0
```

The labels with the subsequent dummy section of instructions will be assigned values relative to the start of the section (the instruction immediately following this directive). No object code would be generated for this section.

An RORG directive is used after a DORG or AORG section to cause the subsequent instructions to be relocatable object code. If no origin directives are included in the assembly language program, all object code is relocatable starting at (referenced to) an address of 0.

Storage Allocation Directives

BES
BSS

These directives reserve a block of memory (range of addresses) for data storage by advancing the location counter by the amount specified in the expression field. Thus, the instruction after the directive will be at an address equal to the expression value plus the address of the instruction just before the directive.

▶7

Basic Formats:

```
BES    Expression
BSS    Expression
```

If a label is included in the BSS directive it is assigned the value of the location counter at the *first byte* if the storage block. If the label is included in the BES directive it is assigned the value of the location counter for the instruction *after* the block.

The Expression designates the number of bytes to be reserved for storage. It is a value or an expression containing no character constants. Expressions must contain only previously defined symbols and result in an absolute value.

Examples:

```
BUFF1   BES   >10
```

A 16 byte buffer is provided. Had the location counter contained the value 100_{16} (FF_{16} was the address of the previous instruction), the new value of the location counter would be 110_{16}, and this would be the value assigned to the symbol BUFF1. The next instruction after the buffer would be at address 110_{16}.

```
BUFF2   BSS   20
```

If the previous instruction is located at FF_{16}, BUFF2 will be assigned the value 100_{16}, and the next instruction will be located at 114_{16}. A 20 byte area of storage with addresses 100_{16} through 113_{16} has been reserved.

Word Boundary — EVEN

This directive causes the location counter to be set to the next even address (beginning of the next word) if it currently contains an odd address. The basic format is:

```
EVEN
```

The label is assigned the value of the location counter prior to the EVEN directive.

Program Listing Control Directives

These directives control the printer, titling, and listing provided by the assembler.

Output Options — OPTION

The basic format of this directive is:

```
OPTION   Keyword-list
```

No label is permitted. The keywords control the listing as follows:

Keyword	*Listing*
XREF	Print a cross reference listing.
OBJ	Print a hexadecimal listing of the object code.
SYMT	Print a symbol table with the object code.

Example:

```
OPTION XREF,SYMT
```

Print a cross reference listing and the symbol table with the object code.

Advance Page — PAGE

This directive causes the assembly listing to continue at the top of the next page. The basic format is:

```
PAGE
```

Page Title — TITL

This directive specifies the title to be printed at the top of each page of the assembler listing. The basic format is:

```
TITL  'String'
```

The String is the title enclosed in single quotes. For example:

```
TITL  'REPORT GENERATOR'
```

Source Listing Control — LIST UNL

These directives control the printing of the source listing. UNL inhibits the printing of the source listing: LIST restores the listing. The basic formats are:

```
UNL
LIST
```

Extended Operation Definition — DXOP

This directive names an extended operation. Its format is:

▶7

```
DXOP  SYMBOL, Term
```

The symbol is the desired name of the extended operation. Term is the corresponding number of the extended operation. For example:

```
DXOP  DADD,13
```

defines DADD as extended operation 13. Once DADD has been so defined, it can be used as the name of a new operation, just as if it were one of the standard instruction mnemonics.

Program Linkage Directives

These directives enable program modules to be assembled separately and then integrated into an executable program.

External Definition — DEF

This directive makes one or more symbols available to other programs for reference. Its basic format is:

```
DEF   Symbol-list
```

Symbol-list contains the symbols to be defined by the program being assembled. For example:

```
DEF   ENTER,  ANS
```

causes the assembler to include the Symbols ENTER and ANS in the object code so that they are available to other programs. When DEF does not precede the source statements that contain the symbols, the assembler identifies the symbols as multi-defined symbols.

External Reference — REF

This directive provides access to symbols defined in other programs. The basic format is:

```
REF   Symbol-list
```

The Symbol-list contains the symbols to be included in the object code and used in the operand fields of subsequent source statements. For example:

```
REF   ARG1,ARG2
```

causes the symbols ARG1 and ARG2 to be included in the object code so that the corresponding address can be obtained from other programs.

Note: If a REF symbol is the first operand of a DATA directive causing the value of the symbol to be in 0 absolute location, the symbol will not be linked correctly in location 0. 7◄

ASSEMBLER OUTPUT

Introduction

The types of information provided by Assemblers include:

Source Listing	— Shows the source statements and the resulting object code.
Error Messages	— Errors in the assembly language program are indicated.
Cross Reference	— Summarizes the label definitions and program references.
Object Code	— Shows the object code in a tagged record format to be passed on to a computer or simulator for execution.

SOURCE LISTING

Assemblers produce a source listing showing the source statements and the resulting object code. A typical listing is shown in *Figure 7-3.*

```
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
```

Figure 7-3. Typical Source Listing.

The first line available in a listing is the title line which will be blank unless a TITL directive has been used. After this line, a line for each source statement is printed. For
▶7 example:

```
0018    0156    C820    MOV    @INIT+3,@3
        0158    012B'
        015A    0003
```

In this case the source statement:

```
MOV   @INIT+3,@3
```

produces 3 lines of object code. The source statement number 18 applies to the entire 3 line entry. Each line has its own location counter value (0156, 0158, and 015A). C820 is the OPCODE for MOV with symbolic memory addressing.

012B' is the value for INIT + 3. 0003 is for the direct address 3. The apostrophe (') after 012B indicates this address is program-relocatable. Source statements are numbered sequentially, whether they are listed or not (listing could be prevented by using the UNLIST directive).

9900
Reference Data

INSTRUCTION FORMAT

FORMAT (USE)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1 (ARITH)	OP CODE			B	T_D		D				T_S		S			
2 (JUMP)	SIGNED DISPLACEMENT*															
3 (LOGICAL)	OP CODE						D				T_S		S			
4 (CRU)	OP CODE						C				T_S		S			
5 (SHIFT)	OP CODE								C				W			
6 (PROGRAM)	OP CODE										T_S		S			
7 (CONTROL)	OP CODE												NOT USED			
	OP CODE											NU	W			
8 (IMMEDIATE)	IMMEDIATE VALUE															
9 (MPY,DIV,XOP)	OP CODE						D				T_S		S			

KEY

B = BYTE INDICATOR
(1 = BYTE, 0 = WORD)
T_D = D ADDR, MODIFICATION
D = DESTINATION ADDR.
T_S = ADDR. MODIFICATION
S = SOURCE ADDR.
C = XFR OR SHIFT LENGTH (COUNT)
W = WORKSPACE REGISTER NO.
* = SIGNED DISPLACEMENT OF −128 TO +127 WORDS
NU = NOT USED

T_D/T_S FIELD

	CODE	EFFECTIVE ADDRESS	MNEMONIC
00:	REGISTER	WP + 2 · [S OR D]	Rn
01:	INDIRECT	(WP + 2 · [S OR D])	*Rn
10:	INDEXED (S OR D ≠ 0)	(WP + 2 · [S OR D]) + (PC); PC ← PC + 2	NUM (Rn)
10:	SYMBOLIC (DIRECT, S OR D = 0)	(PC); PC ← PC + 2	NUM
11:	INDIRECT WITH AUTO INCREMENT	(WP + 2 · [S OR D]); INCREMENT EFF. ADDR.	*Rn+

STATUS REGISTER

▶7

0	1	2	3	4	5	6	7 ... 11	12 ... 15
L>	A>	=	C	O	P	X	RESERVED	INTERRUPT MASK

0 — LOGICAL GREATER THAN
1 — ARITHMETIC GREATER THAN
2 — EQUAL/TB INDICATOR
3 — CARRY FROM MSB
4 — OVERFLOW
5 — PARITY (ODD NO. OF BITS SET)
6 — XOP IN PROGRESS

INTERRUPT MASK
F = ALL INTERRUPTS ENABLED
0 = ONLY LEVEL 0 ENABLED

INTERRUPTS

TRAP ADDR	WP
TRAP ADDR + 2	PC

LEVEL	ID	TRAP ADDR	LEVEL	ID	TRAP ADDR
0	RESET	0000	8	EXTERNAL	0020
1	EXTERNAL	0004	9	EXTERNAL	0024
2	EXTERNAL	0008	10	EXTERNAL	0028
3	EXTERNAL	000C	11	EXTERNAL	002C
4	EXTERNAL	0010	12	EXTERNAL	0030
5	EXTERNAL	0014	13	EXTERNAL	0034
6	EXTERNAL	0018	14	EXTERNAL	0038
7	EXTERNAL	001C	15	EXTERNAL	003C

NOTES: 1) XOP VECTORS 0—15 OCCUPY MEMORY LOCATIONS 0040-007C
2) LOAD VECTOR OCCUPIES MEMORY LOCATIONS FFFC-FFFF

BLWP TRANSFERS	RTWP TRANSFERS	BL TRANSFER	XOP TRANSFER
WP → NEW W13	CURRENT W13 → WP	PC → W11	EFF. ADDR. → NEW W11
PC → NEW W14	CURRENT W14 → PC		WP → NEW W13
ST → NEW W15	CURRENT W15 → ST		PC → NEW W14
			ST → NEW W15
			1 → ST6

INSTRUCTIONS BY MNEMONIC

MNEMONIC	OP CODE	FORMAT	RESULT COMPARED TO ZERO	STATUS AFFECTED	INSTRUCTIONS
A	A000	1	Y	0-4	ADD(WORD)
AB	B000	1	Y	0-5	ADD(BYTE)
ABS	0740	6	Y	0-4	ABSOLUTE VALUE
AI	0220	8	Y	0-4	ADD IMMEDIATE
ANDI	0240	8	Y	0-2	AND IMMEDIATE
B	0440	6	N	—	BRANCH
BL	0680	6	N	—	BRANCH AND LINK (W11)
BLWP	0400	6	N	—	BRANCH LOAD WORKSPACE POINTER
C	8000	1	N	0-2	COMPARE (WORD)
CB	9000	1	N	0-2,5	COMPARE (BYTE)
CI	0280	8	N	0-2	COMPARE IMMEDIATE
CKOF	03C0	7	N	—	EXTERNAL CONTROL
CKON	03A0	7	N	—	EXTERNAL CONTROL
CLR	04C0	6	N	—	CLEAR OPERAND
COC	2000	3	N	2	COMPARE ONES CORRESPONDING
CZC	2400	3	N	2	COMPARE ZEROES CORRESPONDING
DEC	0600	6	Y	0-4	DECREMENT (BY ONE)
DECT	0640	6	Y	0-4	DECREMENT (BY TWO)
DIV	3C00	9	N	4	DIVIDE
IDLE	0340	7	N	—	COMPUTER IDLE
INC	0580	6	Y	0-4	INCREMENT (BY ONE)
INCT	05C0	6	Y	0-4	INCREMENT (BY TWO)
INV	0540	6	Y	0-2	INVERT (ONES COMPLEMENT)
JEQ	1300	2	N	—	JUMP EQUAL (ST2 = 1)

INSTRUCTIONS BY MNEMONIC

JGT	1500	2	N	—	JUMP GREATER THAN (ST1 = 1)
JH	1B00	2	N	—	JUMP HIGH (STO = 1 AND ST2 = 0)
JHE	1400	2	N	—	JUMP HIGH OR EQUAL (STO OR ST2 = 1)
JL	1A00	2	N	—	JUMP LOW (STO AND ST2 = 0)
JLE	1200	2	N	—	JUMP LOW OR EQUAL (STO = 0 OR ST2 = 1)
JLT	1100	2	N	—	JUMP LESS THAN (ST1 AND ST2 = 0)
JMP	1000	2	N	—	JUMP UNCONDITIONAL
JNC	1700	2	N	—	JUMP NO CARRY (ST3 = 0)
JNE	1600	2	N	—	JUMP NOT EQUAL (ST2 = 0)
JNO	1900	2	N	—	JUMP NO OVERFLOW (ST4 = 0)
JOC	1800	2	N	—	JUMP ON CARRY (ST3 = 1)
JOP	1C00	2	N	—	JUMP ODD PARITY (ST5 = 1)
LDCR	3000	4	Y	0-2,5	LOAD CRU
LI	0200	8	N	0-2	LOAD IMMEDIATE
LIMI	0300	8	N	12-15	LOAD IMMEDIATE TO INTERRUPT MASK
LREX	03E0	7	N	12-15	EXTERNAL CONTROL
LWPI	02E0	8	N	—	LOAD IMMEDIATE TO WORKSPACE POINTER
MOV	C000	1	Y	0-2	MOVE (WORD)
MOVB	D000	1	Y	0-2,5	MOVE (BYTE)
MPY	3800	9	N	—	MULTIPLY
NEG	0500	6	Y	0-4	NEGATE (TWO'S COMPLEMENT)
ORI	0260	8	Y	0-2	OR IMMEDIATE
RSET	0360	7	N	12-15	EXTERNAL CONTROL
RTWP	0380	7	N	0-6,12-15	RETURN WORKSPACE POINTER
S	6000	1	Y	0-4	SUBTRACT (WORD)
SB	7000	1	Y	0-5	SUBTRACT (BYTE)
SBO	1D00	2	N	—	SET CRU BIT TO ONE
SBZ	1E00	2	N	—	SET CRU BIT TO ZERO
SETO	0700	6	N	—	SET ONES
SLA	0A00	5	Y	0-4	SHIFT LEFT (ZERO FILL)
SOC	E000	1	Y	0-2	SET ONES CORRESPONDING (WORD)
SOCB	F000	1	Y	0-2,5	SET ONES CORRESPONDING (BYTE)
SRA	0800	5	Y	0-3	SHIFT RIGHT (MSB EXTENDED)
SRC	0800	5	Y	0-3	SHIFT RIGHT CIRCULAR
SRL	0900	5	Y	0-3	SHIFT RIGHT (LEADING ZERO FILL)
STCR	3400	4	Y	0-2,5	STORE FROM CRU
STST	02C0	8	N	—	STORE STATUS REGISTER
STWP	02A0	8	N	—	STORE WORKSPACE POINTER
SWPB	06C0	6	N	—	SWAP BYTES
SZC	4000	1	Y	0-2	SET ZEROES CORRESPONDING (WORD)
SZCB	5000	1	Y	0-2,5	SET ZEREOS CORRESPONDING (BYTE)
TB	1F00	2	N	2	TEST CRU BIT
X	0480	6	N	—	EXECUTE
XOP	2C00	9	N	6	EXTENDED OPERATION
XOR	2800	3	Y	0-2	EXCLUSIVE OR
DCA	2C00	9	N	0-3,5,7	DECIMAL CORRECT ADD
DCS	2C00	9	N	0-3,5,7	DECIMAL CORRECT SUB
LIIM	2C00	9	N	14,15	LOAD INTERRUPT MASK

ILLEGAL OP CODES 0000-01FF;0320-033F;0780-07FF;0C00-OFFF

INSTRUCTIONS BY OP CODE

OP CODE	MNEMONIC	OP CODE	MNEMONIC
0000-01FF	ILLEGAL	1000	JMP
0200	LI	1100	JLT
0220	AI	1200	JLE
0240	ANDI	1300	JEQ
0260	ORI	1400	JHE
0280	CI	1500	JGT
02A0	STWP	1600	JNE
02C0	STST	1700	JNC
02E0	LWPI	1800	JOC
0300	LIMI	1900	JNO
0320-033F	ILLEGAL	1A00	JL
0340	IDLE	1B00	JH
0360	RSET	1C00	JOP
0380	RTWP	1D00	SBO
03A0	CKON	1E00	SBZ
03C0	CKOF	1F00	TB
03E0	LREX	2000	COC
0400	BLWP	2400	CZC
0440	B	2800	XOR
0480	X	2C00	XOP
04C0	CLR	3000	LDCR
0500	NEG	3400	STCR
0540	INV	3800	MPY
0580	INC	3C00	DIV
05C0	INCT	4000	SZC
0600	DEC	5000	SZCB
0640	DECT	6000	S
0680	BL	7000	SB
06C0	SWPB	8000	C
0700	SETO	9000	CB
0740	ABS	A000	A
0780-07FF	ILLEGAL	B000	AB
0800	SRA	C000	MOV
0900	SRL	D000	MOVB
0A00	SLA	E000	SOC
0B00	SRC	F000	SOCB
0C00	ILLEGAL		

PSEUDO-INSTRUCTIONS

MNEMONIC	PSEUDO-INSTRUCTIONS	CODE GENERATED
NOP	NO OPERATION	1000
RT	RETURN	045B

PIN DESCRIPTIONS

PIN #	FUNCTION	PIN #	FUNCTION	PIN #	FUNCTION
1	V_{BB}	23	A1	44	D3
2	V_{CC}	24	A0	45	D4
3	WAIT	25	$\phi 4$	46	D5
4	$\overline{\text{LOAD}}$	26	V_{SS}	47	D6
5	HOLDA	27	V_{DD}	48	D7
6	$\overline{\text{RESET}}$	28	$\phi 3$	49	D8
7	IAQ	29	DBIN	50	D9
8	$\phi 1$	30	CRUOUT	51	D10
9	$\phi 2$	31	CRUIN	52	D11
10	A14	32	$\overline{\text{INTREQ}}$	53	D12
11	A13	33	IC3	54	D13
12	A12	34	IC2	55	D14
13	A11	35	IC1	56	D15
14	A10	36	IC0	57	NC
15	A9	37	NC	58	NC
16	A8	38	NC	59	NC
17	A7	39	NC	60	CRUCLK
18	A6	40	NC	61	$\overline{\text{WE}}$
19	A5	41	D0	62	READY
20	A4	42	D1	63	$\overline{\text{MEMEN}}$
21	A3	43	D2	64	$\overline{\text{HOLD}}$
22	A2				

ASSEMBLER DIRECTIVES

MNEMONIC	DIRECTIVE
AORG	ABSOLUTE ORIGIN
BES	BLOCK ENDING WITH SYMBOL
BSS	BLOCK STARTING WITH SYMBOL
BYTE	INITIALIZE BYTE
DATA	INITIALIZE WORD
DEF	EXTERNAL DEFINITION
DORG	DUMMY ORIGIN
DXOP	DEFINE EXTENDED OPERATION
END	PROGRAM END
EQU	DEFINITE ASSEMBLY — TIME CONSTANT
EVEN	WORD BOUNDARY
IDT	PROGRAM IDENTIFIER
LIST	LIST SOURCE
PAGE	PAGE EJECT
REF	EXTERNAL REFERENCE
RORG	RELOCATABLE ORIGIN
TEXT	INITIALIZE TEXT
TITL	PAGE TITLE
UNL	NO SOURCE LIST

USASCII/HOLLERITH CHARACTER CODE

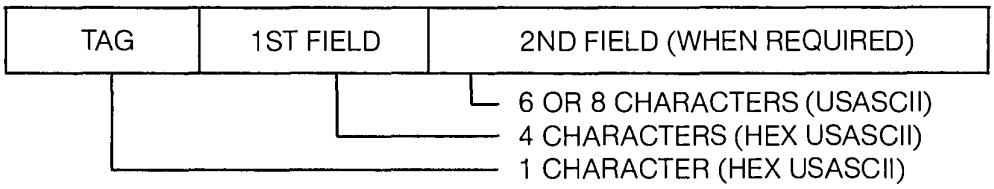
CHAR.	USASCII (HEXADECIMAL)	HOLLERITH*	CHAR.	USASCII (HEXADECIMAL)	HOLLERITH*
NUL	00		3	33	3
SOH	01		4	34	4
STX	02		5	35	5
ETX	03		6	36	6
EOT	04		7	37	7
ENQ	05		8	38	8
ACK	06		9	39	9
BEL	07		:	3A	2-8
BS	08		;	3B	11-6-8
HT	09		<	3C	12-4-8
LF	0A		=	3D	6-8
VT	0B		>	3E	0-6-8
FF	0C		?	3F	0-7-8
CR	0D		@	40	4-8
S0	0E		A/a	41/61	12-1
SI	0F		B/b	42/62	12-2
DLE	10		C/c	43/63	12-3
DC1	11		D/d	44/64	12-4
DC2	12		E/e	45/64	12-5
DC3	13		F/f	46/66	12-6
DC4	14		G/g	47/67	12-7
NAK	15		H/h	48/68	12-8
SYN	16		I/i	49/69	12-9
ETB	17		J/j	4A/6A	11-1
CAN	18		K/k	4B/6B	11-2
EM	19		L/l	4C/6C	11-3
SUB	1A		M/m	4D/6D	11-4
ESC	1B		N/n	4E/6E	11-5
FS	1C		O/o	4F/6F	11-6
GS	1D		P/p	50/70	11-7
RS	1E		Q/q	51/71	11-8
US	1F		R/r	52/72	11-9
SPACE	20	BLANK	S/s	53/73	0-2
!	21	11-2-8	T/t	54/74	0-3
"	22	7-8	U/u	55/75	0-4
#	23	3-8	V/v	56/76	0-5
$	24	11-3-8	W/w	57/77	0-6
%	25	0-4-8	X/x	58/78	0-7
&	26	12	Y/y	59/79	0-8
'	27	5-8	Z/z	5A/7A	0-9
(	28	12-5-8	[	5B	12-2-8
)	29	11-5-8	\	5C	
*	2A	11-4-8	]	5D	12-7-8
+	2B	12-6-8	^	5E	11-7-8
,	2C	0-3-8	_	SF	0-5-8
-	2D	11	`	60	
.	2E	12-3-8	{	7B	
/	2F	0-1	>	7C	
0	30	0	}	7D	
1	31	1	~	7E	
2	32	2	DEL	7F	

*PUNCH IN CARD ROWS

HEX-DECIMAL TABLE

EVEN BYTE				ODD BYTE			
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4,096	1	256	1	16	1	1
2	8,192	2	512	2	32	2	2
3	12,288	3	768	3	48	3	3
4	16,384	4	1,024	4	64	4	4
5	20,480	5	1,280	5	80	5	5
6	24,576	6	1,536	6	96	6	6
7	28,672	7	1,792	7	112	7	7
8	32,766	8	2,048	8	128	8	8
9	36,864	9	2,304	9	144	9	9
A	40,960	A	2,560	A	160	A	10
B	45,066	B	2,816	B	176	B	11
C	49,152	C	3,072	C	192	C	12
D	53,248	D	3,328	D	208	D	13
E	57,344	E	3,584	E	224	E	14
F	61,440	F	3,840	F	240	F	15

OBJECT RECORD FORMAT AND CODE



TAG	FIRST FIELD	SECOND FIELD	MEANING
0	LENGTH OF ALL RELOCATABLE CODE	PROGRAM ID (8-CHARACTER)	PROGRAM START
1	ADDRESS	(NOT USED)	ABSOLUTE ENTRY ADDRESS
2	ADDRESS	(NOT USED)	RELOCATABLE ENTRY ADDRESS
3	LOCATION OF LAST APPEARANCE OF SYMBOL	6 CHARACTER SYMBOL	EXTERNAL REFERENCE LAST USED IN RELOCATABLE CODE
4	LOCATION OF LAST APPEARANCE OF SYMBOL	6 CHARACTER SYMBOL	EXTERNAL REFERENCE LAST USED IN ABSOLUTE CODE
5	LOCATION	6 CHARACTER SYMBOL	RELOCATABLE EXTERNAL DEFINITION
6	LOCATION	6 CHARACTER SYMBOL	ABSOLUTE EXTERNAL DEFINITION
7	CHECKSUM FOR CURRENT RECORD	(NOT USED)	CHECKSUM
8	ANY VALUE	(NOT USED)	IGNORE CHECKSUM VALUE
9	LOAD ADDRESS	(NOT USED)	ABSOLUTE LOAD ADDRESS
A	LOAD SDDRESS	(NOT USED)	RELOCATABLE LOAD ADDRESS
B	DATA	(NOT USED)	ABSOLUTE DATA
C	DATA	(NOT USED)	RELOCATABLE DATA
D	LOAD BIAS	(NOT USED)	LOAD BIAS OR OFFSET (NOT A PART OF ASSEMBLER OUTPUT)
E			ILLEGAL
F	(NOT USED)	(NOT USED)	END OF RECORD