CHAPTER 6

## **Instruction Set**

#### 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.

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.

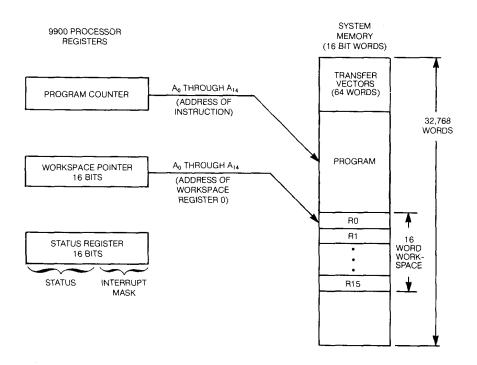


Figure 6-1. 9900 System Memory and Processor Registers.

#### 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.

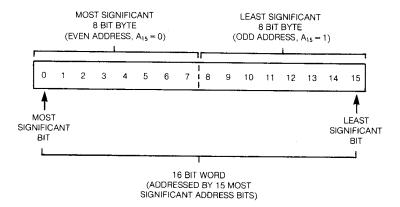


Figure 6-2. Word and Byte Definition.

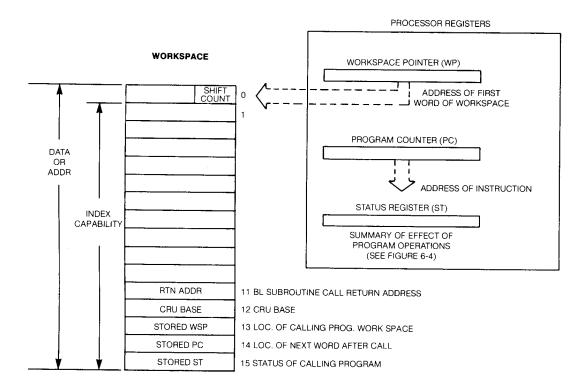


Figure 6-3. Workspace Register Utilization.



- 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:

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

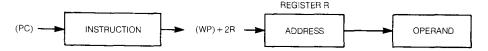


This type of addressing is used to access the often used data contained in the workspace.

#### 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 to clear (CLR) that data word:

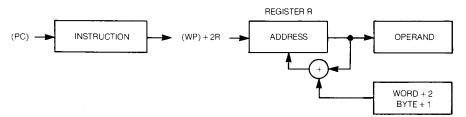
In this instruction the contents of register 3 would not be changed, but the data word addressed by the contents of register 3 would be cleared (set to all zeroes  $-000_{16}$ ). The word address is computed as follows for this type of addressing:



#### 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:

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

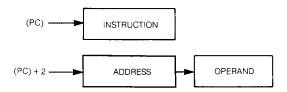
The address of the memory word is contained in the operand field of the instruction and is contained in program memory (ROM) in the word immediately following the operation code word for the instruction. For example, to clear the memory word at address 1000<sub>16</sub>, the following format would be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

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:

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:

$$(PC) + 2$$
  $1000_{16}$  Address of Data

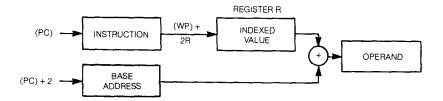
The address of the memory word is thus contained in the instruction itself and is located by the program counter. Since this address is part of the instruction, it cannot be modified by the program. As a result, this type of addressing is used for program variables that occupy a single memory word such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



### SOFTWARE FEATURES OF THE 9900

#### 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:

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:

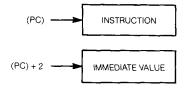
with the value for the symbol DISP defined elsewhere in the assembly language program.

#### 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:

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

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

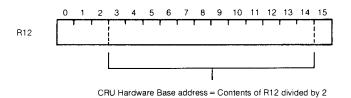
and subtracting from this result the location of LOOP:

$$1A_{16} - 10_{16} = A_{16} = 10$$
 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:



Thus if R12 contains 0400<sub>16</sub> (the software base address), bits 3 through 14 will be 0200<sub>16</sub>. 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<sub>16</sub>, the following CRU instruction would be used:

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

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.

T <sub>d</sub> or T <sub>s</sub> Field Code	00	01	11	10	P) 10
Result	R3 R5	$M(R3) \longrightarrow M(R5)$	$M(R3) \longrightarrow M(R5)$ $R3 + 2 \longrightarrow R3$ $R5 + 2 \longrightarrow R5$	M(ONE)	$M(R3+2) \longrightarrow M(R5+DP)$
MOV Instruction Example Coding	3,5	*. 	*3+,*5+	(aONE, (a)10	(a) 2(3), (a) DP(5)
MC	MOV 3,5	MOV	MOV	MOV	MOV
Memory Location Specified	Workspace Register n Rn	Address given by the contents of workspace register n M(Rn)	As in register Indirect, address register Rn is incremented after the operation (by one for byte operations, by two for word operations)	Address is given by value of exp. M(exp)	Address is the sum of the contents of Rn and the value of exp
Operand Format	u	п *	+ ¤ *	(a) exp	(Dexp(n)
Type of Addressing	Workspace Register	Workspace Register Indirect	Workspace Register Indirect, Autoincrement	Symbolic Memory	Indexed Memory

Figure 6-5. Addressing Modes

Notos.

n is the number of the workspace register:  $0 \le n \le 15$ ; 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.

#### 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:

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<sub>16</sub>.

#### 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:

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.

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*.

Inemonic	L>	<b>'</b> A>	EQ	C	ov	OP	X	Mnemonic	L>	A>	EQ	C	ov	OP	2
A	X	X	X	X	X	-	-	DIV	-	-	_	_	X	-	_
AB	X	X	X	X	$\mathbf{X}$	X	-	IDLE	-	_	_	-	-	-	
ABS	X	X	X	X	$\mathbf{X}$	-	-	INC	X	X	X	X	X	-	
AI	X	X	X	X	X	-	-	INCT	X	X	X	X	X	-	
ANDI	X	X	X	-	-	-	-	INV	X	$\mathbf{X}$	X	-	-	-	
В	-	-	-	-	-	-	-	JEQ	-	-	-	-	-	-	
BL	-	-	-	-	-	-	-	JGT	-	-	-	-	-	-	
BLWP	-	-	-	-	-	-	-	JН	-	-	-	-	-	-	
C	X	X	X	-	-	-	-	JHE	-	-	-	-	-	-	
CB	X	X	X	-	-	X	-	JL	-	-	-	-	-	-	
CI	X	X	X	-	-	-	-	JLE	-	-	-	-	-	-	
CKOF	-	-	-	-	-	-	-	JLT	-	-	-	-	-	-	
CKON	-	-	-	-	-	-	-	JMР	-	-	-	-	-	-	
CLR	-	-	-	-	-	-	-	JNC	-	-	-	-	-	-	
COC	-	-	X	-	-	-	-	JNE	-	-	-	-	-	-	
CZC	-	-	X	-	-	-	-	JNO	-	-	-	-	-	-	
DEC	$\mathbf{X}$	$\mathbf{X}$	X	X	X	-	-	JOC	-	-	-	-	-	-	
DECT	X	X	X	X	X	-	-	JOP	-	-	-	-	-	-	
LDCR	X	$\mathbf{X}$	X	_	_	1	-	SBZ	_	_	_	_	_	_	
LI	X	X	X	-	_	_	_	SETO	-	_	_	-	-	-	
LIMI	-	-	_	-	_	_	-	SLA	X	X	X	X	X	_	
LREX	-	-	_	_	-	_	_	SOC	X	X	$\mathbf{X}$	-	_	_	
LWPI	_	-	-	_	_	-	-	SOCB	X	X	X	_	-	X	
MOV	X	X	X	-	-	-	_	SRA	X	X	X	X	-	-	
MOVB	X	X	$\mathbf{X}$	-	-	X	-	SRC	$\mathbf{X}$	X	$\mathbf{X}$	X	-	-	
MPY	-	-	_	-	_	-	-	SRL	X	X	X	X	-	_	
NEG	X	X	X	X	$\mathbf{X}$	-	-	STCR	X	$\mathbf{X}$	X	_	-	1	
ORI	X	X	$\mathbf{X}$	-	-	-	-	STST	-	-	-	-	-	-	
RSET	-	-	-	-	-	-	-	STWP	-	-	-	_	_	-	
RTWP	X	X	X	X	X	X	X	SWPB	-	-	-	-	-	-	
S	X	X	$\mathbf{X}$	$\mathbf{X}$	X	-	-	SZC	X	X	X	-	_	-	
SB	X	X	X	X	X	X	-	SZCB	X	X	X	-	-	X	
SBO	-	_	-	-	-	-	-	ТВ	-	_	X	_	-	-	
								X	2	2	2	2	2	2	
								XOP	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.

Figure 6-6. Status Bits Affected by Instructions

<sup>2.</sup> 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.

#### 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.

#### 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:

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( $\longrightarrow$ ) 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:

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

$$M(G_s):0$$

which is the operation performed by the MOV instruction:

$$MOV G_s, G_d$$

A specific example of this instruction could be:

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

$$M(ONE) \rightarrow R3$$



#### **DATA TRANSFER INSTRUCTIONS**

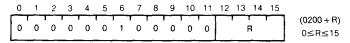
The MOV instructions are used to transfer data from one part of the system to another part. The LOAD instructions are used to initialize registers to desired values. The STORE instructions provide for saving the status register (ST) or the workspace pointer (WP) in a specified workspace register.

#### LOAD IMMEDIATE

LI

Format: LI

₋I 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 <del>→ R</del>7

LI 8,>FF

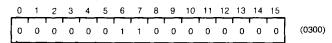
00FF<sub>16</sub> → R8

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

#### 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):

Affect on Status: None

Example: LWPI >0500

Causes 0500<sub>16</sub> 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,,Gd

		2									15	
1	1	0	0	Ta	ĺ	)	-	T <sub>S</sub>	,	S	•	(C)

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:

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

**MOVB** 

### **MOVB**

MOVE Byte

Format: MOVB G,,Gd

_	 2	-									
1	0					7	s .		3	,	(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.

$$MB(G_s) \xrightarrow{} MB(G_d)$$
  
 $MB(G_s):0$ 

Status Bits Affected: LGT, AGT, EQ, OP

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

These instructions would have the following example affects:

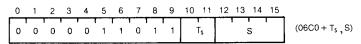
Memory	Contents	Contents
Location	Initially	After Transfer
1C14	<u>20</u> 16	<u>20</u> 16
R3	542B	202B
R8	2123	2123
2123	10 <u>40</u>	1040
R4	0A0C	400C

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 G Format:



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:

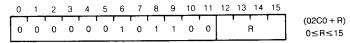
F302 02F3

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

#### STORE STATUS

**STST** 

STST R Format:



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

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

0	1													
0	0	0	0	0	0	1	0	1	0	1	0	- 1	3	(02A0 + R) 0≤R≤15

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

Affect on Status: None

Example: STWP 3

WP is transferred into R3

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

#### **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

Α

Format: A

0	_1	2	3	4	5	6	7	8	9_	10	11	12	13	14	15	
	0	1	0	-	Γ <sub>d</sub>	, <b>1</b>		)		-	s			S		(A)

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:

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

 $G_s,G_d$ 

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

Examples: A 5,@TABLE 
$$R5 + M(TABLE) \longrightarrow M(TABLE)$$
  
A 3,\*2  $R3 + M(R2) \longrightarrow M(R2)$ 

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

$M(G_{s})$	$M(G_d)$	Sum	LGT	AGT**	EQ	$\boldsymbol{C}$	$OV^*$
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

AB G<sub>s</sub>,G<sub>d</sub>

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
	0	1	1							Ţ	s			3		(B)

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) 
$$\longrightarrow$$
 MB(R4), R4 + 2  $\longrightarrow$  R4  $\bigcirc$  AB  $\bigcirc$  TAB,5 MB(TAB) + R5  $\longrightarrow$  R5

To see how the AB works, the following example should be studied: AB @>2120,@>2123

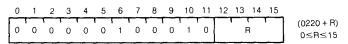
The underlined entries are the addressed and changed bytes.

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.

Al/S

ADD IMMEDIATE

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.

$$R + Value \longrightarrow R$$
,  $R + Value:0$ 

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

Example: AI 6,>C

Adds C<sub>16</sub> to the contents of workspace register 6. If R6 contains 1000<sub>16</sub>, then the instruction will change its contents to 100C<sub>16</sub>, 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  $G_s,G_d$ 

						6								
0	1	1	0	_	T <sub>d</sub>	,	(	)	r –	7	s	,	S	 (6)

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.

$$M(G_d) - M(G_s) \xrightarrow{} M(G_d)$$
  
 $M(G_d) - M(G_s):0$ 

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

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

Memory	Before Subtraction	After Subtraction
Location	Contents	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

		6 7 8 9			_
0 1 1 1	Td	D	T <sub>5</sub>	S	(7)

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.

$$MB(G_d) - MB(G_s) \xrightarrow{\hspace{1cm}} MB(G_d)$$
  
 $MB(G_d) - MB(G_s):0$ 

 $G_sG_d$ 

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

This operation would have the following example result:

Memory	Contents Before	Contents After
Location	Instruction	Instruction
R6	121D	12 <u>1E</u>
121D	3123	4123
R1	1 <u>3</u> 44	<u>F0</u> 44

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

Format:

INC

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.

$$M(G) + 1 \longrightarrow M(G)$$
  
 $M(G) + 1 : 0$ 

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

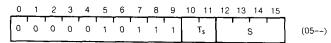
pples: INC @TABL 
$$M(TABL) + 1 \longrightarrow R1$$

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:

$$M(G) + 2 \longrightarrow M(G)$$
  
 $M(G) + 2 : 0$ 

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

Example: INCT 5 (R5) + 2 
$$\longrightarrow$$
 R5

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

DECREMENT

DEC

Format:

DEC

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:

$$\begin{array}{l} M(G) - 1 \longrightarrow M(G) \\ M(G) - 1 : 0 \end{array}$$

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

Example: DEC

@TABL

$$M(TABL) - 1 \longrightarrow M(TABL)$$

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

DECREMENT BY TWO

DECT

DECT G Format:



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:

$$M(G) - 2 \longrightarrow M(G)$$
  
  $M(G) - 2 : 0$ 

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

Example: DECT 3  $(R3) - 2 \longrightarrow R3$ 

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

NEG/ABS

NEGATE NEGATE

Format: N

NEG G

											12 13		
0	0	0	0	0	1	0	1	0	0	T <sub>s</sub>		6	(05)

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

$$-M(G) \xrightarrow{1} M(G)$$
$$-M(G): 0$$

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

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

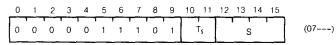
If R5 contained A342<sub>16</sub>, this instruction would cause the R5 contents to changed to  $5CBE_{16}$  and will cause the LGT and AGT status bits to be set to 1.

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.

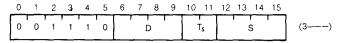
Application: This instruction is used to test the data in location G and then replace the data by its absolute value. This could be used for unsigned arithmetic algorithms such as multiplication.

### **MPY**

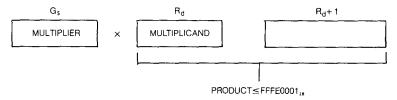
MULTIPLY MPY

Format:

MPY G<sub>s</sub>,R<sub>d</sub>



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.

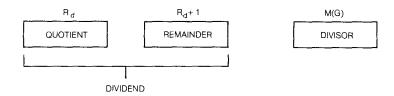
DIV

DIV

Format: DIV Gs,Rd



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.

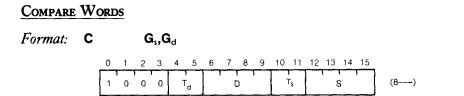
Example: DIV @LOC,2

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.



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:

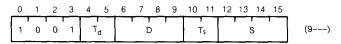
Data at	Data in	Resui	ts of Compa	rison
Location T1	R2	LGT	AGT	EQ
FFFF	0000	1	0	o
7FFF	0000	1	1	0
8000	0000	1	0	0
8000	7FFF	1	0	0
7FFF	7FFF	0	0	1
7FFF	8000	0	1	0

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

COMPARE BYTES

Format:

CB  $G_s,G_d$ 



Operation: The 2's complement 8 bit byte addressed by G<sub>s</sub> is compared to the 2's complement 8 bit byte addressed by G<sub>d</sub>:

 $MB(G_s): MB(G_d)$ 

Status Bits Affected: LGT, AGT, EQ, OP

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

Example: CB

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

			Results of	Comparison	n
R1 data	M(R2) Data	LGT	AGT	EQ	OP
<u>FF</u> FF	FF <u>00</u>	1	0	0	0
<u>7F</u> 00	FF <u>00</u>	1	1	0	1
<u>80</u> 00	FF <u>00</u>	1	0	0	1
8000	FF <u>7F</u>	1	0	0	1
7F00	00 <u>7F</u>	0	0	1	1

The underlined entries indicate the byte addressed.

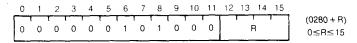
Application: In cases where 8 bit operations are required, CB provides a means of performing byte comparisons for special conversion and recognition problems.

#### COMPARE IMMEDIATE

Cl

Format: C

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<sub>16</sub>, 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

OC G<sub>s</sub>,R

					10				
	1					s		1	(2)

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<sub>16</sub> and location TESTBIT contains C102<sub>16</sub>,

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.

CZC CZC **Instruction Set** 

COMPARE ZEROES CORRESPONDING

Format: CZC

 $G_s,R$ 

_	0_	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
										-	1	s			3	1	(2)

Operation: The data located in the address specified by G<sub>s</sub> 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<sub>s</sub> 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<sub>16</sub> and the R8 location contains 2301<sub>16</sub>,

TESTBIT Data = 
$$\frac{1100\ 0001\ 0000\ 0010}{R8 = \frac{00}{10}10\ 001\frac{1}{X}0000\ 0001}$$

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.

### **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

00240+R) 0 0 0 0 0 0 1 0 0 1 0 0 R 0≤R≤15						10			

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:

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<sub>16</sub>, then (D2AB) AND (6D03) is 4003<sub>16</sub>:

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

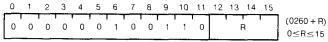
Application: ANDI is used to zero all bits that are not of interest and leave the selected bits (those with ones in Value) unchanged. This can be used to test single bits or isolate portions of the word, such as a four bit group.

ORI

OR IMMEDIATE

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:

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<sub>16</sub>, then R5 will be changed to FFAB<sub>16</sub>:

with LGT being set to 1.

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



Exclusive OR

**XOR** 

Format: XOR

$$G_sR_d$$

				_	10		 _			
0					1	S		s	Υ	(2)

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<sub>16</sub> and R2 contains D2AA<sub>16</sub>, R2 will be changed to BFA9<sub>16</sub>:

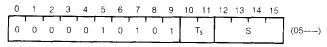
and the LGT status bit will be set to 1. Note that the exclusive OR operation will result in a 1 if only one of the inputs is a 1.

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

#### INVERT

INV

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 1

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.

CLEAR

Format: CLR

G

0 1	2	3	4	5	6	7	8	9	10 11	12 13 14 15	
0 0	0	0	0	1	0	0	1	1	T <sub>5</sub>	S	(04)

Operation: 000016 is placed in the memory location specified by G.

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

Affect on Status: None Example: CLR

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

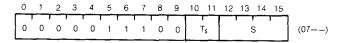
 $0000_{16} \longrightarrow M(R11)$ 

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

SET TO ONE

Format: SETO G

**SETO** 



Operation: FFFF<sub>16</sub> is placed in the memory location specified by G: FFFF<sub>16</sub>  $\longrightarrow$  M(G)

Affect on Status: None

Example: SETO 11

would cause all bits of R11 to be 1.

Application: Similar to CLR

# SOC/SOCB

SET ONES CORRESPONDING

SOC

Format: SOC

				-	_		_	-	10		 		
1	1 -	1	0	T <sub>d</sub>		1	D	l	1	s		3	(E)

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  $M(G_d): 0$ 

 $G_s, G_d$ 

Status Bits Affected: LGT, AGT, EQ

Example: SOC 3,@NEW

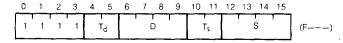
If location NEW contains  $AAAA_{16}$  and  $R_3$  contains  $FF00_{16}$ , the contents at location NEW will be changed to  $FFAA_{16}$  and the LGT status bit will be set to 1.

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

SET ONES CORRESPONDING, BYTE

**SOCB** 

Format: SOCB G,,Gd



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<sub>16</sub> and R8 contains AA24<sub>16</sub>, the most significant byte of R8 will be changed to FA<sub>16</sub> so that R8 will contain FA24<sub>16</sub> and the LGT status bit will be set to 1.

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

SET TO ZEROFS CORRESPONDING

SZC SZC

Format: SZC

 $G_s,G_d$ 

0		_		 			10		-			
0	1	0	0	Γ <sub>d</sub>		0	7	s	_	s S		(4)

Operation: The data addressed by Gs forms a mask for this operation. The bits in the destination data (addressed by G<sub>d</sub>) that correspond to the one bits of the source data (addressed by  $G_{\rm s})$  are cleared. The result is compared to zero and is stored in the  $G_{\rm d}$ location.

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

Status Bits Affected: LGT, AGT, EQ

Example: SZC 5,3

If R5 contains 6D03<sub>16</sub> and R3 contains D2AA<sub>16</sub>, this instruction will cause the R3 contents to change to 92A8<sub>16</sub>:

R5 (Mask) = 
$$0\underline{110}$$
  $\underline{1101}$  0000 00 $\underline{11}$   
R3 = 1101 0010 1010 1010  
Result =  $1\underline{001}$   $\underline{0010}$  1010 10 $\underline{00}$  =  $92A8_{16}$ 

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.

# **SZCB**

SET TO ZEROES CORRESPONDING, BYTES

**SZCB** 

Format: SZCB G,Gd

 	 	5		_	-		 		
		T <sub>d</sub>				s		s S	 (5)

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:

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

Status Bits Affected: LGT, AGT, EQ, OP

Example: SZCB @BITS,@TEST

If location BITS is an *odd* address which locates the data  $18\underline{F0}_{16}$ , and location TEST contains an *even* address which locates the data  $\underline{AA}24_{16}$ , 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.

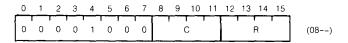
#### SHIFT INSTRUCTIONS

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.

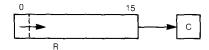
#### SHIFT RIGHT ARITHMETIC

**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.

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
```

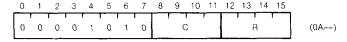
Application: SRA provides binary division by 2<sup>cnt</sup>.

# SLA

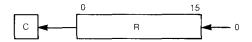
SHIFT LEFT ARITHMETIC

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<sup>Cnt</sup>

SRL/SRC

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<sub>16</sub>, the contents would become 1FFD<sub>16</sub> with the AGT, LGT, and C bits set to 1:

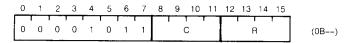
```
R0 Before Shift = 1111 1111 1110 1111 = FFEF<sub>16</sub>
R0 After Shift = 0001 1111 1111 1101 = 1FFD<sub>16</sub>
```

Application: Performs binary division by 2<sup>cnt</sup>

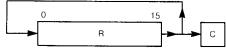
SHIFT RIGHT CIRCULAR

**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

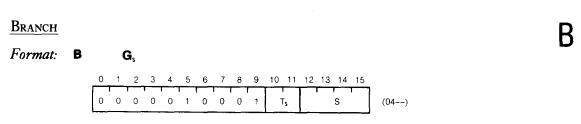
If R2 initially contains FFEF<sub>16</sub>, then after the shift it will contain DFFF<sub>16</sub> with LGT and C set to 1.

```
R2 Before Shift = 1111 1111 1\underline{1101111} = FFEF_{16}
R2 After Shift = \underline{1101} 1111 1111 \underline{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.

#### **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.



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<sub>16</sub>, then the next instruction will be obtained from location 21CC<sub>16</sub>.

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

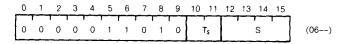
BL

BRANCH AND LINK

BL

Format:

BL G,



Operation: The source address Gs 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: @TRAN

Assume the BL instruction is located at 3200<sub>16</sub> and the value assigned to TRAN is 2000<sub>16</sub>. PC will be loaded with the value 2000<sub>16</sub> (TRAN) and R11 will be loaded with the value 3202<sub>16</sub> (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:

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

### **BLWP**

Branch and Load Workspace Pointer

**BLWP** 

Format: BLWP G.



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.

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.

Instruction Set XOP

Extended Operation XOP

Format: XOP Gs,n

			5						
1			1						(2)

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:

$$M(n \times 4 + 0040_{16}) \longrightarrow WP$$
  
 $M(n \times 4 + 0042_{16}) \longrightarrow PC$   
(Old WP)  $\longrightarrow$  New R13  
(Old PC)  $\longrightarrow$  New R14  
(Old ST)  $\longrightarrow$  New R15  
 $G_s \longrightarrow$  New R11

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

Example: XOP \*1,2

Assume R1 contains 0750<sub>16</sub>. WP is loaded with the word at address 48<sub>16</sub> (first part of transfer vector for extended operation 2) and PC is loaded with the word at address 4A<sub>16</sub>. If location 48<sub>16</sub> contains 0200<sub>16</sub>, this will be the address of R0 of the subprogram workspace. Thus, location 0236<sub>16</sub> (new R11) will be loaded with 0750<sub>16</sub> (contents of R1 in main program), location 023A<sub>16</sub> (new R13) will be loaded with the old WP value, location 023C<sub>16</sub> will be loaded with the old PC value, and location 023E<sub>16</sub> (new R15) will be loaded with the old status value:

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 
$$\longrightarrow$$
 WP  
R14  $\longrightarrow$  PC  
R15  $\longrightarrow$  ST

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

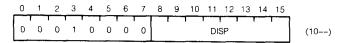
Application: This is used to return from subprograms that were reached by a transfer vector operation such as an interrupt, extended operation, or BLWP instruction.

Unconditional Jump

**JMP** 

Format: JMP

EXP



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.



X

Execute

Format: X

X

X G,



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<sub>16</sub> and location 2000<sub>16</sub> 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.

JH

JL

JHE

JLE

**JGT** 

JLT

JEQ JNE JOC

#### **CONDITIONAL JUMP INSTRUCTIONS**

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

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
0					CC		ĺ			1	DI	SP	ı	1	1	(1)

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	_		_	$L > \bullet \overline{EQ} = 1$	В
JL	X	_	X	_	_	_	L > + EQ = 0	A
JHE	$\mathbf{X}$		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.

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

<sup>\*</sup>CODE is entered in the CODE field of the OPCODE to generate the machine code for the instruction.

SB0

#### **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

SB<sub>0</sub>

Format: SBO



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.

disp

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.

SBZ/ TB

Instruction Set

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

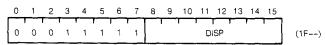
If R12 contains  $0000_{16}$ , the hardware base address is 0 so that the instruction would reset CRU line  $0002_{16}$  to zero.

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

TEST BIT

TR

Format: TB disp



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.

Status Bits Affected: EQ

Example: TB 4

If R12 contains 0140<sub>16</sub>, the hardware base address is A0<sub>16</sub> (which is one-half of 0140<sub>16</sub>):

R12 Contents = 
$$0000 \ \underline{0001} \ \underline{0100} \ \underline{0000}$$

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.

**LDCR** 

LOAD CRU LDCR

Format: LDCR Gs,Cnt



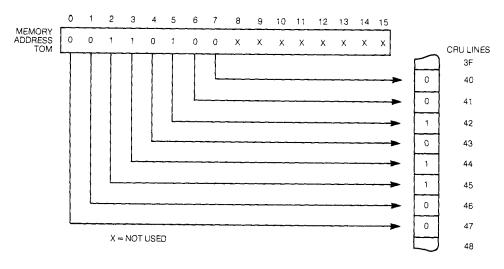
Operation: Cnt specifies the number of bits to be transferred from the data located at the address specified by  $G_s$ , with the first bit transferred from the least significant bit of this data, the next bit from the next least significant bit and so on. If Cnt = 0, the number of bits transferred is 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*.



LDCR @TOM,8 TOM is an even address

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.

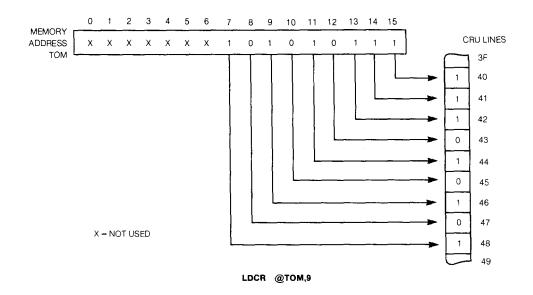


Figure 6-8. LDCR Word transfer

**STCR** 

STORE CRU STCR

Format: STCR G,,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<sub>16</sub> so that the hardware base address is 48<sub>16</sub> for the first bit to be transferred.

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

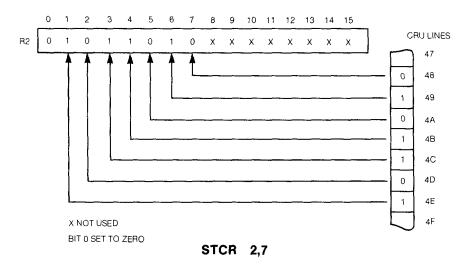


Figure 6-9. STCR Example

#### **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 <sub>o</sub>	A <sub>1</sub>	A <sub>2</sub>	OP CODE
LREX	Н	Н	Н	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 1 0 0 0 0 0 (03E0)
CKOF	Н	Н	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 1 0 0 0 0 0 0 0 (03C0)
CKON	Н	L	Н	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	Н	Н	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 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 0 1 1 0 1 0 0 0 0 0 0 0
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 RESET signal, 2) any interrupt that is enabled, or 3) a 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

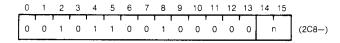
The 9940 instruction set includes the instructions already presented. Two of these instructions are slightly different for the 9940. These are the extended operation and the load interrupt mask immediate instructions. There are two new arithmetic instructions that provide for binary coded decimal (BCD) 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≤n≤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)

#### 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<sub>s</sub>,n

0 0 1 0 1 1 D Ts S (2)				6			11	12	13	14	15	
	1			1			Γs			r S		(2)

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  $(4 \le n \le 15)$  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:

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

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 will be obtained from  $40_{16} + 4 \times 4 = 50_{16}$  and the new PC value will be obtained from  $52_{16}$ .

DCA DCA

DECIMAL CORRECT ADDITION

G,

Format: DCA



Operation: The byte addressed by G<sub>s</sub> 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

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<sub>2</sub> through 1001<sub>2</sub>. It also keeps decimal addition accurate by responding to digit carries. For example, if 8<sub>16</sub> is added to 8<sub>16</sub> in BCD addition, 16<sub>16</sub> should be generated. However, if this operation is performed with binary addition,  $10_{16}$  results:

The DCA detects the digit carry and adds  $0110_2$  to the least significant digit to get the correct 16<sub>16</sub>.

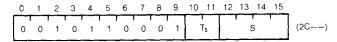
# DCS

DECIMAL CORRECT SUBTRACTION

G,

DCS

Format: DCS



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<sub>2</sub> through 1001<sub>2</sub>.

0	7_	) 8-BIT BYTE CONTAINING RESULT
X	Υ	OF BINARY ADD OR SUBTRACT
MSB	LSB	) OF 2 BCD DIGITS

В	YTE BEFORE	EXECUTI	ON		BYTE AFT	TER DCA			BYTE AFT	ER DCS	
С	Х	DC	Y	С	X	DC	Y	С	X	DC	Y
0	X<10	0	Y<10	0	Х	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	×	0	Y	-	-	-	\   –	0	X + 10	1	Y + 10
0	×	1	Y	-	_	-	\	0	X + 10	0	Y
1	×	0	Y	_	-	_	-	1	×	1	Y + 10
1	×	1	Y	_				1_1_	X	0	Υ

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