990

Computer Family
SYSTEMS HANDBOOK

MANUAL NO. 945250-9701

TEXAS INSTRUMENTS
INCORPORATED

3rd Edition, May 1976

Texas Instruments Incorporated
Digital Systems Division
P.O. Box 2909
Austin, Texas 78767

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

SECTION III

990 INSTRUCTIONS AND ASSEMBLY LANGUAGE

3.1 INTRODUCTION

The instruction set of the Model 990 Computer family readily lends itself to simple and effective programming that results in efficient processing. This section describes the instruction set and the assembler directives that are used with the instruction set to form source programs for the assemblers supported by Texas Instruments. This section also includes guidelines for writing source programs, a description of memory addressing techniques, and an example program. For full details of the assembly language, refer to the *Model 990 Computer Assembly Language Programmer's Guide.*

3.1.1 VERSATILITY OF INSTRUCTIONS. Of the 69 instructions in the Model 990 Computer instruction set, 34 instructions allow a choice of one of five addressing modes for one or both of the operands. The addressing modes are:

- Workspace register addressing
- Workspace register indirect addressing
- Symbolic memory addressing
- Indexed memory addressing
- Workspace register indirect autoincrement addressing.

The versatility provided by these addressing modes allows these instructions to operate on data in workspace registers or data in memory locations. Data in memory locations may be addressed directly or indirectly. Direct addresses may be indexed, and indirect addresses may be automatically incremented. The autoincrement indirect address allows processing every byte or word in an array or table without the necessity of including an add instruction to increment the address.

Any instruction that operates on the contents of a memory location can also perform input to or output from any TILINE device, because the registers of the device controller are accessed in the same manner as memory addresses. Effectively, each of these memory instructions becomes a TILINE Input/Output instruction. Therefore, the instruction set does not include a separate TILINE I/O instruction.

The instructions that transfer groups of bits to or from devices connected to the Communications Register Unit (CRU) also provide versatility. Device controllers connected to the CRU may require one or more fields or groups of bits, and these fields may consist of one or more bits each. Typically, a controller requires a data field, control lines, and status lines. An input instruction transfers a field of one to

sixteen bits from a CRU device controller to memory, and another instruction transfers a field of from one to sixteen bits from memory to a CRU device controller. When the number of bits to be transferred is eight bits or less, both instructions operate as byte instructions. When the number of bits transferred is greater than eight, the instructions operate as word instructions. The instruction set also includes instructions that transfer a single bit to or from the device controller.

3.1.2 EFFICIENCY. A key concept implemented in the Model 990 Computer is the workspace concept. The workspace is a 16-word area of memory that is addressable as workspace registers. Workspace registers may be used to contain operands of instructions, and as accumulators, to contain the results of an operation. Workspace registers may also be used to contain data addresses, or index values to be used in address development.

As shown in figure 3-1, the Workspace Pointer (WP) register contains the address of the first word of the workspace (workspace register 0). Workspace register 0 is optionally used by the shift instructions to contain the shift count. Workspace register 11 is used by the branch and link instruction to store Program Counter (PC) contents as a link to the branching program. Workspace register 12 is used by the CRU instructions as the base address register for CRU I/O. Workspace register 13 contains WP register contents, workspace register 14 contains PC contents, and workspace register 15 contains Status (ST) register contents during context switch operations. Use of the other ten workspace registers is not preempted in any way. The workspace register concept promotes efficiency because an entire new set of workspace register contents becomes applicable simply by changing the contents of the WP register.

Typically, a change of workspace is accomplished as part of a context switch. A context switch is a transfer of control that uses a 2-word transfer vector to define the new environment. A context switch consists of the following operations:

- The first word of a transfer vector is placed in the WP register
- The second word of the same transfer vector is placed in the PC
- The previous contents of the WP register is stored in WR 13 of the new workspace
- The previous contents of the PC is stored in WR 14 of the new workspace
- The contents of the ST register is stored in WR 15 of the new workspace.

A context switch transfers control to a program or subroutine, and activates a workspace associated with the program or subroutine as control passes to the new PC contents. The context switch stores the program environment, that is, the workspace address, the address of the next instruction in sequence, and the program status. A

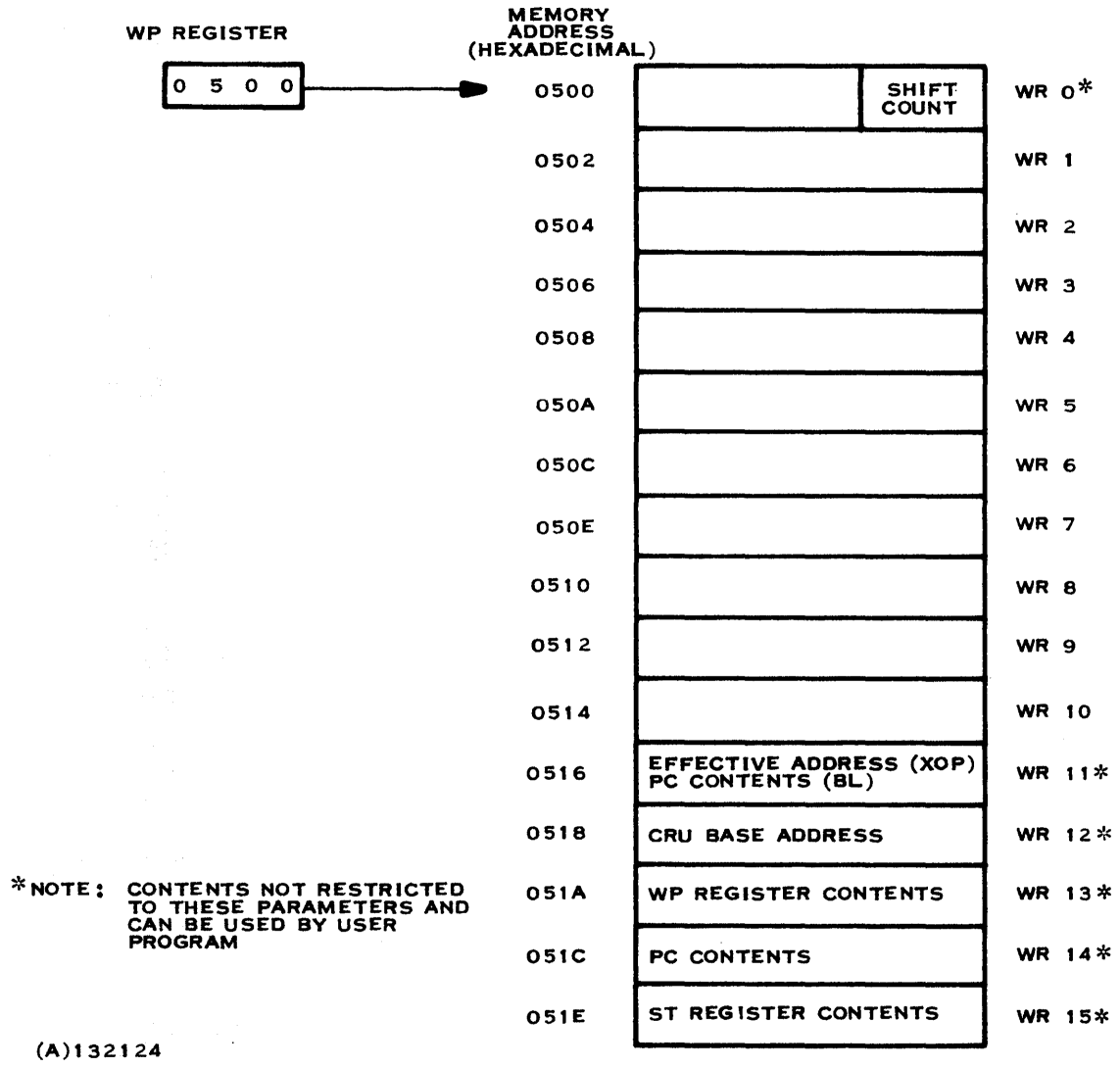


Figure 3-1. Typical Model 990 Computer Workspace

return instruction restores the environment by returning the stored data to the WP register, the PC, and the ST register. Context switches transfer control to subroutines when an interrupt occurs, when an extended operation instruction is executed, or when a branch and load workspace pointer instruction is executed.

3.1.2.1 Interrupt Context Switches. The Model 990 Computer provides vectored interrupts. Vectors for the interrupts are transfer vectors for context switches, and contain a workspace address and an entry point address. Vectors for the available interrupt levels are stored in interrupt level sequence starting at address zero. The level of interrupt is the priority of the interrupt. Level 0 is the highest priority and level 15 is the lowest priority. Interrupts are enabled by means of an interrupt mask, the four least significant bits of the ST register. The value in the mask specifies the lowest enabled level; all higher levels are enabled also. Table 3-1 lists the interrupt levels available in the Model 990/10 Computer. The table also applies to the Model 990/4 Computer, except that only levels 0 through 7 are available.

Table 3-1. Model 990/10 Computer Interrupt Level Data

Interrupt Level	Vector Location (Trap Address)	Assignment	Enabling Mask Values (Hexadecimal)
0	0000	Power On	0 through F
1	0004	Power Failing	1 through F
2	0008	Error	2 through F
3	000C	External Device	3 through F
4	0010	External Device	4 through F
5	0014	Line Frequency Clock (Optional)	5 through F
6	0018	External Device	6 through F
7	001C	External Device	7 through F
8	0020	External Device	8 through F
9	0024	External Device	9 through F
10	0028	External Device	A through F
11	002C	External Device	B through F
12	0030	External Device	C through F
13	0034	External Device	D through F
14	0038	External Device	E and F
15	003C	Line Frequency Clock (Optional)	F only

The vectored interrupt scheme of the Model 990 Computer, together with the transfer to the interrupt routine by means of a context switch, results in highly efficient processing of interrupts. When a pending interrupt condition has a priority higher than or equal to the interrupt mask contents, a context switch occurs following execution of the currently-executing instruction. The interrupt mask is then set to enable the level having the next higher priority, disabling the level of the current interrupt and all lower priority interrupts. The transfer of control, activation of a workspace, and storing of the context for processing of an interrupt are shown in figure 3-2.

In the Model 990/10 Computer, the Privileged Mode bit of the ST register is set to 0 when an interrupt occurs. This enables Privileged Mode instruction execution. When the map option is included, the Map File bit is also set to zero.

The vectored interrupt scheme is faster than other schemes because a single memory access obtains the address of the interrupt routine, rather than an indirect address or an instruction that branches to the routine. The context switch makes workspace registers available for processing the interrupt without having to store previous contents of a register file in the Arithmetic Logic Unit (ALU) and load these registers with values required to process the interrupt.

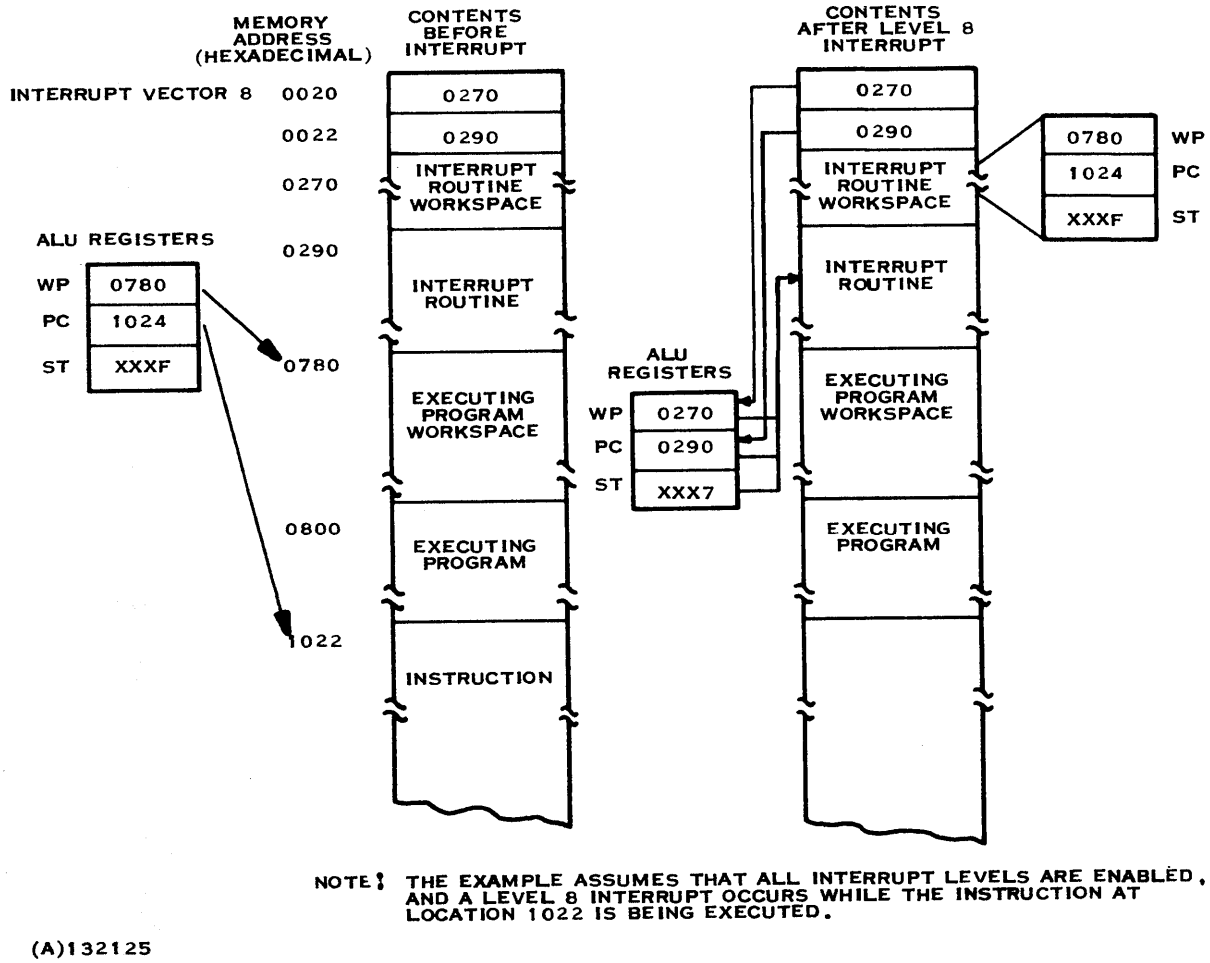


Figure 3-2. Interrupt Processing

Typical timing to accomplish transfer of control to an interrupt routine in the Model 990 Computer is 10 microseconds. This is significantly faster than contemporary computers using other interrupt schemes that require considerably more time to transfer control to the interrupt routine. Figure 3-3 shows the maximum number of interrupts that can be processed per second (as limited by transfer overhead) for three contemporary computers using various less efficient interrupt schemes, and the Model 990 Computer using vectored interrupts and context switching.

3.1.2.2 XOP Context Switches. The Model 990 Computer provides up to 16 extended operations to perform user-defined operations using software routines or hardware modules. The XOP instruction performs a context switch to a specified extended operation routine using a transfer vector in low-order memory. The context switch for an extended operation is similar to that previously described for an interrupt, except that the specified operation number selects a transfer vector in the memory area from 0040_{16} through $007F_{16}$. The computer develops the address in

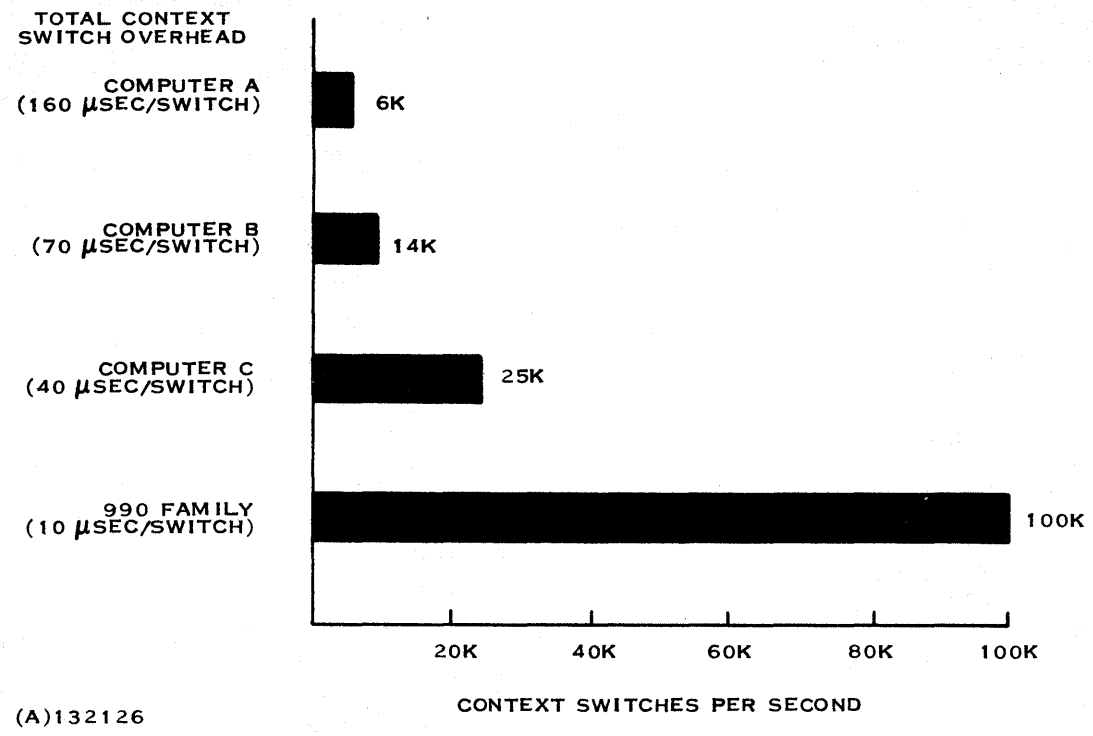


Figure 3-3. Comparison of Context Switching Efficiency

the XOP instruction and places the resulting effective address in workspace register 11 of the new workspace. An example of the use of an extended operation is the supervisor call from a user program to the disc executive, which is implemented as extended operation 15.

For the Model 990/10 Computer, the user may implement one or more (up to 16) extended operations by designing hardware modules to perform the special operations, such a hardware floating point processor. When a hardware module is connected for a given extended operation, the hardware is activated by an XOP instruction that specifies the operation, and no context switch occurs.

3.1.2.3 LREX Context Switch. The Model 990 Computer instruction set includes the LREX (Load or Restart Execution) instruction that performs a context switch using the transfer vector in address $FFFC_{16}$. Typically, the transfer vector and the routine that receives control are in Read Only Memory (ROM). The programmer panel routine is an example of a routine accessed by the LREX instruction.

3.1.2.4 BLWP Context Switch. The user may write subroutines to be accessed by a context switch. The BLWP (Branch and Load Workspace Pointer) instruction is used to access these subroutines. The transfer vector for a BLWP instruction may be placed in any available memory location, (at an even address) and is usually placed in the user area of memory.

3.1.3 INSTRUCTION SUMMARY. The 72 instructions of the Model 990 Computer are summarized in the paragraphs that follow under nine functional categories. They form a versatile and powerful instruction set.

3.1.3.1 Arithmetic Instructions. The following are the arithmetic instructions of the Model 990 Computer instruction set:

Add Words	Increment
Add Bytes	Increment by Two
Add Immediate	Decrement
Subtract Words	Decrement by Two
Subtract Bytes	Absolute Value
Multiply	Negate.
Divide	

The Add Words and Add Bytes instructions add word and byte operands, respectively, replacing the destination operand with the sum. The Add Immediate instruction adds an immediate operand to a workspace register operand, placing the sum in the workspace register. The Subtract Words and Subtract Bytes instructions subtract a word or byte operand from another word or byte operand, and replace the minuend with the remainder. The Multiply instruction multiplies two 16-bit operands as unsigned integers, and places the product in two consecutive workspace registers. The Divide instruction divides a 32-bit dividend in a pair of workspace registers by a 16-bit divisor. Division is performed assuming the operands to be unsigned integers, and the quotient and remainder replace the dividend in the pair of workspace registers. The Increment and Increment by Two instructions add one and two, respectively, to the operand, and replace the operand with the sum. Similarly, the Decrement and Decrement by Two instructions subtract one and two from the operand, and replace the operand with the remainder. The Absolute Value instruction replaces a negative operand with its two's complement. The Negate instruction replaces the operand with its two's complement. The thirteen instructions in the arithmetic category provide a powerful group of arithmetic operations.

3.1.3.2 Branch Instructions. The branch instruction category includes 18 instructions that transfer control to a location other than the next location in sequence, conditionally or unconditionally. The instructions are:

Branch	Branch and Load Workspace Pointer
Branch and Link	Return with Workspace Pointer
Unconditional Jump	Jump if Equal
Jump if Logical High	Jump if Not Equal
Jump if Logical Low	Jump On Carry
Jump if High or Equal	Jump if No Carry
Jump if Low or Equal	Jump if No Overflow
Jump if Greater Than	Jump If Odd Parity
Jump if Less Than	Execute

The Branch instruction branches unconditionally to the specified memory location. The Branch and Link instruction also branches unconditionally to the specified memory location, and stores the PC contents in workspace register 11 as a link to the branching program. The Branch and Load Workspace Pointer instruction performs an unconditional context switch using the transfer vector at the specified location. The

Return with Workspace Pointer instruction restores the calling or interrupted program environment using the values stored by the most recent context switch. The operand of any of the jump instructions is a displacement (in words) in the range of −128 through +127 that is added algebraically to the PC contents. That is, transfers of control by jump instructions are program counter relative. The thirteen jump instructions include an Unconditional Jump, eight jumps related to the results of a comparison, and four jumps on other ST register bits. The Execute instruction specifies an instruction to be executed. The program counter is not altered unless the replacing instruction normally alters the program counter. The branch instruction category provides a great deal of flexibility for transfer of program control.

3.1.3.3 Compare Instructions. The compare instruction category includes five instructions that set and reset the condition code bits of the ST register to indicate the relationship between operands. The instructions are:

Compare Words	Compare Ones Corresponding
Compare Bytes	Compare Zeros Corresponding
Compare Immediate	

The Compare Words and Compare Bytes instructions compare words and bytes, respectively. The Compare Immediate instruction compares an immediate operand with the contents of a workspace register. These instructions compare the operands as two's complement values (arithmetic comparison) and as unsigned values (logical comparison) simultaneously. The other two compare instructions provide a means of comparing selected bits of the contents of a workspace register to ones or zeros. The source operand of each of these instructions is a pattern or mask in which one bits specify the bits to be compared. The Compare Ones Corresponding compares the bits of the destination operand specified by the mask to logical ones. The Compare Zeros Corresponding compares the bits of the destination operand to logical zeros.

3.1.3.4 Control and CRU Instructions. The control and CRU instruction category includes five control instructions and five CRU I/O instructions. The instructions are:

Reset	Set Bit to Logic Zero
Idle	See Bit to Logic One
Clock Off	Test Bit
Clock On	Load CRU
Load or Restart Execution	Store CRU

In the Model 990 Computers, the Reset instruction clears the interrupt mask, disabling all but level 0 interrupts, and resets directly connected I/O devices. The instruction resets CRU devices that provide for reset in the interface with the CRU,

resets pending interrupts, and turns the line frequency clock off. The Idle instruction places the computer in the idle mode. The Clock Off instruction turns the line frequency clock off, and the Clock On instruction turns the line frequency clock on. The Load or Restart Execution instruction performs a context switch using the transfer vector at location $FFFC_{16}$.

In the TMS9900 microprocessor, the control instructions function differently. The Idle instruction places the microprocessor in the idle mode, and generates an external signal. The Reset, Clock Off, Clock On, and Load or Restart Execution instructions perform no function in the TMS9900, but these instructions do generate external signals. In implementing a micro computer using the TMS9900, the user may provide logic to perform any desired function when the signal generated by each of these control instructions is detected.

The Set Bit to Logic One and Set Bit to Logic Zero instructions each transmit a single bit on a CRU line to an I/O device. The Set Bit to Logic One instruction sets the addressed bit to the value of one, and the Set Bit to Logic Zero sets the bit to the value of zero. The Test Bit instruction reads a single bit on a CRU line, and sets the equal bit in the ST register to the value of the addressed line. The single bit CRU I/O instructions specify the desired line as a displacement from the CRU base address. The Load CRU instruction transfers a group or field of 1 to 16 bits from memory to a group or field of contiguous CRU lines. The Store CRU instruction transfers a similar field from CRU lines to memory. For these instructions the CRU base address must be set to the address of the lowest numbered CRU line in the group.

3.1.3.5 Load and Move Instructions. The load and move instruction category includes four instructions that load workspace registers, ALU registers, and mapping registers. The category also includes three move instructions, and two instructions that store ALU register contents. The instructions are:

Load Immediate	Move Byte
Load Interrupt Mask Immediate	Swap Bytes
Load Workspace Pointer Immediate	Store Status
Load Memory Map File	Store Workspace Pointer.
Move Word	

The Load Immediate instruction places the immediate operand into the specified workspace register. The Load Interrupt Mask Immediate instruction places the four least significant bits of the immediate operand into the interrupt mask, the four least significant bits of the ST register. The Load Workspace Pointer Immediate instruction places the immediate operand into the WP register.

The Load Memory Map File instruction applies only to the Model 990/10 Computer with the mapping option. The mapping option allows the 64K addresses available within the instruction format to be mapped to memory addresses in a 20-bit addressing format. The mapping option provides three map files consisting of six registers each. Three of the registers contain limits in one's complement form, and the other three registers contain bias addresses. (Refer to figure 3-4.) ALU addresses are compared to the one's complement of the limit registers. ALU addresses between zero and the contents of Limit 1 (L1) are added to Bias 1 (B1). ALU addresses greater than the contents of L1 and less than or equal to the contents of Limit 2 (L2) are added to Bias 2 (B2). ALU addresses greater than the contents of L2 and less than or equal to the contents of Limit 3 (L3) are added to Bias 3 (B3). ALU addresses greater than the contents of L3 are not accessible. The addition of ALU addresses to B1, B2, and B3 results in 20-bit addresses. The contents of the bias register is shifted to the left five bit positions, and added to the ALU address. The five least significant bits of the ALU address are retained, and the least significant bit is ignored.

The Load Memory Map File instruction places the contents of a six-word area in memory at the address contained in the specified workspace register into a map file. Map files 0 or 1 can be loaded by this instruction. Use of map file 2 is described in a subsequent paragraph.

The Move Word and Move Byte instructions copy the data in the source operand into the destination operand. The Swap Bytes instruction exchanges the bytes in a workspace register or memory word.

The Store Status instruction stores the contents of the ST register in a workspace register, and the Store Workspace Pointer instruction stores the contents of the WP register in a workspace register.

3.1.3.6 Logical Instructions. The logical instruction category includes 10 instructions that perform logical functions on words or bytes. The instructions are:

AND Immediate	Set to Ones
OR Immediate	Set Ones Corresponding
Exclusive OR	Set Ones Corresponding, Byte
Invert	Set Zeros Corresponding
Clear	Set Zeros Corresponding, Byte.

The AND Immediate instruction performs an AND operation between the immediate operand and the contents of a workspace register, and places the result in a workspace register. The OR Immediate instruction performs an OR operation between the immediate operand and the contents of a workspace register, and places the result in a workspace register. The Exclusive OR instruction performs an exclusive OR operation between an operand in memory or a workspace register and another operand in a

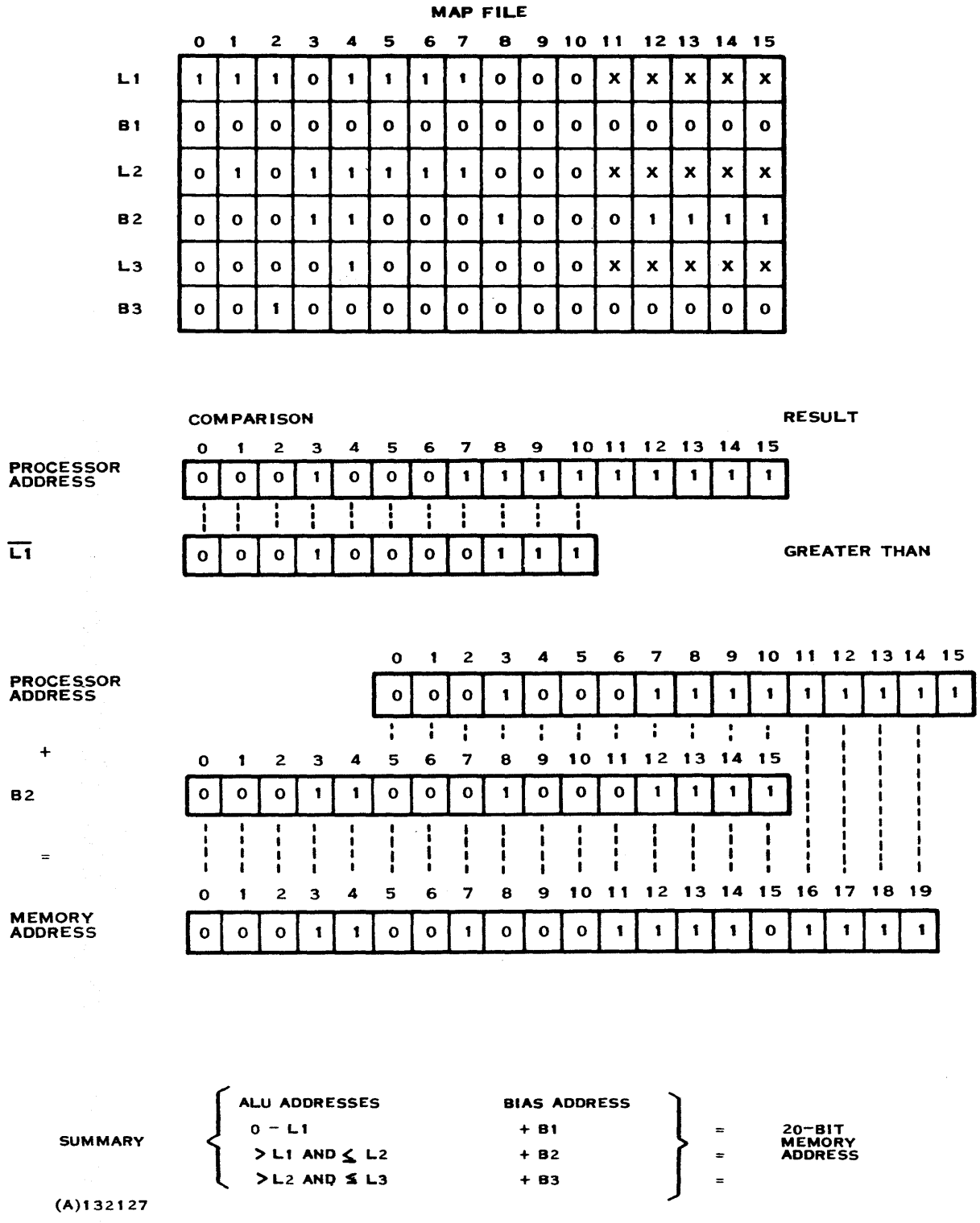


Figure 3-4. Development of Memory Address

workspace register, and replaces the operand in the workspace register. The Invert instruction replaces the operand with its one's complement, and the Clear instruction sets the bits of the operand to zeros. The Set to One instruction sets the bits of the operand to ones. The remaining logical instructions all use the source operand as a pattern or mask to select bits of the destination operand to be set. Ones in the source operand correspond to bits to be set, and zeros in the source operand correspond to bits to remain unaltered. The Set Zeros Corresponding sets bits of the destination word to zeros, corresponding to the mask. The Set Zeros Corresponding, Byte sets bits of the destination byte to zeros. The Set Zeros Corresponding instructions are effectively AND instructions that combine the one's complement of the mask with the destination operand. The Set Ones Corresponding instruction sets bits of the destination word to ones, corresponding to the mask. The Set Ones Corresponding, Byte instruction sets bits of the destination byte to ones. The Set Ones Corresponding instructions are effectively OR instructions that combine the two operands.

3.1.3.7 Shift Instructions. The shift instructions shift the bits in a workspace register according to the shift count. The instructions are:

Shift Right Arithmetic — Shift Right Logical

Shift Left Arithmetic — Shift Right Circular.

Shift instructions take the shift count from an operand, or from the four least significant bits of workspace register 0 when the shift count operand contains zero. When both the shift count operand and the four least significant bits of workspace register 0 contain zero, the register is shifted 16 bit positions. The carry bit of the ST register contains the last bit shifted out of the register.

The Shift Right Arithmetic instruction shifts the contents of a specified workspace register to the right, filling the vacated bit positions with the sign bit. The Shift Left Arithmetic instruction shifts the contents of the specified workspace register to the left, filling the vacated bit positions with zeros. The Shift Right Logical instruction shifts the contents of the specified workspace register to the right, filling the vacated bit positions with zeros. The Shift Right Circular instruction shifts the contents of the specified workspace register to the right, filling the vacated bit positions with the bits shifted off the right end of the workspace register.

3.1.3.8 Extended Operation Instruction. The Extended Operation instruction is a means of adding up to 16 additional computer operations, as required, to the Model 990 Computer instruction set. The user may supply a subroutine to perform the operation and place a transfer vector consisting of the workspace address and the entry address in the appropriate address (0040_{16} through $007C_{16}$). The Extended Operation instruction performs a context switch using the transfer vector corresponding to the extended operation number. In the Model 990/10 Computer, the user may supply a hardware module to perform the operation. When a hardware module is connected for the specified extended operation, the instruction activates the hardware instead of performing the context switch.

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

Long Distance Source Long Distance Destination.

Each instruction places the contents of a 6-word area of memory in map file 2. The first word of the 6-word area is at the source operand address. The Long Distance Source instruction loads map file 2 and assigns map file 2 to the source address of the next instruction. The Long Distance Destination address loads map file 2 and assigns map file 2 to the destination address of the next instruction.

3.2 MACHINE INSTRUCTIONS

The machine instructions of the Model 990 Computer instructions are listed in table 3-2. The machine formats are shown on the following pages, along with the operation code, execution results, and list of status bits affected. The operation codes (Op Codes) are shown as four digit hexadecimal numbers in which operand fields contain zero. The machine instruction formats use the following abbreviations:

- T_s Mode bits for source address, defined in table 3-2.
- S Workspace register field for source address, specifies the workspace register that contains the operand, the address of the operand, or the index value for the address of the operand.
- T_d Mode bits for destination address, defined in table 3-2.
- D Workspace register field for destination address, specifies the workspace register that contains the operand, the address of the operand, or the index value for the address of the operand.
- W Workspace register field for a workspace operand.
- C Count field, containing either a bit count or a shift count.
- M Map file field

In the execution results, the following conventions and abbreviations apply:

- ga_s General address, source. An address in one of the modes defined in table 3-2.
- ga_d General address, destination. An address in one of the modes defined in table 3-2.
- wa Workspace register address. Specifies a workspace register that contains an operand.

- iop Immediate operand
- ( ) Specifies the contents of an address or register.
- | | Specifies an absolute value
- → Specifies "replaces"

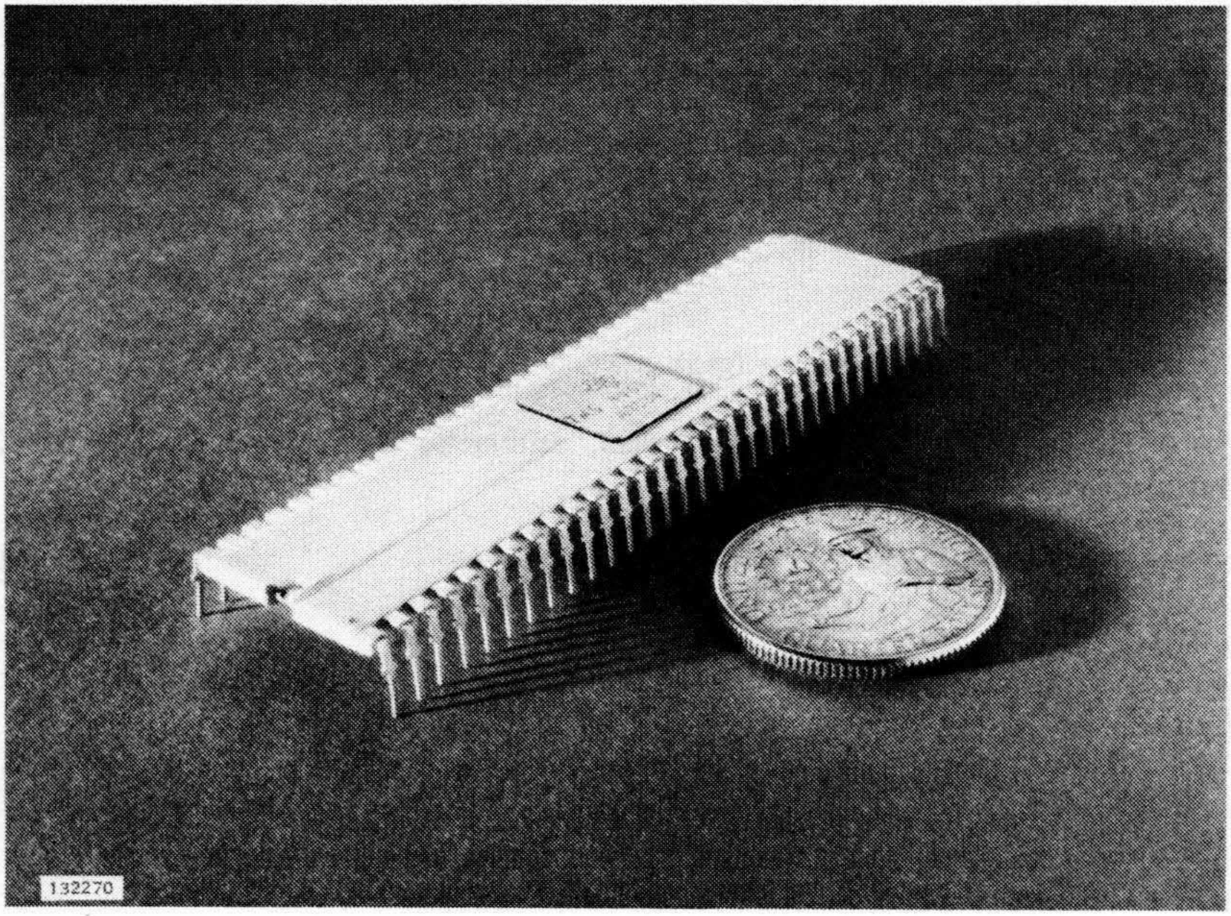
Table 3-2. Machine Instruction Addressing Modes

Mode Bits	Addressing Mode
00_2	Workspace Register Addressing
01_2	Workspace Register Indirect Addressing
10_2	Symbolic Memory Addressing, when corresponding workspace register field contains 0
10_2	Indexed Memory Addressing, when corresponding workspace register field contains a value greater than 0
11_2	Workspace Register Indirect Autoincrement Addressing

The status bits affected by the instruction are bits in the ST register, defined as follows:

- Logical greater than - result of a comparison of operands as unsigned numbers.
- Arithmetic greater than - result of a comparison of operands as two's complement numbers.
- Equal - result of a comparison.
- Carry - carry out of most significant bit of an operand.
- Overflow - result too large or too small to be correctly represented in a byte or word (as applicable) in two's complement form.
- Odd parity - result of operation is a byte that contains an odd number of one bits.
- Extended operation - set as a software-implemented extended operation is initiated.

The comparison that sets or resets the logical greater than, arithmetic greater than, and equal bits is a comparison of the result to zero, except for compare instructions. Compare instructions set or reset these bits according to the result of comparing two operands.

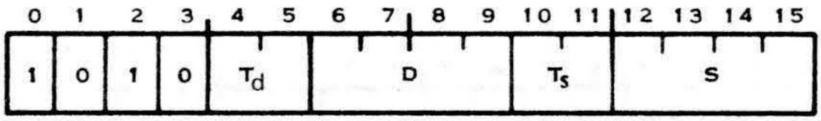


The instructions apply to the TMS9900 microprocessor and the Model 990 Computer except as noted.

3.2.1 ADD WORDS A

Op Code: A000

Format:



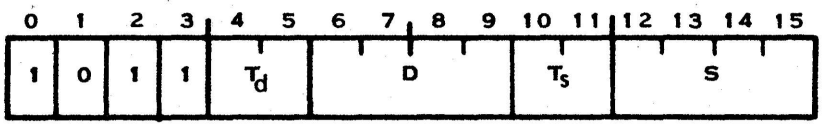
Execution results: $(ga_s) + (ga_d) \rightarrow (ga_d)$

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

3.2.2 ADD BYTES AB

Op Code: B000

Format:



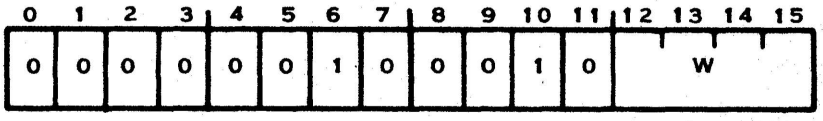
Execution results: $(ga_s) + (ga_d) \rightarrow (ga_d)$

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

3.2.3 ADD IMMEDIATE AI

Op Code: 0200

Format:



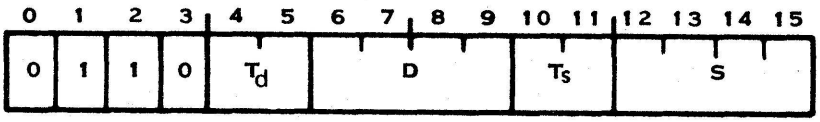
Execution results: (wa) + iop → (wa)

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

3.2.4 SUBTRACT WORDS S

Op Code: 6000

Format:



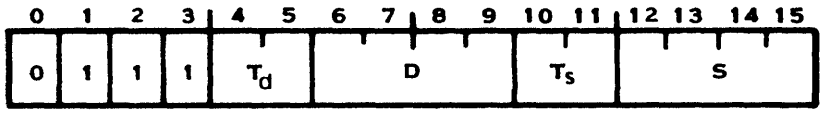
Execution results: $(ga_d) - (ga_s) \rightarrow (ga_d)$

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

3.2.5 SUBTRACT BYTES SB

Op Code: 7000

Format:



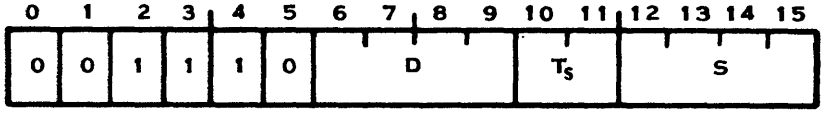
Execution results: $(ga_d) - (ga_s) \rightarrow (ga_d)$

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

3.2.6 MULTIPLY MPY

Op Code: 3800

Format:



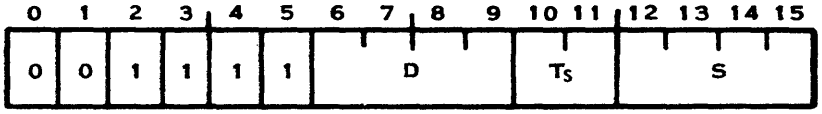
Execution results: $(ga_s) \cdot (wa_d)$. The product (32-bit magnitude) is placed in wa_d and $wa_d + 1$, with the most significant half in wa_d.

Status bits affected: None

3.2.7 DIVIDE DIV

Op Code: 3C00

Format:



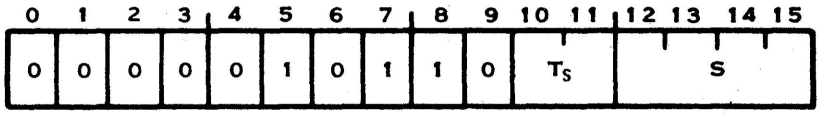
Execution results: The contents of wa_d and $wa_d + 1$ (32-bit magnitude) are divided by the contents of ga_s and the quotient is placed in wa_d. The remainder is placed in $wa_d + 1$.

Status bits affected: Overflow.

3.2.8 INCREMENT INC

Op Code: 0580

Format:



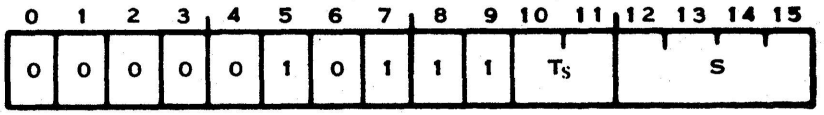
Execution results: $(ga_s) + 1 \rightarrow (ga_s)$

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

3.2.9 INCREMENT BY TWO INCT

Op Code: 05C0

Format:



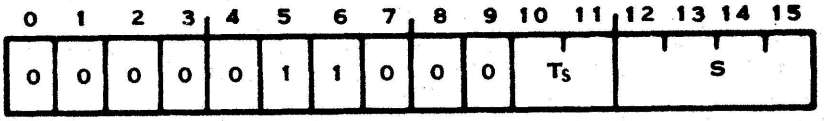
Execution results: $(ga_s) + 2 \rightarrow (ga_s)$

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

3.2.10 DECREMENT DEC

Op Code: 0600

Format:



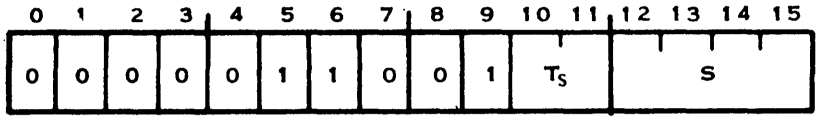
Execution results: $(ga_S) - 1 \rightarrow (ga_S)$

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

3.2.11 DECREMENT BY TWO DECT

Op Code: 0640

Format:



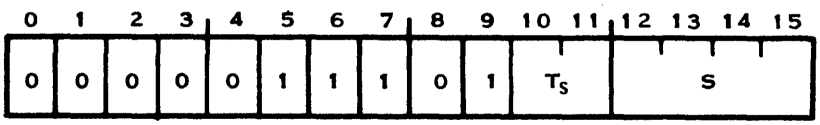
Execution results: $(ga_S) - 2 \rightarrow (ga_S)$

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

3.2.12 ABSOLUTE VALUE ABS

Op Code: 0740

Format:



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

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

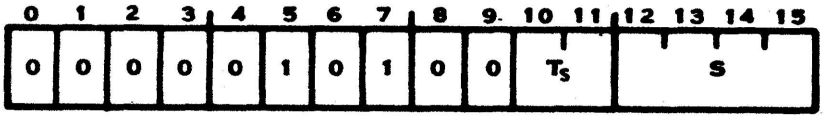
NOTE

These status bits are affected by contents of the source operand before instruction execution.

3.2.13 NEGATE NEG

Op Code: 0500

Format:



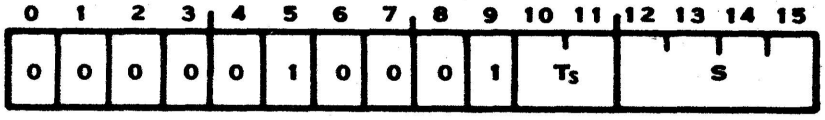
Execution results: $-(ga_S) \rightarrow (ga_S)$

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

3.2.14 BRANCH B

Op Code: 0440

Format:



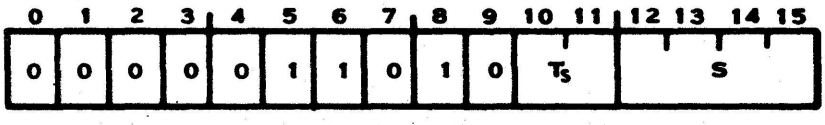
Execution results: $ga_S \rightarrow (PC)$

Status bits affected: None.

3.2.15 BRANCH AND LINK BL

Op Code: 0680

Format:



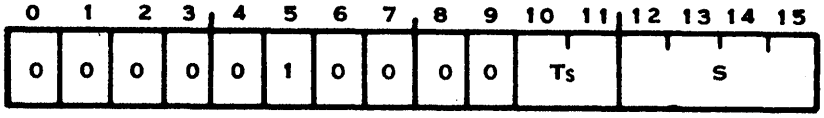
Execution results: (PC) → (Workspace register 11)
$ga_S \rightarrow (PC)$:

Status bits affected: None.

3.2.16 BRANCH AND LOAD WORKSPACE POINTER BLWP

Op Code: 0400

Format:



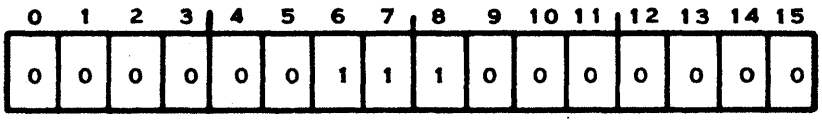
Execution results: (WP)→ (Workspace register 13)
(PC) → (Workspace register 14)
(ST) → (Workspace register 15)
} "New" Workspace
(ga_s) → (WP)
(ga_s + 2) → (PC)

Status bits affected: None.

3.2.17 RETURN WITH WORKSPACE POINTER RTWP

Op Code: 0380

Format:



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

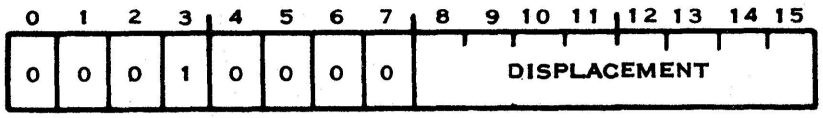
Status bits affected: Restores all status bits to the value contained in workspace register 15.

Model 990/10 Computer: In the Model 990/10 Computer with the Privileged Mode bit (bit 7) of the ST register set to 1, only bits 0 through 6 of workspace register 15 are placed in bits 0 through 6 of the ST register. When bit 7 of the ST register is set to 0, the instruction places all 16 bits of workspace register 15 in the ST register.

3.2.18 UNCONDITIONAL JUMP JMP

Op Code: 1000

Format:



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

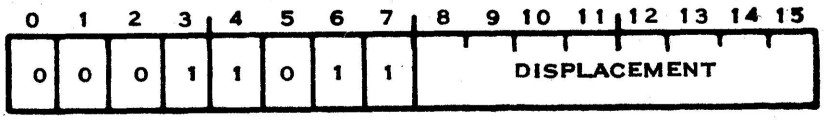
The PC is incremented to the address of the next instruction prior to execution of an instruction. The execution results of jump instructions refer to the PC contents after the contents have incremented to address the next instruction in sequence. The displacement (in words) is shifted to the left one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

Status bits affected: None.

3.2.19 JUMP IF LOGICAL HIGH JH

Op Code: 1B00

Format:



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

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

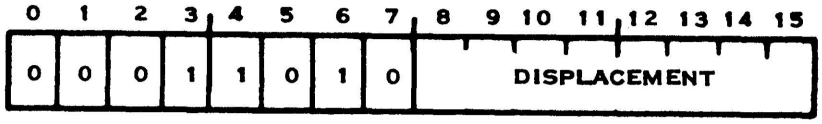
Refer to explanation of execution in paragraph 3.2.18

Status bits affected: None.

3.2.20 JUMP IF LOGICAL LOW JL

Op Code: 1A00

Format:



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

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

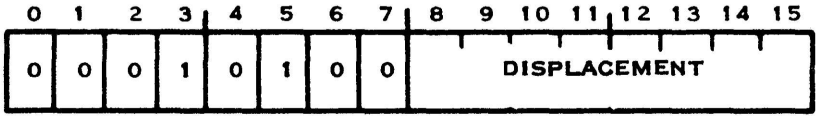
Refer to explanation of execution in paragraph 3.2.18

Status bits affected: None

3.2.21 JUMP IF HIGH OR EQUAL JHE

Op Code: 1400

Format:



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

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

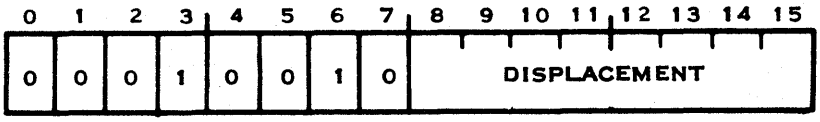
Refer to explanation of execution in paragraph 3.2.18

Status bits affected: None

3.2.22 JUMP IF LOW OR EQUAL JLE

Op Code: 1200

Format:



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

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

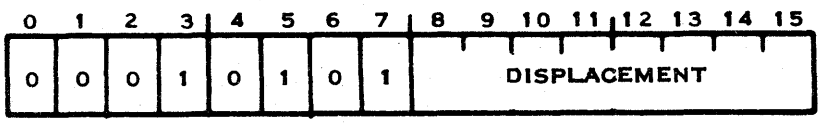
Refer to explanation of execution in paragraph 3.2.18.

Status bits affected: None.

3.2.23 JUMP IF GREATER THAN JGT

Op Code: 1500

Format:



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

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

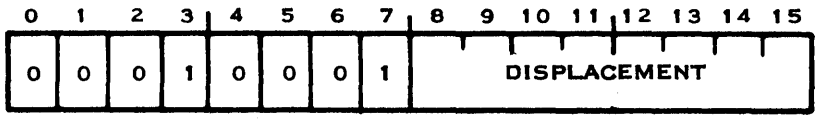
Refer to explanation of execution in paragraph 3.2.18.

Status bits affected: None.

3.2.24 JUMP IF LESS THAN JLT

Op Code: 1100

Format:



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

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

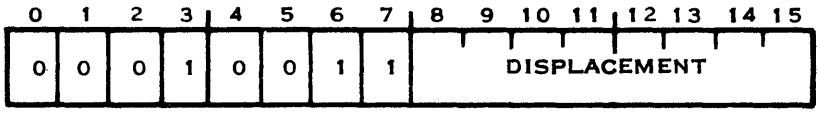
Refer to explanation of execution in paragraph 3.2.18.

Status bits affected: None

3.2.25 JUMP IF EQUAL JEQ

Op Code: 1300

Format:



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

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

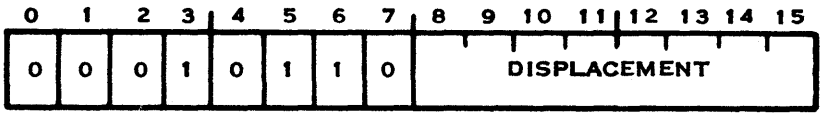
Refer to explanation of execution in paragraph 3.2.18.

Status bits affected: None.

3.2.26 JUMP IF NOT EQUAL JNE

Op Code: 1600

Format:



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

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

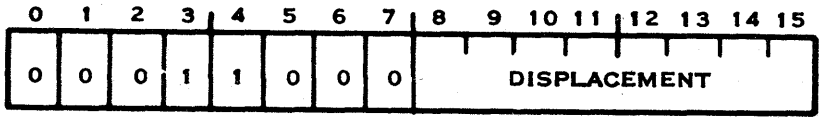
Refer to explanation of execution in paragraph 3.2.18.

Status bits affected: None.

3.2.27 JUMP ON CARRY JOC

Op Code: 1800

Format:



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

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

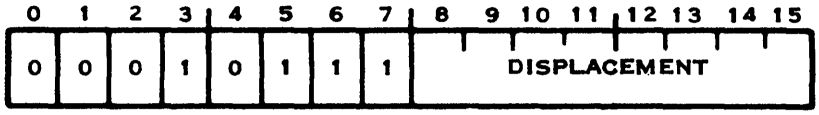
Refer to explanation of execution in paragraph 3.2.18

Status bits affected: None

3.3.28 JUMP IF NO CARRY JNC

Op Code: 1700

Format:



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

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

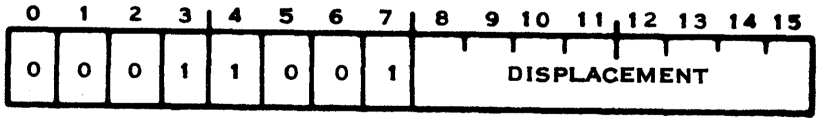
Refer to explanation of execution in paragraph 3.2.18.

Status bits affected: None.

3.2.29 JUMP IF NO OVERFLOW JNO

Op Code: 1900

Format:



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

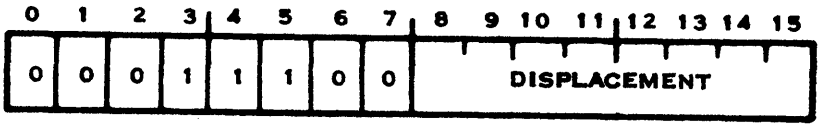
If overflow bit is equal to 1: (PC) → (PC)

Refer to explanation of execution in paragraph 3.2.18.

3.2.30 JUMP IF ODD PARITY JOP

Op Code: 1C00

Format:



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

If odd parity bit is equal to 0: (PC) + 2 → (PC)

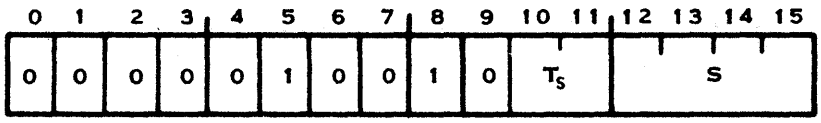
Refer to explanation of execution in paragraph 3.2.18.

Status bits affected: None.

3.2.31 EXECUTE X

Op Code: 0480

Format:

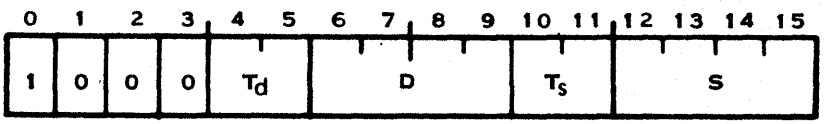


Execution results: An instruction at ga_s is executed. Status bits affected: None, but substituted instruction affects status bits normally.

3.2.32 COMPARE WORDS C

Op Code: 8000

Format:



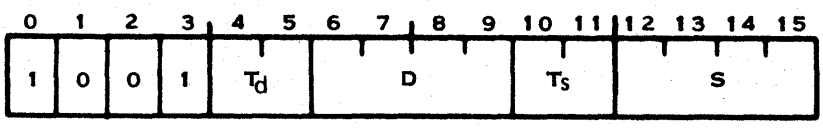
Execution results: (ga_s) : (ga_d)

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

3.3.33 COMPARE BYTES CB

Op Code: 9000

Format:



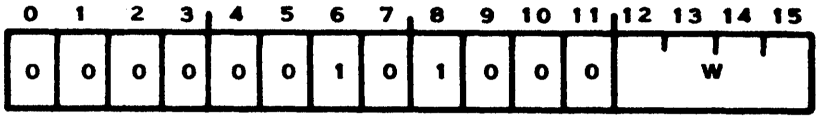
Execution results: $(ga_s):(ga_d)$

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

3.2.34 COMPARE IMMEDIATE CI

Op Code: 0280

Format:



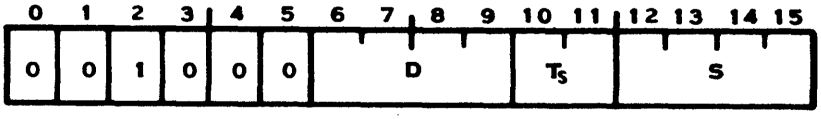
Execution results: (wa): iop

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

3.2.35 COMPARE ONES CORRESPONDING COC

Op Code: 2000

Format:



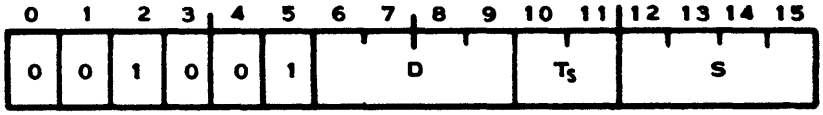
Execution results: Equal bit set if all bits of (wa) that correspond to the bits of (ga_s) that are equal to 1 are also equal to 1.

Status bit affected: Equal.

3.2.36 COMPARE ZEROS CORRESPONDING CZC

Op Code: 2400

Format:



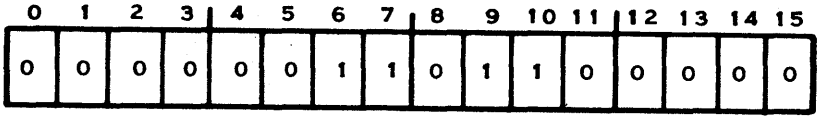
Execution results: Equal bit set if all bits of (wa) that correspond to the bits of (ga_S) that are equal to 1 are equal to 0.

Status bit affected: Equal

3.2.37 RESET RSET

Op Code: 0360

Format:



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

Status bits affected: Interrupt Mask

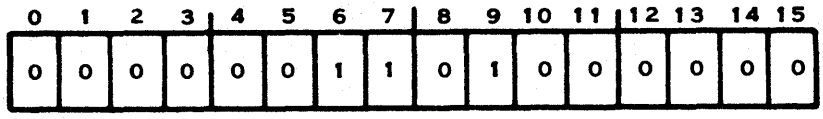
TMS9900 Microprocessor: Provides a signal that a RSET instruction is identified, but performs no processing. User may implement hardware to perform desired processing when the signal is present.

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

3.2.38 IDLE IDLE

Op Code: 0340

Format:



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

Status bits affected: None.

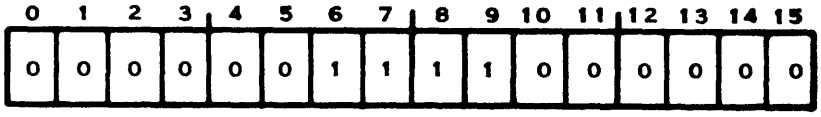
TMS9900 Microprocessor: Provides a signal that an IDLE instruction is being executed, and places the microprocessor in the idle mode. User may implement hardware to perform additional processing when the signal is present.

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

3.2.39 CLOCK OFF CKOF

Op Code: 03C0

Format:



Execution results: Line frequency clock disabled.

Status bits affected: None.

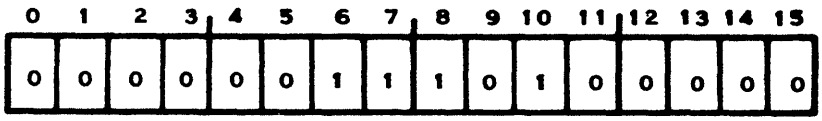
TMS9900 Microprocessor: Provides a signal that a CKOF instruction is identified, but performs no processing. User may implement hardware to perform desired processing when signal is present.

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

3.2.40 CLOCK ON CKON

Op Code: 03A0

Format:



Execution results: Line frequency clock enabled.

Status bits affected: None.

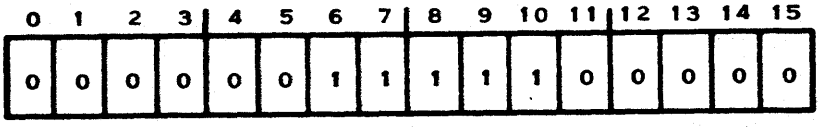
TMS9900 Microprocessor: Provides a signal that a CKON instruction is identified, but performs no processing. User may implement hardware to perform desired processing when signal is present.

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

3.2.41 LOAD OR RESTART EXECUTION LREX

Op Code: 03E0

Format:



Execution Results: Performs a context switch using the transfer vector at location $FFFC_{16}$, and sets the interrupt mask to 0. The transfer vector and the subroutine to which control is transferred are often in Read Only Memory (ROM).

Status bits affected: Interrupt Mask

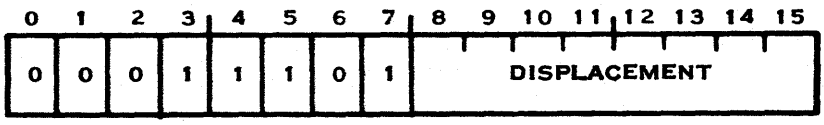
TMS9900 Microprocessor: Provides a signal that an LREX instruction is identified, but performs no processing. User may implement hardware to perform desired processing when signal is present.

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

3.2.42 SET BIT TO LOGIC ONE SBO

Op Code: 1D00

Format:



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

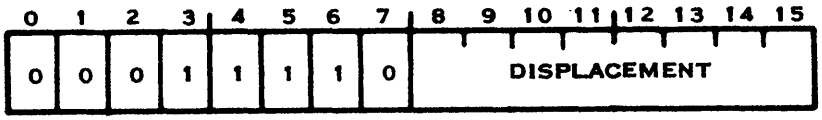
Status bits affected: None.

Model 990/10 Computer: When the Privileged Mode bit (bit 7) is set to 0, the SBO instruction executes normally. When the Privileged Mode bit is set to 1, and the effective CRU address is equal to or greater than $E00_{16}$, an error interrupt occurs.

3.2.43 SET BIT TO LOGIC ZERO SBZ

Op Code: 1E00

Format



Execution results: CRU bit addressed by the sum of the contents of workspace register 12 + displacement is set to 0.

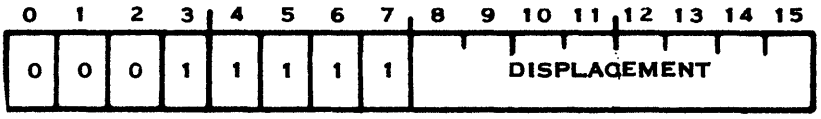
Status bits affected: None.

Model 990/10 Computer: When the Privileged Mode bit (bit 7) of the ST register is set to 0, the SBZ instruction executes normally. When bit 7 is set to 1 and the effective CRU address is equal to or greater than $E00_{16}$, an error interrupt occurs.

3.2.44 TEST BIT TB

Op Code: 1F00

Format:



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

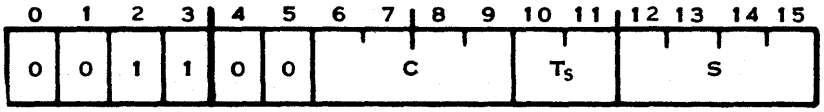
Status bit affected: Equal.

Model 990/10 Computer: When the Privileged Mode bit (bit 7) of the ST register is set to 0, the TB instruction executes normally. When bit 7 is set to 1 and the effective CRU address is equal to or greater than $E00_{16}$, an error interrupt occurs.

3.2.45 LOAD CRU LDCR

Op Code: 3000

Format:



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

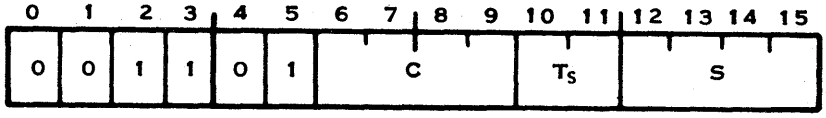
Status bits affected: Logical greater than, arithmetic greater than, and equal. When C is less than 9, odd parity is also set or reset.

Model 990/10 Computer: When the Privileged Mode bit (bit 7) of the ST register is set to 0, the LDCR instruction executes normally. When bit 7 is set to 1 and the effective CRU address is equal to or greater than $E00_{16}$, an error interrupt occurs.

3.2.46 STORE CRU STCR

Op Code: 3400

Format:



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

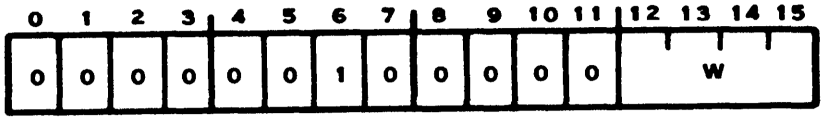
Status bits affected: Logical greater than, arithmetic greater than, and equal. When C is less than 9, odd parity is also set or reset.

Model 990/10 Computer: When the Privileged Mode bit (bit 7) of the ST register is set to 0, the STCR instruction executes normally. When bit 7 is set to 1 and the effective CRU address is equal to or greater than $E00_{16}$, an error interrupt occurs.

3.2.47 LOAD IMMEDIATE LI

Op Code: 0200

Format:



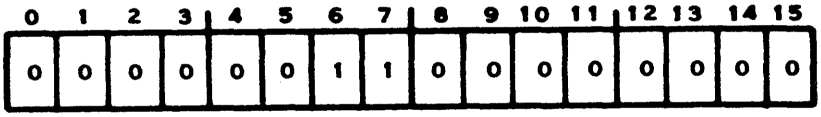
Execution results: iop → (wa)

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

3.2.48 LOAD INTERRUPT MASK IMMEDIATE LIMI

Op Code: 0300

Format:



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

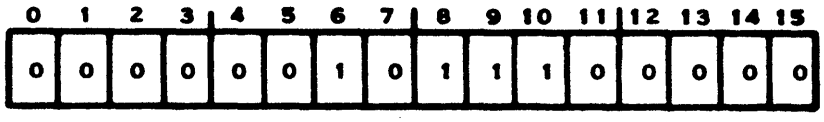
Status bits affected: Interrupt Mask

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

3.2.49 LOAD WORKSPACE POINTER IMMEDIATE LWPI

Op Code: 02E0

Format:



Execution results: iop → (WP)

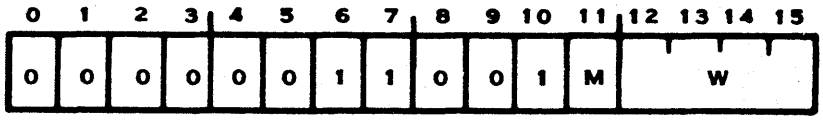
Status bits affected: None.

3.2.50 LOAD MEMORY MAP FILE LMF

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

Op Code: 0320

Format:



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

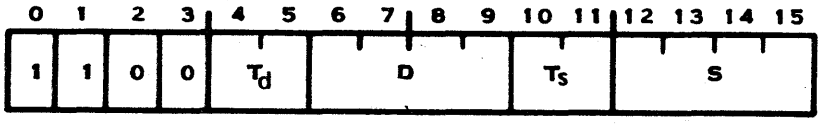
When Privileged Mode bit is set to 1: Error interrupt.

Status bits affected: None.

3.2.51 MOVE WORD MOV

Op Code: C000

Format:



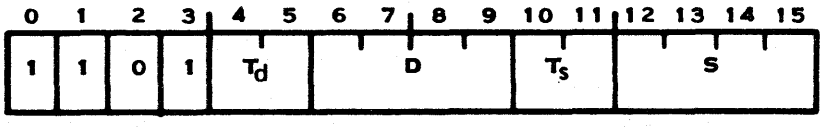
Execution results: $(ga_S) \rightarrow (ga_d)$

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

3.2.52 MOVE BYTE MOVB

Op Code: D000

Format



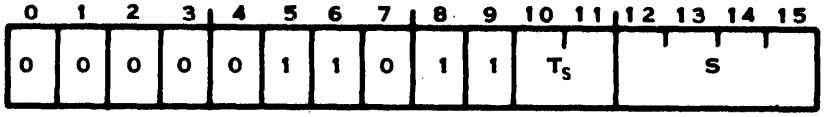
Execution results: $(ga_S) \rightarrow (ga_d)$

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

3.2.53 SWAP BYTES SWPB

Op Code: 06C0

Format:



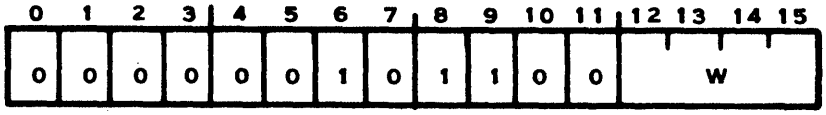
Execution results: Exchanges left and right bytes of word (ga_S).

Status bits affected: None.

3.2.54 STORE STATUS STST

Op Code: 02C0

Format:



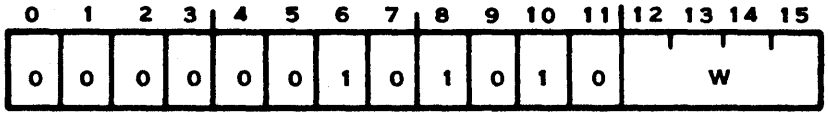
Execution results: (ST) → (wa)

Status bits affected: None.

3.2.55 STORE WORKSPACE POINTER STWP

Op Code: 02A0

Format:



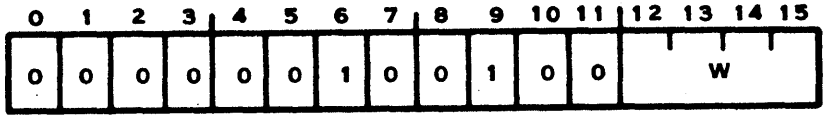
Execution results: (WP) → (wa)

Status bits affected: None.

3.2.56 AND IMMEDIATE ANDI

Op Code: 0240

Format:



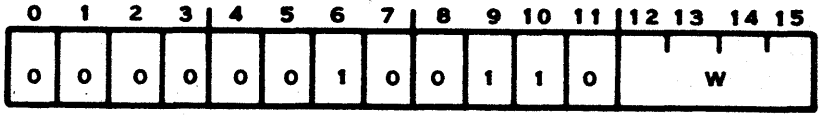
Execution results: (wa) AND iop → (wa)

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

3.2.57 OR IMMEDIATE ORI

Op Code: 0260

Format:



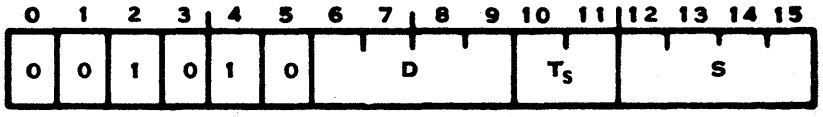
Execution results: (wa) OR iop → (wa)

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

3.2.58 EXCLUSIVE OR XOR

Op Code: 2800

Format:



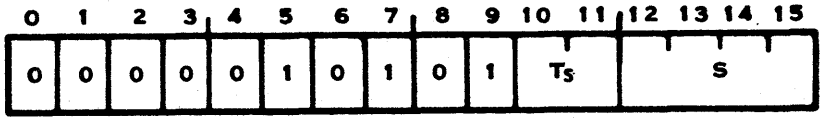
Execution results: (ga_s) XOR (wa_d) → (wa_d)

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

3.2.59 INVERT INV

Op Code: 0540

Format:



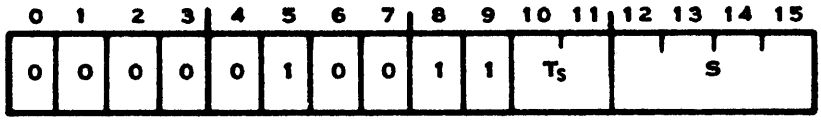
Execution results: The one's complement of (ga_s) is placed in (ga_s).

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

3.2.60 CLEAR CLR

Op Code: 04C0

Format:



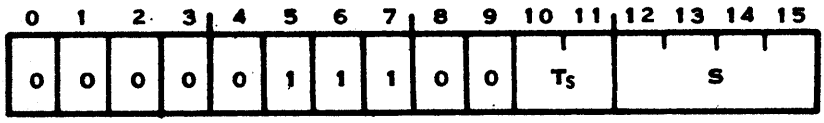
Execution results: $0 \rightarrow (ga_S)$

Status bits affected: None.

3.2.61 SET TO ONE SETO

Op Code: 0700

Format:



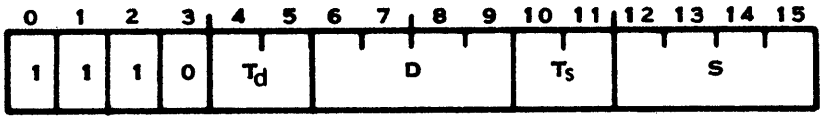
Execution results: $FFFF_{16} \rightarrow (ga_S)$

Status bits affected: None.

3.2.62 SET ONES CORRESPONDING SOC

Op Code: E000

Format:



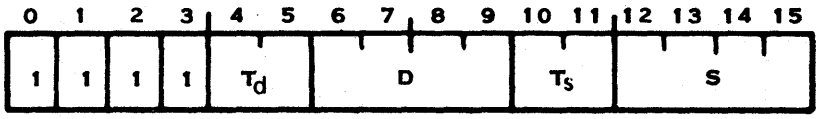
Execution results: Bits of (ga_d) corresponding to bits of (ga_S) equal to 1 are set to 1.

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

3.2.63 SET ONES CORRESPONDING, BYTE SOCB

Op Code: F000

Format:



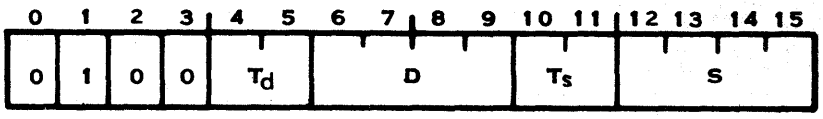
Execution results: Bits of (ga_d) corresponding to bits of (ga_s) equal to 1 are set to 1.

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

3.2.64 SET ZEROS CORRESPONDING SZC

Op Code: 4000

Format:



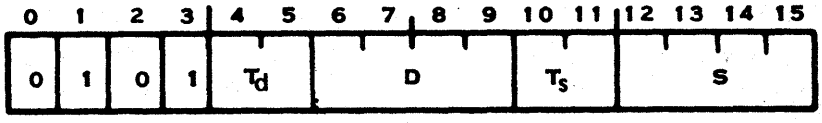
Execution results: Bits of (ga_d) corresponding to bits of (ga_s) equal to 1 are set to 0.

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

3.2.65 SET ZEROS CORRESPONDING, BYTE SZCB

Op Code: 5000

Format:



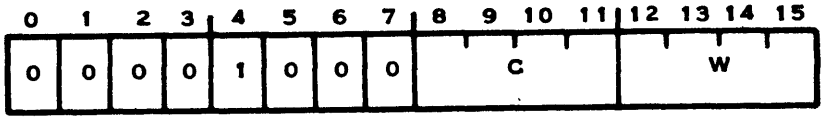
Execution results: Bits of (ga_d) corresponding to bits of (ga_s) equal to 1 are set to 0.

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

3.2.66 SHIFT RIGHT ARITHMETIC SRA

Op Code: 0800

Format:



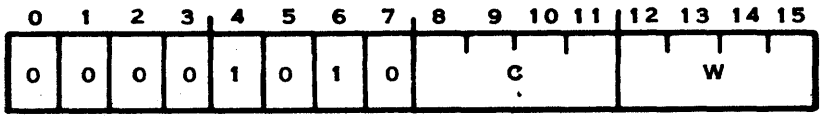
Execution results: Shift the bits of (wa) to the right, extending the sign bit to the right to fill vacated bit positions. When the contents of C is greater than 0, shift the number of bit positions contained in C. Otherwise shift the number of bit positions contained in the four least significant bits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

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

3.2.67 SHIFT LEFT ARITHMETIC SLA

Op Code: 0A00

Format:



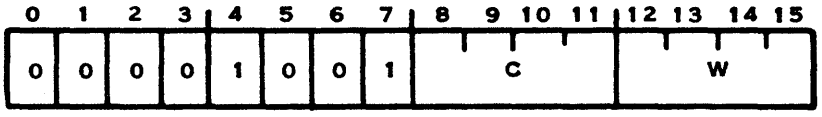
Execution results: Shift the bits of (wa) to the left, filling the vacated bit positions with zeros. The number of bit positions is determined by the value of C or the contents of workspace register 0 as described in paragraph 3.2.66.

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

3.2.68 SHIFT RIGHT LOGICAL SRL

Op Code: 0900

Format:



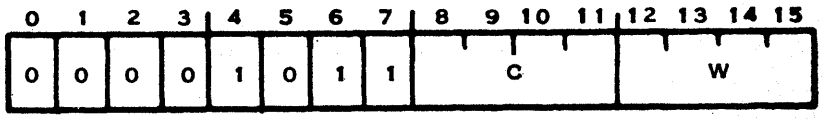
Execution results: Shift the bits of (wa) to the right, filling the vacated bit positions with zeros. The number of bit positions is determined by the value of C or the contents of workspace register 0 as described in paragraph 3.2.66.

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

3.2.69 SHIFT RIGHT CIRCULAR SRC

Op Code: 0B00

Format:



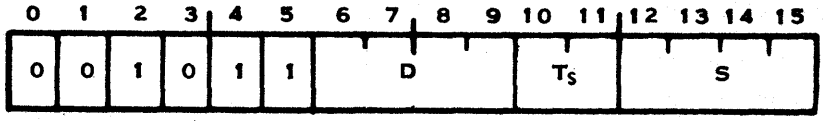
Execution results: Shift the bits of (wa) to the right, filling the vacated bit positions with the bits shifted out at the right. The number of bit positions shifted is determined by the value of C or the contents of workspace register 0 as described in paragraph 3.2.66.

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

3.2.70 EXTENDED OPERATION XOP

Op Code: 2C00

Format:



Execution results: $ga_S \rightarrow$ (workspace register 11)
$(0040_{16} + (D *4)) \rightarrow (WP)$
$(0042_{16} + (D *4)) \rightarrow (PC)$

(WP) → (workspace register 13)
(PC) → (workspace register 14)
(ST) → (workspace register 15)

Status bits affected: Extended operation.

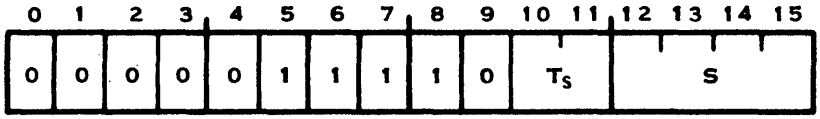
Model 990/10 Computer: An extended operation may be alternatively implemented by user-supplied hardware. When hardware is connected for the specified operation no context switch occurs, and the hardware performs the operation. When a Model 990/10 Computer performs a software-implemented extended operation,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

3.2.71 LONG DISTANCE SOURCE LDS

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

Op Code: 0780

Format:



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

When Privileged Mode bit is set to 1: Error interrupt

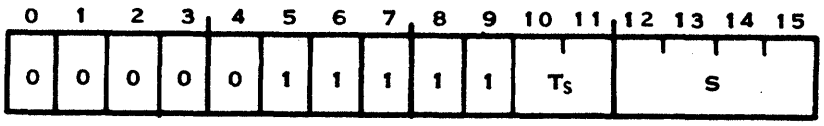
Status bits affected: None.

3.2.72 LONG DISTANCE DESTINATION LDD

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

Op Code: 07C0

Format:



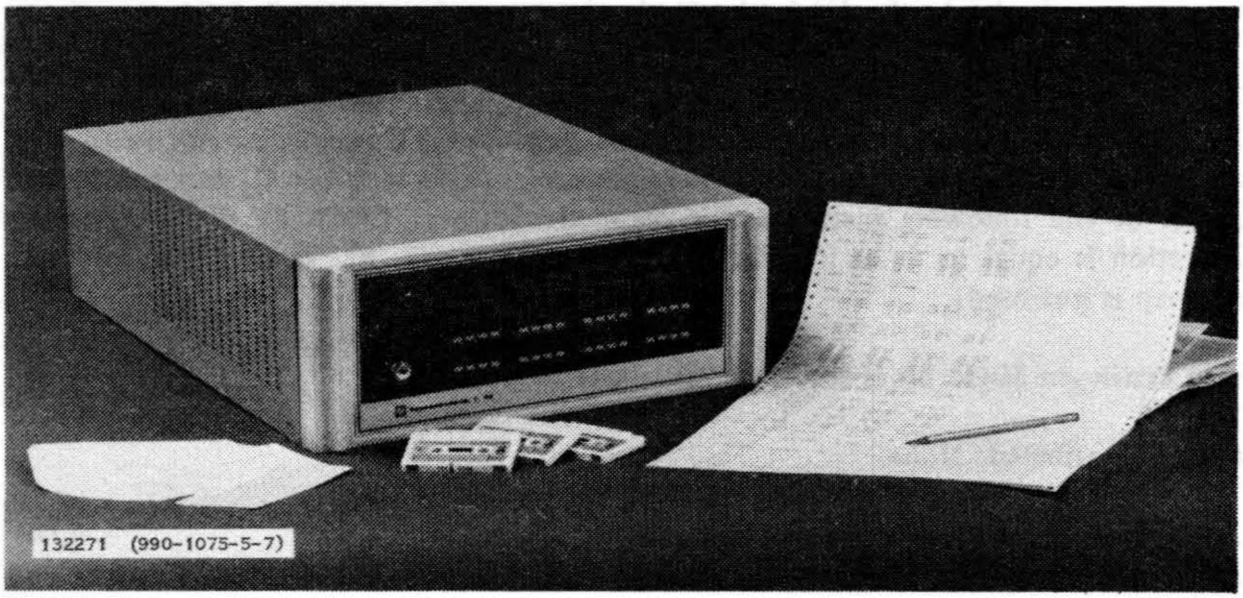
Execution results: When Privileged Mode bit (bit 7 of ST register) is set to 0: The contents of a six-word area at address ga_S are placed in map file 2, and the destination address of the following instruction is mapped with map file 2. (If T_d of the following instruction is equal to 0, or if the destination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

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

Status bits affected: None.

3.3 ASSEMBLY LANGUAGE CODING

The assembly language provides a means of writing a program without having to be concerned with actual memory addresses or machine instruction formats. The following paragraphs are an introduction to assembly language coding. Refer to the *Model 990 Computer Assembly Language Programmer's Guide*, for complete details of the language.



3.3.1 SYMBOLIC ADDRESSES. One of the big difficulties of programming in machine language is keeping track of memory addresses. The assembly language solves this problem by using symbolic addresses. The programmer assigns symbols to the locations and instructions as labels and refers to these locations by symbol rather than by address.

The assembler maintains a location counter to provide addresses for the symbols in the label field (labels) as they are read in the source program. As the assembler reads source program statements, it advances the location counter by one for each byte of object code resulting from each source statement. When there is a symbol in the label field, the assembler places the symbol and the corresponding location counter value in a symbol table. When the complete program has been read, the assembler has assigned

a machine address to each symbolic address (label) of the program. The assembler assembles object code using the address corresponding to the label in the entry in the symbol table.

A one-pass assembler reads the source statements of the program one time only, and builds the symbol table as it reads the statements. For this reason, the one-pass assembler requires that many of the labels used as operands be defined prior to their use as operands. Two-pass and multi-pass assemblers read the source statements two or more times, and build the complete symbol table before beginning to substitute machine addresses in a subsequent pass. These assemblers require fewer of the labels to be defined prior to their use as operands. The Model 990 Computer Assembly Language is used with a one-pass assembler (PX9ASM), a two-pass assembler (Cross Assembler), and a multi-pass assembler (SDSMAC).

3.3.2 SYMBOLIC OPERATION CODES. The assembly language includes a mnemonic operation code for each machine instruction of the instruction set. During the appropriate pass of the assembler, it substitutes the machine code for the mnemonic operation code as it assembles the instruction. Refer to table 3-3 for a summary of mnemonics.

The assembly language allows a label to be assigned to a machine instruction, and operands to be specified, according to machine instruction syntax, as follows:

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

This syntax definition means that a machine instruction may have a label, which is defined by the user. One or more blanks separate the label from the opcode. When the label is omitted, at least one blank must precede the opcode. The opcode, a generic name, may be any one of the set of mnemonic operation codes defined for the machine instructions. The opcode may also be an operation code defined in an assembler directive described in a subsequent paragraph. One or more blanks separate the opcode from an operand, if an operand is required. When a second operand is required, it is separated from the first operand by a comma. One or more blanks separate the operand or operands from the comments. Comments are optional, and have no effect on the assembly processing. They are printed in the listing to document the program.

Table 3-3. List of 990 Machine Instructions

MNEMONIC	OP CODE	INSTRUCTION
A	A000	ADD (WORD)
AB	B000	ADD (BYTE)
ABS	0740	ABSOLUTE VALUE
AI	0220	ADD IMMEDIATE
ANDI	0240	AND IMMEDIATE
B	0440	BRANCH
BL	0680	BRANCH AND LINK (W11)
BLWP	0400	BRANCH; LOAD WORKSPACE POINTER
C	8000	COMPARE (WORD)
CB	9000	COMPARE (BYTE)
CI	0280	COMPARE IMMEDIATE
CKOF	03C0	CLOCK OFF (NOTE 1,3)
CKON	03A0	CLOCK ON (NOTE 1,3)
CLR	04C0	CLEAR OPERAND
COC	2000	COMPARE ONES CORRESPONDING
CZC	2400	COMPARE ZEROES CORRESPONDING
DEC	0600	DECREMENT (BY ONE)
DECT	0640	DECREMENT (BY TWO)
DIV	3C00	DIVIDE
IDLE	0340	COMPUTER IDLE (NOTE 1,3)
INC	0580	INCREMENT (BY ONE)
INCT	05C0	INCREMENT (BY TWO)
INV	0540	INVERT
JEQ	1300	JUMP EQUAL (ST2=1)
JGT	1500	JUMP GREATER THAN (ST1=1)
JH	1B00	JUMP HIGH (ST0=1 AND ST2=0)
JHE	1400	JUMP HIGH OR EQUAL (ST0 OR ST2=1)
JL	1A00	JUMP LOW (ST0 AND ST2=0)
JLE	1200	JUMP LOW OR EQUAL (ST0 = 0 OR ST2 = 1)
JLT	1100	JUMP LESS THAN (ST1 AND ST2=0)
JMP	1000	JUMP UNCONDITIONAL
JNC	1700	JUMP NO CARRY (ST3 = 0)
JNE	1600	JUMP NOT EQUAL (ST2 = 0)
JNO	1900	JUMP NO OVERFLOW (ST4 = 0)
JOC	1800	JUMP ON CARRY (ST3 = 1)
JOP	1C00	JUMP ODD PARITY (ST5 = 1)
LDCR	3000	LOAD CRU
LDD	07C0	LONG DISTANCE DESTINATION (NOTE 1,2)
LDS	0780	LONG DISTANCE SOURCE (NOTE 1,2)
LI	0200	LOAD IMMEDIATE
LIMI	0300	LOAD INTERRUPT MASK IMMEDIATE (NOTE 1)
LMF	0320	LOAD MAP FILE (NOTE 1,2)
LREX	03E0	LOAD OR RESTART EXECUTION (NOTE 1,3)
LWPI	02E0	LOAD IMMEDIATE TO WORKSPACE POINTER
MOV	C000	MOVE (WORD)
MOVB	D000	MOVE (BYTE)
MPY	3800	MULTIPLY
NEG	0500	NEGATE (TWO'S COMPLEMENT)
ORI	0260	OR IMMEDIATE
RSET	0360	RESET AU (NOTE 1,3)
RTWP	0380	RETURN FROM INT. SUBR. (NOTE 1)
S	6000	SUBTRACT (WORD)
SB	7000	SUBTRACT (BYTE)
SBO	1D00	SET CRU BIT TO ONE
SBZ	1E00	SET CRU BIT TO ZERO
SETO	0700	SET ONES
SLA	0A00	SHIFT LEFT ARITHMETIC
SOC	E000	SET ONES CORRESPONDING (WORD)
SOCB	F000	SET ONES CORRESPONDING (BYTE)
SRA	0800	SHIFT RIGHT (MSB EXTENDED)
SRC	0B00	SHIFT RIGHT CIRCULAR
SRL	0900	SHIFT RIGHT LOGICAL
STCR	3400	STORE FROM CRU
STST	02C0	STORE STATUS REGISTER
STWP	02A0	STORE WORKSPACE POINTER
SWPB	06C0	SWAP BYTES
SZC	4000	SET ZEROES CORRESPONDING (WORD)
SZCB	5000	SET ZEROES CORRESPONDING (BYTE)
TB	1F00	TEST CRU BIT
X	0480	EXECUTE
XOP	2C00	EXTENDED OPERATION
XOR	2800	EXCLUSIVE OR

NOTES

1. PRIVILEGED INSTRUCTIONS – MODEL 990/10
2. MODEL 990/10 WITH MAP ONLY
3. NOT IMPLEMENTED IN TMS9900

The syntax definition just described illustrates the following conventions used in the syntax definitions for assembler directives in succeeding paragraphs:

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

3.4 ASSEMBLER DIRECTIVES

The assembly language includes assembler directives that supplement the machine instructions to form a source program. The assembler directives control the assembler, and define data for in the program. Subsequent paragraphs describe the five categories of directives, and additional paragraphs describe individual directives and the syntax for the directives.

3.4.1 INITIALIZING OR MODIFYING LOCATION COUNTER CONTENTS. The directives in this category are:

Absolute Origin	Block Starting With Symbol
Relocatable Origin	Block Ending With Symbol
Dummy Origin	Work Boundary.

The assembly language includes three directives that initialize the location counter, the Absolute Origin directive, the Relocatable Origin directive, and the Dummy Origin directive. If neither of the three directives is included before the first statement that results in object code, the assembler initializes the location counter to zero and assembles object code with relocatable addresses. When the assembler reads an Absolute Origin directive, it initializes the location counter with the value of the operand and assembles object code with absolute addresses. When the assembler reads a Relocatable Origin directive, the assembler initializes the location counter and assembles object code with absolute addresses. When the assembler reads a Dummy Origin directive, it initializes the location counter to the value of the operand, and processes source statements without producing object code.

The other three directives in this category modify the contents of the location counter. The Block Starting With Symbol statement reserves a specified area of memory and optionally assigns a label to the first location of the area. The Block Ending With Symbol statement also reserves an area of memory and optionally assigns a label to the location that follows the last location of the area. The Even directive forces word alignment by incrementing the location counter by one if it contains an odd address. It should be used wherever word alignment is required.

3.4.2 DEFINING THE ASSEMBLER OUTPUT. The directives in this category are:

Program Identifier	List Source
Page Title	No Source List
Output Options	Page Eject.

The Program Identifier directive assigns an identifier to the program and places the identifier in the object code. The Page Title directive specifies a title to be printed at the top of each page of the listing. The Output Options directive specifies output options for the assembly. Different options are available with each assembler. Refer to the *Model 990 Computer/TMS9900 Microprocessor Assembly Language Programmer's Guide* for the options of each assembler. The List Source directive relates to the No Source List directive. These directives allow the user to specify areas of his program to be omitted in the listing. The No Source List directive disables printing of the listing, and the List Source directive restores printing of the listing. The Page Eject directive allows the user to force a new page in the listing, which aids documentation of the program.

3.4.3 INITIALIZING CONSTANTS. The directives that initialize constants for the program are:

Initialize Byte	Initialize Text
Initialize Word	Define Assembly-Time Constant.

The Initialize Byte directive specifies values to be placed in one or more bytes of the object program. These may be decimal or hexadecimal values, or single characters. The assembler converts decimal and hexadecimal values to binary numbers, and characters to ASCII representation. The Initialize Word directive specifies values to be placed in one or more words of the object program, performing the same conversions. When characters are specified with the Initialize Word directive, the user must specify a pair of characters to be placed in a word. The Initialize Text directive specifies a string of characters to be placed in successive bytes of the object program. The characters are converted to ASCII representation. The Define Assembly-Time Constant directive allows the user to assign values to symbols to be used during assembly of the program.

3.4.4 DEFINING PROGRAM LINKAGE. The directives that define linkage between modules of a program are:

External Definition External Reference.

These directives allow a program to be assembled as a number of modules and linked to form an integrated program. Linking requires that any symbols in one module that are referenced in one or more other modules be identified. Linking also requires that any symbols referenced in a module that appear in another module be identified. The External Definition directive defines one or more symbols that appear as labels in the program module and are referenced in other modules of the program. The External Reference directive defines one or more symbols that are used as operands in the program module but appear as labels in other modules of the program.

3.4.5 ADDITIONAL DIRECTIVES. The following two miscellaneous directives are available in all assemblers:

Define Extended Operation Program End.

The Define Extended Operation directive allows the user to assign a symbolic operator code to an extended operation. Following entry of this directive, the symbol defined by the directive may be used as an opcode in a source statement. The Program End directive is required in every source program to define the end of the program.

The Macro Assembler of the Software Development System (SDSMAC) supports the following additional directives:

Define Operation Copy Source File

Workspace Pointer.

The Define Operation directive allows the user to define a synonym for an opcode. The synonym may then be used as an opcode in a source statement. The Workspace Pointer directive defines the current workspace to the assembler, allowing SDSMAC to recognize workspace register addresses expressed as symbolic memory addresses. The Copy Source File directive copies source statements from source files, incorporating the copied statements into the current source program.

3.4.6 ABSOLUTE ORIGIN. The absolute origin directive defines location counter contents as absolute, and initializes the location counter with the specified value. The syntax of the AORG directive is defined as follows:

[<label>]b...AORGb...<wd exp>b...[<comment>]

The operand (wd exp) must be a well-defined expression, which is an expression that has an absolute value, any symbol of which is previously defined. The following is an example of an AORG directive:

AORG	>40	Initializes the location counter to absolute address 0040_{16}.

3.4.7 RELOCATABLE ORIGIN. The relocatable origin directive defines location counter contents as relocatable, and initializes the location counter with the specified value. When the operand is omitted, the directive initializes the location counter to the value following the previous relocatable code of the program, or to zero. The syntax of the RORG directive is defined as follows:

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

The following are examples of RORG directives:

START	RORG	256	Initializes the location counter to relocatable address 0100_{16} and assigns that value to label START.
	RORG		If there is no previous RORG directive in the program, initializes the location counter to relocatable address zero. If a previous RORG directive had resulted in a block of relocatable code ending at location $00FF_{16}$, initializes the location counter to relocatable address 0100_{16}.

3.4.8 DUMMY ORIGIN. The dummy origin directive defines location counter contents as a dummy value, and initializes the location counter with the specified value. The syntax of the DORG directive is defined as follows:

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

The following is an example of a DORG directive:

TDATA	DORG	0	Initializes the location counter to provide a dummy section in which addresses are relative to zero.

3.4.9 BLOCK STARTING WITH SYMBOL. The block starting with symbol directive advances the location counter a specified number of bytes, and optionally assigns a label to the first location of the block. The syntax of the BSS directive is defined as follows:

[<label>]b...BSSb...<wd exp>b...[<comment>]

The following is an example of a BSS directive:

BUFF	BSS	80	Reserves an 80-byte area at location BUFF.

3.4.10 BLOCK ENDING WITH SYMBOL. The block ending with symbol directive advances the location counter a specified number of bytes, and optionally assigns a symbol to the location following the block. The syntax of the BES directive is defined as follows:

[<label>]b...BESb...<wd exp>b...[<comment>]

The following is an example of a BES directive:

	BES	24	Reserves a 24-byte area of memory.

3.4.11 WORD BOUNDARY. The word boundary directive advances the location counter to the next word boundary (even) address when the location counter contains an odd address. The syntax of the EVEN directive is as follows:

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

The following is an example of an EVEN directive:

MSG	EVEN	If the location counter contains an even address, leaves the location counter unaltered. Otherwise, adds one to the location counter value, and places the sum in the location counter. Assigns the previous value to label MSG.

3.4.12 PROGRAM IDENTIFIER. The program identifier directive places the specified identifier in the object output. The syntax of the IDT directive is defined as follows:

[<label>]b...IDTb...<string>b...[<comment>]

The operand is a string of up to eight characters enclosed in single quotes (' '). The following is an example of an IDT directive:

```
IDT     'PROG1'     Assigns identifier PROG1 to the
                    program, and places the identifier
                    in the object program.
```

3.4.13 PAGE TITLE. The page title directive specifies a title to be placed in the heading of each page of the listing. The directive must be the first source statement if it is to be printed on the first page. The syntax of the TITL directive is defined as follows:

[<label>]b...TITLb...<string>b...[<comment>]

The operand consists of a string of up to 50 characters. The following is an example of a TITL directive:

```
TITL    'MAIN PROGRAM'    Specifies the page title
                          MAIN PROGRAM to be printed on
                          the pages of the listing.
```

3.4.14 OUTPUT OPTIONS. The output options directive specifies options for the assembly. This directive does not apply to the one-pass assembler, PX9ASM. The Cross Assembler and the Macro Assembler, SDSMAC, include different options. Refer to the *Model 990 Computer Assembly Language Programmer's Guide* for the keywords and options applicable to the assembler being used. The syntax of the OPTION directive is defined as follows:

b...OPTIONb...<keyword>[,<keyword>]...b...[<comment>]

The following is an example of an OPTION directive:

```
OPTION    XREF          Specifies the printing of a cross
                        reference listing of the assembly.
```

3.4.15 LIST SOURCE. The list source directive restores printing of the source listing. The syntax of the LIST directive is defined as follows:

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

The following is an example of a LIST directive:

```
LIST          Causes listing of source statements
              to resume.
```

3.4.16 NO SOURCE LIST. The no source list directive inhibits printing of the source listing. The syntax of the UNL directive is defined as follows:

[<label>]ƀ...UNLƀ...[<comment>]

The following is an example of an UNL directive:

UNL	Causes listing of source statements to be suspended.

3.4.17 PAGE EJECT. The page eject directive causes the assembler to continue printing the source listing at the top of the next page. The syntax of the PAGE directive is defined as follows:

[<label>]ƀ...PAGEƀ...[<comment>]

The following is an example of a PAGE directive:

PAGE	Causes the printer on which the source listing is being printed to skip to the top of the next page.

3.4.18 INITIALIZE BYTE. The initialize byte directive places one or more values in one or more successive bytes of memory. The values may be expressed as decimal or hexadecimal numbers, labels, or as single ASCII characters. The syntax of the BYTE directive is defined as follows:

[<label>]ƀ...BYTEƀ...<exp>[,<exp>]...ƀ...[<comment>]

The following is an example of a BYTE directive:

KON	BYTE	12,>F,'A'	Initializes a three-byte area at location KON. The bytes contain $0C_{16}$, $0F_{16}$, and 41_{16}, respectively.

3.4.19 INITIALIZE WORD. The initialize word directive places one or more values in one or more successive words of memory. The values may be expressed as decimal or hexadecimal numbers, labels, or as pairs of ASCII characters. The syntax of the DATA directive is defined as follows:

[<label>]ƀ...DATAƀ...<exp>[,<exp>]...ƀ...[<comment>]

The following is an example of a DATA directive:

D1	DATA	BUFF,>08FE,'AB'	Initializes a three-word area at location D1. The words contain the value of label BUFF, $08FE_{16}$, and 4142_{16}, respectively.

3.4.20 INITIALIZE TEXT. The initialize text directive places the ASCII representations of a string of characters (up to 52) in successive bytes of memory. When a minus sign (–) is placed before the string, the ASCII character representation of the last character is negated (two's complement). The syntax of the TEXT directive is defined as follows:

[<label>]b...TEXTb...[–]<string>b...[<comment>]

The following is an example of a TEXT directive:

MSG1	TEXT	'ENTER NAME'	Places ten ASCII characters in ten successive bytes, as follows: 45_{16}, $4E_{16}$, 54_{16}, 45_{16}, 52_{16}, 20_{16}, $4E_{16}$, $4A_{16}$, $4D_{16}$, 45_{16}.

3.4.21 DEFINE ASSEMBLY-TIME CONSTANT. The define assembly-time constant directive defines a constant to use during assembly. The syntax of the EQU directive is defined as follows:

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

The following is an example of an EQU directive:

TWO	EQU	2	Assigns the value 2 to the label TWO.

3.4.22 EXTERNAL DEFINITION. The external definition directive makes one or more symbols in a program module available for use in other modules. The syntax for the DEF directive is defined as follows:

[<label>]b...DEFb...<symbol>[,<symbol>]...b...[<comment>]

The following is an example of a DEF directive:

	DEF	BUFF,MSG1	Causes the assembler to make the values of labels BUFF and MSG1 available for linking in other modules.

3.4.23 EXTERNAL REFERENCE. The external reference directive accesses one or more symbols in other program modules. The syntax for the REF directive is defined as follows:

[<label>]b...REFb...<symbol>[,<symbol>]...b...[<comment>]

The following is an example of an REF directive:

REF	DAT1,BUFF2	Causes the assembler to make labels DAT1 and BUFF2 available for linking to other modules.

3.4.24 WORKSPACE POINTER. The workspace pointer directive is available with the SDSMAC assembler only. The workspace pointer directive defines the current workspace to the assembler. The syntax for the WPNT directive is defined as follows:

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

The following is an example of a WPNT directive:

WPNT	WS1	Defines the workspace at location WS1 to the assembler as the current workspace.

3.4.25 COPY SOURCE FILE. The copy source file directive is available with the SDSMAC assembler only. The copy source file directive causes the assembler to obtain source statements form the specified file. The syntax for the COPY directive is defined as follows:

[<label>]b...COPYb...<file name>b...[<comment>]

The following is an example of a COPY directive:

COPY	FILE1	Includes the contents of FILE1 in the current source program at this point.

3.4.26 DEFINE OPERATION. The define operation directive is available with the SDSMAC assembler only. The directive defines a synonym for an operation. The syntax for the DFOP directive is defined as follows:

[<label>]b...DFOPb...<symbol>,<operation>b...[<comment>]

The following is an example of a DFOP directive:

ADD	DFOP	ADD,A	Defines ADD as a synonym for operation code A.

3.4.27 DEFINE EXTENDED OPERATION. The define extended operation directive assigns a symbol for an extended operation. The syntax for the DXOP directive is defined as follows:

[<label>]b...DXOPb...<symbol>,<term>b...[<comment>]

The term is a number, 0 through 15, that specifies an extended operation. The symbol is the symbol to be used as an operation code for the extended operation. The following is an example of the DXOP directive:

DXOP	FADD,7	Assigns the symbol FADD as an opcode for extended operation 7.

3.4.28 PROGRAM END. The program end directive terminates a program module or program. The syntax for the END directive is as follows:

[<label>]b...ENDb...[<symbol>]b...[<comment>]

Including the optional symbol operand causes the assembler to place the value of the symbol in the object code as the entry point. The following is an example of the END directive:

END	START	Terminates the program module and defines START as the entry point.

3.5 PSEUDO-INSTRUCTIONS

The assemblers for the Model 990 Computers support two pseudo-instructions, each of which assembles an instruction with a specific operand. In addition, the SDSMAC assembler supports an additional pseudo-instruction that assembles three assembler directives.

The no-operation pseudo-instruction is used to provide an instruction that has no effect on the execution of the program. It has the effect of reserving a word location in the executable portion of the program. The return pseudo-instruction is used to return from a subroutine when a link to the calling program is stored in workspace register 11. The BL instruction stores a link in workspace register 11 when it transfers control to a subroutine. The transfer vector pseudo-instruction (SDSMAC only) provides a transfer vector for a subroutine to which control is passed by a context switch with a BLWP instruction. The pseudo-instruction also defines the new workspace to the assembler.

3.5.1 NO OPERATION. The no operation pseudo-instruction has no effect on the execution of the program. The assembler substitutes a jump instruction to the next instruction in sequence. The syntax for the NOP pseudo-instruction is defined as follows:

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

3.5.2 RETURN. The return pseudo-instruction returns control from a common workspace subroutine entered by a BL instruction. The assembler substitutes a branch instruction to the address stored in workspace register 11. The syntax for the RT pseudo-instruction is defined as follows:

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

3.5.3 TRANSFER VECTOR. The transfer vector pseudo-instruction provides a transfer vector consisting of an address to be placed in the WP register and another address to be placed in the PC. The assembler substitutes two DATA directives to build the vector, and a WPNT directive to define the workspace to the assembler. The syntax for the XVEC pseudo instruction is defined as follows:

<label>b...XVECb...<wp address>[,<subr address>]b...[<comment>]

The first operand is the workspace address, to be placed in the WP register. The optional second operand is the subroutine address, to be placed in the PC. When the second operand is omitted, the assembler assumes that the entry point of the subroutine is the instruction that follows the XVEC directive. The following is an example of an XVEC directive:

```
CONV    XVEC    WS2,SUB2
```

Provides a transfer vector at location CONV consisting of address WS2 and address SBU2. Also defines WS2 to the assembler as the current workspace. Equivalent to:

```
CONV    DATA    WS2
        DATA    SUB2
        WPNT    WS2
```

3.6 MEMORY ADDRESSING

The following paragraphs provide some guidelines for addressing the memory of the Model 990 Computer. The following five modes are addressing options for one or both operands of many instructions:

- Workspace register addressing
- Indirect workspace register addressing

- Symbolic memory addressing
- Indexed memory addressing
- Indirect autoincrement workspace register addressing.

Other instructions use one of the following specific types of addresses:

- Jump addresses
- CRU addresses
- Immediate addresses.

3.6.1 CODING OF WORKSPACE REGISTER ADDRESSES. The workspace register address is one of five addressing options for many instructions, and is the specific address type for other instructions.

The workspace concept is an important feature of the Model 990 Computer. A workspace is a set of 16 consecutive memory words that may be addressed as workspace registers 0 through 15. Beginning with the initial context switch that occurs as power is applied, a value is placed in the WP register during each context switch. This value is the address of the first word of the workspace (workspace register 0), and may be any address in memory. It should be greater than 80_{16}, and less than or equal to the highest address in memory minus 31. Execution of a BLWP, LREX, XOP or RTWP instruction or processing of an interrupt causes a context switch. Execution of an LWPI instruction activates a new workspace without performing a context switch.

A workspace register address may be an integer from 0 through 15, or a symbol having a value in that range. The addressed workspace register contains the operand. The assemblers have preassigned symbols for workspace register addresses. These symbols are R0 through R15, for workspace registers 0 through 15, respectively. Figure 3-5 shows an instruction having two workspace register addresses, in memory location 300_{16}. The assembly language form of the instruction is shown, but actually memory would contain the machine language assembled for the instruction. The WP register contents, 100_{16}, is the address of workspace register 0. The PC contains 302_{16}, the address following the current instruction. The ST register contents are unimportant to the execution of the instruction. The current instruction is an Add instruction, that adds the contents of workspace register 5 to the contents of workspace register 6, and places the sum in workspace register 6. The contents of workspace registers 5 and 6 are shown prior to execution. After executing the instruction, workspace register 5 still contains 3, and workspace register 6 contains 10.

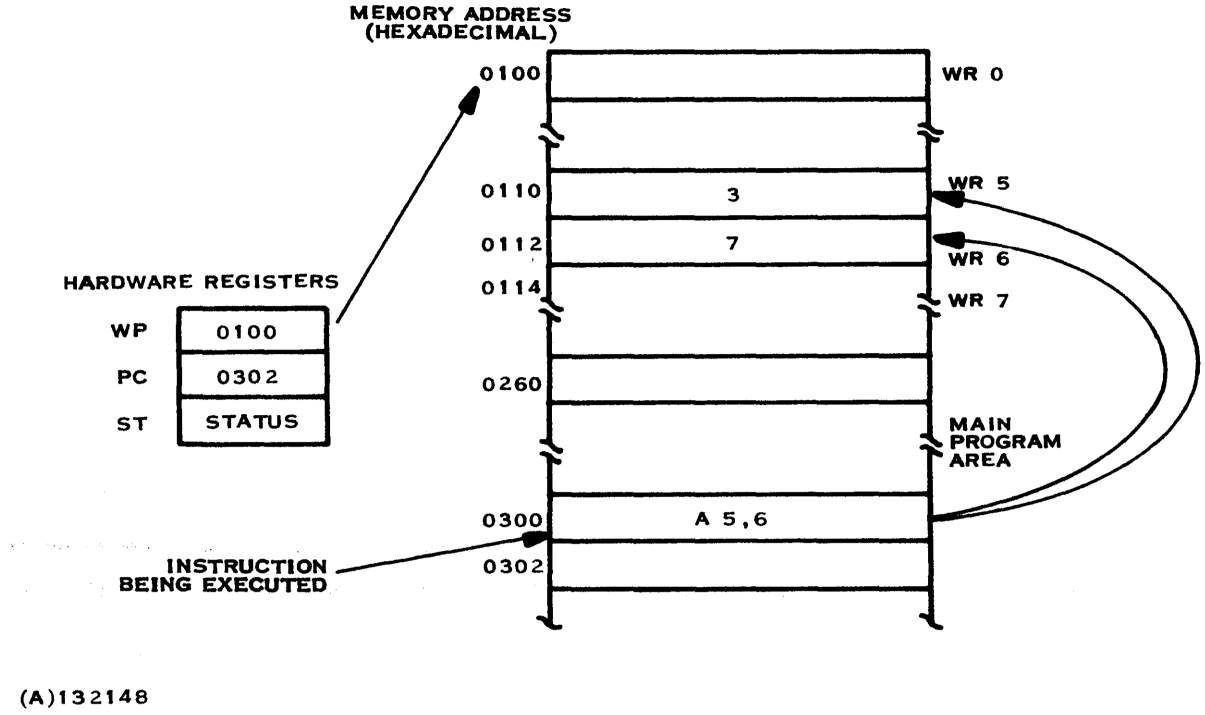


Figure 3-5. Workspace Register Addressing

The following are additional examples of coding of workspace register addresses:

	DECT	6	Decrement the contents of workspace register 6 by two.
	SB	R1,R5	Subtract the contents of the left byte of workspace register 1 from the contents of the left byte of workspace register 5, and place the remainder in the left byte of workspace register 5.
SUM	EQU	7	Assigns the value 7 to label SUM.
	CLR	SUM	Sets the contents of workspace register 7 to zero.

3.6.2 · CODING OF INDIRECT ADDRESSES. Another of the five addressing options is the indirect workspace register address. The workspace register is specified in the same manner as in a workspace register address, but the number or symbol is preceded by an asterisk (*). The workspace register contains the address of the operand.

Figure 3-6 shows an example of indirect addressing. The instruction in address 0304_{16} has a workspace register address for the source operand, and an indirect workspace

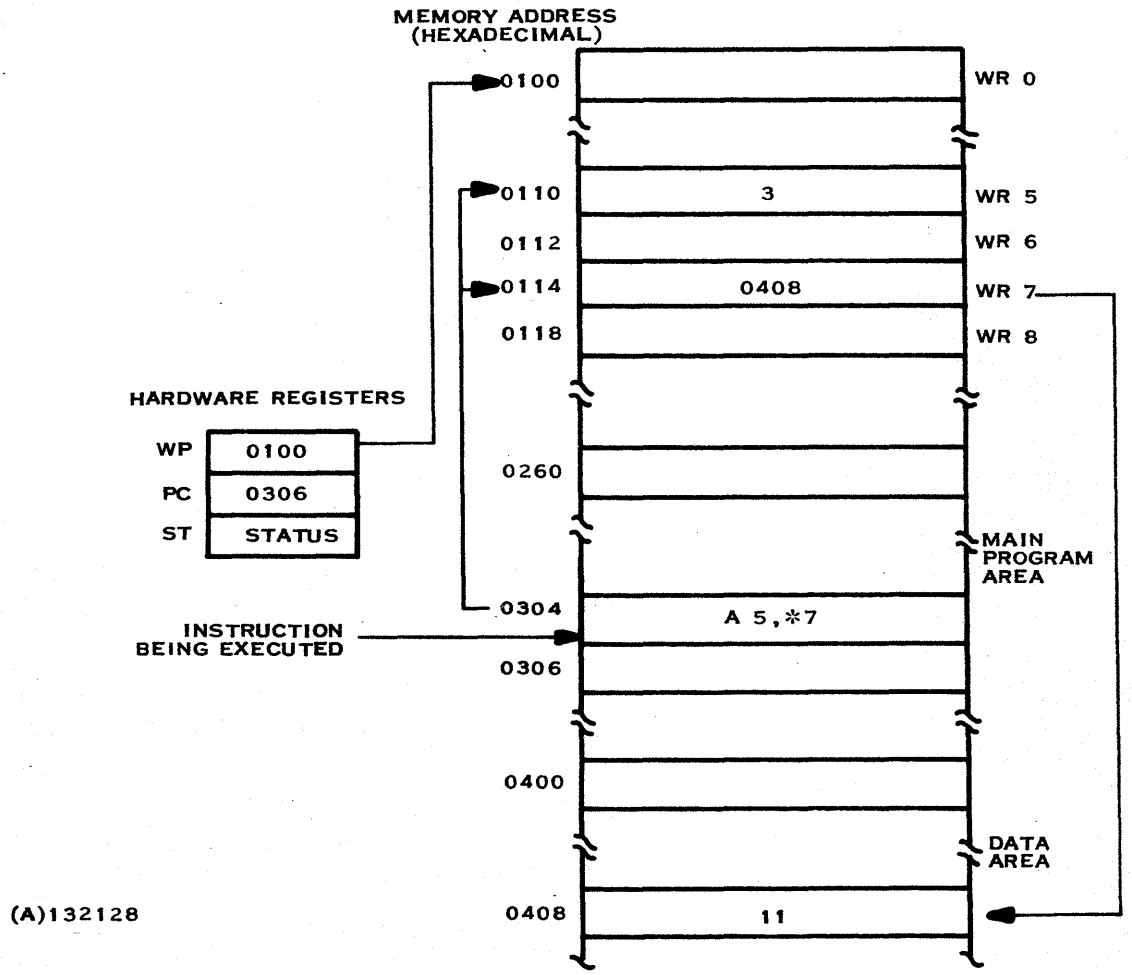


Figure 3-6. Indirect Workspace Register Addressing

register address for the destination operand. The WP register contents defines the same workspace as in the previous example, and the PC contains 0306_{16}, the address of the following instruction. The ST register contents are unimportant to the execution of the instruction. The current instruction is an Add instruction that adds the contents of workspace register 5 to the operand addressed by workspace register 7. Since workspace register 7 contains 0408_{16}, the operand is the contents of address 0408_{16}. The instruction places the sum in the destination operand, the contents of address 0408_{16}. Figure 3-6 shows the operands prior to execution of the instruction. After execution, workspace register 5 still contains 3, workspace register 7 still contains 0408_{16}, and address 0408_{16} now contains the sum, 14.

Indirect workspace register addressing may be used to access data at an address that has been placed in a workspace register. For example, the BL instruction places the address of the word following the instruction in workspace register 11. This word may contain data related to the BL instruction. The following source code could be used in a subroutine to access data in the word following the BL instruction that transfers control to the subroutine:

```
      BL      @SUB
      DATA    10
        .
        .
        .
SUB   MOV     *11,R2        Copy data in the address in workspace
                            register 11 into workspace register 2.
```

After the instructions that perform the processing of the subroutine, the subroutine may include the following instructions:

```
      INCT    11            Add two to the contents of work-
                            space register 11, to return to
                            the instruction following the data
                            word.

      B       *11           Return to the address in work-
                            space register 11.
```

Other examples of coding indirect workspace register addresses are as follows:

```
      A       *1,*5         Add the contents of the word at
                            the location in workspace register
                            1 to the contents of the word
                            at the location in workspace register
                            5, and place the sum in the location
                            in workspace register 5.

      SB      *R1,*R5       Subtract the byte at the address in
                            workspace register 1 from the byte
                            at the address in workspace
                            register 5, and place the remainder
                            in the byte at the address in
                            workspace register 5.
```

3.6.3 CODING OF SYMBOLIC ADDRESSES. The third of the five addressing options is the symbolic memory address. When this mode is used, the assembler supplies the memory address corresponding to the symbol in a word following the instruction. The address is coded as an at sign (@) followed by a symbol or expression that represents the address. Integer addresses may be used when the addresses are known; for example, when they are absolute addresses.

Figure 3-7 illustrates an instruction with a symbolic address. The instruction at address $030C_{16}$ has a workspace register address for the source operand and a symbolic memory address for the destination operand. The WP register defines 0100_{16} as the address of the workspace, and the PC contains 0310_{16}, the address of the next instruction. The ST register contents are unimportant to the execution of the instruction. The current instruction is an Add instruction that adds the contents of workspace register 5 to the contents of location PRIME, address 0400_{16}. Figure 3-7 shows the contents of the operands prior to execution of the instruction. After execution, workspace register 5 still contains 3, but address 0400_{16} contains the sum, 5.

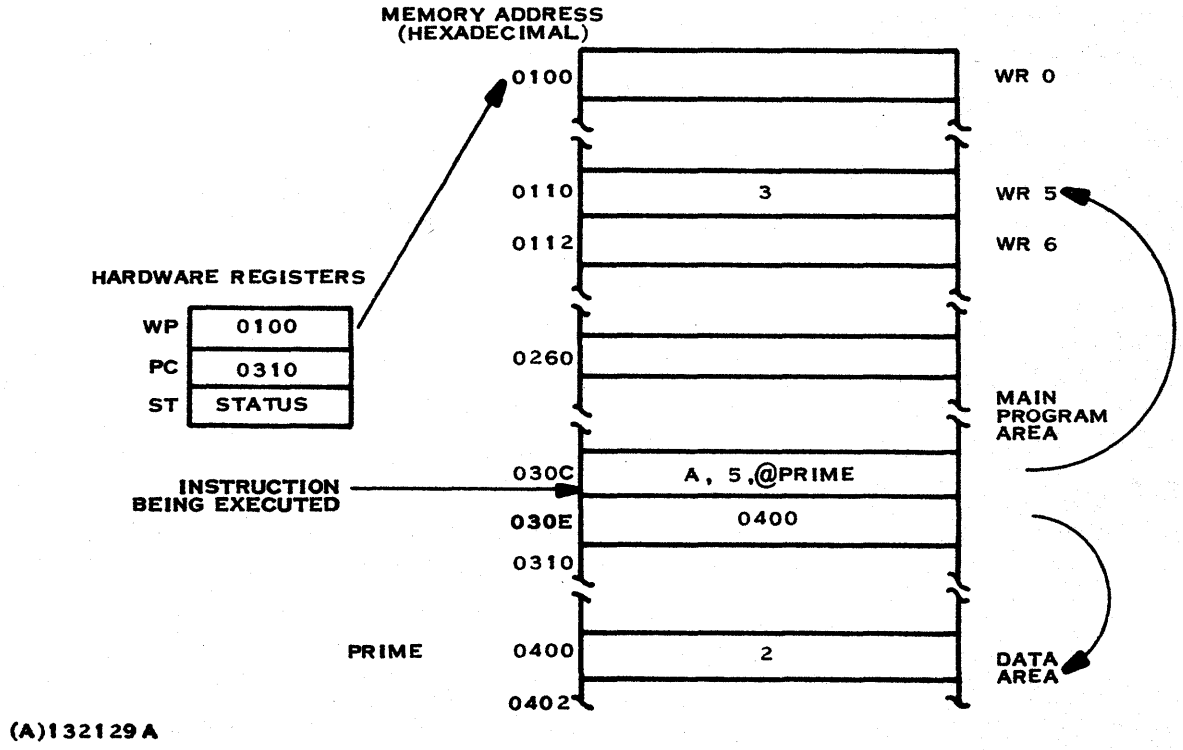


Figure 3-7. Symbolic Memory Addressing

The following are additional examples of symbolic memory address coding:

MOV	@IN,@OUT+2	Copy the contents of location IN into the word location following location OUT.
BL	@>42	Branch to absolute location 0042_{16}, and store a link in workspace register 11.
C	@>100,@WP1	Compare the contents of address 0100_{16} to the contents of location WP1.

Symbolic memory addresses for the SDSMAC Assembler may be coded without the at sign. SDSMAC maintains a record of the current workspace address, and translates a symbolic memory address into a workspace register address when the value of the symbol is neither less than the workspace address nor greater than the sum of the workspace address plus 30. Processing of symbolic addresses in this manner makes it unnecessary to require the use of the at sign(@). The following source statements would both be accepted by SDSMAC, and would result in identical object code:

ABS	@TOTAL	Places the absolute value of the contents of location TOTAL in location TOTAL.
ABS	TOTAL	Places the absolute value of the contents of location TOTAL in location TOTAL.

3.6.4 CODING OF INDEXED ADDRESSES. The fourth of the five addressing options is the indexed memory address. When this mode is used, the address includes a workspace register designated as an index register. The computer adds the contents of this register to the address at execution time to form the effective address for the instruction. An indexed memory address is coded as an at sign followed by a symbol, expression, or integer, and an index register enclosed in parentheses. The index register is specified as an integer in the range of 1 through 15, or as a symbol having a value in that range. In other words, an indexed memory address is a symbolic memory address followed by an index register specification.

Figure 3-8 shows an example of indexed memory addressing. The instruction at address 0310_{16} has a workspace register address for the source operand, and an indexed memory address for the destination operand. The WP register defines 0100_{16} as the current workspace, and the PC contains 0314_{16}, the address of the next instruction. The ST register contents are unimportant to the execution of the

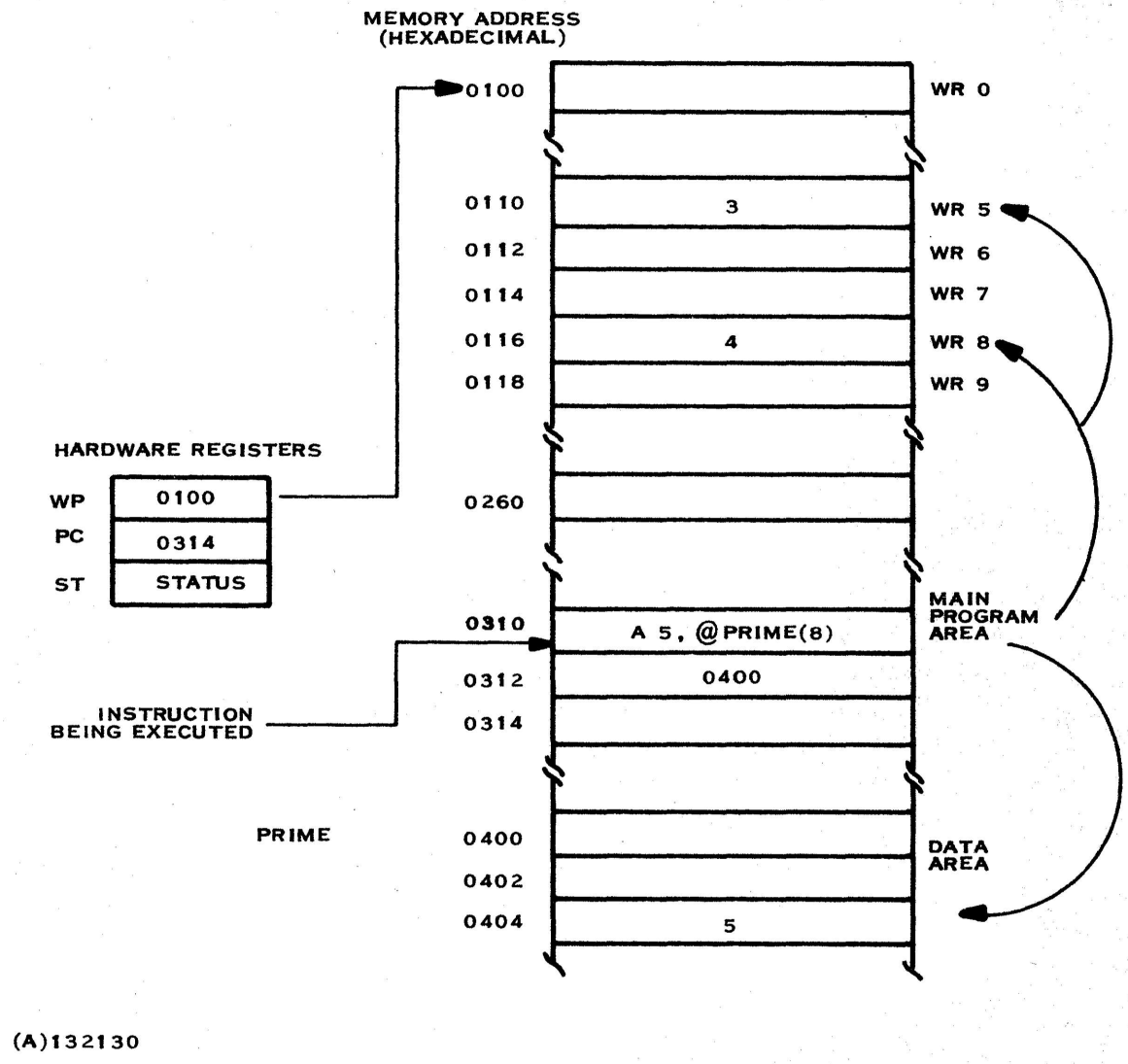


Figure 3-8. Indexed Memory Addressing

instruction. The current instruction is an Add instruction that adds the contents of workspace register 5 to the contents of an address that is the sum of the contents of workspace register 8 and the value of symbol PRIME. The value of symbol PRIME is 0400_{16}, and the contents of workspace register 8 is 4, so the destination operand is the contents of location 0404_{16}. The instruction places the sum in this address. Figure 3-8 shows the contents of the operands prior to the execution of the instruction. After execution, workspace register 5 still contains 3, workspace register 8 still contains 4, but address 0404_{16} now contains the sum, 8.

When the address of a buffer, table, or array is placed in a workspace register other than workspace register 0, that register may be used as an index register to access the

data. For example, if the address of a buffer were in workspace register 6, the following code would access specific bytes and words of the buffer:

MOV	@6(6),@VAL	Copy the contents of the fourth word of the buffer (buffer address + 6) into location VAL.
CB	@1(6),@KEY	Compare the contents of the second byte of the buffer (buffer address +1) with the contents of the byte at location KEY.
SZC	@MASK,@8(6)	Mask off the fifth word of the buffer (buffer address + 8) according to the one bits of location MASK.

3.6.5 CODING OF AUTOINCREMENT ADDRESSES. The fifth addressing option is the indirect workspace register autoincrement address. In this mode, the contents of the specified workspace register is the address of the operand, and are incremented after the operand is accessed. The increment is two for word instructions or one for byte instructions. An indirect workspace register autoincrement address is coded as an asterisk followed by an integer from 0 through 15, or a symbol or expression having a value in that range, and a plus sign (+). In other words, an indirect workspace register autoincrement address is an indirect workspace register address followed by a plus sign.

Figure 3-9 shows an example of autoincrement addressing. The instruction at address 0308_{16} has a workspace address for the source operand and an indirect autoincrement address for the destination operand. The WP register defines the current workspace at address 0100_{16}, and the PC contains the address of the following instruction, $030A_{16}$. The ST register contents are unimportant to the execution of the instruction. The current instruction is an Add instruction that adds the contents of workspace register 5 to the contents of the address in workspace register 7, places the sum in that address, and increments workspace register 7 by two. Figure 3-9 shows the contents of the operands prior to the execution of the instruction. After the instruction is executed, workspace register 5 still contains 3, workspace register 7 contains $040A_{16}$, and address 0408_{16} contains the sum, 14.

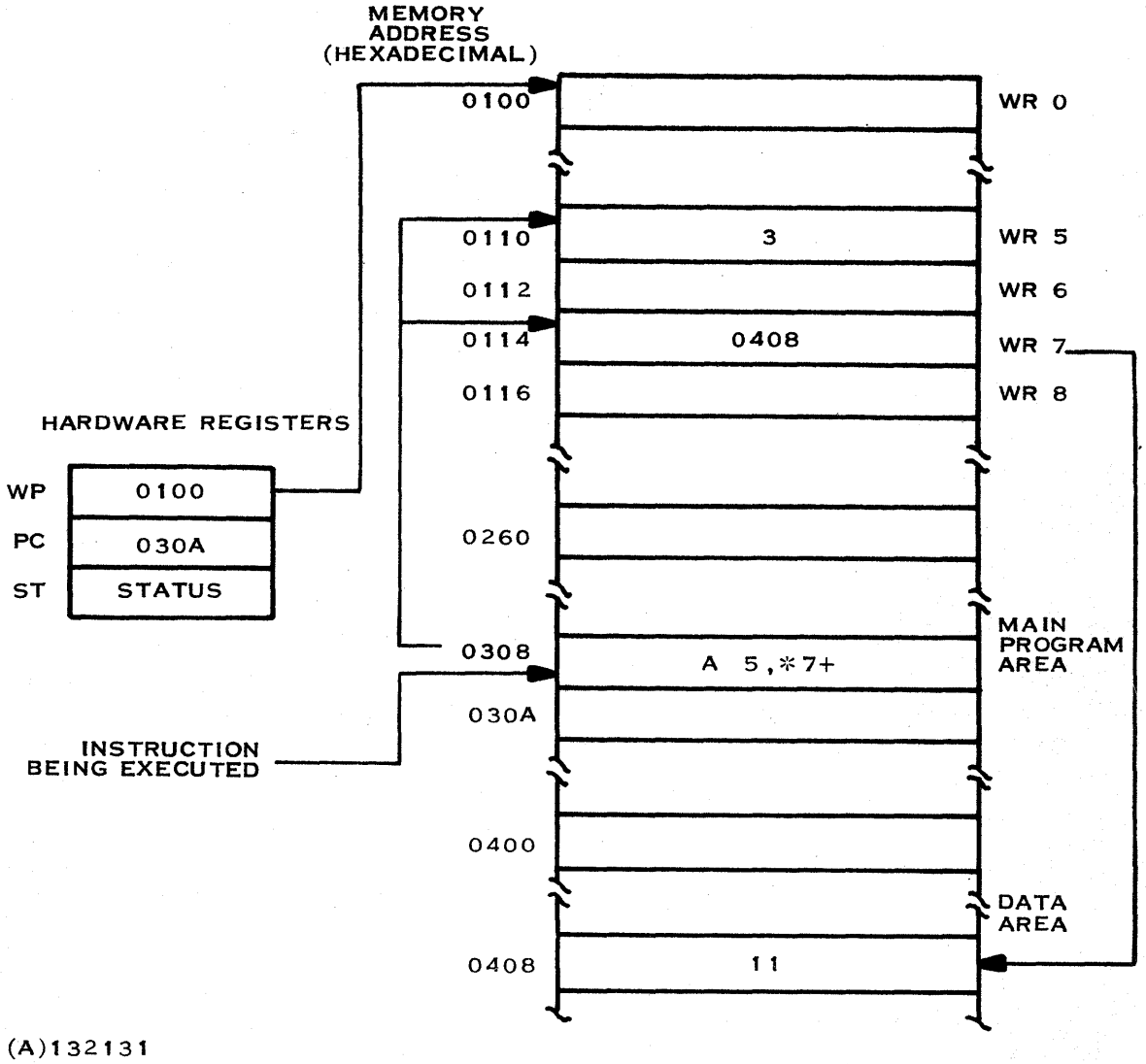


Figure 3-9. Indirect Workspace Register Autoincrement Addressing

This addressing mode is very useful for processing an array or table. For example, if an array of 12 numbers at location IN is to be added to the numbers in an array at location SUM, and the sums placed in the second array, the following instructions could be used:

LI	3,12	Place 12 in workspace register 3
LI	4,IN	Place the value assigned to label IN in workspace register 4.
LI	5,SUM	Place the value assigned to label SUM in workspace register 5.

```
LOOP    A       *4+,*5+     Add an element of array IN
                            to the corresponding element
                            of array SUM, and place the sum
                            in the element of array
                            SUM. Increment each address by
                            two.

        DEC     3           Subtract one from the
                            contents of workspace
                            register 3.

        JNE     LOOP        Branch to location LOOP
                            until the remainder in
                            workspace register 3 is
                            zero.
```

The Add instruction is executed a total of 12 times, once for each element of the arrays. Workspace register 4 contains the address of the word following array IN, and workspace register 5 contains the address of the word following array SUM, following execution of the code shown.

The following are additional examples of coding of indirect workspace register autoincrement addressing:

```
        MOV     *R11+,@ARG1 Copy the contents of
                            the address in workspace
                            register 11 into
                            location ARG1 and increment
                            the contents of workspace
                            register 11 by two.

        SB      3,*7+       Subtract the contents of
                            the most significant byte of
                            workspace register 3 from
                            the contents of the address
                            in workspace register 7,
                            and increment the contents
                            of workspace register 7
                            by one.

        CLR     *R3+        Clear the contents of the
                            address in workspace
                            register 3 to zero, and
                            increment the address in
                            workspace register 3 by
                            two.
```

3.6.6 CODING OF JUMP ADDRESSES. The jump instructions require symbolic addresses or expressions that are not preceded by at signs. The assembler computes the displacement required in the machine instruction by evaluating the address, subtracting the location counter value from the address value, and dividing the difference by two. This displacement must be in the range of −128 through +127. A symbolic address that results in a displacement outside of this range is an invalid address.

The following are examples of coding of jump addresses:

JMP	START	Transfer control to the instruction at location START.
JNE	LOOP	Transfer control to the instruction at location LOOP if the equal status bit is equal to zero, or to the following instruction if the equal status bit is equal to one.
JOP	ODD	Transfer control to the instruction at location ODD if the odd parity status bit is equal to one, or to the following instruction if the odd parity status bit is equal to zero.
JOC	CARRY	Transfer control to the instruction at location CARRY if the carry status bit is equal to one, or to the following instruction if the carry status bit is equal to zero.

3.6.7 CODING OF CRU ADDRESSES. The Communication Register Unit is addressed using base addressing. The CRU instructions that transfer or test a single bit specify that bit as a displacement in the range of −128 through +127 from the base address in workspace register 12. The CRU instructions that transfer a group of consecutive bits require that the base address in workspace register 12 be the lowest bit address in the group of bits. In either case, the base address in workspace register 12 is shifted one bit position to the left, placing it in bits 3 through 14 of the register. This means that the base address is effectively twice the bit number.

The displacement of an SBO, SBZ, or TB instruction is an integer in the range of −128 through +127, or an expression having a value within that range. If an expression is used, any symbol in the expression must have been previously defined. The following are examples of coding CRU instructions:

LI	R12,>40	Place the value 40_{16} into CRU base register. This value corresponds to CRU line 32.
SBO	3	Set CRU line 35 to one.
SBZ	A	Set CRU line 42 to zero.
TB	4	Place the value on CRU line 36 into the equal bit of the status register.

The first operand for the LDCR and STCR instructions is an address in either of the five addressing modes that specifies the memory locations for the transfer of data. The second operand is the number of bits to be transferred, expressed as an integer 0 through 15, or an expression or symbol having a value in that range. The value of zero transfers 16 bits. The following are additional examples of coding CRU instructions:

LI	R12,>20	Place the value 20_{16} in the CRU base register. This value corresponds to CRU line 16.
STCR	*R5,0	Place the value on CRU lines 16 through 31 into a word of memory at address in workspace register 5. CRU line 16 goes into bit 15, CRU line 31 into bit 0.
LI	R12,>28	Place the value 28_{16} in the CRU base register. This value corresponds to CRU line 20.
LDCR	@CHAR,8	Place the eight bits of a byte of memory at location CHAR on CRU lines 20 through 27. Bit 7 goes to CRU line 20, and bit 0 to CRU line 27.

3.6.8 CODING OF IMMEDIATE ADDRESSES. The Model 990 Computer instruction set includes seven instructions that use immediate addresses. The assembler places the immediate address in the word following the instruction word. The immediate address is used by the instruction as an operand. An immediate address (operand) may be coded as an integer, a symbol, or an expression. The following examples show coding of immediate operands:

LI	R4,>AD45	Place immediate value $AD45_{16}$ into workspace register 4.
LWPI	WS2	Place immediate value WS2 into WP register.
LIMI	15	Place immediate value 15 into interrupt mask, enabling all interrupt levels.
AI	6,-7	Add immediate value, -7, to the contents of workspace register 6, and place the result in workspace register 6.
CI	5,BUFF	Compare the contents of workspace register 5 to an immediate value, the value of label BUFF.
ANDI	R7,>0FF0	Perform an AND operation on the contents of workspace register 7 and an immediate value, $0FF0_{16}$, and place the result in workspace register 7.

3.7 EXAMPLE PROGRAM

A simple program has been prepared to show the coding and assembling of a program on the Model 990 Computer. The program is organized as a data division and a procedure assembled as a single module. A ten-digit account number is placed in a buffer called NUMBER as the program is assembled. The program computes a check digit for the account number exclusive of the least significant digit and replaces the least significant digit with the computed check digit. The check digit is the least significant digit of the sum of the second, fourth, sixth, and eighth digits plus the product of the sum of the remaining digits times three. This is one of several check digit algorithms used in business applications to promote accuracy.

The example does not include input and output programming because this is ordinarily done by the executive under which the program is executed. Refer to the user's guide for a specific executive for I/O programming information.

3.7.1 CODING THE SOURCE PROGRAM. Before beginning to code a program, the programmer should thoroughly define the problem. Then he should draw a flowchart of the desired solution to the problem. After checking the flowchart for accuracy, he is ready to begin writing the code. The source statements may be written on coding sheets as shown in figure 3-10. Include an adequate number of comment statements and use the comment fields of other source statements. Good comments make it easy to debug the program and to make any modifications required at a later time.

After the coding is complete, the data of the source statements should be transcribed onto the appropriate medium for assembly. The data may be keypunched on punched cards, or written onto a cassette using the 733 ASR Data Terminal. The data may also be entered directly from the keyboard of a 733 ASR Data Terminal or a CRT Display Terminal.

3.7.2 ASSEMBLING THE SOURCE CODE. Refer to the user's guide for a specific executive for information on loading and executing the assembler under that executive, or to the *Model 990 Computer Cross Support System User's Guide* for information on executing the Cross Assembler. Place the source code in the input device and execute the assembler. The assembler should print a listing similar to that shown in figure 3-11.

The data division consists of a three-word block at location TST, workspace WS, and a buffer at location NUMBER. Six workspace registers are initialized with constants required in the program. The block at location TST is typical of the interface required by the executives for the Model 990 Computer. Refer to the specific executive user's guide for information. The first BSS directive reserves 8 bytes at location WS for workspace registers 0 through 3. The next statement is a DATA directive that places the address of buffer NUMBER in the word that becomes workspace register 4. Similarly, the next DATA statement places a mask constant, $F0F0_{16}$ in the succeeding word for workspace register 5. Next, another DATA statement places constant 3030_{16} in the next word, for workspace register 6. The next statement is a DATA directive that places the address corresponding to label CORR in a word, for workspace register 7. Two more DATA directives place constants $0A00_{16}$ and 0600_{16} in the next two words, for workspace registers 8 and 9, respectively. The DATA directives are followed by a BSS directive that reserves 12 bytes for workspace registers 10 through 15, completing the workspace. The last statement in the Data Division is a DATA statement that initializes buffer NUMBER with the account number to be processed. The digits of the account number are represented as ASCII characters, two per word. The even-numbered bytes of buffer NUMBER contain the first, third, fifth, seventh, and ninth digits of the account number; i.e., the odd-numbered digits, as they are numbered in the algorithm for the check digit.

TI 990 ASSEMBLY CODING FORM

LABEL (1–6)	OPER (8–11)	OPERAND (13–25)	COMMENTS (26–60)
	TITL	'SAMPLE PROGRAM'	
	IDT	'SPLE1'	
*			
* DATA	DIVISION		
*			
TST	DATA	WS,PC,>F	EXECUTIVE INTERFACE
WS	BSS	8	R0 - R3
	DATA	NUMBER	R4
	DATA	>FOFO	R5
	DATA	>3030	R6
	DATA	CORR	R7
	DATA	>0A00	R8
	DATA	>0600	R9
WS1	BSS	12	R10 - R15
NUMBER	DATA	'92','47','62','81','95'	
*			
* PROCEDURE	-	COMPUTES CHECK DIGIT FOR ACCOUNT NUMBER	
*			
PC	MOV	R4,R1	INITIALIZE ADDRESS
	LI	R2,5	INITIALIZE COUNT
L1	SZC	R5,*R1+	CLEAR MS BITS OF NUMBERS
	DEC	R2	DECREMENT COUNT
	JNE	L1	REPEAT FOR EACH WORD
	CLR	R3	CLEAR SUM
	MOV	R4,R1	INITIALIZE ADDRESS
	LI	R2,5	INITIALIZE COUNT

PROGRAM | PROGRAMMED BY | CHARGE | PAGE | OF

(A)132132 (1/3)

Figure 3-10. Example Program (Sheet 1 of 3)

TI 990 ASSEMBLY CODING FORM

LABEL	OPER	OPERAND	COMMENTS
L2	AB	*R1,R3	ADD ODD DIGITS
	BL	*R7	BCD CORRECTION
	INCT	R1	INCREMENT ADDRESS
	DEC	R2	DECREMENT COUNT
	JNE	L2	REPEAT FOR ODD DIGITS
	MOV	R3,R10	MOVE SUN TO R10
	SLA	R3,1	MULTIPLY SUM BY 2
	BL	*R7	BCD CORRECTION
	A	R10,R3	ADD SUM (SUM X 3)
	BL	*R7	BCD CORRECTION
	MOV	R4,R1	INITIALIZE ADDRESS
	INC	R1	FOR ODD BYTE
	LI	R2,4	INITIALIZE COUNT
L3	AB	*R1,R3	ADD EVEN DIGITS
	BL	*R7	BCD CORRECTION
	INCT	R1	INCREMENT ADDRESS
	DEC	R2	DECREMENT COUNT
	JNE	L3	REPEAT FOR EVEN DIGITS
	S2CB	R5,R3	STRIP OFF LEFT BITS
	MOVB	R3,@NUMBER+9	INSERT CHECK DIGIT
	MOV	R4,R1	INITIALIZE ADDRESS
	LI	R2,5	INITIALIZE COUNT
L4	SOC	R6,*R1+	INSERT ZONE BITS
	DEC	R2	DECREMENT COUNT
	JNE	L4	REPEAT FOR ALL WORDS
	XOP	@TERM,15	EXECUTIVE INTERFACE FOR EXIT

PROGRAM | PROGRAMMED BY | CHARGE | PAGE | OF

(A)132132 (2/3)

Figure 3-10. Example Program (Sheet 2 of 3)

TI 990 ASSEMBLY CODING FORM

LABEL (1–6)	OPER (8–11)	OPERAND (13–25)	COMMENTS (26–60)
CORR	C	R3,R8	VALID BCD?
	JLT	OK	JUMP IF SO
	A	R ,R3	CORRECT INVALID BCD
	SZCB	R5,R3	STRIP OFF MS BITS
OK	RT		
	END		

PROGRAM	PROGRAMMED BY	CHARGE	PAGE	OF

(A)132132 (3/3)

Figure 3-10. Example Program (Sheet 3 of 3)

```
SAMPLE PROGRAM                                          PAGE 0001

0002                        IDT   'SPLF1'
0003                  *
0004                  *  DATA DIVISION
0005                  *
0006  0000  0000'    TST   DATA  WS,PC,>F    EXECUTIVE INTERFACE
      0002  0032'
      0004  000F
0007  0006          WS    BSS   8           R0 - R3
0008  000E  0026'        DATA  NUMBER      R4
0009  0010  F0F0        DATA  >F0F0       R5
0010  0012  3030        DATA  >3030       R6
0011  0014  0082'        DATA  CORR        R7
0012  0016  0A00        DATA  >0A00       R8
0013  0018  0000        DATA  >0000       R9
0014  001A          WS1   BSS   12          R10 - R15
0015  0026  3932    NUMBER DATA  '92','47','62','81','95'
      0028  3437
      002A  3632
      002C  3831
      002E  3935
0034                  *
0035                  *  PROCEDURE - COMPUTES CHECK DIGIT FOR ACCOUNT NUMBER
0036                  *
0037  0030  C044    PC    MOV   R4,R1        INITIALIZE ADDRESS
0038  0032  0202          LI    R2,5        INITIALIZE COUNT
      0034  0005
0039  0036  4C45    L1    SZC   R5,*R1+     CLEAR MS BITS OF NUMBERS
0040  0038  0602          DEC   R2          DECREMENT COUNT
0041  003A  16FD          JNE   L1          REPEAT FOR EACH WORD
0042  003C  04C3          CLR   R3          CLEAR SUM
0043  003E  C044          MOV   R4,R1       INITIALIZE ADDRESS
0044  0040  0202          LI    R2,5        INITIALIZE COUNT
      0042  0005
0045  0044  A0D1    L2    AB    *R1,R3      ADD ODD DIGITS
0046  0046  0697          BL    *R7         BCD CORRECTION
0047  0048  05C1          INCT  R1          INCREMENT ADDRESS
0048  004A  0602          DEC   R2          DECREMENT COUNT
0049  004C  16F8          JNE   L2          REPEAT FOR ODD DIGITS
0050  004E  C283          MOV   R3,R10      MOVE SUM TO R10
0051  0050  0A13          SLA   R3,1        MULTIPLY SUM BY 2
0052  0052  0697          BL    *R7         BCD CORRECTION
0053  0054  A0CA          A     R10,R3      ADD SUM (SUM X 3)
0054  0056  0697          BL    *R7         BCD CORRECTION
0055  0058  C044          MOV   R4,R1       INITIALIZE ADDRESS
0056  005A  0581          INC   R1            FOR ODD BYTE
0057  005C  0202          LI    R2,4        INITIALIZE COUNT
      005E  0004
0058  0060  A0D1    L3    AB    *R1,R3      ADD EVEN DIGITS
0059  0062  0697          BL    *R7         BCD CORRECTION
0060  0064  05C1          INCT  R1          INCREMENT ADDRESS
0061  0066  0602          DEC   R2          DECREMENT COUNT
0062  0068  16F8          JNE   L3          REPEAT FOR EVEN DIGITS
0063  006A  50C5          SZCB  R5,R3       STRIP OFF LEFT BITS
0064  006C  D803          MOVB  R3,@NUMBER+9  INSERT CHECK DIGIT
      006E  002F'
0065  0070  C044          MOV   R4,R1       INITIALIZE ADDRESS
0066  0072  0202          LI    R2,5        INITIALIZE COUNT
      0074  0005
0067  0076  EC46    L4    SOC   R6,*R1+      INSERT ZONE BITS
0068  0078  0602          DEC   R2          DECREMENT COUNT
0069  007A  16FD          JNE   L4          REPEAT FOR ALL WORDS
0070  007C  2FE0          XOP   @TERM,15    EXECUTIVE INTERFACE FOR EXIT
      007E  0030'
0071  0080  8203    CORR  C     R3,R8       VALID BCD?
0072  0082  1102          JLT   OK          JUMP IF SO
0073  0084  A0C9          A     R9,R3       CORRECT INVALID BCD
0074  0086  50C5          SZCB  R5,R3       STRIP OFF MS BITS
0075  0088  0458    OK    RT
0076                  END
0000 ERS

(B)132133
```

Figure 3-11. Example Program Listing

The procedure begins with a MOV instruction that copies the addresss of buffer NUMBER in workspace register 4 into workspace register 1, where it is used to address the words of the buffer. The next instruction, a LI instruction places 5 in workspace register 2 as the word count. The next instruction, at location L1, is an SZC instruction that masks off the bits in the word at the address in workspace register 1 according to the mask in workspace register 5, and increments workspace register 1 by two. Since workspace register 1 contains the address of the first word in buffer NUMBER, and workspace register 5 contains $F0F0_{16}$, the effect of this instruction is to clear the four most significant bits of the characters in the buffer to zero, leaving the Binary Coded Decimal (BCD) values of the digits in the buffer. The next instruction, a DEC instruction, decrements the count in workspace register 2 and compares the remainder to zero. This instruction is followed by a JNE instruction that causes the instruction at location LI to be repeated for each of the five words of the buffer.

The next three instructions initialize variables for the summation of the odd-numbered digits of the account number. The CLR instruction zeros workspace register 3 in which the sum is to be accumulated. The MOV instruction re-initializes workspace register 1 with the buffer address, and the LI instruction re-initializes workspace register 2 with the word count of 5. The instruction at location L2, an AB instruction, adds the first character BCD value (at the address in workspace register 1) to the contents of workspace register 3, and places the sum in workspace register 3. The next instruction, a BL instruction, branches to the address in workspace register 7, the address of location CORR. This instruction transfers control to a subroutine that tests the sum in workspace register 3 to determine if it is a valid BCD value, and corrects the sum to the BCD equivalent when it is not a valid BCD value. The subroutine is described in a subsequent paragraph. It returns control to the instruction following the BL instruction with a valid BCD value in workspace register 3. The next instruction, an INCT instruction, increments the address in workspace register 1 to the next word in the buffer. Next, a DEC instruction reduces the word count in workspace register 2, and tests for a zero count. The last instruction in the loop, a JNE instruction, jumps to the instruction at location L2 until all five words of the buffer have been processed.

The next five instructions multiply the sum in workspace register 3 by 3, maintaining a valid BCD value. The first instruction, a MOV instruction, copies the sum from workspace register 3 into workspace register 10. The next instruction, a SLA instruction, shifts the contents of workspace register 3 to the left one bit position. This in effect multiplies the sum by two. The SLA instruction is followed by a branch to the BCD correction subroutine, and an A instruction. The A instruction adds the contents of workspace register 10 to the contents of workspace register 3, and places the sum in workspace register 3. This instruction is also followed by a branch to location CORR. When control returns following this branch, the valid BCD product of the sum multiplied by 3 is in workspace register 3.

The next three instructions initialize variables for adding the sum of the remaining digits to the value in workspace register 3. The MOV instruction re-initializes the address in workspace register 1, and the INC instruction adds one to the address. The effect of the INC instruction is to address the odd (least significant) bytes of the words in the buffer, which contain the even-numbered digits of the account number. The third instruction, an LI instruction, initializes the word count to four. Only four of the even-numbered digits of the account number are involved in computing the check digit. The instruction at location L3, an AB instruction, adds the contents of the byte at the address in workspace register 1 to the sum in workspace register 3, and places the new sum in register 3. The next instruction, a BL instruction, branches to the BCD correction subroutine, which performs correction, if needed, to provide a valid BCD value in workspace register 3. The BL instruction is followed by an INCT instruction that increments the address in workspace register 1 to the next word of the buffer. The next instruction is a DEC instruction, that decrements the word count in workspace register 2, and compares the result to zero. The next instruction, a JNE instruction, jumps to location L3 until all four digits have been added to the sum.

The next two instructions place the computed check digit in the buffer. The first, an SZCB instruction, sets the bits of the check digit in workspace register 3 that correspond to the one bits in the most significant byte of workspace register 5. Workspace register 5 contains $F0F0_{16}$, so the result of the SZCB instruction is to strip off the most significant four bits of the character that contains the check digit. The MOVB instruction copies the check digit in workspace 3 into the high order byte of buffer NUMBER.

The next two instructions initialize variables for restoring the BCD values in buffer NUMBER to ASCII characters. The first of these instructions, a MOV instruction, re-initializes the address in workspace register 1, and the second, an LI instruction, initializes the word count to five. The instruction at location L4, a SOC instruction, sets ones in the word at the address in workspace register 1 corresponding to the ones in workspace register 6, and increments the address in workspace register 1. Workspace register 6 contains 3030_{16}, so the result of this instruction is to place a hexadecimal 3 ahead of the BCD value in each byte of the word of the buffer. This converts the BCD values to the equivalent ASCII characters. The SOC instruction is followed by a DEC instruction that decrements the word count and compares the result to zero. It is followed by a JNE instruction that jumps to the instruction at location L4 until all words in the buffer have been converted to ASCII.

The next instruction, an XOP instruction, performs extended operation 15 using a value at location TERM. This instruction is typical of the interface with the executives for the Model 990 Computers to return control to the executive. Refer to the user's guide for a specific executive for more information.

The subroutine at location CORR tests the contents of workspace register 3 to determine whether it contains a valid BCD value, and to correct the contents to the equivalent BCD value. A valid BCD number is in the range of 0 through 9. When computation results in a value greater than 9, the least significant digit may be

corrected by adding 6 to the value, and clearing the four most significant bits to zero. The first instruction of the subroutine, a C instruction, compares the contents of workspace register 3 to the contents of workspace register 8. Workspace register 8 contains $0A00_{16}$. The next instruction, a JLT instruction, jumps to the instruction at location OK when the result of the comparison is less than. If the contents of workspace register 3 are not less than $0A00_{16}$, correction is required. The A instruction adds the contents of workspace register 9 to the contents of workspace register 3, and places the result in workspace register 3. The next instruction is a SZCB instruction that sets bits of the contents of the most significant byte of workspace register 3 to one that correspond to the one bits in the most significant byte of workspace register 5. Workspace register 5 contains $F0F0_{16}$, so the instruction clears the four most significant bits of the contents of workspace register 3. The instruction at label OK is an RT pseudo-instruction that returns control to the calling program at the address stored in workspace register 11. The last statement is an END directive that terminates the source program.

3.7.3 EXECUTING THE PROGRAM. After assembling the program, load the object code into memory and debug the program. Refer to the user's guide for the specific executive under which the program is to be executed for loading instructions. Different debug programs are available with different executives, so which of these is used also depends upon the applicable executive. When a program is assembled on the Cross Assembler, it may be debugged using the TMS9900 Simulator as described in the **Model 990 Computer Cross Support System User's Guide.** The correct contents of buffer NUMBER following execution of the program consists of the following ASCII characters:

9247628190