TEXAS INSTRUMENTS

Improving Man's Effectiveness Through Electronics

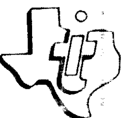
Model 980 Computer

Assembly Language

Programmer's Reference Manual

MANUAL NO. 943013-9701
ORIGINAL ISSUE 15 DECEMBER 1974
REVISED AND REISSUED 1 MARCH 1975
INCLUDES
CHANGE 1...............1 MARCH 1976

Digital Systems Division



© Texas Instruments Incorporated 1976
All Rights Reserved

The information and/or drawings set forth in this document and all rights in and to inventions disclosed herein and patents which might be granted thereon disclosing or employing the materials, methods, techniques or apparatus described herein are the exclusive property of Texas Instruments Incorporated.

No disclosure of the information or drawings shall be made to any other person or organization without the prior consent of Texas Instruments Incorporated.

LIST OF EFFECTIVE PAGES

INSERT LATEST CHANGED PAGES DESTROY SUPERSEDED PAGES

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

Model 980 Computer Assembly Language Programmer's Reference Manual (943013-9701)

Original Issue . . . . 15 December 1974
Revised and Reissued . . . . 1 March 1975 (ECN 388070)
Change 1 . . . . 1 March 1976 (ECN 407116)

Total number of pages in this publication is 166 consisting of the following:

PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.
Cover	1	4-20	1		
Effective Pages	1	4-21 - 4-22	0		
iii - v	0	Appendix A Divider	0		
vi - viii	1	A-1 - A-4	0		
1-1 - 1-2	1	Appendix B Divider	0		
2-1 - 2-8	0	B-1 - B-6	0		
3-1 - 3-7	0	Appendix C Divider	0		
3-8	1	C-1 - C-2	0		
3-8A/3-8B	1	User's Response	0		
3-9 - 3-55	0	Business Reply	1		
3-56	1	Cover Blank	0		
3-57 - 3-92	0	Cover	0		
3-93	1				
3-94 - 3-96	0				
4-1	0				
4-2	1				
4-2A/4-2B	1				
4-3	1				
4-4 - 4-5	0				
4-6 - 4-8	1				
4-8A/4-8B	1				
4-9 - 4-10	1				
4-11	0				
4-12	1				
4-12A/4-12B	1				
4-13 - 4-19	0				

TABLE OF CONTENTS

Paragraph	Title	Page
	SECTION I. GENERAL INFORMATION	
1.1	Scope of Manual	1-1
1.2	References	1-1
	SECTION II. HARDWARE FEATURES	
2.1	General	2-1
2.2	Computer Organization	2-1
2.3	Data and Instruction Formats	2-3
2.4	Register Organization	2-4
2.5	Memory Protect/Privileged Instruction Feature	2-6
2.6	Program Relocation Feature	2-7
2.7	Priority Interrupt Feature	2-7
	SECTION III. MACHINE INSTRUCTIONS AND CODING CONVENTIONS	
3.1	General	3-1
3.1.1	Instruction Descriptions	3-1
3.1.2	Addressing Modes	3-5
3.1.3	Extended Format Addressing	3-8
3.2	Load Instructions	3-8
3.2.1	Double Load Registers A and E (DLD)	3-8
3.2.2	Load Register A (LDA)	3-9
3.2.3	Load Register E (LDE)	3-10
3.2.4	Load Register M (LDM)	3-10
3.2.5	Load Register X (LDX)	3-11
3.2.6	Load Register File (LRF)	3-12
3.3	Store Instructions	3-13
3.3.1	Double Store Registers A and E (DST)	3-13
3.3.2	Store Register File (SRF)	3-13
3.3.3	Store Register A (STA)	3-14
3.3.4	Store Register E (STE)	3-15
3.3.5	Store Register X (STX)	3-16
3.4	Branch Instructions	3-16
3.4.1	Branch on Incremented Index (BIX)	3-17
3.4.2	Branch and Link (BRL)	3-18
3.4.3	Branch Unconditional (BRU)	3-19
3.4.4	Idle (IDL)	3-20

TABLE OF CONTENTS (Continued)

Paragraph	Title	Page
3.4.5	Load Status Block and Branch (LSB)	3-21
3.4.6	Load Status Block, Reset Interrupt, and Branch (LSR)	3-22
3.4.7	Store Status Block and Branch (SSB)	3-23
3.5	Arithmetic Instructions	3-24
3.5.1	Add to Register A (ADD)	3-25
3.5.2	Double Length Add (DAD)	3-25
3.5.3	Divide (DIV)	3-26
3.5.4	Double Length Subtract (DSB)	3-27
3.5.5	Increment Memory by One (IMO)	3-28
3.5.6	Multiply (MPY)	3-29
3.5.7	Register Add (RAD)	3-30
3.5.8	Register Complement (RCO)	3-31
3.5.9	Register Decrement (RDE)	3-31
3.5.10	Register Increment (RIN)	3-32
3.5.11	Register Invert (RIV)	3-33
3.5.12	Register Subtract (RSU)	3-34
3.5.13	Subtract from Register A (SUB)	3-35
3.6	Compare Instructions	3-36
3.6.1	Compare Logical Character String (CLC)	3-36
3.6.2	Compare Algebraic (CPA)	3-38
3.6.3	Compare Logical (CPL)	3-39
3.6.4	Register Compare Algebraic (RCA)	3-40
3.6.5	Register Compare Logical (RCL)	3-41
3.7	Skip Instructions	3-42
3.7.1	Decrement Memory and Test (DMT)	3-42
3.7.2	Skip on Equal (SEQ)	3-43
3.7.3	Skip on Even (SEV)	3-44
3.7.4	Skip on Greater than or Equal (SGE)	3-44
3.7.5	Skip on Greater Than (SGT)	3-45
3.7.6	Skip on Less Than or Equal (SLE)	3-46
3.7.7	Skip on Less Than (SLT)	3-47
3.7.8	Skip on Minus (SMI)	3-48
3.7.9	Skip on No Carry (SNC)	3-48
3.7.10	Skip on Not Equal (SNE)	3-49
3.7.11	Skip on Not All Ones (SNO)	3-50
3.7.12	Skip on No Overflow (SNV)	3-51
3.7.13	Skip on Not All Zeros (SNZ)	3-52
3.7.14	Skip on Carry (SOC)	3-52
3.7.15	Skip on Odd (SOD)	3-53
3.7.16	Skip on All Ones (SOO)	3-54

TABLE OF CONTENTS (Continued)

Paragraph	Title	Page
3.7.17	Skip on Overflow (SOV)	3-55
3.7.18	Skip on Plus (SPL)	3-55
3.7.19	Skip on Sense Switch Equal (SSE)	3-56
3.7.20	Skip on Sense Switch not Equal (SSN)	3-57
3.7.21	Skip on Zero (SZE)	3-58
3.8	Shift Instructions	3-58
3.8.1	Arithmetic Left Shift Register A (ALA)	3-59
3.8.2	Arithmetic Left Shift Double (ALD)	3-59
3.8.3	Arithmetic Right Shift Register A (ARA)	3-60
3.8.4	Arithmetic Right Shift Double (ARD)	3-61
3.8.5	Circular Left Shift Double (CLD)	3-62
3.8.6	Circular Right Shift Register A (CRA)	3-62
3.8.7	Circular Right Shift Register B (CRB)	3-63
3.8.8	Circular Right Shift Double (CRD)	3-64
3.8.9	Circular Right Shift Register E (CRE)	3-64
3.8.10	Circular Right Shift Register L (CRL)	3-65
3.8.11	Circular Right Shift Register M (CRM)	3-66
3.8.12	Circular Right Shift Register S (CRS)	3-66
3.8.13	Circular Right Shift Register X (CRX)	3-67
3.8.14	Logical Left Shift Register A (LLA)	3-68
3.8.15	Logical Left Shift Double (LLD)	3-68
3.8.16	Logical Right Shift Register A (LRA)	3-69
3.8.17	Logical Right Shift Double (LRD)	3-70
3.8.18	Left Test for Ones in Register A (LTO)	3-70
3.8.19	Left Test for Zeros in Register A (LTZ)	3-71
3.8.20	Normalize (NRM)	3-72
3.8.21	Right Test for Ones in Register A (RTO)	3-73
3.8.22	Right Test for Zeros in Register A (RTZ)	3-74
3.9	Logical Instructions	3-75
3.9.1	Logical AND with Register A (AND)	3-75
3.9.2	Logical OR with Register A (IOR)	3-76
3.9.3	Register AND (RAN)	3-77
3.9.4	Register Exclusive OR (REO)	3-77
3.9.5	Register OR (ROR)	3-78
3.10	Bit Manipulation Instructions	3-79
3.10.1	Set Register A Bit to One (SABO)	3-79
3.10.2	Set Register A Bit to Zero (SABZ)	3-80
3.10.3	Set Memory Bit to One (SMBO)	3-81
3.10.4	Set Memory Bit to Zero (SMBZ)	3-82
3.10.5	Test Register A Bit for One (TABO)	3-83
3.10.6	Test Register A Bit for Zero (TABZ)	3-83

TABLE OF CONTENTS (Continued)

Paragraph	Title	Page
3.10.7	Test Memory Bit for One (TMBO)	3-84
3.10.8	Test Memory Bit for Zero (TMBZ)	3-85
3.11	Move Instructions	3-86
3.11.1	Move Character String (MVC)	3-86
3.11.2	Register Exchange (REX)	3-88
3.11.3	Register Move (RMO)	3-88
3.12	Input/Output Instructions	3-89
3.12.1	Auxiliary Processor Initiate (API)	3-89
3.12.2	Automatic Transfer Instruction (ATI)	3-91
3.12.3	Read Direct Single (RDS)	3-92
3.12.4	Write Direct Single (WDS)	3-94

SECTION IV. ASSEMBLER CHARACTERISTICS AND DIRECTIVES

Paragraph	Title	Page
4.1	General	4-1
4.2	Symbolic Assembly Program (SAP)	4-1
4.2.1	SAP Coding Line Format	4-5
4.2.2	Segmented Source Programs	4-7
4.2.3	SAP Object Format	4-8
4.2.4	SAP Error Messages	4-8
4.3	Assembler Directives	4-8
4.3.1	Block Ending Symbol (BES)	4-9
4.3.2	Base Register Reset (BRR)	4-10
4.3.3	Base Register Set (BRS)	4-10
4.3.4	Block Starting Symbol (BSS)	4-11
4.3.5	Generate Byte Address (BYTE)	4-12
4.3.6	Blank Common (COMM)	4-12
4.3.7	Generate Word Address or Data (DATA)	4-13
4.3.8	Define Entry Point Symbol (DEF)	4-13
4.3.9	End of Source (END)	4-15
4.3.10	Equate (EQU)	4-16
4.3.11	Flag Bit Address (FLAG)	4-16
4.3.12	Format a New Instruction (FRM)	4-17
4.3.13	Page Heading (HED)	4-18
4.3.14	Object Identifier (IDT)	4-18
4.3.15	Conditional Assembly (IF)	4-18
4.3.16	Start Listing (LIS)	4-19
4.3.17	Operation Define (OPD)	4-19
4.3.18	Origin (ORG)	4-21
4.3.19	Page Eject (PEJ)	4-21
4.3.20	Referenced External Symbols (REF)	4-22
4.3.21	Stop Listing (UNL)	4-22

APPENDIXES

Appendix	Title	Page
A	Instruction Execution Times	A-1
B	Alphabetical and Hexadecimal Instruction Indexes	B-1
C	Illegal Instruction Operation Codes	C-1

LIST OF ILLUSTRATIONS

Figure	Title	Page
2-1	Model 980 Computer Block Diagram	2-1
3-1	Register-Memory Instruction Fields	3-5
4-1	Source Coded Main Program	4-1
4-2	Source Coded Subroutine	4-2A
4-3	Assembled Main Program	4-3
4-4	Assembled Subroutine	4-4
4-5	Example of BYTE and DATA Usage	4-14

LIST OF TABLES

Table	Title	Page
1-1	Related Manuals	1-1
2-1	Model 980 Computer Characteristics	2-2
2-2	Model 980 Computer Addressable Registers	2-4
2-3	Status Register Bit Functions	2-5
2-4	Model 980 Computer Interrupts	2-8
3-1	Model 980 Computer Machine Instructions by Functional Group	3-1
3-2	Assembly Language Coding Format and Instruction Execution Symbols	3-5
3-3	Register-Memory Instruction Addressing Modes and Coding Conventions	3-6
4-1	SAP Error Messages	4-8A
4-2	Model 980 Computer SAP Assembler Directives	4-9

SECTION I
GENERAL INFORMATION

1.1 SCOPE OF MANUAL

This is one of two manuals covering the Model 980 Computer assembly language. This manual describes all of the Model 980 Computer machine instructions and the associated symbolic assembly language coding conventions. Beginning with Section II, an overview of the Model 980 Computer is presented with specific information on the hardware features that affect assembly language. Section III presents the machine instructions and the symbolic coding conventions. Section IV follows with a general description of the Symbolic Assembly Program (SAP) and a list of assembler directives. Included in Section IV are sample assembly listings produced by SAP. The appendixes at the rear of the manual contain instruction execution times, an alphabetical and numerical listing of instruction operation codes, and a table of illegal operation codes.

1.2 REFERENCES

The second of the two manuals covering the Model 980 Computer assembly language is Model 980 Computer Assembly Language Input/Output. It provides the information necessary to program input/output devices available with the 980 at the assembly language level. The Model 980 Computer Basic System Use and Operation manual or the DX980 GPOS Programmer's Guide should be referenced for information on how to assemble, load, and execute an assembly language program. The related software manuals and their respective manual numbers are listed in table 1-1.

Table 1-1. Related Manuals

Manual	Manual No.
Model 980 Computer Assembly Language Input/Output	961961-9734
Model 980 Computer Basic System Use and Operation	961961-9710
Model 980 Computer Programming Card	943000-9701
DX980 General Purpose Operating System Programmer's Guide	943005-9701

SECTION II

HARDWARE FEATURES

2.1 GENERAL

This section contains a brief block diagram discussion of the computer, a table of computer characteristics, and a list of programmable registers. Included is a bit-by-bit breakdown of the status register.

2.2 COMPUTER ORGANIZATION

The computer is functionally organized into a central processing unit (CPU), a memory, an input/output (I/O) unit, and a power supply. Figure 2-1 shows a block diagram of the basic system. The Direct Memory Access Channel (DMAC) is an I/O channel used for peripheral devices having a relatively fast

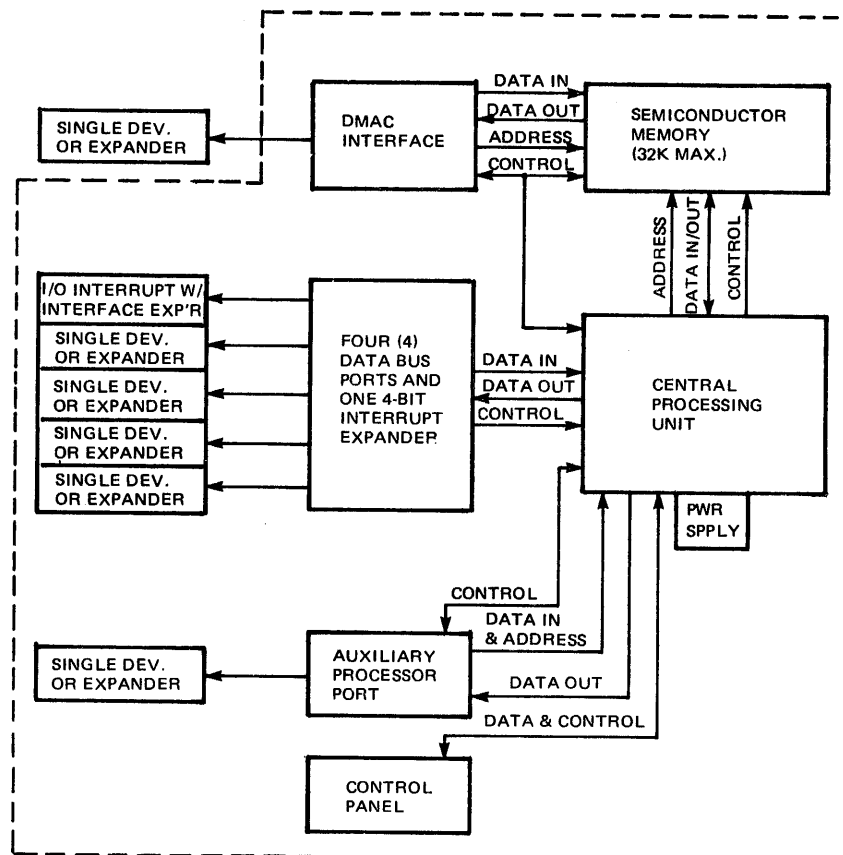


Figure 2-1. Model 980 Computer Block Diagram

rate of data transfer. The Data Bus is an I/O channel used for peripheral devices having a relatively slow rate of data transfer. An auxiliary processor (AP) is used to add to the standard 980 instruction set. For example, floating point arithmetic may be added or the instructions for another computer may be emulated. Expansion of the DMAC, Data Bus, and AP ports may be accomplished by using the optional twelve (12) connector chassis within the Model 980 mainframe and/or an expansion chassis external to the mainframe. Table 2-1 lists some of the more important characteristics of the computer.

Tabel 2-1. Model 980 Computer Characteristics

Organization

- Parallel operation
- Single level indirect addressing
- Two's complement arithmetic
- Eight addressable registers, plus status register
- Bipolar ROM control for CPU

Memory

- Dynamic MOS/LSI semiconductor array memory
- 16-bit word length plus even parity (980A); 16-bit word length plus 6-bit error correction/detection code (980B)
- Capacity, in 4096-word increments (980A); in 8192-word increments (980B)

  4096 words minimum (980A); 8192 words minimum (980B)
  65536 words maximum (980A and 980B)
  32768 in CPU chassis, 32768 external (980A); 65536 in CPU chassis (980B)

- All of memory can be directly addressed
- Power failure protection
- 750 nanosecond read or write cycle
- 500 nanosecond memory access

Input/Output

- One direct memory access channel (DMAC) port, expandable to eight

  Single word parallel transfer
  One million words per second burst rate

Table 2-1. Model 980 Computer Characteristics (Continued)

- A processor-controlled data bus with 4 ports, expandable to 256 ports

  One 4-bit interrupt expander

  16-bit parallel transfer

- Three priority interrupts

  Vectored interrupts (highest priority)
  DMAC interrupts
  Data bus interrupts (lowest priority)

Instruction set

- 99 basic instructions (covered in Section III)

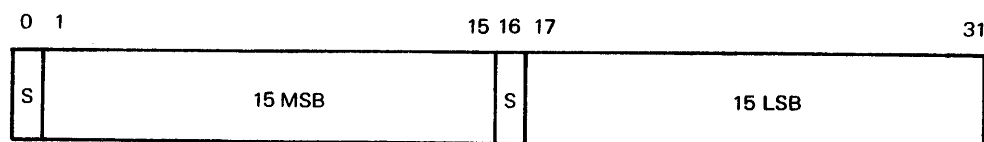
Other features

- Memory protect/privileged instruction feature (standard)
- Vectored (priority) interrupt option, up to 32 hardware vectored interrupts (optional)
- Auxiliary Processor option (optional)
- Hardware bootstrap loader (standard)
- Internal expansion chassis for DMAC, data bus, and auxiliary processors (optional)
- Internal battery for maintaining memory contents when power is off (optional)

2.3 DATA AND INSTRUCTION FORMATS

Both the data and instruction words are 16 bits long. The bit positions within a word are numbered 0 (most significant bit) through 15 (least significant bit). Data is represented in binary two's complement form with bit 0 indicating the algebraic sign. A zero in the first bit indicates a positive sign. The range of integers representable in one 16-bit word is from -2^{15} to $+2^{15}$ -1.

Double length operands such as products from multiplication, dividends for divides, and quantities for double-length arithmetic shifts have the following format:



Input, output, and status register related instructions are 32 bits long and occupy two consecutive 16-bit words. The register-to-memory instructions may be 16, 32, or 48 bits long.

2.4 REGISTER ORGANIZATION

Eight 16-bit registers are directly addressable via the instruction formats involving registers. These registers with their respective address, designation, and function are listed in table 2-2.

Table 2-2. Model 980 Computer Addressable Registers

Register Address	Designation	Function
0	A	Primary arithmetic register.
1	E	Secondary (extension) arithmetic register.
2	X	Index register for operand address modification.
3	M	Maintenance register for temporary storage.
4	S	Storage register for temporary storage.
5	L	Link register to hold return address for subroutine linkage.
6	B	Base register to hold base address for operands.
7	PC	Program counter to hold the address of the next instruction.

In addition to these eight registers, the status register may be directly affected by the instruction set. The status register is used to hold the present condition of the computer and to enable or disable interrupts. The status register together with the program counter constitutes the "status block". The functions of the status register bits are listed in table 2-3.

Table 2-3. Status Register Bit Functions

Bits	Function
0, 1	Compare Indicators - Indicate the result of the last compare operation. 00 - less than 01 - equal to 10 - greater than 11 - unused bit setting
2	Overflow Indicator - Turned on or off by those instructions that may result in a number that is outside of the range of the associated register(s).
3	Carry Indicator - Turned on or off by an add or subtract instruction that may result in a carry into the sign bit of a register.
4	Privileged Instruction and Memory Protect Interrupt Control 0 - Disabled 1 - Enabled
5	Memory Protect Address Violation - May not be set under program control. 0 - No Violation 1 - Violation
6	PIF* Instruction Violation - May not be set under program control. 0 - No Violation 1 - Violation
7	Data Bus Interrupt Control 0 - Disabled 1 - Enabled
8	Vectored Interrupt Feature 0 - Disabled 1 - Enabled

*PIF - Privileged Instruction Feature

Table 2-3. Status Register Bit Functions (Continued)

Bits	Function
9	PIF* Lower Limit Address Bias 0 - Disabled 1 - Enabled
10	Index Control 0 - Post Indexing 1 - Pre-indexing
11	Memory Parity Error Interrupt Control 0 - Disabled 1 - Enabled
12	DMAC Interrupt Control 0 - Disabled 1 - Enabled
13	Not Used
14	Memory Parity Error Indicator - May not be set under program control. 0 - No Error 1 - Error
15	Power Fail Indicator - One millisecond (980A) or 20 millisecond (980B) warning that power failure is imminent. May not be set under program control. 0 - Power Up 1 - Power Failure Imminent

*PIF - Privileged Instruction Feature

2.5 MEMORY PROTECT/PRIVILEGED INSTRUCTION FEATURE

When enabled, the memory protect/privileged instruction feature (MP/PIF) allows program execution to occur only within a specified area of memory. It also causes certain instructions to be treated as illegal. This feature may be used to protect the operating environment from destruction during execution of an undebugged program.

The system may use this feature to prevent a user program from inadvertently storing data over a system program or another user program. The

MP/PIF can also prevent program execution from proceeding beyond the region that the given program occupies in memory; thus, a program cannot inadvertently branch into the middle of another program. Finally, when the MP/PIF is enabled, a user can neither disrupt input/output activity that the system has in progress nor bring the computer to an idle.

Before enabling the MP/PIF feature, it is first necessary to load the MP/PIF lower limit and upper limit registers that define the limits within which execution will be constrained. Both registers are loaded using the WDS instruction (refer to paragraph 3.12.4) just as if the MP/PIF registers were external to the computer. Register address zero defines the lower limit and register address one defines the upper limit. These boundary locations and all memory outside of the boundaries are protected by the MP/PIF feature. The MP/PIF feature is then enabled by setting bit 4 of the status register.

2.6 PROGRAM RELOCATION FEATURE

The program relocation feature (PRF) allows a program to be loaded anywhere within the 980 memory, but to execute as though it were loaded starting at memory location zero. When used by a system program, this allows programs to be moved from one point in memory to another with no affect on the operation of the program. It also allows programs to be stored in an absolute rather than relocatable form, thus requiring less storage space.

The lower limit register used by the MP/PIF is also used by the PRF. If the system sets bit 9 of the status register at the time control is transferred to the user program, the contents of the lower limit register plus one is added into the address calculations for each memory access. For example, suppose a program is assembled as an absolute program with origin at location 0000_{16}. Also, suppose that the entry point to the program is location 0020_{16}, and that it is convenient for the system to load the program at location 1000_{16}. The system loads the program starting at 1000_{16}, places $0FFF_{16}$ in the lower limit register, and performs an LSB instruction (refer to paragraph 3.4.5) to transfer to the program. The LSB must set bit 9 of the status register and load the program counter with 0020_{16}. Note, that although the instruction executed is at 1020_{16}, the program counter contains 0020_{16}. If, for instance, a trap were to occur, the value 0020_{16} in the program counter would be saved for the return.

2.7 PRIORITY INTERRUPT FEATURE

The Model 980 Computer responds to four different types of interrupts. These interrupts, in order of priority include: internal interrupt, vectored interrupt option, DMAC interrupt, and data bus interrupt. The three lower priority interrupts are input/output interrupts, and their occurrence depends on the system hardware configuration. The internal interrupts include the detection of imminent power failure, an illegal operation code, a memory parity error, a memory protect violation, and a privileged instruction violation. When any internal or input/output interrupt occurs, computer control

traps to low order memory as listed in table 2-4, assuming the proper status register bits are set to enable the interrupt. Note that the power failure and illegal operation code interrupts cannot be masked by the status register.

Table 2-4. Model 980 Computer Interrupts

Interrupt Type	Trap Address (Hex)	Status Register Bits	
		Mask Bit	Interrupt Bit
Internal			
Power fail	0002	-	15
Illegal op-code	0002	-	- ①
Parity error	0002	11	14
MP violation	0002	4	5
PIF violation	0002	4	6
Vectored (Optional)	0008② : 0046	8	-
DMAC	0004	12	-
Data Bus	0006	7	-

NOTES:

① The illegal op-code interrupt is detected when none of the other internal interrupts cause the trap to 0002_{16}.

② The optional vectored interrupt feature may include up to 32 separate trap locations, beginning with the highest priorities at 0008_{16}, $000A_{16}$, $000C_{16}$, etc. to the lowest priority at 0046_{16}.

Programming all four types of interrupts is covered in detail in the Model 980 Computer Assembly Language Input/Output manual.

SECTION III

MACHINE INSTRUCTIONS AND CODING CONVENTIONS

3.1 GENERAL

This section describes the machine instructions and the related assembly language coding conventions for the Model 980 Computer. Table 3-1 groups the 99 instructions by function, and references a separate paragraph on each instruction for more detailed information. Appendix B contains an alphabetical and hexadecimal index to these same paragraph numbers. General coding conventions applicable to the label, operation, operand, and comment fields of the symbolic assembly language are covered in Section IV of this manual.

3.1.1 INSTRUCTION DESCRIPTIONS

Each instruction description referenced in table 3-1 contains the following information about the instruction:

- Instruction word field breakdown
- Description of instruction execution
- Status register bits affected by instruction execution
- Execution time
- Assembly language coding conventions
- Instruction example

Table 3-1. Model 980 Computer Machine Instructions by Functional Group

Mnemonic	Description	Paragraph No.
• Load Instructions		3.2
DLD	Double Load Registers A and E	3.2.1
LDA	Load Register A	3.2.2
LDE	Load Register E	3.2.3
LDM	Load Register M	3.2.4
LDX	Load Register X	3.2.5
LRF	Load Register File	3.2.6
• Store Instructions		3.3
DST	Double Store Registers A and E	3.3.1
SRF	Store Register File	3.3.2
STA	Store Register A	3.3.3
STE	Store Register E	3.3.4
STX	Store Register X	3.3.5

Table 3-1. Model 980 Computer Machine Instructions by Functional Group (Continued)

Mnemonic	Description	Paragraph No.
• Branch Instructions		3.4
BIX	Branch on Incremented Index	3.4.1
BRL	Branch and Link	3.4.2
BRU	Branch Unconditional	3.4.3
IDL	Idle	3.4.4
LSB	Load Status Block and Branch	3.4.5
LSR	Load Status Block, Reset Interrupt, and Branch	3.4.6
SSB	Store Status Block and Branch	3.4.7
• Arithmetic Instructions		3.5
ADD	Add to Register A	3.5.1
DAD	Double Length Add	3.5.2
DIV	Divide	3.5.3
DSB	Double Length Subtract	3.5.4
IMO	Increment Memory by One	3.5.5
MPY	Multiply	3.5.6
RAD	Register Add	3.5.7
RCO	Register Complement	3.5.8
RDE	Register Decrement	3.5.9
RIN	Register Increment	3.5.10
RIV	Register Invert	3.5.11
RSU	Register Subtract	3.5.12
SUB	Subtract from Register A	3.5.13
• Compare Instructions		3.6
CLC	Compare Logical Character String	3.6.1
CPA	Compare Algebraic	3.6.2
CPL	Compare Logical	3.6.3
RCA	Register Compare Algebraic	3.6.4
RCL	Register Compare Logical	3.6.5
• Skip Instructions		3.7
DMT	Decrement Memory and Test	3.7.1
SEQ	Skip on Equal	3.7.2
SEV	Skip on Even	3.7.3
SGE	Skip on Greater Than or Equal	3.7.4
SGT	Skip on Greater Than	3.7.5
SLE	Skip on Less Than or Equal	3.7.6
SLT	Skip on Less Than	3.7.7

Table 3-1. Model 980 Computer Machine Instructions by Functional Group (Continued)

Mnemonic	Description	Paragraph No.
• Skip Instructions (Continued)		
SMI	Skip on Minus	3.7.8
SNC	Skip on No Carry	3.7.9
SNE	Skip on Not Equal	3.7.10
SNO	Skip on Not All Ones	3.7.11
SNV	Skip on No Overflow	3.7.12
SNZ	Skip on Not All Zeros	3.7.13
SOC	Skip on Carry	3.7.14
SOD	Skip on Odd	3.7.15
SOO	Skip on All Ones	3.7.16
SOV	Skip on Overflow	3.7.17
SPL	Skip on Plus	3.7.18
SSE	Skip on Sense Switch Equal	3.7.19
SSN	Skip on Sense Switch Not Equal	3.7.20
SZE	Skip on Zero	3.7.21
• Shift Instructions		3.8
ALA	Arithmetic Left Shift Register A	3.8.1
ALD	Arithmetic Left Shift Double	3.8.2
ARA	Arithmetic Right Shift Register A	3.8.3
ARD	Arithmetic Right Shift Double	3.8.4
CLD	Circular Left Shift Double	3.8.5
CRA	Circular Right Shift Register A	3.8.6
CRB	Circular Right Shift Register B	3.8.7
CRD	Circular Right Shift Double	3.8.8
CRE	Circular Right Shift Register E	3.8.9
CRL	Circular Right Shift Register L	3.8.10
CRM	Circular Right Shift Register M	3.8.11
CRS	Circular Right Shift Register S	3.8.12
CRX	Circular Right Shift Register X	3.8.13
LLA	Logical Left Shift Register A	3.8.14
LLD	Logical Left Shift Double	3.8.15
LRA	Logical Right Shift Register A	3.8.16
LRD	Logical Right Shift Double	3.8.17
LTO	Left Test for Ones in Register A	3.8.18
LTZ	Left Test for Zeros in Register A	3.8.19
NRM	Normalize	3.8.20
RTO	Right Test for Ones in Register A	3.8.21
RTZ	Right Test for Zeros in Register A	3.8.22

Table 3-1. Model 980 Computer Machine Instructions by Functional Group (Continued)

Mnemonic	Description	Paragraph No.
• Logical Instructions		3.9
AND	Logical AND with Register A	3.9.1
IOR	Logical OR with Register A	3.9.2
RAN	Register AND	3.9.3
REO	Register Exclusive OR	3.9.4
ROR	Register OR	3.9.5
• Bit Manipulation Instructions		3.10
SABO	Set Register A Bit to One	3.10.1
SABZ	Set Register A Bit to Zero	3.10.2
SMBO	Set Memory Bit to One	3.10.3
SMBZ	Set Memory Bit to Zero	3.10.4
TABO	Test Register A Bit for One	3.10.5
TABZ	Test Register A Bit for Zero	3.10.6
TMBO	Test Memory Bit for One	3.10.7
TMBZ	Test Memory Bit for Zero	3.10.8
• Move Instructions		3.11
MVC	Move Character String	3.11.1
REX	Register Exchange	3.11.2
RMO	Register Move	3.11.3
• Input/Output Instructions		3.12
API	Auxiliary Processor Initiate	3.12.1
ATI	Automatic Transfer Instruction	3.12.2
RDS	Read Direct Single	3.12.3
WDS	Write Direct Single	3.12.4

The status register bits are defined in table 2-3. The symbols used in presenting the instruction assembly language coding formats and the symbols used in presenting an abbreviated form of instruction execution are listed in table 3-2. The symbols and directives used in the instruction examples are explained in Section IV.

Table 3-2. Assembly Language Coding Format and Instruction Execution Symbols

	Symbol	Definition
Instruction Execution	( )	Contents of enclosed register or address
	⟶	Replaces
Assembly Language Coding Format	*	Indirect addressing
	@	Extended format
	=	Immediate operand
	[ ]	Optional item
	Lower case alphabetic characters	User supplied item
	b̸	Required blank space (one or more)

3.1.2 ADDRESSING MODES

The computer instruction set can be broken down into a number of different format types. The addressing modes associated with all but one of the format types are straightforward, and are included in the individual instruction descriptions. The remaining instruction format type, register-memory instructions, is more involved and is described in this paragraph and referenced by the instruction descriptions when applicable.

The format of register-memory instructions is shown in figure 3-1. The addressing mode is determined by the I, X, and B fields as shown in table 3-3.

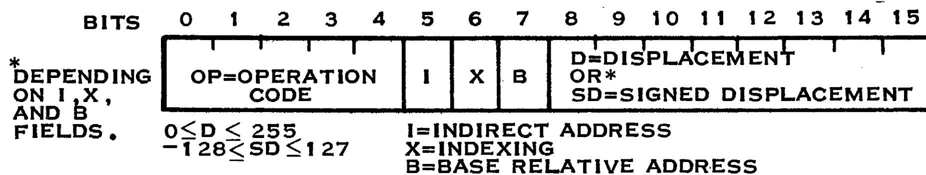


Figure 3-1. Register-Memory Instruction Fields

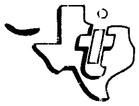


Table 3-3. Register-Memory Instruction Addressing Modes and Coding Conventions

IXB Bits	Effective Operand Address, EOA	Symbolic Coding Conventions		Addressing Mode
		Operation	Operand	
000	(PC)(1) + SD	MNU(1) @MNU @MNU	ADRS(2) =NUM(3), (5) NUM, 7(5)	PC relative
001	(B)(1) + D	MNU MNU	ADRS, 1 ADRS(4)	Base register relative
010	(PC) + (X)(1) + SD	MNU	ADRS, 2	Indexed PC relative
011	(B) + (X) + D	MNU MNU	ADRS, 3 ADRS, 2(4)	Indexed base register relative
100	((PC) + SD)	MNU MNU MNU @MNU	*ADRS *ADRS, 4 ADRS, 4 ADRS(5)	Indirect PC relative
101	((B) + D)	MNU MNU MNU MNU	*ADRS, 1 *ADRS, 5 ADRS, 5 *ADRS(4)	Indirect base register relative
110	((PC) + SD) + (X)(7) ((PC) + (X) + SD)(8)	MNU MNU MNU @MNU	*ADRS, 6 *ADRS, 2 ADRS, 6 ADRS, 2(5), (6)	Indirect, indexed, PC relative
111	Immediate value is the SD	MNU MNU	=NUM NUM, 7	Immediate

NOTES:

(1) PC - Program Counter (points to next instruction); B - Base Register; X - Index Register; MNU - Instruction Mnemonic.

(2) Symbolic name of address.

(3) Number, literal, or address.

(4) Under BRS directive.

(5) All extended format instructions are regarded as PC relative because the assembler zeros the SD field. This means the computer must add the PC to the zeroed SD to locate the extended data/address. Note that the computer increments the PC to the next location before the instruction is executed.

(6) Post-indexing, regardless of status register bit 10.

(7) Post-indexing if status register bit 10 = 0.

(8) Pre-indexing if status register bit 10 = 1.

NOTE

To fully understand table 3-3, all of paragraph 3.1.2 and 3.1.3 must be read.

In general, calculation of the Effective Operand Address (EOA) of the memory data involved in the instruction includes indirect addressing if bit I is set, indexing if bit X is set, and base relative addressing if bit B is set. If all three of these bits are set, an immediate operand is assumed by the computer. If immediate addressing is specified for a load, add, subtract, or algebraic compare instruction, the displacement field (D) is treated as an 8-bit signed quantity and bit eight is extended through bits 0 to 7 to provide a 16-bit operand. If immediate addressing is specified for a store instruction, D is treated as the EOA.

The index control bit 10 in the status register permits optional pre-indexing or post-indexing. This controls the relation of indexing to indirect addressing. If the index control bit is one, indexing precedes indirect addressing. If the index control bit is zero, indexing follows indirect addressing. If indirect addressing is not involved, the two modes are equivalent. Additional addressing capability is available with the optional memory protect/privileged instruction feature. If status register bit 9 is set, the lower limit address is added to the computer calculated address for every memory access.

Table 3-3 also lists the symbolic coding conventions available with register-memory instructions, and hence shows the transliteration process performed by the assembler in developing the I, X, and B fields. In order to translate the operand address expression of a register-memory instruction, the assembler first evaluates the expression as a 16-bit number and then modifies the expression in one of the following ways:

- For program counter relative instructions, a number one greater than the assembler location counter is subtracted.
- For base register relative instructions, the base register value or the number associated with a BRS directive (refer to Section IV of this manual) is subtracted.
- For extended format instructions (described in next paragraph), the expression remains unmodified.
- For single length immediate instructions, or base register relative instructions under the BRR directive (refer to Section VII of this manual), the expression is truncated to an eight-bit value.
- If the resulting address is unattainable under the defined conditions, a field size error is indicated by the assembler.

3.1.3 EXTENDED FORMAT ADDRESSING

It is possible to extend the format of certain register-memory instructions and to include data or indirect addresses within these instructions. When this feature is used, the instruction is referred to as an extended format instruction. The extended format instruction coding forms are flagged by note 5 in table 3-3. The assembler interprets the coded instruction and fills the I, X, B and SD fields as follows:

- If the I, X, B, and SD fields are 0, 0, 0, 0, respectively, the next sequential location in memory is used for the operand, and the program counter is incremented a second time. (The first increment is normal to locate the next word in memory). If the instruction is of the double precision type, such as DLD, DST, DAD, or DSB, the next two sequential memory locations are used for the operand, and the program counter is incremented a third time. The assembler, in this case, generates only one word of data for these double-length instructions. The programmer must supply the second word, typically with a DATA directive.
- If the I, X, B, and SD fields are 1, 0, 0, 0, respectively, the effective address is obtained from the next sequential location in memory, and the program counter is incremented a second time.
- If the I, X, B, and SD fields are 1, 1, 0, 0, respectively, the content of the next sequential memory location is added to the content of the index register to form the effective address, and the program counter is incremented a second time.

NOTE

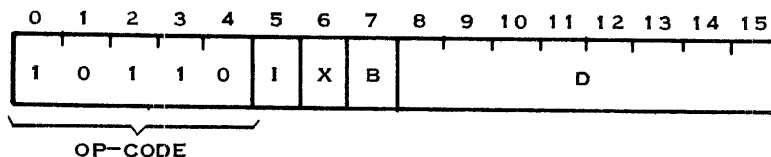
The indexing is unconditionally performed as post-indexing for double-word instructions; bit 10 of the status word is ignored in this case.

3.2 LOAD INSTRUCTIONS

The load instructions listed in table 3-1 are described in the following paragraphs.

3.2.1 DOUBLE LOAD REGISTERS A AND E (DLD)

Machine Format:



Instruction Execution: (EOA, EOA+1) ⟶ (A, E) where EOA is developed in accordance with table 3-3.

Description: Register A is loaded with the contents of the effective operand address, EOA, and register E is loaded with the contents of the EOA plus one. If the IXB fields are 7_{16} (immediate addressing), load E with the sign extended displacement field, D, and load A with the extended sign (all zeros or all ones).

Status Affected: None

Execution Time: 1.00 to 4.00 microseconds (refer to Appendix A)

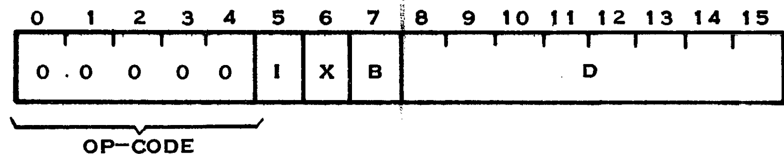
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the DLD instruction. The DLD mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Examples:

						Before	After
	DLD	$+1	⇒	(A)	=	0054_{16}	$AE30_{16}$
	DATA	>AE30, >3239		(E)	=	$16BC_{16}$	3239_{16}
				(EOA)	=	$AE30_{16}$	No change
				(EOA+1)	=	3239_{16}	No change
	@DLD	BASE					
	⋮		⇒	(A)	=	$CC45_{16}$	1064_{16}
BASE	DATA	>1064, >7558		(E)	=	$A0A0_{16}$	7558_{16}
				(EOA)	=	1064_{16}	No change
				(EOA+1)	=	7558_{16}	No change

3.2.2 LOAD REGISTER A (LDA)

Machine Format:



Instruction Execution: (EOA)⟶(A) where EOA is developed in accordance with table 3-3.

Description: Register A is loaded with the contents of the effective operand address, EOA. If the IXB fields are 7_{16} (immediate addressing), load A with the sign extended displacement field, D.

Status Affected: None

Execution Time: 0.75 to 2.75 microseconds (refer to Appendix A)

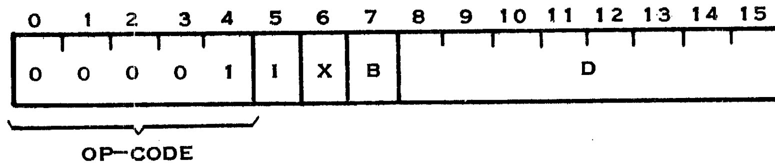
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the LDA instruction. The LDA mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Examples:

						Before	After
	LDA	= -1	=>	(A)	=	$05A3_{16}$	$FFFF_{16}$
				(EOA)	=	$07FF_{16}$	No change
HERE	LDA	$	=>	(A)	=	$F6EF_{16}$	$00FF_{16}$
				(HERE)	=	$00FF_{16}$	No change

3.2.3 LOAD REGISTER E (LDE)

Machine Format:



Instruction Execution: (EOA) ⟶ (E) where EOA is developed in accordance with table 3-3.

Description: Register E is loaded with the contents of the effective operand address, EOA. If the IXB fields are 7_{16} (immediate addressing), load E with the sign extended displacement field, D.

Status Affected: None

Execution Time: 0.75 to 2.75 microseconds (refer to Appendix A)

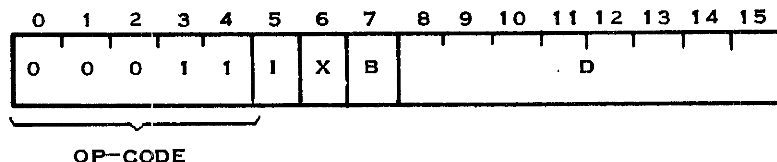
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the LDE instruction. The LDE mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

						Before	After
	LDE	BOT,2	=>	(E)	=	$A6B7_{16}$	0333_{16}
	:			(X)	=	0001_{16}	No change
BOT	DATA	>F,>0333		(EOA)	=	0333_{16}	No change

3.2.4 LOAD REGISTER M (LDM)

Machine Format:



Instruction Execution: (EOA) ⟶ (M) where EOA is developed in accordance with table 3-3.

Description: Register M is loaded with the contents of the effective operand address, EOA. If the IXB fields are 7_{16} (immediate addressing), load M with the sign extended displacement field, D.

Status Affected: None

Execution Time: 0.75 to 2.75 microseconds (refer to Appendix A)

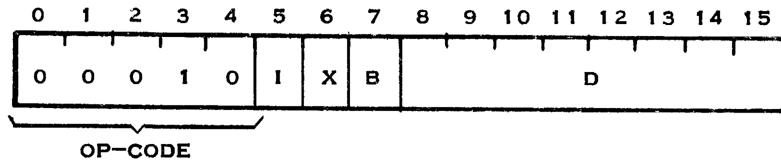
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the LDM instruction. The LDM mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

						Before	After
EXEC	@LDM	=PRB		(M)	=	1124_{16}	Address of PRB
	:		=>				
PRB	DATA	>0006		(EXEC+1)	=	Address of PRB	No change
	DATA	>0000					
	DATA	>0050, BUFFER					

3.2.5 LOAD REGISTER X (LDX)

Machine Format:



Instruction Execution: (EOA) ⟶ (X) where EOA is developed in accordance with table 3-3.

Description: Register X is loaded with the contents of the effective operand address, EOA. If the IXB fields are 7_{16} (immediate addressing), load X with the sign extended displacement field, D.

Status Affected: None

Execution Time: 0.75 to 2.75 microseconds (refer to Appendix A)

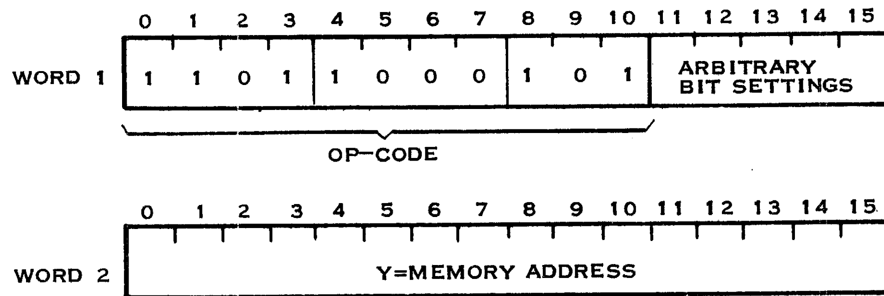
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the LDX instruction. The LDX mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

						Before	After
CHCT	LDX	= -32	=>	(X)	=	0000_{16}	$FFE0_{16}$
				(CHCT)	=	$17E0_{16}$	No change

3.2.6 LOAD REGISTER FILE (LRF)

Machine Format:



Instruction Execution: (Y, Y+1, Y+2, Y+3, Y+4, Y+5, Y+6) ⟶ (A, E, X, M, S, L, B)

Description: Registers A, E, X, M, S, L, and B (the register file) are loaded from sequential memory locations starting at the address specified by Y (second word of the instruction).

Status Affected: None

Execution Time: 7.00 microseconds

Symbolic Coding: The assembly language coding formats for the LRF instructions are as follows:

Label		Operation		Operand		Comment
[label]	ƀ	@LRF	ƀ	adrs	ƀ	[comment]
			or			
[label]	ƀ	LRF	ƀ			[comment]
[label]	ƀ	DATA	ƀ	adrs	ƀ	[comment]

where "adrs" is the symbolic name of a 16-bit memory address.

Example:

```
        @LRF    MEMA
          :
MEMA    DATA    >0300,>06AA,>FFE0,>1A61,>0000,>1121,>8A04
```

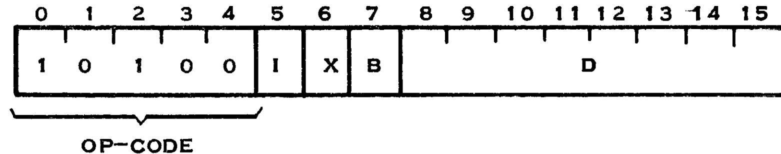
	Before (Hex)	After (Hex)
Register file	(A) = 0000	0300
	(E) = 0002	06AA
	(X) = FFFF	FFE0
	(M) = 200D	1A61
	(S) = 0C00	0000
	(L) = FA00	1121
	(B) = 0601	8A04

3.3 STORE INSTRUCTIONS

The store instructions listed in table 3-1 are described in the following paragraphs.

3.3.1 DOUBLE STORE REGISTERS A AND E (DST)

Machine Format:



Instruction Execution: (A, E)⟶(EOA, EOA+1) where EOA is developed in accordance with table 3-3.

Description: Store the contents of register A into the contents of the effective operand address, EOA, and store the contents of register E into the contents of EOA plus one. If the IXB fields are 7_{16} (immediate addressing), the displacement field, D, is the EOA.

Status Affected: None

Execution Time: 2.75 to 4.00 microseconds (refer to Appendix A)

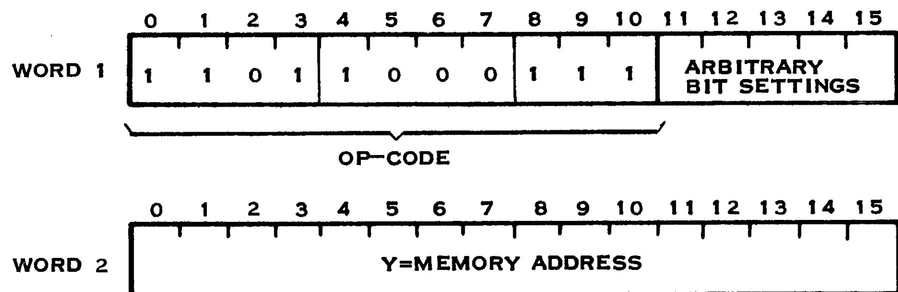
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the DST instruction. The DST mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

					Before	After
	DST	TOP		(A, E) =	4441_{16}, $4D4E_{16}$	No change
	.		⇒			
	.					
TOP	BSS	2		(TOP, TOP+1) =	$4C55_{16}$, $434B_{16}$	4441_{16}, $4D4E_{16}$

3.3.2 STORE REGISTER FILE (SRF)

Machine Format:



Instruction Execution: (A, E, X, M, S, L, B) ⟶ (Y, Y+1, Y+2, Y+3, Y+4, Y+5, Y+6)

Description: Store the contents of registers A, E, X, M, S, L, and B (register file) into sequential memory locations starting at the address specified by Y (second word of the instruction).

Status Affected: None

Execution Time: 7.00 microseconds

Symbolic Coding: The assembly language coding formats for the SRF instruction are as follows:

Label		Operation		Operand		Comment
[label]	ƀ	@SRF	ƀ	adrs	ƀ	[comment]
			or			
[label]	ƀ	SRF	ƀ			[comment]
[label]	ƀ	DATA	ƀ	adrs	ƀ	[comment]

where "adrs" is the symbolic name of a 16-bit memory address.

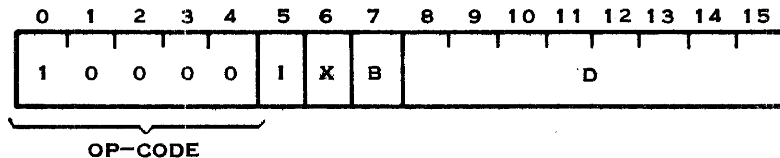
Example:

```
        SRF
        DATA    SAVE
          :
SAVE    BSS     7
```

		Before (Hex)	After (Hex)
Register file	(A)	0001	No change
	(E)	DE03	
	(X)	0004	
	(M)	0101	
	(S)	FFFF	
	(L)	23A3	
	(B)	0800	
Memory locations	(SAVE)	FA03	0001
	(SAVE+1)	0004	DE03
	(SAVE+2)	FFDE	0004
	(SAVE+3)	DE80	0101
	(SAVE+4)	3A40	FFFF
	(SAVE+5)	11AB	23A3
	(SAVE+6)	CE00	0800

3.3.3 STORE REGISTER A (STA)

Machine Format:



Instruction Execution: (A) ⟶ (EOA) where EOA is developed in accordance with table 3-3.

Description: Store the contents of register A into the contents of the effective operand address, EOA. If the IXB fields are 7_{16} (immediate addressing), the displacement field, D, is the EOA.

Status Affected: None

Execution Time: 2.00 to 3.00 microseconds (refer to Appendix A)

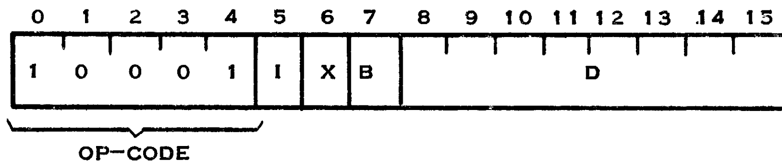
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the STA instruction. The STA mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

				Before	After
STA	DEST, 1	=>	(A) =	$D8C0_{16}$	No change
			(DEST) =	0642_{16}	$D8C0_{16}$

3.3.4 STORE REGISTER E (STE)

Machine Format:



Instruction Execution: (E) ⟶ (EOA) where EOA is developed in accordance with table 3-3.

Description: Store the contents of register E into the contents of the effective operand address, EOA. If the IXB fields are 7_{16} (immediate addressing), the displacement field, D, is the EOA.

Status Affected: None

Execution Time: 2.00 to 3.00 microseconds (refer to Appendix A)

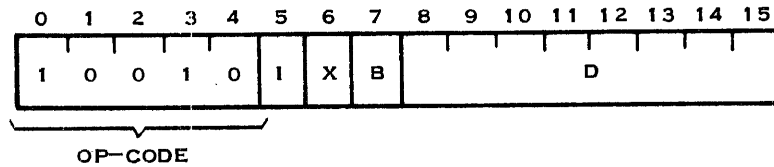
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the STE instruction. The STE mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

				Before	After
STE	=6	=>			
			(E) =	$1AE9_{16}$	No change
			(Memory location 6) =	$788B_{16}$	$1AE9_{16}$

3.3.5 STORE REGISTER X (STX)

Machine Format:



Instruction Execution: (X)→(EOA) where EOA is developed in accordance with table 3-3.

Description: Store the contents of register X into the contents of the effective operand address, EOA. If the IXB fields are 7_{16} (immediate addressing), the displacement field, D, is the EOA.

Status Affected: None

Execution Time: 2.00 to 3.00 microseconds (refer to Appendix A)

Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the STX instruction. The STX mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

@STX FARAWY,2 ⇒			Before	After
	(X)	=	0002_{16}	No change
	(FARAWY+2)	=	1007_{16}	0002_{16}

NOTE

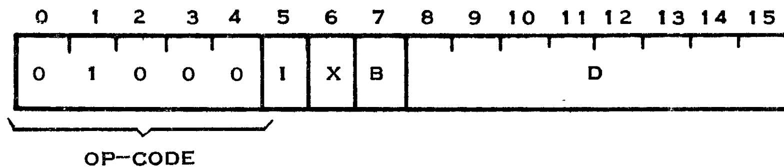
The content of register X is both stored and used as the index.

3.4 BRANCH INSTRUCTIONS

The branch instructions listed in table 3-1 are described in the following paragraphs.

3.4.1 BRANCH ON INCREMENTED INDEX (BIX)

Machine Format:



Instruction Execution: (X)+1 ⟶ (X); if (X) ≠ 0, EOA ⟶ PC
if (X) = 0, PC is not affected
where EOA is developed in accordance with table 3-3.

Description: Increment the contents of register X by one: if the resulting X register value is non-zero, place the effective operand address, EOA, in the program counter and continue execution from that point; if the resulting X register value is zero, continue execution with the next sequential instruction. If the IXB fields are 7_{16} (immediate addressing), the displacement field, D, is the EOA. The BIX instruction is commonly used in loop control where register X contains a negative loop count.

NOTE

The extended format BIX instruction is allowed since an extra program counter increment occurs on the fall through condition. If the BIX instruction is single length, the IXB bits are zero, and the displacement field is zero, the next word is skipped when the X register is incremented to zero. When the X register is incremented to a non-zero quantity, the next word is executed.

Status Affected: None

Execution Time: 1.25 to 2.25 microseconds (refer to Appendix A)

Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the BIX instruction. The BIX mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

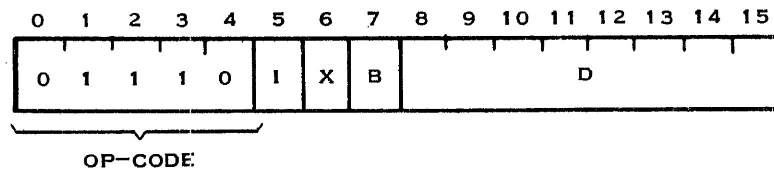
					Before	After	
BIX	DOG	⇒	(X)	=	$FFA6_{16}$	$FFA7_{16}$	where the BIX instruction is at $1B64_{16}$ and DOG is at $1B20_{16}$.
			(PC)	=	$1B64_{16}$	$1B20_{16}$	

The following instruction application example illustrates use of the BIX instruction to sum a buffer's contents.

```
                 :
          LDX    =-32
          LDA    =0
LOOP      ADD    BUFFER+32,2
          BIX    LOOP
                 :
BUFFER    BSS    32
```

3.4.2 BRANCH AND LINK (BRL)

Machine Format:



Instruction Execution: (PC) → (L); EOA → (PC) where EOA is developed in accordance with table 3-3.

Description: Load the contents of the program counter into the link register, L, place the effective operand address, EOA, in the program counter, and continue execution from that point. If the IXB fields are 7_{16} (immediate addressing), the displacement field, D, is the EOA. The BRL instruction is commonly used for subroutine linkage. To return, the subroutine typically uses either an RMO L, P or REX L, P instruction. The return may also be accomplished by storing the contents of the link register in memory and branching indirectly through that memory location with a BRU instruction.

NOTE

> The extended format BRL instruction places the address of the first word beyond the double-length BRL instruction in the link register.

Status Affected: None

Execution Time: 1.50 to 2.50 microseconds (refer to Appendix A)

Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the BRL instruction. The BRL mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

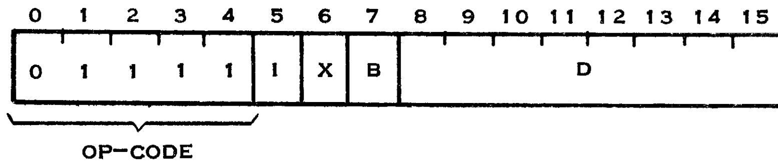
				Before	After	
BRL	CAREA	⇒ (L)	=	$032A_{16}$	$055E_{16}$	where CAREA is at 0580_{16} and in the range $-128 \leq$ PC ≤ 127.
		(PC)	=	$055D_{16}$	0580_{16}	

The following instruction application example illustrates use of the BRL instruction to execute a subroutine.

```
              (Main program)
                  :
                BRL    WRITE
                  :
      WRITE     EQU    $       (Write subroutine)
                  :
                RMO    5,7     (Return to instruction following BRL WRITE)
```

3.4.3 BRANCH UNCONDITIONAL (BRU)

Machine Format:



Instruction Execution: EOA ⟶ (PC) where EOA is developed in accordance with table 3-3.

Description: Place the effective operand address, EOA, in the program counter and continue execution from that point. If the IXB fields are 7_{16} (immediate addressing), the displacement field, D, is the EOA.

NOTE

The extended format BRU instruction alters the program counter in the same manner as single-length BRU instructions.

Status Affected: None

Execution Time: 1.00 to 2.25 microseconds (refer to Appendix A)

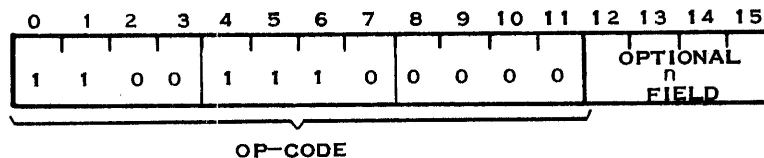
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the BRU instruction. The BRU mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

			Before	After	
@BRU	TAB, 2 ⇒ (PC)	=	$1B13_{16}$	0850_{16}	where TAB is at 0800_{16}.
	(X)	=	0050_{16}	No change	

3.4.4 IDLE (IDL)

Machine Format:



Instruction Execution: HALT

Description: The idle instruction causes the computer to pause. If the idle instruction is encountered in the RUN mode, the RUN indicator is turned off and the IDLE indicator is turned on. If the MODE switch is left in the RUN position, the computer re-enters the RUN mode if an interrupt occurs or if the START switch is activated. The IDLE indicator is turned off if the MODE switch is placed in the HALT position. If the MODE switch is placed in the SIE position and an idle instruction is encountered during single instruction execution, the IDLE indicator is turned on. If an interrupt occurs in the SIE mode after encountering an idle instruction, the instruction in the appropriate trap location is automatically executed and the IDLE indicator is turned off. The idle instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled.

NOTE

This instruction is commonly used in catastrophic sequences such as a power failure condition. All conditions and registers are preserved in memory, specific interrupt mask conditions are established, and the IDL is executed. Subsequently, when power is restored, or an interrupt is issued which indicates a clearing of the catastrophic situation, the program will resume from the appropriate interrupt entrance.

Status Affected: None

Execution Time: 1.00 microseconds

Symbolic Coding: The assembly language coding format for the IDL instruction is as follows:

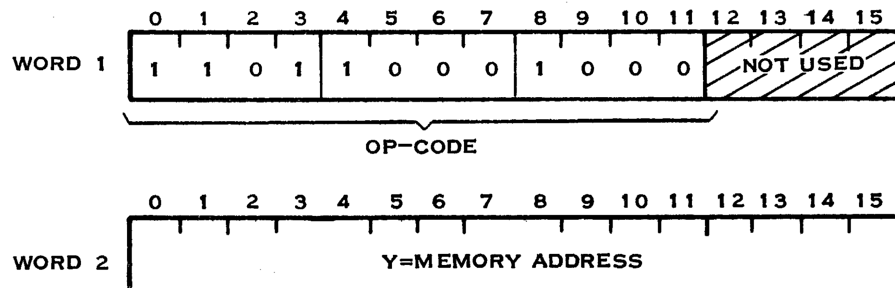
Label	Operation	Operand	Comment
[label] ƀ	IDL ƀ	n ƀ	[comment]

where "n" can be used to flag the reason for the idle when the instruction register is displayed on the computer front panel. If no flag is desired, "n" may be coded as a zero. ($0 \leq n \leq 15$).

Example: IDL 1

3.4.5 LOAD STATUS BLOCK AND BRANCH (LSB)

Machine Format:



Instruction Execution: (Y, Y+1) → (PC, ST)

Description: The program counter is loaded with the contents of memory location Y and the status register is loaded with the contents of memory location Y+1. Program execution continues at the location specified by the new contents of the program counter. Status register bits 5 (memory protect violation), 6 (PIF violation), 14 (memory parity error), and 15 (power fail) are unconditionally cleared to zero by the LSB instruction. The instruction is also restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled. Interrupts, other than internal, are inhibited for one instruction following an LSB.

NOTE

The LSB instruction is commonly used for an exit from interrupt processing or for a return from a subroutine. The address Y points to the program counter and status register preserved by an SSB instruction upon entrance to an interrupt processing or subroutine program.

Status Affected: All status register bits are affected as indicated by memory location Y+1, with the following exceptions: bits 5, 6, 14, and 15 are unconditionally cleared to zero.

Execution Time: 3.25 microseconds

Symbolic Coding: The assembly language coding formats for the LSB instruction are as follows:

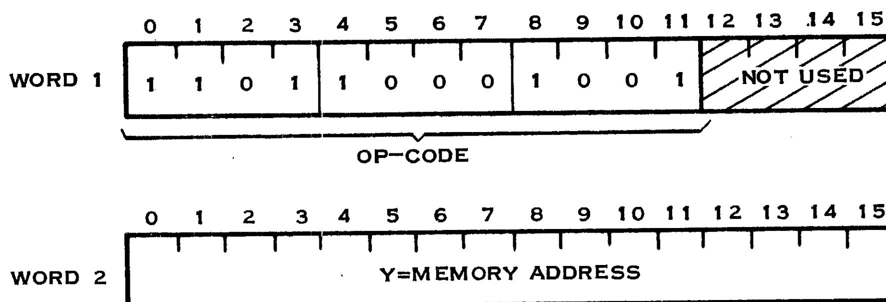
Label		Operation		Operand		Comment	
[label]	ƀ	@LSB	ƀ	adrs	ƀ	[comment]	where "adrs" is the symbolic name of a 16-bit memory address.
		or					
[label]	ƀ	LSB	ƀ			[comment]	
[label]	ƀ	DATA	ƀ	adrs	ƀ	[comment]	

Example:

			Before (Hex)	After (Hex)
@LSB	PROG5 ⇒	(PC, ST)	= 0400, 0850	1A69, 0010
		(PROG5, PROG5+1)	= 1A69, 0010	No change

3.4.6 LOAD STATUS BLOCK, RESET INTERRUPT, AND BRANCH (LSR)

Machine Format:



Instruction Execution: (Y, Y+1)⟶ (PC, ST); reset highest priority vectored interrupt if applicable.

Description: Execution of the LSR instruction is identical to LSB (paragraph 3.4.5), except that the highest priority vectored interrupt present in the vectored interrupt option is additionally reset. If the computer does not include the vectored interrupt option, the LSR instruction is identical to LSB.

Status Affected: All status register bits are affected as indicated by memory location Y+1, with the following exceptions: bits 5 (memory protect violation), 6 (PIF violation), 14 (memory parity error), and 15 (power fail) are unconditionally cleared to zero.

Execution Time: 3.25 microseconds

Symbolic Coding: The assembly language coding formats for the LSR instruction are as follows:

Label		Operation		Operand		Comment
[label]	b̸	@LSB	b̸	adrs	b̸	[comment]
		or				
[label]	b̸	LSB	b̸			[comment]
[label]	b̸	DATA	b̸	adrs	b̸	[comment]

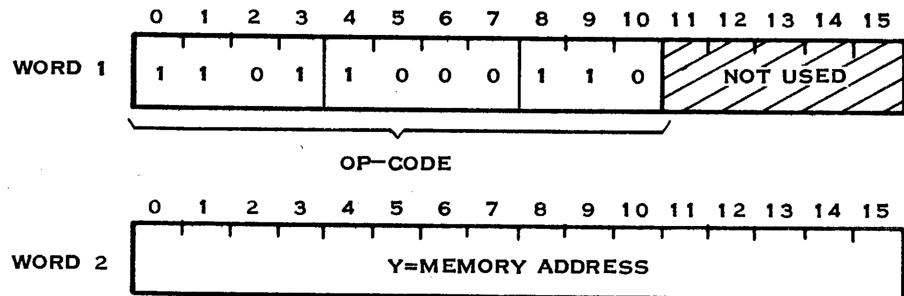
where "adrs" is the symbolic name of a 16-bit memory address.

Example:

				Before (Hex)	After (Hex)
LSR		⇒			
DATA	CATA		(PC, ST) =	13A5, 0110	075D, 0010
			(CATA, CATA+1) =	075D, 0010	No change

3.4.7 STORE STATUS BLOCK AND BRANCH (SSB)

Machine Format:



Instruction Execution: (PC, ST) → (Y, Y+1); Y+2 → (PC)

Description: The program counter is stored in memory location Y and the status register is stored in memory location Y+1. Program execution continues at memory location Y+2. Interrupts, other than internal, are inhibited for one instruction following an SSB.

NOTE

The SSB instruction is commonly used for entrance to interrupt processing and subroutine programs. Return from these type of programs is accomplished by an LSB instruction.

Status Affected: Bits 7 (data bus interrupt), 8 (vectored interrupt), and 12 (DMAC interrupt) of the status register are cleared to zero according to the computer interrupt priority scheme. These bits are cleared so that when an interrupt occurs, all interrupts of lower or equal priority are disabled. The four types of interrupts in order of priority are as follows: internal interrupt, vectored interrupt, DMAC interrupt, and data bus interrupt.

Execution Time: 3.25 microseconds

Symbolic Coding: The assembly language coding formats for the SSB instruction are as follows:

Label		Operation		Operand		Comment
[label]	ƀ	@SSB	ƀ	adrs	ƀ	[comment]
		or				
[label]	ƀ	SSB	ƀ			[comment]
[label]	ƀ	DATA	ƀ	adrs	ƀ	[comment]

where "adrs" is the symbolic name of a 16-bit memory address.

Example:

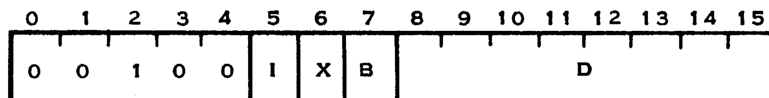
				Before (Hex)	After (Hex)
SSB		=>			
DATA	>0A23		(PC, ST)	= 07A2, 0110	0A25, 0110
			$(0A23_{16}, 0A24_{16})$	= 08B6, 0010	07A2, 0110

3.5 ARITHMETIC INSTRUCTIONS

The arithmetic instructions listed in table 3-1 are described in the following paragraphs.

3.5.1 ADD TO REGISTER A (ADD)

Machine Format:



Instruction Execution: (EOA) + (A) → (A) where EOA is developed in accordance with table 3-3.

Description: Add the contents of the effective operand address, EOA, to the contents of register A and place the sum in register A. If the IXB fields are 7_{16} (immediate addressing), the sign extended displacement field, D, is added to register A.

Status Affected: If the sum from the ADD instruction is outside the range of -2^{-15} to $2^{15}-1$, the overflow indicator (bit 2 of the status register) is turned on. If the sum is within the same range, the overflow indicator is turned off. If the add operation results in a carry into the sign position (bit 0), the carry indicator (bit 3 of the status register) is turned on; otherwise, it is turned off.

Execution Time: 0.75 to 2.75 microseconds (refer to Appendix A)

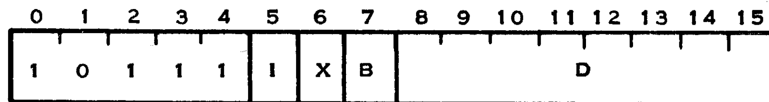
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the ADD instruction. The ADD mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

			Before	After
ADD	*BSC ⇒ (A)	=	$4B10_{16}$	$5F0C_{16}$
	(BSC)	=	$003A_{16}$	No change
	$(003A_{16})$	=	$13FC_{16}$	No change

3.5.2 DOUBLE LENGTH ADD (DAD)

Machine Format:



Instruction Execution: (EOA, EOA+1) + (A, E) → (A, E) where EOA is developed in accordance with table 3-3.

Description: Add the concatenation of the contents of the effective operand address, EOA, and EOA+1 to the concatenation of registers A and E (register A is the most significant half of the second concatenation). At completion of the add operation, bit 0 of register E is forced to agree with bit 0 of register A. If the IXB fields are 7_{16} (immediate addressing), the displacement field, D, with its sign extended 24 bits becomes the double-length operand.

NOTE

Prior to the addition, ensure that the two sign bits associated with each double-length word are identical. If the two sign bits in the same double-length word are different, the result of the add may not be valid.

Status Affected: If the sum from the DAD instruction is outside the range of -2^{30} to $2^{30}-1$, the overflow indicator (bit 2 of the status register) is turned on; otherwise, the overflow indicator is turned off. If the add operation results in a carry into the sign position (bit 0 of register A), the carry indicator (bit 3 of the status register) is turned on; otherwise, the carry indicator is turned off.

Execution Time: 1.00 to 4.00 microseconds (refer to Appendix A).

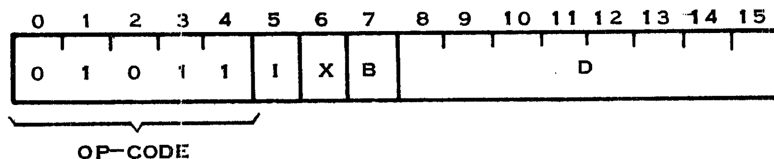
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the DAD instruction. The DAD mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

		Before (Hex)	After (Hex)
DAD PRICE			
	⇒ (A, E)	= 0069, 73B4	016A, 5034
	(PRICE, PRICE+1)	= 0100, 5C80	No change

3.5.3 DIVIDE (DIV)

Machine Format:



Instruction Execution: $(A, E) / (EOA) \rightarrow (A_{quo}, E_{rem})$ where EOA is developed in accordance with table 3-3.

Description: Divide the concatenation of registers A and E (with the most significant half in register A) by the contents of the effective operand address, EOA. Place the quotient in register A and the remainder in register E. The sign of the remainder will be the same as the sign of the original dividend, except when the sign is set positive in the case of a zero remainder. If the IXB fields are 7_{16} (immediate addressing), the displacement field, D, with its sign extended eight bits is used as the divisor.

Status Affected: If the magnitude of the most significant half of the dividend (register A) is greater than or equal to the magnitude of the divisor, the overflow indicator (bit 2 of the status register) is turned on and the contents of registers A and E remain unchanged. Otherwise, the overflow indicator is turned off.

Execution Time: 1.50 to 8.75 microseconds (refer to Appendix A)

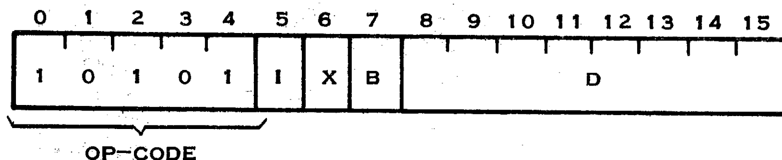
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the DIV instruction. The DIV mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

			Before (Hex)	After (Hex)
@DIV	=600 ⇒	(A, E) =	0019, 78A0	0588, 01E0
		(EOA) =	0258	No change

3.5.4 DOUBLE LENGTH SUBTRACT (DSB)

Machine Format:



Instruction Execution: $(A, E) - (EOA, EOA+1) \rightarrow (A, E)$ where EOA is developed in accordance with table 3-3.

Description: Add the two's complement of the concatenation of the contents of the effective operand address, EOA, and EOA+1 to the concatenation of registers A and E (register A is the most significant half of the second concatenation). Place the result in registers A and E. At the completion of the

two's complement addition, bit 0 of register E is forced to agree with bit 0 of register A. If the IXB fields are 7_{16} (immediate addressing), the displacement field, D, with its sign extended 24 bits becomes the subtrahend.

NOTE

Prior to the subtraction, ensure that the two sign bits associated with each double-length word are identical. If the two sign bits in the same double-length word are different, the result of the subtract may not be valid.

Status Affected: If the result of the DSB instruction is outside the range of -2^{30} to $2^{30}-1$, the overflow indicator (bit 2 of the status register) is turned on; otherwise, the overflow indicator is turned off. If there is a carry into the sign position (bit 0 of register A), the carry indicator (bit 3 of the status register) is turned on; otherwise, the carry indicator is turned off.

Execution Time: 1.00 to 4.00 microseconds (refer to Appendix A)

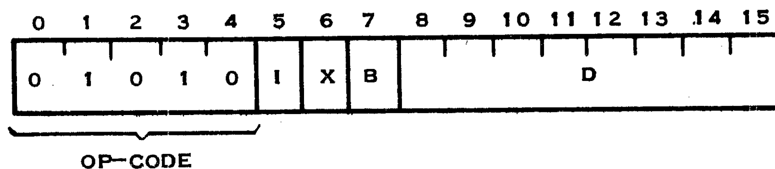
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the DSB instruction. The DSB mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

				Before (Hex)	After (Hex)
DSB	DECMAL, 5 ⇒	(A, E)	=	6D11, 6F51	5268, 5ACB
		(DECMAL)	=	0396	No change
		(0396_{16}, 0397_{16})	=	1AA9, 1486	No change

3.5.5 INCREMENT MEMORY BY ONE (IMO)

Machine Format:



Instruction Execution: (EOA) + 1 ⟶ (EOA) where EOA is developed in accordance with table 3-3.

Description: Increment the contents of the effective operand address, EOA, by one, and replace the contents of the EOA with the result. If the IXB fields are 7_{16} (immediate addressing), the displacement field, D, becomes the EOA.

Status Affected: None

Execution Time: 2.75 to 3.75 microseconds (refer to Appendix A)

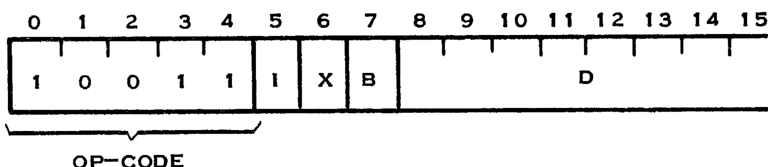
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the IMO instruction. The IMO mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

				Before	After
@IMO	BOX, 2 ⇒	(X)	=	0008_{16}	No change
		(BOX+8)	=	$634A_{16}$	$634B_{16}$

3.5.6 MULTIPLY (MPY)

Machine Format:



Instruction Execution: (A) x (EOA) ⟶ (A, E) where EOA is developed in accordance with table 3-3.

Description: Multiply register A by the contents of the effective operand address, EOA. Place the double-length result in registers A and E, the most significant part being in register A. At completion of the multiplication, bit 0 of register E is forced to agree with bit 0 of register A. If the IXB fields are 7_{16} (immediate addressing), the displacement field, D, with its sign extended eight bits becomes the operand.

Status Affected: If both operands are equal to the maximum negative number (-2^{15}), the overflow indicator (bit 2 of the status register) is turned on and the result in registers A and E will be indeterminate. Otherwise, the overflow indicator is turned off.

Execution Time: 1.25 to 7.25 microseconds (refer to Appendix A)

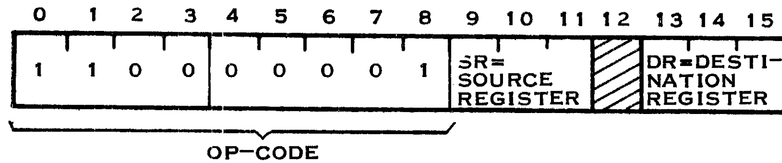
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the MPY instruction. The MPY mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

			Before (Hex)	After (Hex)
MPY	ARG, 1			
	⇒	(A, E)	0003, 1020	FFFF, FFFD
		(ARG)	FFFF	No change

3.5.7 REGISTER ADD (RAD)

Machine Format:



Instruction Execution: (SR) + (DR) ⟶ (DR)

Description: Add the contents of the registers specified by the SR and DR fields. Place the result in the register specified by the DR field. If bit 12 of the machine format is set to one and bits 13 to 15 are zeroed, the status register is specified as the destination register. In this case the instruction is restricted, meaning it is considered illegal if the memory protect/ privileged instruction feature is enabled. Interrupts, other than internal, are inhibited for one instruction following this special case of the RAD instruction.

Status Affected: If the result of the RAD instruction is outside the range of -2^{15} to $2^{15}-1$, the overflow indicator (bit 2 of the status register) is turned on; otherwise, the overflow indicator is turned off. If there is a carry into the sign position (bit 0), the carry indicator (bit 3 of the status register) is turned on; otherwise, the carry indicator is turned off.

Execution Time: 1.25 microseconds

Symbolic Coding: The assembly language coding format for the RAD instruction is as follows:

Label	Operation	Operand	Comment
[label] ƀ	RAD ƀ	sreg, dreg ƀ	[comment]

where "sreg" and "dreg" are expressions that address the source and destination registers, respectively, in accordance with table 2-2. The special case when "dreg" equals eight is covered in the "Description" paragraph.

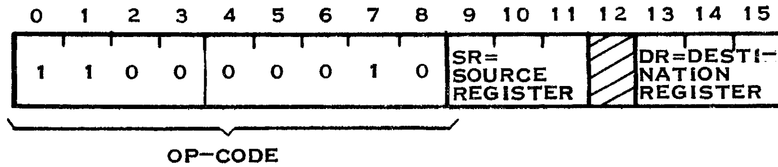
Example:

```
A   EQU   0
X   EQU   2
     :
    RAD   A,X
```

	Before	After
⇒ (X) =	4456_{16}	6622_{16}
(A) =	$21CC_{16}$	No change

3.5.8 REGISTER COMPLEMENT (RCO)

Machine Format:



Instruction Execution: -(SR) ⟶ (DR)

Description: Replace the contents of the register specified by the DR field with the two's complement of the contents of the register specified by the SR field. If bit 12 of the machine format is set to one and bits 13 to 15 are zeroed, the status register is specified as the destination register. In this case the instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled. Interrupts, other than internal, are inhibited for one instruction following this special case of the RCO instruction.

Status Affected: If the SR register contains -2^{15}, the overflow indicator (bit 2 of the status register) is turned on and the DR register is set to -2^{15}; otherwise, the overflow indicator is turned off.

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the RCO instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	RCO	ƀ	sreg, dreg	ƀ	[comment]

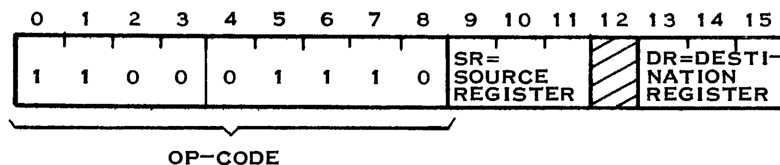
where "sreg" and "dreg" are expressions that address the source and destination registers, respectively, in accordance with table 2-2. The special case when "dreg" equals eight is covered in the "Description" paragraph.

Example:

			Before	After
RCO	2, 2	=> (X) =	$000F_{16}$	$FFF1_{16}$

3.5.9 REGISTER DECREMENT (RDE)

Machine Format:



Instruction Execution: (SR)-1 ⟶ (DR)

Description: Subtract one from the contents of the register specified by the SR field and place the result in the register specified by the DR field.

NOTE

> If the maximum negative number (-32768) is decremented, the maximum positive number (+32767) is placed in the DR register.

If bit 12 of the machine format is set to one and bits 13 to 15 are zeroed, the status register is specified as the destination register. In this case the instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled. Interrupts, other than internal, are inhibited for one instruction following this special case of the RDE instruction.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the RDE instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	RDE	ƀ	sreg, dreg	ƀ	[comment]

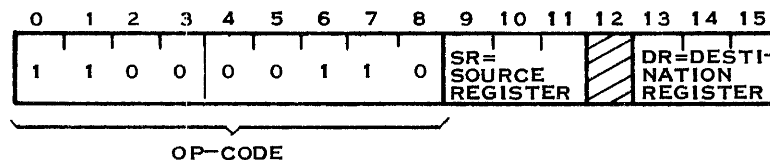
where "sreg" and "dreg" are expressions that address the source and destination registers, respectively, in accordance with table 2-2. The special case when "dreg" equals eight is covered in the "Description" paragraph.

Example:

```
S    EQU   4   =>          Before      After
 :                 (S) =   0044_16     0043_16
 :
     RDE   S,S
```

3.5.10 REGISTER INCREMENT (RIN)

Machine Format:



Instruction Execution: (SR)+1 → (DR)

Description: Add one to the contents of the register specified by the SR field and place the result in the register specified by the DR field.

NOTE

If the result of the RIN is considered to be a 15-bit signed number, incrementing the maximum positive number (+32767) results in the maximum negative number (-32768). If the result of the RIN is considered to be a 16-bit positive number (as in address calculation), incrementing the maximum positive number (65535) results in zero.

If bit 12 of the machine format is set to one and bits 13 to 15 are zeroed, the status register is specified as the destination register. In this case the instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled. Interrupts, other than internal, are inhibited for one instruction following this special case of the RIN instruction.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the RIN instruction is as follows:

Label		Operation		Operand		Comment
[label]	b̸	RIN	b̸	sreg, dreg	b̸	[comment]

where "sreg" and "dreg" are expressions that address the source and destination registers, respectively, in accordance with table 2-2. The special case when "dreg" equals eight is covered in the "Description" paragraph.

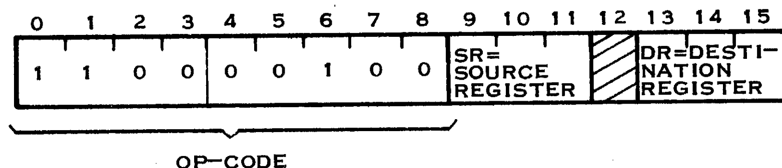
Example:

RIN 7,5

			Before	After
=>	(L)	=	$622B_{16}$	0226_{16}
	(PC)	=	0225_{16}	No change

3.5.11 REGISTER INVERT (RIV)

Machine Format:



Instruction Execution: -(SR)-1⟶(DR)

Description: Replace the contents of the register specified by the DR field with the one's complement of the contents of the register specified by the SR field. This means each bit of the SR register is complemented individually. If bit 12 of the machine format is set to one and bits 13 to 15 are zeroed, the status register is specified as the destination register. In this case the instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled. Interrupts, other than internal, are inhibited for one instruction following this special case of the RIV instruction.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the RIV instruction is as follows:

Label	Operation	Operand	Comment
[label] ƀ	RIV	ƀ sreg, dreg ƀ	[comment]

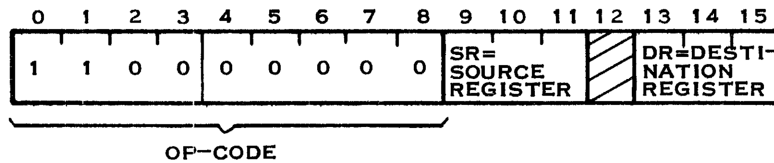
where "sreg" and "dreg" are expressions that address the source and destination registers, respectively, in accordance with table 2-2. The special case when "dreg" equals eight is covered in the "Description" paragraph.

Example:

```
E    EQU   1                  Before      After
X    EQU   2          (X) =   121C16      FCFA16
 :               =>
                      (E) =   030516      No change
     RIV   E,X
```

3.5.12 REGISTER SUBTRACT (RSU)

Machine Format:



Instruction Execution: (DR) - (SR)⟶(DR)

Description: Subtract the contents of the register specified by the SR field from the contents of the register specified by the DR field. Place the result in the register specified by the DR field. If bit 12 of the machine format is

set to one and bits 13 to 15 are zeroed, the status register is specified as the destination register. In this case the instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled. Interrupts, other than internal, are inhibited for one instruction following this special case of the RSU instruction.

Status Affected: If the result of the RSU instruction is outside the range of -2^{15} to 2^{15}-1, the overflow indicator (bit 2 of the status register) is turned on; otherwise, the overflow indicator is turned off. If there is a carry into the sign position (bit 0), the carry indicator (bit 3 of the status register) is turned on; otherwise, the carry indicator is turned off.

Execution Time: 1.25 microseconds

Symbolic Coding: The assembly language coding format for the RSU instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	RSU	ƀ	sreg, dreg	ƀ	[comment]

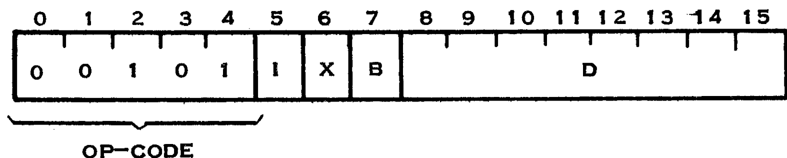
where "sreg" and "dreg" are expressions that address the source and destination registers, respectively, in accordance with table 2-2. The special case when "dreg" equals eight is covered in the "Description" paragraph.

Example:

RSU 6,5			Before	After
	=> (L)	=	$56A2_{16}$	5567_{16}
	(B)	=	$013B_{16}$	No change

3.5.13 SUBTRACT FROM REGISTER A (SUB)

Machine Format:



Instruction Execution: (A) - (EOA) ⟶ (A) where EOA is developed in accordance with table 3-3.

Description: Add the two's complement of the contents of the effective operand address, EOA, to the contents of register A. Place the result in register A. If the IXB fields are 7_{16} (immediate addressing), the sign extended displacement field, D, is subtracted from register A.

Status Affected: If the result of the SUB instruction is outside the range of -2^{15} to $2^{15}-1$, the overflow indicator (bit 2 of the status register) is turned on; otherwise, the overflow indicator is turned off. If there is a carry into the sign position (bit 0), the carry indicator (bit 3 of the status register) is turned on; otherwse, the carry indicator is turned off.

Execution Time: 0.75 to 2.75 microseconds (refer to Appendix A)

Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the SUB instruction. The SUB mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

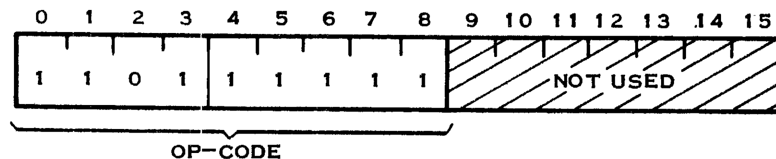
THIS SUB =28			Before	After
=>	(A)	=	0005_{16}	$FFE9_{16}$
	(THIS)	=	$2F1C_{16}$	No change

3.6 COMPARE INSTRUCTIONS

The compare instructions listed in table 3-1 are described in the following paragraphs.

3.6.1 COMPARE LOGICAL CHARACTER STRING (CLC)

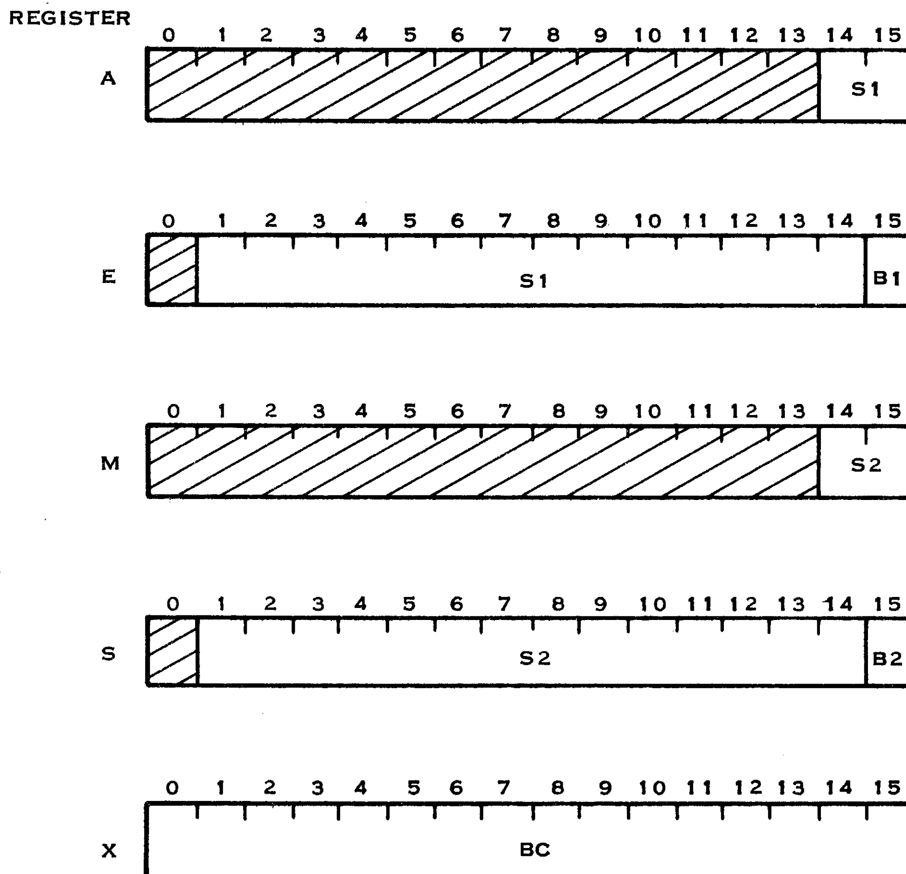
Machine Format:



Instruction Execution: $(M_1):(Y_1)$, $(M_2):(Y_2)$, ... $(M_n):(Y_n)$

where $M_1, M_2, \ldots M_n$ and $Y_1, Y_2, \ldots Y_n$ are byte strings in memory

Description: Perform a consecutive byte-by-byte logical comparison of two byte strings in memory defined in general registers as follows:



where, S1 and S2 are the starting word addresses of the two byte strings. The most significant bits of the S1 and S2 addresses are in the A and M registers, respectively.

B1 and B2 indicate the position of the first byte in the words addressed by S1 and S2, respectively. A logic zero indicates the first byte is in the most significant half (left half) of the first word; a logic one indicates the first byte is in the least significant half (right half) of the first word.

BC indicates the number of bytes to be compared (up to 65,535).

The first non-equal comparison encountered terminates the CLC instruction with the number of bytes left to be compared loaded in register X. In addition, registers A and E will contain the byte address of the next byte that would have been processed in string 1 and registers M and S will contain the byte address of the next byte that would have been processed in string 2. If the CLC instruction is interrupted, the general registers contain the same information as that described for a non-equal comparison when the interrupt is taken. Note that register X will contain all zeros only when all byte comparisons, or all but the last byte comparison, are found to be equal.

Status Affected: Bits 0 and 1 of the status register are modified as follows by the CLC instruction.

	Bit 0	Bit 1
Each Compare Equal	0	1
$Byte_1 > Byte_2$	1	0
$Byte_1 < Byte_2$	0	0
Unused Bit Setting	1	1

If the byte count (BC) in register X is specified as zero, no comparison is performed and status register bits 0 and 1 are set to 01 unconditionally.

Execution Time: 5.00 + 2.25 X (no. of bytes compared) microseconds

Symbolic Coding: The assembly language coding format for the CLC instruction is as follows:

Label		Operation		Operand	Comment
[label]	b̸	CLC	b̸		[comment]

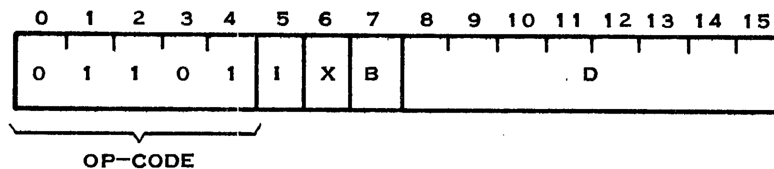
Example:

CLC

		Before (Hex)	After (Hex)
⇒	(A) =	0000	0000
	(E) =	0574	0578
	(M) =	0000	0000
	(S) =	06A6	06AA
	(X) =	000B	0007
	(02BA, 02BB, ...) =	5123, 64AC, ...	No change
	(0353, 0354, ...) =	5123, 64AD, ...	No change

3.6.2 COMPARE ALGEBRAIC (CPA)

Machine Format:



Instruction Execution: (A):(EOA), algebraically where EOA is developed in accordance with table 3-3.

Description: Perform an algebraic compare (bit 0 reflects sign) between the contents of register A and the contents of the effective operand address, EOA. The contents of register A and the contents of EOA are not affected by the compare. Set status register bits to indicate the result of the compare (refer to the next paragraph). If the IXB fields are 7_{16} (immediate addressing), the displacement field, D, sign extended to 16 bits is compared with register A.

Status Affected: Bits 0 and 1 of the status register are modified as follows by the CPA instruction.

	Bit 0	Bit 1
(A) > (EOA)	0	0
(A) = (EOA)	0	1
(A) < (EOA)	1	0
Unused Bit Setting	1	1

Execution Time: 0.75 to 2.75 microseconds (refer to Appendix A)

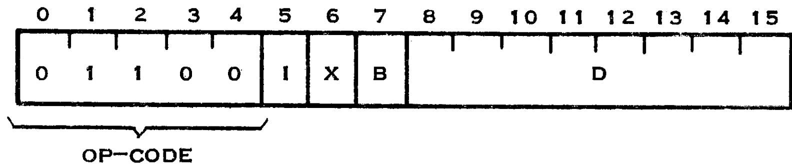
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the CPA instruction. The CPA mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

CPA H4000, 1 ⇒ (A) = $7FFF_{16}$; (H4000) = 4000_{16} } Status register bits 0 and 1 equal 00

3.6.3 COMPARE LOGICAL (CPL)

Machine Format:



Instruction Execution: (A):(EOA), logically where EOA is developed in accordance with table 3-3.

Description: Perform a logical compare (unsigned numbers) between the contents of register A and the contents of the effective operand address, EOA. The contents of register A and the contents of EOA are not affected by the compare. Set the status register bits as described for the CPA instruction in paragraph 3.6.2. If the IXB fields are 7_{16} (immediate addressing), the eight bits of the displacement field, D, are compared with the low order eight bits of register A.

Status Affected: Refer to paragraph 3.6.2.

Execution Time: 0.75 to 2.75 microseconds (refer to Appendix A)

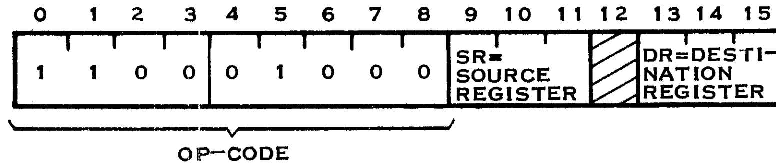
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the CPL instruction. The CPL mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

@CPL	=DOZEN ⇒	(A)	= $A6BB_{16}$	Status register bits 0 and 1 set to 00
		(DOZEN)	= $18F4_{16}$	

3.6.4 REGISTER COMPARE ALGEBRAIC (RCA)

Machine Format:



Instruction Execution: (SR) : (SR), algebraically

Description: Perform an algebraic compare (bit 0 reflects sign) between the contents of the register specified by the SR field and the contents of the register specified by the DR field. The status register bits are set to indicate the result of the compare (refer to the next paragraph). If bit 12 of the machine format is set to one and bits 13 to 15 are zeroed, the status register is specified as the destination register. In this case the instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled. Interrupts, other than internal, are inhibited for one instruction following this special case of the RCA instruction.

Status Affected: Bits 0 and 1 of the status register are modified as follows by the RCA instruction.

	Bit 0	Bit 1
(SR) < (DR)	0	0
(SR) = (DR)	0	1
(SR) > (DR)	1	0
Unused Bit Setting	1	1

Execution Time: 1.25 microseconds

Symbolic Coding: The assembly language coding format for the RCA instruction is as follows:

Label		Operation		Operand		Comment
[label]	￠	RCA	￠	sreg, dreg	￠	[comment]

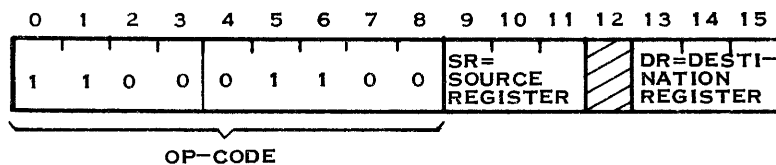
where "sreg" and "dreg" are expressions that address the source and destination registers, respectively, in accordance with table 2-2. The special case when "dreg" equals eight is covered in the "Description" paragraph.

Example:

```
S   EQU   4        (S) = 1054₁₆ }   Status register bits 0 and 1
X   EQU   2   =>                }   set to 00
    :              (X) = B666₁₆ }
    RCA   X,S
```

3.6.5 REGISTER COMPARE LOGICAL (RCL)

Machine Format:



Instruction Execution: (SR) : (DR), logically

Description: Perform a logical compare (unsigned numbers) between the contents of the register specified by the SR field and the contents of the register specified by the DR field. The status register bits are set to indicate the result of the compare as detailed in paragraph 3.6.4. If bit 12 of the machine format is set to one and bits 13 to 15 are zeroed, the status register is specified as the destination register. In this case the instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled. Interrupts, other than internal, are inhibited for one instruction following this special case of the RCL instruction.

Status Affected: Refer to paragraph 3.6.4.

Execution Time: 1.25 microseconds

Symbolic Coding: The assembly language coding format for the RCL instruction is as follows:

Label		Operation		Operand		Comment
[label]	b̸	RCL	b̸	sreg, dreg	b̸	[comment]

where "sreg" and "dreg" are expressions that address the source and destination registers, respectively, in accordance with table 2-2. The special case when "dreg" equals eight is covered in the "Description" paragraph.

Example:

RCL 2,4 ⇒ (S) = 1054_{16} } Status register bits 0 and 1 set to 10
(X) = $B666_{16}$

3.7 SKIP INSTRUCTIONS

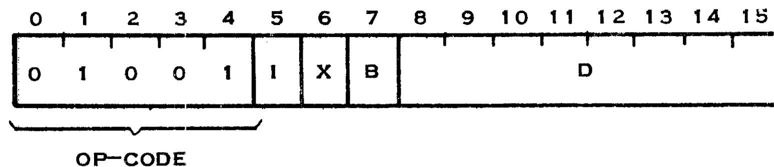
The skip instructions listed in table 3-1 are described in the following paragraphs.

CAUTION

When a skip is taken, only one word is skipped. For this reason, a double or triple length instruction should not immediately follow a skip instruction.

3.7.1 DECREMENT MEMORY AND TEST (DMT)

Machine Format:



Instruction Execution: (EOA)-1⟶(EOA); skip next word if (EOA) = 0

where EOA is developed in accordance with table 3-3.

Description: Decrement the contents of the effective operand address, EOA, by one and replace the contents of the EOA with the result. If the result is zero, skip the next sequential word. If the IXB fields are 7_{16} (immediately addressing), the displacement field, D, is the EOA.

NOTE

The DMT instruction is typically used for loop control where the contents of some memory location is used as a counter.

Status Affected: None

Execution Time: 2.75 to 3.75 microseconds (refer to Appendix A)

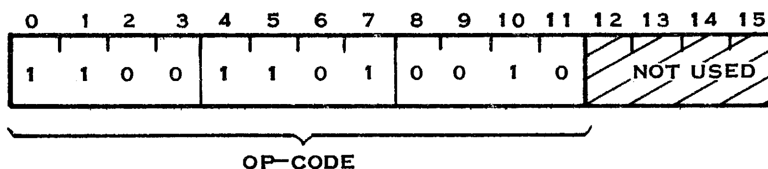
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the DMT instruction. The DMT mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

```
    :
    DMT    BASE, 2                      Before     After
    BRU    $-10     =>  (X)         =  0009₁₆     No change
    BRU    RESET                                          }  Control will
    :                   (BASE+9)    =  0001₁₆     0000₁₆  }  now branch
                                                          }  to RESET
```

3.7.2 SKIP ON EQUAL (SEQ)

Machine Format:



Instruction Execution: $(ST)_{0,1} = 01$, skip next word

$(ST)_{0,1} \neq 01$, execute next word

Description: Skip the next sequential word if the result of the last compare operation was equal (status register bits 0 and 1 set to 01). If the result was something other than equal, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SEQ instruction is as follows:

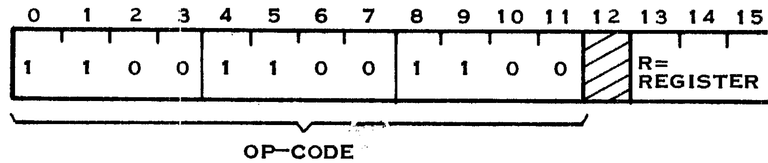
Label	Operation	Operand	Comment
[label] ƀ	SEQ	ƀ	[comment]

Example: The SEQ instruction in the following example will skip a word only if the contents of registers S and X are equal.

```
    :
    RCL   2,4
    SEQ
    :
```

3.7.3 SKIP ON EVEN (SEV)

Machine Format:



Instruction Execution: $(R)_{15}$ = 0, skip next word

$(R)_{15}$ = 1, execute next word

Description: If bit position 15 of the register specified by the R field is zero, skip the next sequential word; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SEV instruction is as follows:

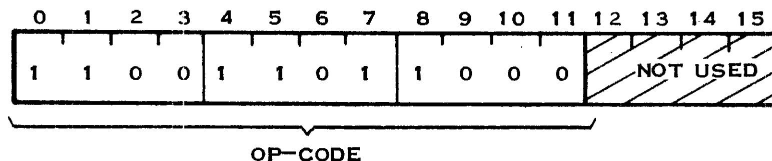
Label		Operation		Operand		Comment	
[label]	ƀ	SEV	ƀ	reg	ƀ	[comment]	where "reg" is an expression that addresses a register in accordance with table 2-2.

Example:

```
A    EQU   0                 Before        After
  :               =>   (A)  = A620₁₆       No change
SEV   A               (PC) = 0132₁₆       0134₁₆  (skip)
```

3.7.4 SKIP ON GREATER THAN OR EQUAL (SGE)

Machine Format:



Instruction Execution: $(ST)_{0,1} \neq 00$, skip next word
$(ST)_{0,1} = 00$, execute next word

Description: If the result of the last compare operation was greater than or equal (status register bits 0 and 1 other than 00), skip the next sequential word; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SGE instruction is as follows:

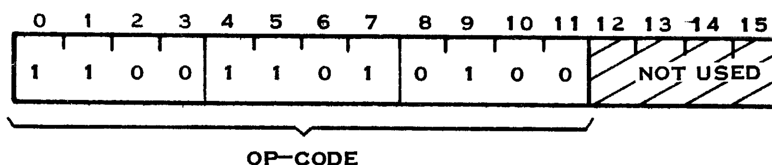
Label	Operation	Operand	Comment
[label] ฿	SGE	฿	[comment]

Example: The SGE instruction in the following example will skip a word only if the content of register X is logically greater than or equal to the content of register S.

```
   :
RCL   2,4
SGE
   :
```

3.7.5 SKIP ON GREATER THAN (SGT)

Machine Format:



Instruction Execution: $(ST)_{0,1} = 10$, skip next word
$(ST)_{0,1} \neq 10$, execute next word

Description: If the result of the last compare operation was greater than (status register bits 0 and 1 set to 10), skip the next word; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SGT instruction is as follows:

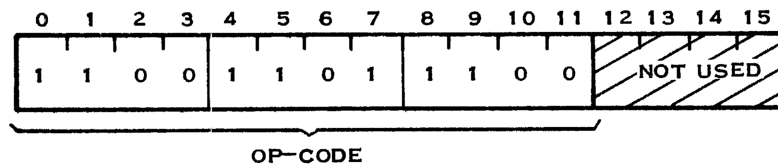
Label	Operation	Operand	Comment
[label] b̸	SGT	b̸	[comment]

Example: The SGT instruction in the following example will skip a word only if the content of register X is logically greater than the content of register S.

```
   :
RCL   2,4
SGT
   :
```

3.7.6 SKIP ON LESS THAN OR EQUAL (SLE)

Machine Format:



Instruction Execution: $(ST)_{0,1} \neq 10$, skip next word

$(ST)_{0,1} = 10$, execute next word

Description: If the result of the last compare operation was less than or equal (status register bits 0 and 1 other than 10), skip the next sequential word; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SLE instruction is as follows:

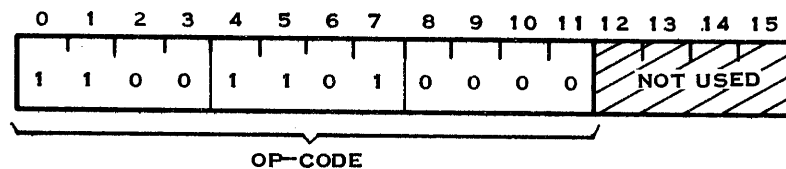
Label	Operation	Operand	Comment
[label] b̸	SLE	b̸	[comment]

Example: The SLE instruction in the following example will skip a word only if the content of register X is logically less than or equal to the content of register S.

```
  :
RCL   2,4
SLE
  :
```

3.7.7 SKIP ON LESS THAN (SLT)

Machine Format:



Instruction Execution: $(ST)_{0,1} = 00$, skip next word

$(ST)_{0,1} \neq 00$, execute next word

Description: If the result of the last compare operation was less than (status register bits 0 and 1 both set to zero), skip the next word; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SLT instruction is as follows:

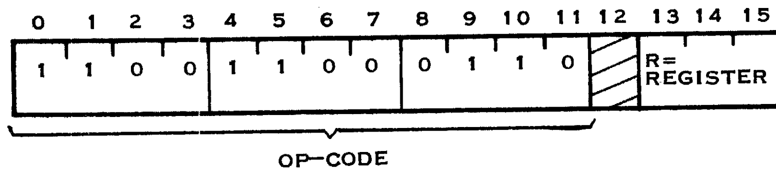
Label	Operation	Operand	Comment
[label] ƀ	SLT	ƀ	[comment]

Example: The SLT instruction in the following example will skip a word only if the content of register X is logically less than the content of register S.

```
  :
RCL   2,4
SLT
  :
```

3.7.8 SKIP ON MINUS (SMI)

Machine Format:



Instruction Execution: $(R)_0 = 1$, skip next word

$(R)_0 = 0$, execute next word

Description: If bit position 0 of the register specified by the R field is one, skip the next word; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SMI instruction is as follows:

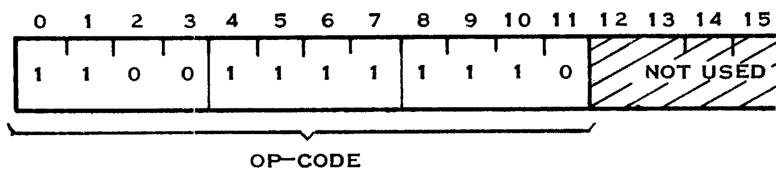
Label		Operation		Operand		Comment	
[label]	ƀ	SMI	ƀ	reg	ƀ	[comment]	where "reg" is an expression that addresses a register in accordance with table 2-2.

Example:

SMI 3			Before	After	
=>	(M)	=	$62AE_{16}$	No change	
	(PC)	=	$23FE_{16}$	$23FF_{16}$	(no skip)

3.7.9 SKIP ON NO CARRY (SNC)

Machine Format:



Instruction Execution: $(ST)_3$ = 0, skip next word

$(ST)_3$ = 1, execute next word

Description: If the last instruction affecting the carry indicator (bit 3 of the status register) did not turn it on, the next word is skipped; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SNC instruction is as follows:

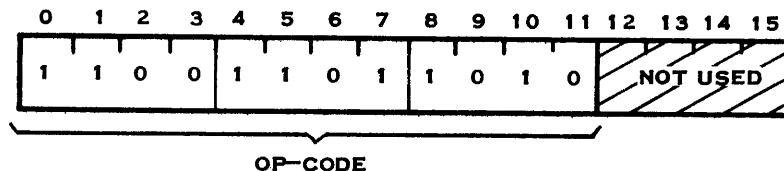
Label		Operation		Operand	Comment
[label]	ƀ	SNC	ƀ		[comment]

Example: The SNC instruction in the following example will skip a word if the sum of register A and the contents of location TABLE did not produce a carry into bit 0.

```
  .
  .
ADD   TABLE
SNC
  .
  .
```

3.7.10 SKIP ON NOT EQUAL (SNE)

Machine Format:



Instruction Execution: $(ST)_{0,1}$ ≠ 01, skip next word

$(ST)_{0,1}$ = 01, execute next word

Description: If the result of the last compare operation was less than or greater than (status register bits 0 and 1 other than 01), skip the next word; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SNE instruction is as follows:

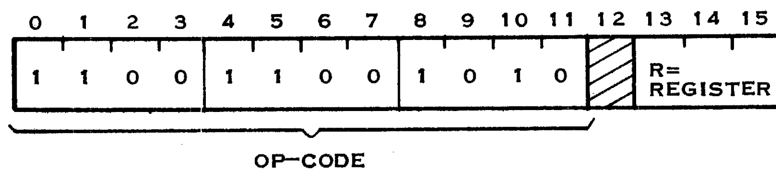
Label	Operation	Operand	Comment
[label] ␢	SNE	␢	[comment]

Example: The SNE instruction in the following example will skip a word if the content of register X is logically less than or greater than the content of register S.

```
  :
RCL   2,4
SNE
  :
```

3.7.11 SKIP ON NOT ALL ONES (SNO)

Machine Format:



Instruction Execution: (R) ≠ $FFFF_{16}$, skip next word

(R) = $FFFF_{16}$, execute next word

Description: If at least one bit position of the register specified by the R field is zero, skip the next word; if all bit positions are ones, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SNO instruction is as follows:

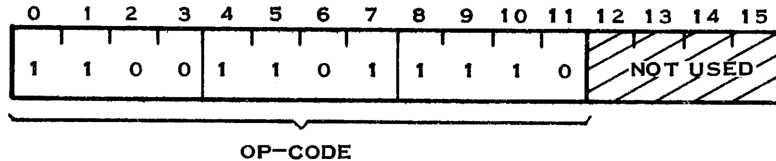
Label	Operation	Operand	Comment	
[label] ␢	SNO ␢	reg ␢	[comment]	where "reg" is an expression that addresses a register in accordance with table 2-2.

Example:

```
X    EQU   2                          Before        After
     :              =>   (X)    =   FFEF16        No change
     SNO   X             (PC)   =   211116        211316  (skip)
```

3.7.12 SKIP ON NO OVERFLOW (SNV)

Machine Format:



Instruction Execution: $(ST)_2$ = 0, skip next word

$(ST)_2$ = 1, execute next word

Description: If the last instruction affecting the overflow indicator (bit 2 of the status register) did not turn it on, the next word is skipped; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SNV instruction is as follows:

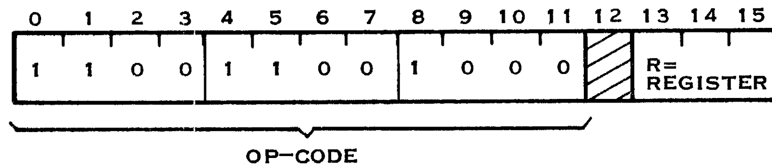
Label		Operation		Operand	Comment
[label]	ƀ	SNV	ƀ		[comment]

Example: The SNV instruction in the following example will skip a word if the sum of register A and the contents of location TABLE did not cause an overflow.

```
 :
ADD   TABLE
SNV
 :
```

3.7.13 SKIP ON NOT ALL ZEROS (SNZ)

Machine Format:



Instruction Execution: (R) ≠ 0, skip next word
(R) = 0, execute next word

Description: If at least one bit position of the register specified by the R field is one, skip the next word; if all bit positions are zeros, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SNZ instruction is as follows:

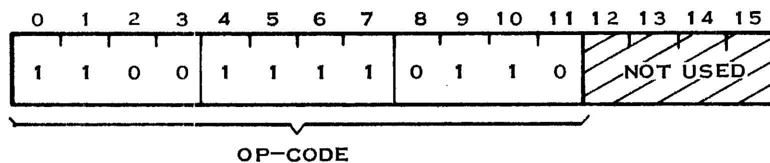
Label		Operation		Operand		Comment	
[label]	ƀ	SNZ	ƀ	reg	ƀ	[comment]	where "reg" is an expression that addresses a register in accordance with table 2-2.

Example:

SNZ 1			Before	After	
⇒	(E)	=	2100_{16}	No change	
	(PC)	=	1103_{16}	1105_{16}	(skip)

3.7.14 SKIP ON CARRY (SOC)

Machine Format:



Instruction Execution: $(ST)_3$ = 1, skip next word

$(ST)_3$ = 0, execute next word

Description: If the last instruction affecting the carry indicator (bit 3 of the status register) turned it on, the next word is skipped; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SOC instruction is as follows:

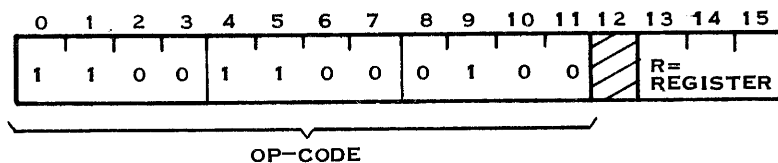
Label		Operation		Operand	Comment
[label]	ɓ	SOC	ɓ		[comment]

Example: The SOC instruction in the following example will skip an instruction if the sum of register A and the contents of location TABLE results in a carry into bit 0.

```
  :
ADD   TABLE
SOC
  :
```

3.7.15 SKIP ON ODD (SOD)

Machine Format:



Instruction Execution: $(R)_{15}$ = 1, skip next word

$(R)_{15}$ = 0, execute next word

Description: If bit position 15 of the register specified by the R field is one, skip the next word; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SOD instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	SOD	ƀ	reg	ƀ	[comment]

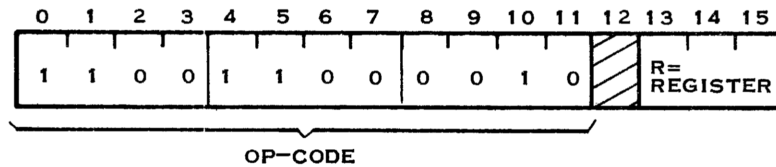
where "reg" is an expression that addresses a register in accordance with table 2-2.

Example:

						Before	After	
X	EQU	2	⇒	(X)	=	0004_{16}	No change	
	:							
	SOD	X		(PC)	=	0010_{16}	0011_{16}	(no skip)

3.7.16 SKIP ON ALL ONES (SOO)

Machine Format:



Instruction Execution: (R) = $FFFF_{16}$, skip next word

(R) ≠ $FFFF_{16}$, execute next word

Description: If all bit positions of the register specified by the R field are one, skip the next word; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SOO instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	SOO	ƀ	reg	ƀ	[comment]

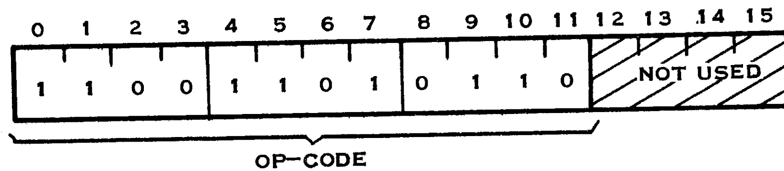
where "reg" is an expression that addresses a register in accordance with table 2-2.

Example:

					Before	After	
SOO	0	⇒	(A)	=	$FFFF_{16}$	No change	
			(PC)	=	0101_{16}	0103_{16}	(skip)

3.7.17 SKIP ON OVERFLOW (SOV)

Machine Format:



Instruction Execution: $(ST)_2$ = 1, skip next word

$(ST)_2$ = 0, execute next word

Description: If the last instruction affecting the overflow indicator (bit 2 of the status register) turned the indicator on, the next word is skipped; otherwise, the next word is executed.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SOV instruction is as follows:

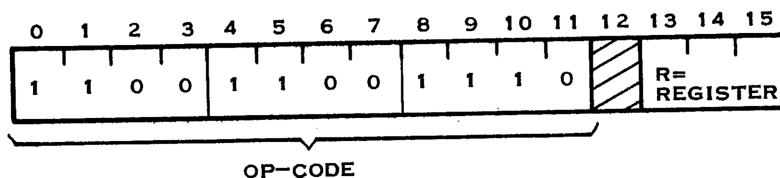
Label		Operation	Operand	Comment
[label]	ƀ	SOV	ƀ	[comment]

Example: The SOV instruction in the following example will skip a word if the sum of register A and the contents of location TABLE causes an overflow.

```
    :
ADD   TABLE
SOV
    :
```

3.7.18 SKIP ON PLUS (SPL)

Machine Format:



Instruction Execution: $(R)_0$ = 0, skip next word

$(R)_0$ = 1, execute next word

Description: If bit position zero of the register specified by the R field is zero, skip the next word; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SPL instruction is as follows:

Label	Operation	Operand	Comment
[label] ƀ	SPL ƀ	reg ƀ	[comment]

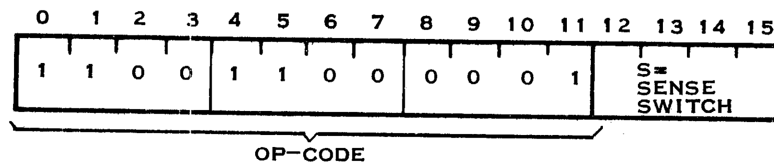
where "reg" is an expression that addresses a register in accordance with table 2-2.

Example:

					Before	After	
L	EQU	5	⇒	(L) =	$F32B_{16}$	No change	
	:						
	SPL	L		(PC) =	0908_{16}	0909_{16}	(no skip)

3.7.19 SKIP ON SENSE SWITCH EQUAL (SSE)

Machine Format:



Instruction Execution: Refer to "description" paragraph.

Description: The S field bits of the machine format correspond to the computer front panel sense switches as follows:

Sense Switch	S Field Bit
1	12
2	13
3	14
4	15

Test only the sense switches whose corresponding S field bits are one. If the tested switches are on (up position), skip the next word; otherwise, execute the next word. If all S field bits are zero, SSE will always skip and SSN will never skip.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SSE instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	SSE	ƀ	ss	ƀ	[comment]

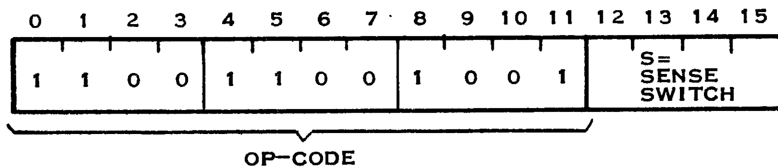
where "ss" is an expression that specifies the sense switches to be tested.

Example: The following SSE instruction will skip a word if sense switches 2 and 3 are on (switches 1 and 4 are not tested).

```
    :
SSE 6
    :
```

3.7.20 SKIP ON SENSE SWITCH NOT EQUAL (SSN)

Machine Format:



Instruction Execution: Refer to "description" paragraph.

Description: Refer to paragraph 3.7.19 for the relationship between the machine format S field bits and the computer front panel sense switches. Test only the sense switches whose corresponding S field bits are one. If any of the test switches are off (down position), skip the next word; otherwise, execute the next word.

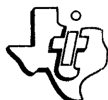
Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SSN instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	SSN	ƀ	ss	ƀ	[comment]

where "ss" is an expression that specifies the sense switches to be tested.

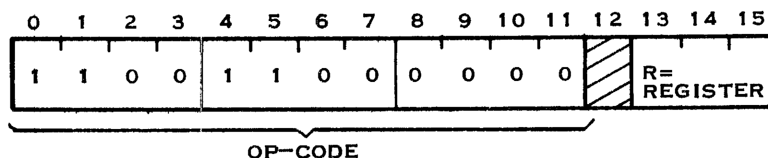


Example: The following SSN instruction will skip a word if sense switch 1 is off (switches 2, 3, and 4 are not tested).

```
    :
    SSN    8
    :
```

3.7.21 SKIP ON ZERO (SZE)

Machine Format:



Instruction Execution: (R) = 0, skip next word
(R) ≠ 0, execute next word

Description: If the content of the register specified by the R field is zero, skip the next word; otherwise, execute the next word.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SZE instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	SZE	ƀ	reg	ƀ	[comment]

where "reg" is an expression that addresses a register in accordance with table 2-2.

Example:

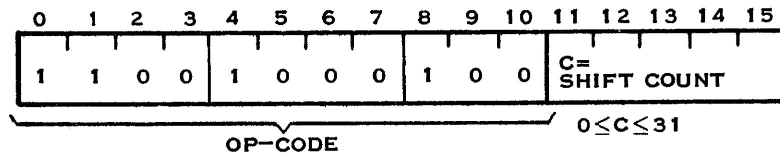
						Before	After
B	EQU	6					
	:		=>	(B)	=	0010_{16}	No change
	SZE	B		(PC)	=	1188_{16}	1189_{16}

3.8 SHIFT INSTRUCTIONS

The shift instructions listed in table 3-1 are described in the following paragraphs.

3.8.1 ARITHMETIC LEFT SHIFT REGISTER A (ALA)

Machine Format:



Instruction Execution: Shift (A) left C places; zero fill vacated bits

Description: Shift bits 1 through 15 of register A to the left the number of bit positions specified by the C field. The sign bit (bit 0) of register A is not affected by the shift. Bit positions vacated are filled with zeros and bits shifted off the left end (from bit 1) are lost. If the C field is zero, no shift takes place.

Status Affected: If the sign bit and bit 1 of register A differ at any time during the shift operation, the overflow indicator (bit 2 of the status register) is turned on; otherwise, it is turned off. In either case, the sign bit is not affected.

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the ALA instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	ALA	ƀ	count	ƀ	[comment]

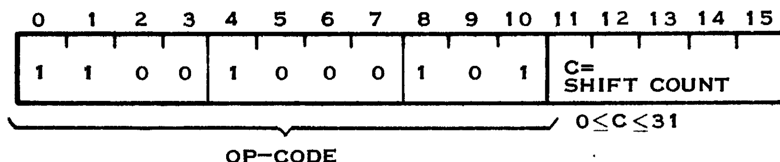
where "count" is an expression that specifies the shift count.

Example:

ALA 5 ⇒ (A) = Before $537B_{16}$ After $6F60_{16}$ (the overflow indicator is turned on)

3.8.2 ARITHMETIC LEFT SHIFT DOUBLE (ALD)

Machine Format:



Instruction Execution: Shift (A, E) left C places; zero fill vacated bits

Description: Shift the double-length word formed by bits 1 through 15 of both registers A and E to the left the number of bit positions specified by the C field. The sign bits (bit 0) of registers A and E are not involved in the shift. Bit 0 of register E is forced to agree with bit 0 of register A and bits shifted out of bit 1 of register E are shifted into bit 15 of register A. Bit positions vacated by the shift are filled with zeros and bits shifted off the left end (bit 1 of register A) are lost. If the C field is zero, no shift takes place but the sign of register E is forced to agree with the sign of register A.

Status Affected: If the sign bit and bit 1 of register A differ at any time during the shift operation, the overflow indicator (bit 2 of the status register) is turned on; otherwise, it is turned off. In either case, the sign bit is not affected.

Execution Time: 1.00 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the ALD instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	ALD	ƀ	count	ƀ	[comment]

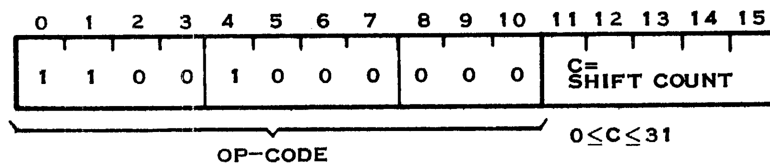
where "count" is an expression that specifies the shift count.

Example:

			Before (Hex)	After (Hex)	
ALD 10	=>	(A, E) =	C3C1, 86A1	8435, 8400	(the overflow indicator is turned on)

3.8.3 ARITHMETIC RIGHT SHIFT REGISTER A (ARA)

Machine Format:



Instruction Execution: Shift (A) right C places; sign fill vacated bits

Description: Shift the contents of register A to the right the number of bit positions specified by the C field. Bit positions vacated are filled with the original sign bit (bit 0) and bits shifted off the right end are lost. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the ARA instruction is as follows:

Label	Operation	Operand	Comment
[label] ƀ	ARA	ƀ count	ƀ [comment]

where "count" is an expression that specifies the shift count.

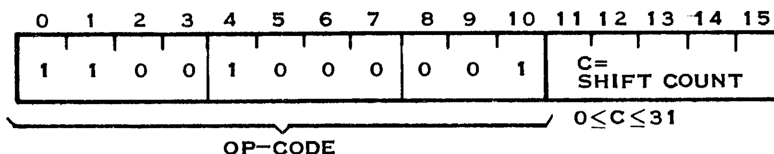
Example:

ARA 3

	Before	After
⇒ (A) =	8321_{16}	$F064_{16}$

3.8.4 ARITHMETIC RIGHT SHIFT DOUBLE (ARD)

Machine Format:



Instruction Execution: Shift (A, E) right C places; sign fill vacated bits

Description: Shift the double-length word formed by registers A and E to the right the number of bit positions specified by the C field. Bit 0 of register E is forced to agree with bit 0 of register A and bits shifted out of bit 15 of register A are shifted into bit 1 of register E. Bit positions vacated by the shift are filled with the original sign bit (bit 0 of register A) and bits shifted off the right end are lost. If the C field is zero, no shift takes place but the sign of register E is forced to agree with the sign of register A.

Status Affected. None

Execution Time: 1.00 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the ARD instruction is as follows:

Label	Operation	Operand	Comment
[label] ƀ	ARD	ƀ count	ƀ [comment]

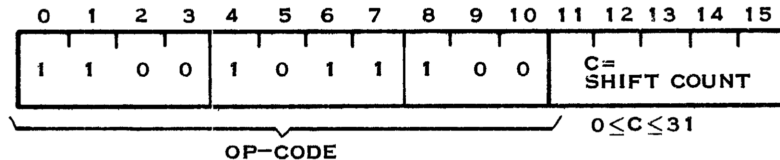
where "count" is an expression that specifies the shift count.

Example:

FIVE	EQU	5			Before (Hex)	After (Hex)
	.		⇒	(A, E) =	2F03, 1100	0178, 0C88
	.					
	ARD	FIVE				

3.8.5 CIRCULAR LEFT SHIFT DOUBLE (CLD)

Machine Format:



Instruction Execution: Shift (A, E) left C places, circularly

Description: Shift the double-length word formed by registers A and E to the left the number of bit positions specified by the C field. Bits shifted out of bit 0 of register A are shifted into bit 15 of register E. Bits shifted out of bit 0 of register E are shifted into bit 15 of register A. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the CLD instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	CLD	ƀ	count	ƀ	[comment]

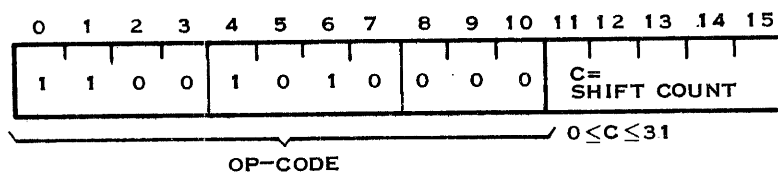
where "count" is an expression that specifies the shift count.

Example:

CLD	8			Before (Hex)	After (Hex)
		⇒	(A, E) =	5350, 4F54	504F, 5453

3.8.6 CIRCULAR RIGHT SHIFT REGISTER A (CRA)

Machine Format:



Instruction Execution: Shift (A) right C places, circularly

Description: Shift the contents of register A to the right the number of bit positions specified by the C field. Bits shifted out of position 15 are shifted into position 0. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the CRA instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	CRA	ƀ	count	ƀ	[comment]

where "count" is an expression that specifies the shift count.

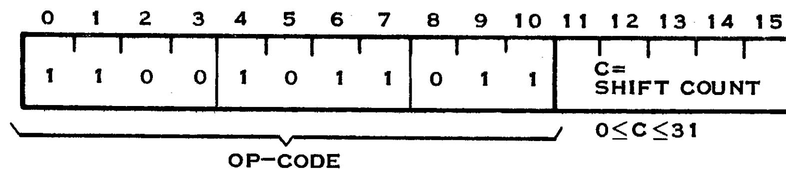
Example:

```
FOUR    EQU    4
         :
        CRA    FOUR
```

⇒ (A) = Before $FAD9_{16}$ After $9FAD_{16}$

3.8.7 CIRCULAR RIGHT SHIFT REGISTER B (CRB)

Machine Format:



Instruction Execution: Shift (B) right C places, circularly

Description: Shift the contents of register B to the right the number of bit positions specified by the C field. Bits shifted out of position 15 are shifted into position 0. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the CRB instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	CRB	ƀ	count	ƀ	[comment]

where "count" is an expression that specifies the shift count.

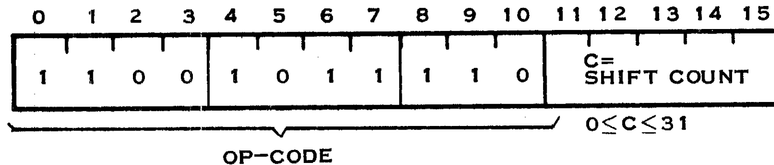
Example:

CRB 15

	Before	After
⇒ (B) =	0105_{16}	$020A_{16}$

3.8.8 CIRCULAR RIGHT SHIFT DOUBLE (CRD)

Machine Format:



Instruction Execution: Shift (A, E) right C places, circularly

Description: Shift the double-length word formed by registers A and E to the right the number of bit positions specified by the C field. Bits shifted out of position 15 of register E are shifted into position 0 of register A. Bits shifted out of position 15 of register A are shifted into position 0 of register E. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the CRD instruction is as follows:

Label		Operation		Operand		Comment
[label]	b̸	CRD	b̸	count	b̸	[comment]

where "count" is an expression that specifies the shift count.

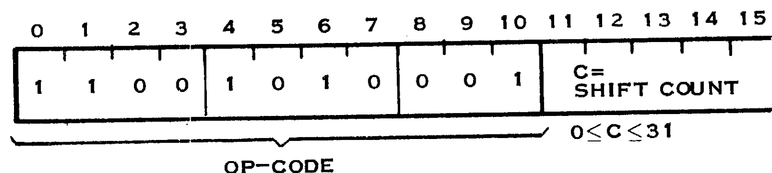
Example:

CRD 6

	Before (Hex)	After (Hex)
⇒ (A, E) =	F6A9, 24B1	C7DA, A492

3.8.9 CIRCULAR RIGHT SHIFT REGISTER E (CRE)

Machine Format:



Instruction Execution: Shift (E) right C places, circularly

Description: Shift the contents of register E to the right the number of bit positions specified by the C field. Bits shifted out of position 15 are shifted into position 0. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the CRE instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	CRE	ƀ	count	ƀ	[comment]

where "count" is an expression that specifies the shift count.

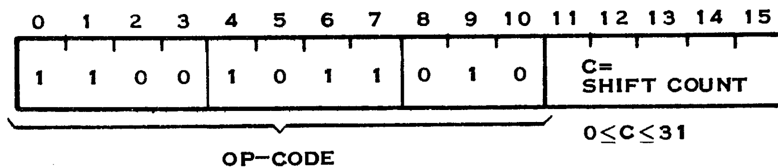
Example:

```
ONE   EQU   1
       :
      CRE   ONE
```

⇒ (E) = Before $24AC_{16}$, After 1256_{16}

3.8.10 CIRCULAR RIGHT SHIFT REGISTER L (CRL)

Machine Format:



Instruction Execution: Shift (L) right C places, circularly

Description: Shift the contents of register L to the right the number of bit positions specified by the C field. Bits shifted out of position 15 are shifted into position 0. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the CRL instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	CRL	ƀ	count	ƀ	[comment]

where "count" is an expression that specifies the shift count.

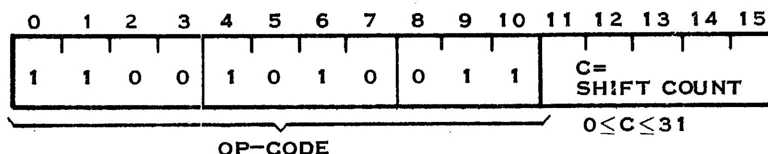
Example:

CRL 5

		Before	After
⇒ (L)	=	$62FF_{16}$	$FB17_{16}$

3.8.11 CIRCULAR RIGHT SHIFT REGISTER M (CRM)

Machine Format:



Instruction Execution: Shift (M) right C places, circularly

Description: Shift the contents of register M to the right the number of bit positions specified by the C field. Bits shifted out of position 15 are shifted into position 0. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the CRM instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	CRM	ƀ	count	ƀ	[comment]

where "count" is an expression that specifies the shift count.

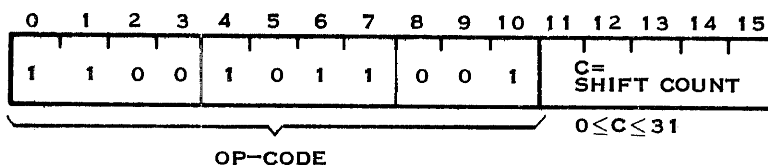
Example:

CRM 8

		Before	After
⇒ (M)	=	2630_{16}	3026_{16}

3.8.12 CIRCULAR RIGHT SHIFT REGISTER S (CRS)

Machine Format:



Instruction Execution: Shift (S) right C places, circularly

Description: Shift the contents of register S to the right the number of bit positions specified by the C field. Bits shifted out of position 15 are shifted into position 0. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the CRM instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	CRS	ƀ	count	ƀ	[comment]

where "count" is an expression that specifies the shift count.

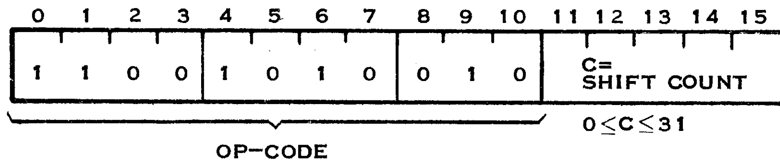
Example:

CRS 2

		Before	After
⇒	(S) =	$CD94_{16}$	3365_{16}

3.8.13 CIRCULAR RIGHT SHIFT REGISTER X (CRX)

Machine Format:



Instruction Execution: Shift (X) right C places, circularly

Description: Shift the contents of register X to the right the number of bit positions specified by the C field. Bits shifted out of position 15 are shifted into position 0. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the CRX instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	CRX	ƀ	count	ƀ	[comment]

where "count" is an expression that specifies the shift count.

Example:

F15 EQU 15

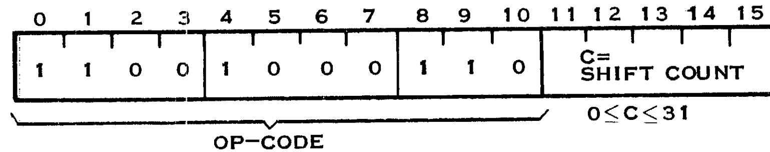
:

CRX F15

⇒ (X) = Before $00B2_{16}$ After 0164_{16}

3.8.14 LOGICAL LEFT SHIFT REGISTER A (LLA)

Machine Format:



Instruction Execution: Shift (A) left C places; zero fill vacated bits

Description: Shift the contents of register A to the left the number of bit positions specified by the C field. Bit positions vacated are filled with zeros and bits shifted off the left end are lost. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the LLA instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	LLA	ƀ	count	ƀ	[comment]

where "count" is an expression that specifies the shift count.

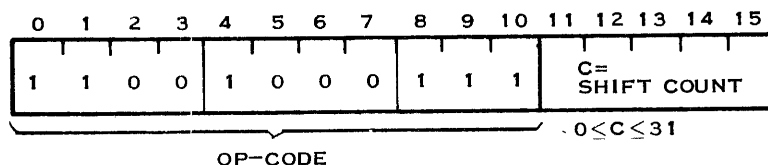
Example:

LLA 4

⇒ (A) = Before $F409_{16}$ After 4090_{16}

3.8.15 LOGICAL LEFT SHIFT DOUBLE (LLD)

Machine Format:



Instruction Execution: Shift (A, E) left C places; zero fill vacated bits

Description: Shift the double-length word formed by registers A and E to the left the number of bit positions specified by the C field. Bit positions vacated are filled with zeros, bits shifted out of position 0 of register A are lost, and bits shifted out of position 0 of register E are shifted into position 15 of register A. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the LLD instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	LLD	ƀ	count	ƀ	[comment]

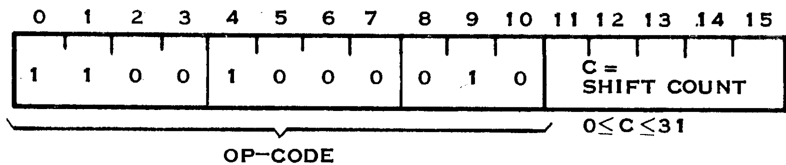
where "count" is an expression that specifies the shift count.

Example:

			Before (Hex)	After (Hex)
LLD 3	⇒	(A, E) =	F2F0, 1108	9780, 8840

3.8.16 LOGICAL RIGHT SHIFT REGISTER A (LRA)

Machine Format:



Instruction Execution: Shift (A) right C places; zero fill vacated bits

Description: Shift the contents of register A to the right the number of bit positions specified by the C field. Bit positions vacated are filled with zeros and bits shifted off the right end are lost. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the LRA instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	LRA	ƀ	count	ƀ	[comment]

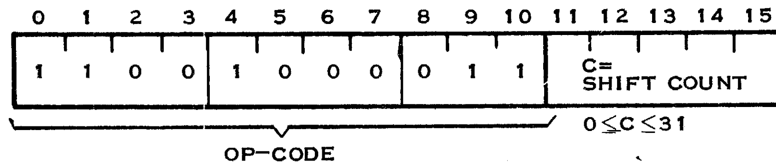
where "count" is an expression that specifies the shift count.

Example:

					Before	After
SEVN	EQU	7				
	:		⇒	(A) =	$3CF1_{16}$	0079_{16}
	LRA	SEVN				

3.8.17 LOGICAL RIGHT SHIFT DOUBLE (LRD)

Machine Format:



Instruction Execution: Shift (A, E) right C places; zero fill vacated bits

Description: Shift the double-length word formed by registers A and E to the right the number of bit positions specified by the C field. Bit positions vacated are filled with zeros, bits shifted out of position 15 of register A are shifted into position 0 of register E, and bits shifted out of position 15 of register E are lost. If the C field is zero, no shift takes place.

Status Affected: None

Execution Time: 0.75 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the LRD instruction is as follows:

Label	Operation	Operand	Comment
[label] b̸	LRD b̸	count b̸	[comment]

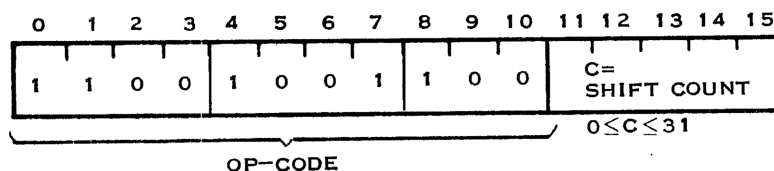
where "count" is an expression that specifies the shift count.

Example:

				Before (Hex)	After (Hex)
LRD	12				
		⇒	(A, E) =	0214, 5F67	0000, 2145

3.8.18 LEFT TEST FOR ONES IN REGISTER A (LTO)

Machine Format:



Instruction Execution: Shift (A) left C places or until a one is found in bit 0; leading zeros count → (X); zero fill vacated bits

Description: Logically shift the contents of register A to the left the number of bit positions specified by the C field or until a one appears in bit 0 of register A. Bit positions vacated by the shift are filled with zeros. If a one is shifted into bit 0, it is set to zero and register X is loaded with a count of the number of zeros shifted out of bit 0. If a one is not found after shifting the number of bits specified by the C field, register X is loaded with the value of the C field. If the C field is zero, bit 0 of register A is complemented and register X remains unchanged.

NOTE

The LTO instruction is commonly used to determine which bits of a status word returned from a peripheral device are set.

Status Affected: None

Execution Time: 1.00 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the LTO instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	LTO	ƀ	count	ƀ	[comment]

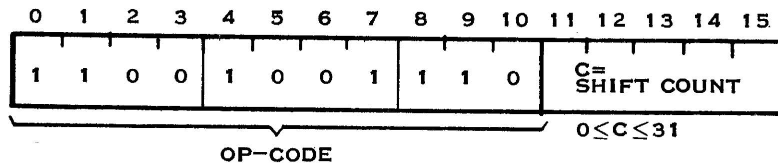
where "count" is an expression that specifies the shift count.

Example:

```
SIX   EQU   6                    Before        After
 :              =>    (A)  =  3C2B16        70AC16
 :
      LTO   SIX       (X)  =  FF0316        000216   ("one" found after
                                                      two shifts)
```

3.8.19 LEFT TEST FOR ZEROS IN REGISTER A (LTZ)

Machine Format:



Instruction Execution: Shift (A) left C places or until a zero is found in bit 0; leading ones count → (X); zero fill vacated bits

Description: Logically shift the contents of register A to the left the number of bit positions specified by the C field or until a zero appears in bit 0 of register A. Bit positions vacated by the shift are filled with zeros. If a zero is shifted into bit 0, it is set to one and register X is loaded with a count of the number of ones shifted out of bit 0. If a zero is not found after shifting the number of bits specified by the C field, register X is loaded with the value of the C field. If the C field is zero, bit 0 of register A is complemented and register X remains unchanged.

Status Affected: None

Execution Time: 1.00 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the LTZ instruction is as follows:

Label		Operation		Operand		Comment
[label]	b̸	LTZ	b̸	count	b̸	[comment]

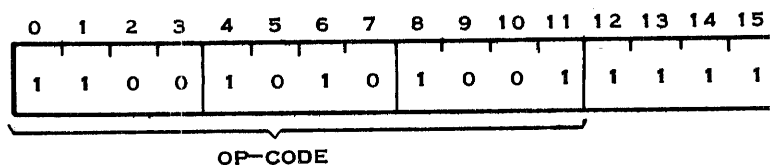
where "count" is an expression that specifies the shift count.

Example:

LTZ 3		Before	After	
⇒	(A) =	$FC02_{16}$	$E010_{16}$	
	(X) =	0080_{16}	0003_{16}	(no "zeros" found in three shifts)

3.8.20 NORMALIZE (NRM)

Machine Format:



Instruction Execution: Shift (A, E) left until $(A)_0 \neq (A)_1$; shift count → (X); zero fill vacated bits

Description: Shift the double-length word formed by registers A and E to the left until bit 0 of register A is different from bit 1 of register A. Bit positions vacated by the shift are filled with zeros and bit 0 of register E is forced to agree with bit 0 of register A. Bits shifted out of bit 1 of register E are shifted into bit 15 of register A. The total number of bits shifted to perform the normalization is loaded in register X. If the contents of registers

A and E are both zero and the NRM instruction is executed, a count of 31 is stored in register X and registers A and E remain at zero. If registers A and E are all ones and the NRM instruction is executed, a count of 30 is stored in register X and registers A and E both contain 8000_{16}.

Status Affected: None

Execution Time: 1.00 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the NRM instruction is as follows:

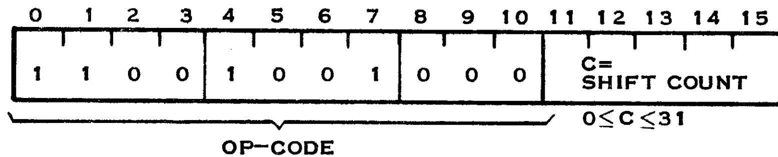
Label		Operation		Comment
[label]	ƀ	NRM	ƀ	[comment]

Example:

NRM			Before (Hex)	After (Hex)
⇒	(A, E)	=	0062, B87A	6238, 7A00
	(X)	=	0AB2	0008

3.8.21 RIGHT TEST FOR ONES IN REGISTER A (RTO)

Machine Format:



Instruction Execution: Shift (A) right C places or until a one appears in bit 15; trailing zeros count → (X); zero fill vacated bits

Description: Logically shift the contents of register A to the right the number of bit positions specified by the C field or until a one appears in bit 15. Bit positions vacated by the shift are filled with zeros. If a one is shifted into bit 15, it is set to zero and register X is loaded with a count of the number of zeros shifted out of bit 15. If a one is not found after shifting the number of bits specified by the C field, register X is loaded with the value of the C field. If the C field is zero, bit 15 of register A is complemented and register X remains unchanged.

Status Affected: None

Execution Time: 1.00 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the RTO instruction is as follows:

Label	Operation	Operand	Comment
[label] b̸	RTO	b̸ count	b̸ [comment]

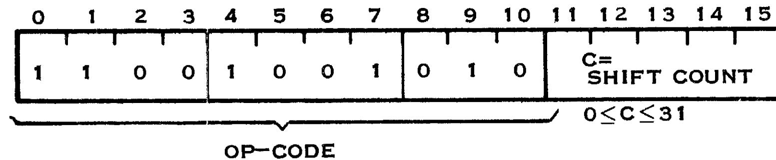
where "count" is an expression that specifies the shift count.

Example:

```
EGHT    EQU    8                Before     After
  :                     (A) =  6BA4₁₆     1AE8₁₆
        RTO    EGHT     (X) =  0905₁₆     0002₁₆
```

3.8.22 RIGHT TEST FOR ZEROS IN REGISTER A (RTZ)

Machine Format:



Instruction Execution: Shift (A) right C places or until a zero appears in bit 15; trailing ones count → (X)

Description: Logically shift the contents of register A to the right the number of bit positions specified by the C field or until a zero appears in bit 15. Bit positions vacated by the shift are filled with zeros. If a zero is shifted into bit 15, it is set to one and register X is loaded with a count of the number of ones shifted out of bit 15. If a zero is not found after shifting the number of bits specified by the C field, register X is loaded with the value of the C field. If the C field is zero, bit 15 of register A is complemented and register X remains unchanged.

Status Affected: None

Execution Time: 1.00 + (shift count/4) microseconds

Symbolic Coding: The assembly language coding format for the RTZ instruction is as follows:

Label	Operation	Operand	Comment
[label] b̸	RTZ	b̸ count	b̸ [comment]

where "count" is an expression that specifies the shift count.

Example:

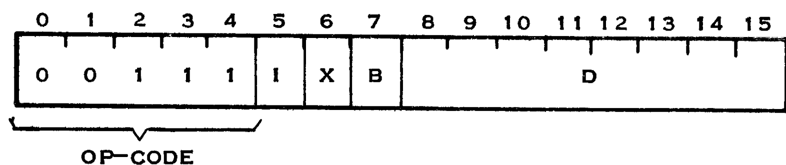
			Before	After
RTZ	5	(A) =	$F601_{16}$	$7B01_{16}$
		(X) =	$FFFF_{16}$	0001_{16}

3.9 LOGICAL INSTRUCTIONS

The logical instructions listed in table 3-1 are described in the following paragraphs.

3.9.1 LOGICAL AND WITH REGISTER A (AND)

Machine Format:



Instruction Execution: (A) AND (EOA) → (A) where EOA is developed in accordance with table 3-3.

Description: Perform a bit-by-bit logical AND between the contents of register A and the contents of the effective operand address, EOA. Place the result in register A. If the IXB fields are 7_{16} (immediate addressing), the operand to be AND'ed with register A consists of zeros in bits 0 to 7 and the displacement field, D, in bits 8 to 15. The Logical AND operation is defined as follows:

(A) Bit	(EOA) Bit	Result
0	0	0
0	1	0
1	0	0
1	1	1

Status Affected: None

Execution Time: 0.75 to 2.75 microseconds (refer to Appendix A)

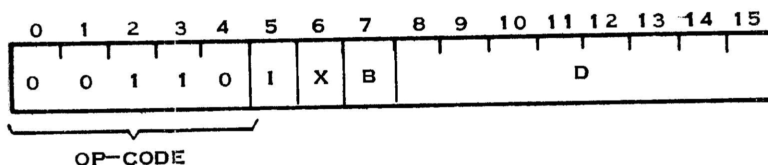
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the AND instruction. The AND mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

				Before	After
MASK AND =>B6	⇒	(A)	=	$F637_{16}$	0036_{16}
		(MASK)	=	$3FB6_{16}$	No change

3.9.2 LOGICAL OR WITH REGISTER A (IOR)

Machine Format:



Instruction Execution: (A) OR (EOA) → (A)

Description: Perform a bit-by-bit logical OR between the contents of register A and the contents of the effective operand address, EOA. Place the result in register A. If the IXB fields are 7_{16} (immediate addressing), the operand to be OR'ed with register A consists of zeros in bits 0 to 7 and the displacement field, D, in bits 8 to 15. The logical OR operation is defined as follows:

(A) Bit	(EOA) Bit	Result
0	0	0
0	1	1
1	0	1
1	1	1

Status Affected: None

Execution Time: 0.75 to 2.75 microseconds (refer to Appendix A)

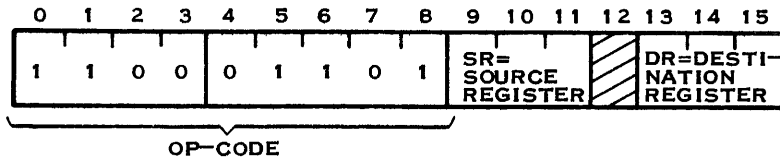
Symbolic Coding: Refer to table 3-3 for the assembly language coding formats available with the IOR instruction. The IOR mnemonic replaces the MNU operation field (in table 3-3) and optional label and comment fields may be used.

Example:

				Before	After	
IOR HEX, 2	⇒	(A)	=	0108_{16}	3138_{16}	where, (X) = 0018_{16}
		(HEX + 18_{16})	=	3030_{16}	No change	

3.9.3 REGISTER AND (RAN)

Machine Format:



Instruction Execution: (SR) AND (DR)⟶(DR)

Description: Perform a bit-by-bit logical AND between the contents of the registers specified by the SR and DR fields. Place the result in the register specified by the DR field. The logical AND operation is defined in paragraph 3.9.1. If bit 12 of the machine format is set to one and bits 13 to 15 are zeroed, the status register is specified as the destination register. In this case the instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled. Interrupts, other than internal, are inhibited for one instruction following this special case of the RAN instruction.

Status Affected: None

Execution Time: 1.25 microseconds

Symbolic Coding: The assembly language coding format for the RAN instruction is as follows:

Label		Operation		Operand		Comment
[label]	b̸	RAN	b̸	sreg, dreg	b̸	[comment]

where "sreg" and "dreg" are expressions that address the source and destination registers, respectively, in accordance with table 2-2. The special case when "dreg" equals eight is covered in the "Description" paragraph.

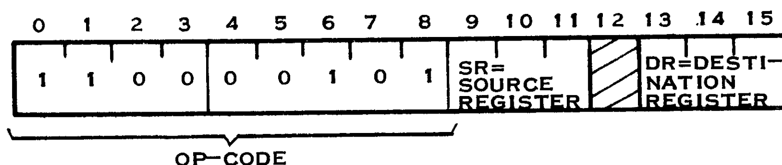
Example:

RAN 0,3

		Before	After
⇒	(M) =	$B8A5_{16}$	0820_{16}
	(A) =	$0F70_{16}$	No change

3.9.4 REGISTER EXCLUSIVE OR (REO)

Machine Format:



Instruction Execution: (SR) exclusive OR (DR) → (DR)

Description: Perform a bit-by-bit logical exclusive OR between the contents of the registers specified by the SR and DR fields. Place the result in the register specified by the DR field. The exclusive OR operation is defined as follows:

(SR) Bit	(DR) Bit	Result
0	0	0
0	1	1
1	0	1
1	1	0

If bit 12 of the machine format is set to one and bits 13 to 15 are zeroed, the status register is specified as the destination register. In this case the instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled. Interrupts, other than internal, are inhibited for one instruction following this special case of the REO instruction.

Status Affected: None

Execution Time: 1.25 microseconds

Symbolic Coding: The assembly language coding format for the REO instruction is as follows:

Label		Operation		Operand		Comment
[label]	ɓ	REO	ɓ	sreg, dreg	ɓ	[comment]

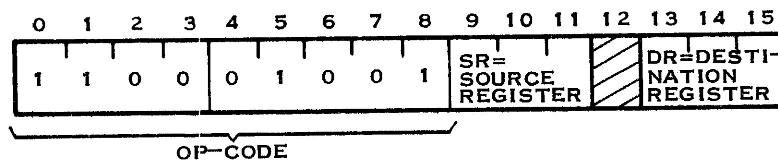
where "sreg" and "dreg" are expressions that address the source and destination registers, respectively, in accordance with table 2-2. The special case when "dreg" equals eight is covered in the "Description" paragraph.

Example:

						Before	After
A	EQU	0					
S	EQU	4	⇒	(S)	=	3862_{16}	$63C3_{16}$
	:						
	REO	A, S		(A)	=	$5BA1_{16}$	No change

3.9.5 REGISTER OR (ROR)

Machine Format:



Instruction Execution: (SR) OR (DR) → (DR)

Description: Perform a bit-by-bit logical OR between the contents of the registers specified by the SR and DR fields. Place the result in the register specified by the DR field. The logical OR operation is defined in paragraph 3.9.2. If bit 12 of the machine format is set to one and bits 13 to 15 are zeroed, the status register is specified as the destination register. In this case the instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled. Interrupts, other than internal, are inhibited for one instruction following this special case of the ROR instruction.

Status Affected: None

Execution Time: 1.25 microseconds

Symbolic Coding: The assembly language coding format for the ROR instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	ROR	ƀ	sreg, dreg	ƀ	[comment]

where "sreg" and "dreg" are expressions that address the source and destination registers, respectively, in accordance with table 2-2. The special case when "dreg" equals eight is covered in the "Description" paragraph.

Example:

ROR 4,3

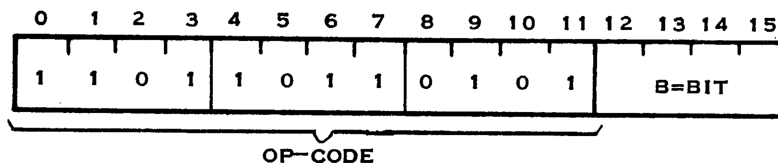
	Before	After
⇒ (M) =	0005_{16}	0035_{16}
(S) =	0030_{16}	No change

3.10 BIT MANIPULATION INSTRUCTIONS

The bit manipulation instructions listed in table 3-1 are described in the following paragraphs.

3.10.1 SET REGISTER A BIT TO ONE (SABO)

Machine Format:



Instruction Execution: $1 \rightarrow (A)_{\text{bit B}}$

Description: Set the bit in register A specified by the B field to one.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SABO instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	SABO	ƀ	bit	ƀ	[comment]

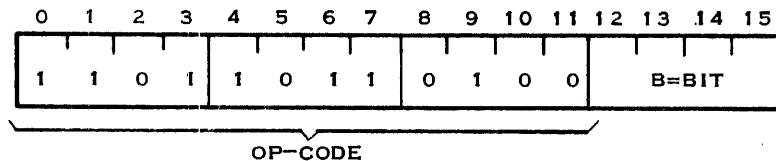
where "bit" is an expression that specifies the bit in register A to be set to one.

Example:

			Before	After
SABO 4	⇒	(A) =	2200_{16}	$2A00_{16}$

3.10.2 SET REGISTER A BIT TO ZERO (SABZ)

Machine Format:



Instruction Execution: $0 \rightarrow (A)_{\text{bit B}}$

Description: Set the bit in register A specified by the B field to zero.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the SABZ instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	SABZ	ƀ	bit	ƀ	[comment]

where "bit" is an expression that specifies the bit in register A to be set to zero.

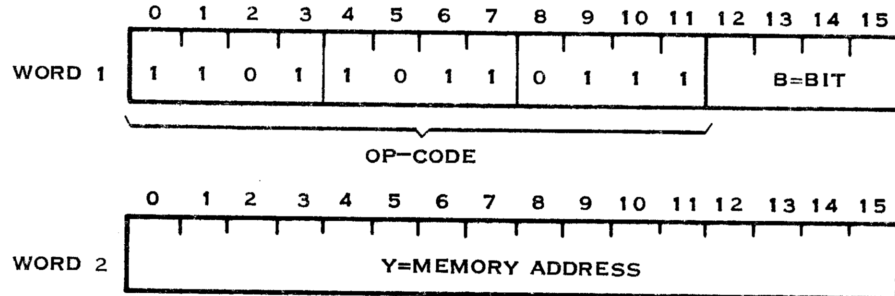
Example:

```
FIFTN   EQU     15
          :
        SABZ    FIFTN
```

		Before	After
⇒	(A) =	$FFFF_{16}$	$FFFE_{16}$

3.10.3 SET MEMORY BIT TO ONE (SMBO)

Machine Format:



Instruction Execution: $1 \rightarrow (Y)_{bit\ B}$

Description: Set the bit, in memory location Y, specified by the B field to one.

Status Affected: None

Execution Time: 3.25 microseconds

Symbolic Coding: The assembly language coding formats for the SMBO instruction are as follows:

NOTE

The FLAG directive in the second coding format is described in Section IV.

Label		Operation		Operand		Comment
[label]	b̸	SMBO	b̸	bit, adrs	b̸	[comment]
			or			
[label]	b̸	FLAG	b̸	adrs	b̸	[comment]
[label]	b̸	SMBO	b̸	bit	b̸	[comment]

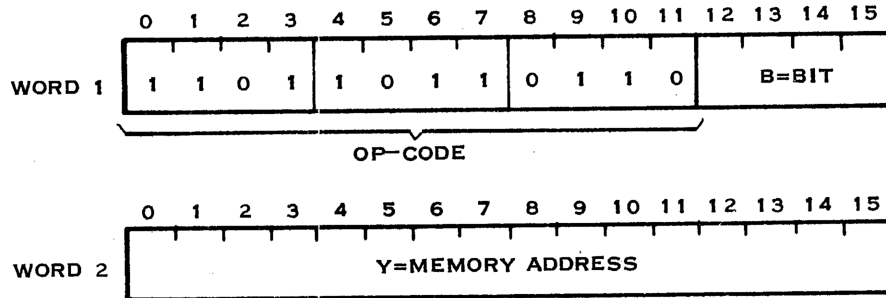
where "bit" and "adrs" are expressions that must be evaluated to specify a bit in memory to be set to one. First, the "bit" expression is divided by 16. The resulting quotient is added to the value of the "adrs" expression to form the memory word address, Y. The remainder becomes the B field and specifies the bit in word Y to be set to one.

Example:

			Before	After
SMBO 17, STATUS				
	⇒	(STATUS+1) =	0013_{16}	4013_{16}

3.10.4 SET MEMORY BIT TO ZERO (SMBZ)

Machine Format:



Instruction Execution: $0 \rightarrow (Y)_{bit\ B}$

Description: Set the bit, in memory location Y, specified by the B field to zero.

Status Affected: None

Execution Time: 3.25 microseconds

Symbolic Coding: The assembly language coding formats for the SMBZ instruction are as follows:

NOTE

The FLAG directive in the second coding format is described in Section IV.

Label		Operation		Operand		Comment
[label]	b̸	SMBZ	b̸	bit, adrs	b̸	[comment]
			or			
[label]	b̸	FLAG	b̸	adrs	b̸	[comment]
[label]	b̸	SMBZ	b̸	bit	b̸	[comment]

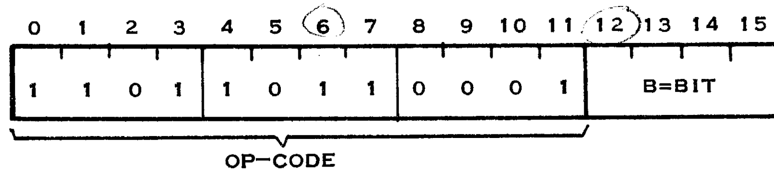
where "bit" and "adrs" are expressions that must be evaluated to specify a bit in memory to be set to zero. First, the value of the "bit" expression is divided by 16. The resulting quotient is added to the value of the "adrs" expression to form the memory word address, Y. The remainder becomes the B field and specifies the bit in word Y to be set to zero.

Example:

			Before	After
SMBZ 15, MEM	⇒	(MEM) =	$2A23_{16}$	$2A22_{16}$

3.10.5 TEST REGISTER A BIT FOR ONE (TABO)

Machine Format:



Instruction Execution: $(A)_{bit\ B} = 1$; skip next word
$(A)_{bit\ B} = 0$; execute next word

Description: If the bit in register A specified by the B field is a one, skip the next word. If the bit is a zero, execute the next word.

Status Affected: None

Execution Time: 1.25 microseconds

Symbolic Coding: The assembly language coding format for the TABO instruction is as follows:

Label		Operation		Operand		Comment
[label]	␢	TABO	␢	bit	␢	[comment]

where "bit" is an expression that specifies the bit in register A to be tested.

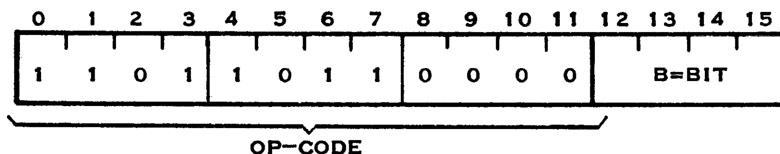
Example:

TABO 6

		Before	After
⇒ (A)	=	$02A3_{16}$	No change
(PC)	=	1179_{16}	$117B_{16}$

3.10.6 TEST REGISTER A BIT FOR ZERO (TABZ)

Machine Format:



Instruction Execution: $(A)_{bit\ B} = 0$; skip next word
$(A)_{bit\ B} = 1$; execute next word

Description: If the bit in register A specified by the B field is zero, skip the next word. If the bit is one, execute the next word.

Status Affected: None

Execution Time: 1.25 microseconds

Symbolic Coding: The assembly language coding format for the TABZ instruction is as follows:

Label		Operation		Operand		Comment
[label]	b̸	TABZ	b̸	bit	b̸	[comment]

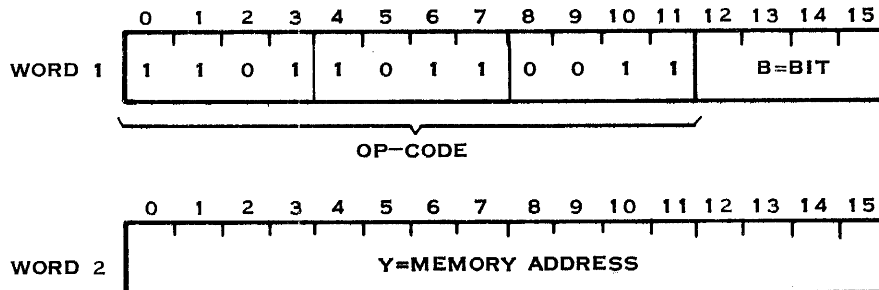
where "bit" is an expression that specifies the bit in register A to be tested.

Example:

					Before	After
SEVN	EQU	7				
	:		⇒	(A) =	$F5C6_{16}$	No change
	TABZ	SEVN		(PC) =	1311_{16}	1312_{16}

3.10.7 TEST MEMORY BIT FOR ONE (TMBO)

Machine Format:



Instruction Execution: $(Y)_{bit\ B} = 1$; skip next word

$(Y)_{bit\ B} = 0$; execute next word

Description: If the bit, in memory location Y, specified by the B field is one, skip the next word. If the bit is zero, execute the next word.

Status Affected: None

Execution Time: 2.75 microseconds

Symbolic Coding: The assembly language coding formats for the TMBO instruction are as follows:

NOTE

The FLAG directive in the second coding format is described in Section IV.

Label		Operation		Operand		Comment
[label]	ƀ	TMBO	ƀ	bit, adrs	ƀ	[comment]
			or			
[label]	ƀ	FLAG	ƀ	adrs	ƀ	[comment]
[label]	ƀ	TMBO	ƀ	bit	ƀ	[comment]

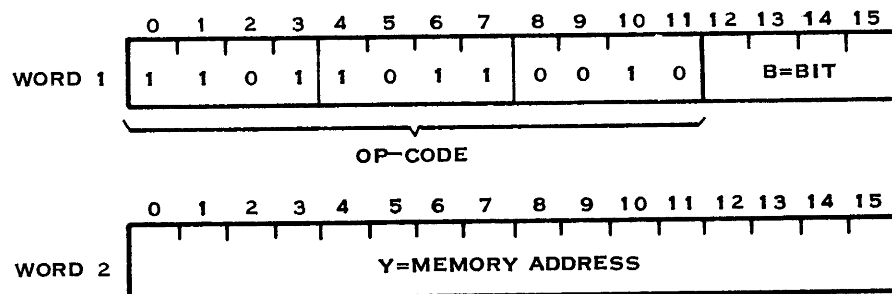
where "bit" and "adrs" are expressions that must be evaluated to specify a bit in memory to be tested. First, the value of the "bit" expression is divided by 16. The resulting quotient is added to the value of the "adrs" expression to form the memory word address, Y. The remainder becomes the B field and specifies the bit in word Y to be tested.

Example:

			Before	After
TMBO 4, TEST	⇒	(TEST) =	0800_{16}	No change
		(PC) =	$2AEF_{16}$	$2AF1_{16}$

3.10.8 TEST MEMORY BIT FOR ZERO (TMBZ)

Machine Format:



Instruction Execution: $(Y)_{\text{bit B}} = 0$; skip next word

$(Y)_{\text{bit B}} = 1$; execute next word

Description: If the bit, in memory location Y, specified by the B field is zero, skip the next word. If the bit is one, execute the next word.

Status Affected: None

Execution Time: 2.75 microseconds

Symbolic Coding: The assembly language coding formats for the TMBZ instruction are as follows:

NOTE

The FLAG directive in the second coding format is described in Section IV.

Label		Operation		Operand		Comment
[label]	ƀ	TMBZ	ƀ	bit, adrs	ƀ	[comment]
			or			
[label]	ƀ	FLAG	ƀ	adrs	ƀ	[comment]
[label]	ƀ	TMBZ	ƀ	bit	ƀ	[comment]

where "bit" and "adrs" are expressions that must be evaluated to specify a bit in memory to be tested. First, the value of the "bit" expression is divided by 16. The resulting quotient is added to the value of the "adrs" expression to form the memory word address, Y. The remainder becomes the B field and specifies the bit in word Y to be tested.

Example:

TMBZ 0, LOC

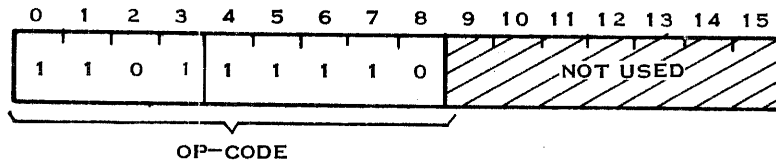
⇒		Before	After
(LOC)	=	$808A_{16}$	No change
(PC)	=	$077D_{16}$	$077E_{16}$

3.11 MOVE INSTRUCTIONS

The move instructions listed in table 3-1 are described in the following paragraphs.

3.11.1 MOVE CHARACTER STRING (MVC)

Machine Format:



Instruction Execution: $(M_1, M_2, \ldots M_n) \rightarrow (Y_1, Y_2, \ldots Y_n)$

where $M_1, M_2, \ldots M_n$ and $Y_1, Y_2, \ldots Y_n$ are byte strings in memory

Description: Move a string of consecutive bytes from one location in memory to a second location in memory. The starting addresses of the two memory locations (S1, B1 moved to S2, B2) and the number of bytes to be moved (BC) are established in general registers as described in paragraph 3.6.1. The content of byte address S1, B1 is moved to S2, B2, and then the two byte addresses are incremented. The byte move and address increment process is repeated until BC bytes have been moved in this manner.

CAUTION

If the displacement between S1, B1 and S2, B2 is less than the length of the byte string (BC) to be moved, and S1, B1 is less than S2, B2, the bytes from the source string (S1, B1) in the overlap addresses will be replaced before they are to be moved. In particular, if the move displacement is one byte, the first byte of the source string will be placed in all of the destination addresses.

Status Affected: None

Execution Time: 4.75 + 2.75 X (no. of bytes moved) microseconds

Symbolic Coding: The assembly language coding format for the MVC instruction is as follows:

Label		Operation		Operand	Comment
[label]	b̸	MVC	b̸		[comment]

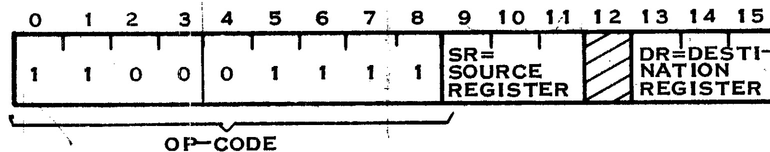
Example:

MVC

			Before (Hex)	After (Hex)
⇒	(A)	=	0000	0000
	(E)	=	0574	0577
	(M)	=	0000	0000
	(S)	=	06A6	06A9
	(X)	=	0003	0000
	(02BA, 02BB)	=	5123, 64AC	No change
	(0353, 0354)	=	F125, 0398	5123, 6498

3.11.2 REGISTER EXCHANGE (REX)

Machine Format:



Instruction Execution: (SR) → (DR); (DR) → (SR)

Description: Exchange the contents of the registers specified by the SR and DR fields. If bit 12 of the machine format is set to one and bits 13 to 15 are zeroed, the status register is specified as the destination register. In this case the instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled. Interrupts other than internal, are inhibited for one instruction following this special case of the REX instruction.

Status Affected: None

Execution Time: 1.50 microseconds

Symbolic Coding: The assembly language coding format for the REX instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	REX	ƀ	sreg, dreg	ƀ	[comment]

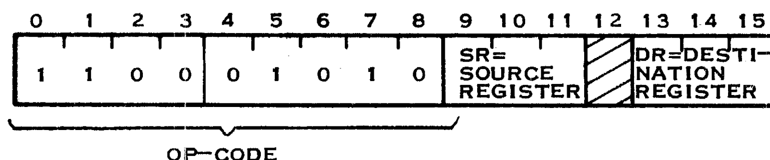
where "sreg" and "dreg" are expressions that address the source and destination registers, respectively, in accordance with table 2-2. The special case when "dreg" equals eight is covered in the "Description" paragraph.

Example:

						Before	After
B	EQU	6					
M	EQU	3	⇒	(M)	=	0032_{16}	$1FA0_{16}$
	•						
	•						
	REX	B, M		(B)	=	$1FA0_{16}$	0032_{16}

3.11.3 REGISTER MOVE (RMO)

Machine Format:



Instruction Execution: (SR)→(DR)

Description: Move the contents of the register specified by the SR field to the register specified by the DR field. The contents of the register specified by the SR field remain unchanged. If bit 12 of the machine format is set to one and bits 13 to 15 are zeroed, the status register is specified as the destination register. In this case the instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled. Interrupts other than internal, are inhibited for one instruction following this special case of the RMO instruction.

Status Affected: None

Execution Time: 1.00 microsecond

Symbolic Coding: The assembly language coding format for the RMO instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	RMO	ƀ	sreg, dreg	ƀ	[comment]

where "sreg" and "dreg" are expressions that address the source and destination registers, respectively, in accordance with table 2-2. The special case when "dreg" equals eight is covered in the "Description" paragraph.

Example:

RMO 5,0

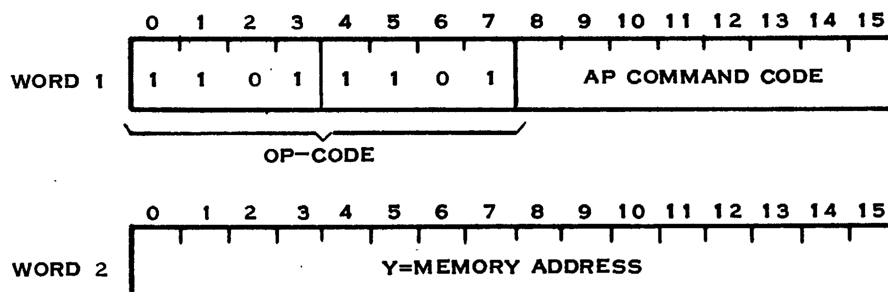
		Before	After
⇒	(A) =	0003_{16}	$1C25_{16}$
	(L) =	$1C25_{16}$	No change

3.12 INPUT/OUTPUT INSTRUCTIONS

The input/output instructions listed in table 3-1 are described in the following paragraphs.

3.12.1 AUXILIARY PROCESSOR INITIATE (API)

Machine Format:



Instruction Execution: (Y) → (AP) ; (AP) → (Y)   where AP is Auxiliary Processor

Description: The computer sends the two words comprising the API instruction to the AP port. The computer then enters a wait state while both the AP and DMAC ports are given access to memory. If the AP port command code in word one of the instruction is not recognized by the controller(s) being used, the computer treats the API instruction as illegal. The AP port is also capable of suspending its operation with appropriate return information stored in memory when the computer recognizes an interrupt. The AP uses the command code in word one of the instruction and the memory address in word two of the instruction to perform operations not included in the 980 instruction set (floating point arithmetic, emulation of other computer instruction sets, etc.). Following a successful AP operation, the AP port issues a release signal to the computer so the computer may resume processing.

NOTE

The AP physically interfaces with the computer at a card slot in the input/output expansion area of the computer chassis.

Status Affected: None

Execution Time: Variable, depending on the complexity of the AP operation.

Symbolic Coding: The assembly language coding format for the API instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	API	ƀ	cmd	ƀ	[comment]
[label]	ƀ	DATA	ƀ	adrs	ƀ	[comment]

where "cmd" is an expression which, when evaluated, identifies to the AP the command to be executed. The expression "adrs" is the symbolic name for a 16-bit memory address containing the necessary information to execute the command.

An optional method of issuing API instructions is through use of the OPD assembler directive (described in Section IV of this manual). The example in the next paragraph illustrates this method in detail.

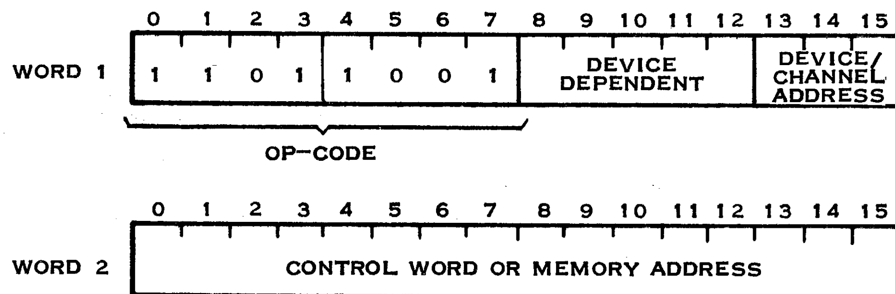
Example: The following example assumes an AP is available to perform a vector dot product. The three OPD directives establish the word one bit patterns of the three API instructions issued later in the extended version of the register-memory format. The extended instructions then reference

symbolic names for the memory addresses that comprise word two of the respective API instructions.

Label	Operation	Operand	Comment
VLD	OPD	DD00, 1	Vector Load Command
VDOT	OPD	DD80, 1	Vector Dot Command
SST	OPD	DDC0, 1	Scalar Store Comment
	.		
	.		
	VLD	Vect1	Load Vect1
	VDOT	Vect2	Vect1 Dot Vect2
	SST	Result	Store in result
	.		
	.		

3.12.2 AUTOMATIC TRANSFER INSTRUCTION (ATI)

Machine Format:



Instruction Execution: External device data →Memory, or
Memory data → External device

Description: The ATI instruction is used to control the Direct Memory Access Channel (DMAC). The first word of the ATI instruction addresses one of eight possible device controllers (bits 13 to 15) and supplies any necessary device dependent data (bits 8 to 12). The second word of the ATI instruction is interpreted by the addressed device controller as a single word functional command or as an address pointing to a list in memory containing command related data. After the second word has been interpreted, the specified DMAC data transfer takes place. The ATI instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled.

NOTE

The ATI instruction and DMAC are covered in more detail in the Model 980 Computer Assembly Language Input/Output manual.

Status Affected: None

Execution Time: 2.50 microseconds

Symbolic Coding: The assembly language coding format for the ATI instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	ATI	ƀ	dev	ƀ	[comment]
[label]	ƀ	DATA	ƀ	adrs	ƀ	[comment]

where "dev" is the symbolic name for the least significant eight bits of word one of the ATI instruction and "adrs" is the symbolic name of the 16-bit address comprising word two.

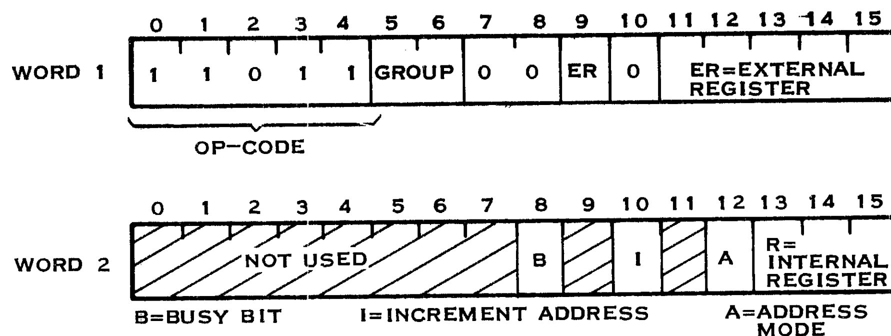
All standard Texas Instruments software addresses the DMAC devices (bits 13 to 15 of word one in the ATI instruction) as follows:

Address (Hex)	Device
0	Fixed-Head Disc or DS 330 Disc
1	Moving-Head Disc
2	Magnetic Tape
5	High-Speed Line Printer

Example: Examples of ATI instructions for the fixed-head disc, moving-head disc, magnetic tape, and high-speed line printer are included in the Model 980 Computer Assembly Language Input/Output manual.

3.12.3 READ DIRECT SINGLE (RDS)

Machine Format:



Instruction Execution: External device data→(R) or ((R))

Description: The RDS instruction uses the input/output data bus to read one word of data from an external device to a register or memory location. The external device is specified by the GROUP and ER fields of word one of the RDS instruction. The GROUP field selects 1 of 4 groups and the ER field picks 1 of 64 external devices in the chosen group. This allows for a maximum of 256 data bus ports, however, in most cases GROUP zero is specified. The destination register or memory location is specified by the A and R fields of word two of the RDS instruction. The R field selects 1 of 8 registers in accordance with table 2-2 and the A field is the associated indirect bit. If the A field is zero, the destination of the read is a register; if the A field is one, the destination of the read is the memory address contained in the selected register. If the A field is one, the I field bit in word two is set to a one or zero to increment or decrement, respectively, the memory address in the selected register each time the RDS instruction is executed. The B field is set to a one when the device addressed by the GROUP and ER fields may not be ready to transfer data when queried by the RDS instruction. If the B field bit is one and no data transfer takes place, the instruction following the RDS instruction is executed. If the B field bit is one and a successful data transfer takes place, the instruction following the RDS instruction is skipped (dependent on physical device - see manual for particular device). If the B field bit is zero, the instruction following the RDS instruction is unconditionally executed. The RDS instruction is considered illegal if the memory protect/privileged instruction feature is enabled.

NOTE

The RDS instruction and input/output data bus are covered in more detail in the Model 980 Computer Input/Output manual.

Status Affected: None

Execution Time: 3.00 to 4.75 microseconds

Symbolic Coding: The assembly language coding format for the RDS instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	RDS	ƀ	dev	ƀ	[comment]
[label]	ƀ	DATA	ƀ	biar	ƀ	[comment]

where "dev" is the symbolic name of a 16-bit number that is OR'ed with the RDS op-code to develop word one of the instruction. "biar" is the symbolic name of a 16-bit number that represents the B, I, A, and R fields of word two.

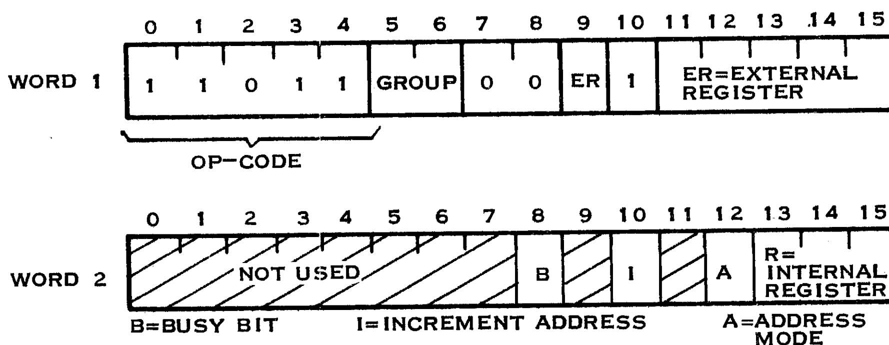
Example: The following example reads a word from the device connected to external register 18_{16} into register A. The busy bit option is also used.

```
    :
RDS     >18
DATA    >80
    :
```

Refer to the Model 980 Computer Assembly Language Input/Output manual for additional examples of the RDS instruction and the standard input/output data bus external register addresses used by Texas Instruments software.

3.12.4 WRITE DIRECT SINGLE (WDS)

Machine Format:



Instruction Execution: (R) or ((R)) → External device

Description: The WDS instruction uses the input/output data bus to write one word of data from a register or memory location to an external device. The source register or memory location is specified by the A and R fields of WDS word two and the destination device is specified by the GROUP and ER fields of WDS word one. These fields along with the B and I fields of WDS word two perform the same function as those described in paragraph 3.12.3 for the RDS instruction. The WDS instruction is restricted, meaning it is considered illegal if the memory protect/privileged instruction feature is enabled.

NOTE

The WDS instruction and input/output data bus are covered in more detail in the Model 980 Computer Input/Output manual.

Status Affected: None

Execution Time: 3.00 to 5.00 microseconds

Symbolic Coding: The assembly language coding format for the WDS instruction is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	WDS	ƀ	dev	ƀ	[comment]
[label]	ƀ	DATA	ƀ	biar	ƀ	[comment]

where "dev" is the symbolic name of a 16-bit number that is OR'ed with the WDS op-code to develop word one of the instruction. "biar" is the symbolic name of a 16-bit number that represents the B, I, A, and R fields of word two.

Example: The following example writes a word in register A to the external device connected to external register 10_{16}. The busy bit option is not used.

```
  :
WDS     >10
DATA    >0
  :
```

Refer to the Model 980 Computer Assembly Language Input/Output manual for additional examples of the WDS instruction and the standard input/output data bus external register addresses used by Texas Instruments software.

SECTION IV

ASSEMBLER CHARACTERISTICS AND DIRECTIVES

4.1 GENERAL

This section describes the Symbolic Assembly Program (SAP) from the user point of view and the 22 assembler directives available to the assembly language programmer. The SAP description covers source program coding fields, object program output, error messages that may accompany the assembly listing, and sample source programs and associated assembly listings. Operation of the two versions of SAP, SAPG and SAP733, is covered in the Model 980 Computer Basic System Use and Operation manual.

4.2 SYMBOLIC ASSEMBLY PROGRAM (SAP)

The two versions of SAP, SAPG and SAP733, are available to translate symbolic assembly language coding into object language acceptable to the Model 980 Computer. The difference between SAPG and SAP733 is due to the media handled. SAPG is a general assembler that handles paper tape, card, magnetic tape, and disc media. SAP733 is used only with cassette media on the 733 ASR data terminal. Figure 4-1 is a sample source main program, written in symbolic assembly language and ready to be punched into cards or

SYMBOLIC CODING FORM

```
    5    10   15   20   25   30   35   40   45

       HED  MODEL 980  MAIN PROGRAM
       IDT  ILLUS     6 CHARS. FOR OBJECT
       ORG  1000      TELL SAP RUN-TIME
       BRS  1000        ORIGIN AND BASE
BASE   DATA 1000
       REF  SUB       EXT. REF. FOR LINKING
START  LDA  BASE      ACTUALLY SET BASE
       RMO  0,6         FOR EXECUTION
       @BRL SUB       ADD 2 NOS. TOGETHER
       DATA ADDR1       ADDR.OF FIRST NO.
       DATA ADDR2       ADDR.OF SECOND NO.
       STA  ANSWER      ANSWER IN REG.A
.      .              MORE EXECUTABLE
.      .                INSTRUCTIONS AND
.      .                ASSEMBLER
.      .                DIRECTIVES.
ADDR1  DATA 7         FIRST NO.
ADDR2  DATA 8         SECOND NO.
ANSWER BSS  1
       END  START
```

Figure 4-1. Source Coded Main Program

paper tape, or otherwise prepared for input. Figure 4-2 is a source subroutine. Source programs input to SAP generate two outputs. The first output is an object program that can be loaded into the computer and executed or linked with other object programs. The object program can be output on cassette, paper tape, or other media. The second output is an assembly listing as depicted in figure 4-3 for the main program and figure 4-4 for the subroutine. Note the following about the assembly listings:

- The items listed under A are an exact reproduction of the handwritten entries on the coding sheet.
- The items under B are a hexadecimal representation of the corresponding instructions and constants as assembled by SAP.
- The items under C show the hexadecimal addresses of the instructions, constants, and areas of storage specified by the programmer.
- The items under D show the decimal line or sequence number of the source statements to be used in case the program is changed.
- Under DX980, the date and time of assembly is obtained and printed in the heading of every sheet of the assembly and placed in characters 18-22 of the IDT record as follows:

18-19	month
20-21	day
22-23	year
24-25	hours
26-27	minutes
28-29	seconds

SAP is a two-pass assembler, meaning it scans the source program twice. During the first pass, the source program is read and a symbol table is generated. This is accomplished with the use of a location counter in the assembler. The location counter keeps track of the storage locations that will be

SYMBOLIC CODING FORM

```
        5    10   15   20   25   30   35   40   45
|----|----|----|----|----|----|----|----|----|

       IDT  SUB       6 CHARS.FOR OBJECT
       DEF  SUB       DEFINE ENTRY POINT FOR
A      EQU  0           LINKING
L      EQU  5         GIVE REGISTERS SYMBOLIC
P      EQU  7           NAMES.
POINT  BSS  2         RESERVE LOCATIONS.
HERE   BSS  2
SUB    RMO  L,A       L POINTS TO FIRST DATA
       STA  POINT       WORD AFTER @BRL
       RIN  A,A       POINTER TO SECOND DATA
       STA  POINT+1     WORD AFTER @BRL
       LDA  *POINT    GET ADDR1
       STA  HERE      STORE ADDRESS IN THIS
       LDA  *POINT+1    SUBROUTINE
       STA  HERE+1    GET AND SAVE ADDR2
       LDA  *HERE     PICK UP FIRST NO.
       ADD  *HERE+1   ADD SECOND NO.
       RIN  L,L       MOVE POINTER PAST DATA
       RIN  L,P         WORDS AND RETURN.
       END  SUB
```

Figure 4-2. Source Coded Subroutine

```
   C     B    D                          A
MODEL 980  MAIN PROGRAM                                                   SHEET 0001

                0001          HED     MODEL 980 MAIN PROGRAM
                0002          IDT     ILLUS     6 CHARS.FOR OBJECT
  03E8          0003          ORG     1000      TELL SAP RUN-TIME
        03E8    0004          BRS     1000        ORIGIN AND BASE
  03E8  03E8    0005   BASE   DATA    1000
                0006          REF     SUB       EXT.REF.FOR LINKING
  03E9  00FE    0007   START  LDA     BASE      ACTUALLY SET BASE
  03EA  C506    0008          RMO     0,6         FOR EXECUTION
  03EB  7400    0009          @BRL    SUB       ADD 2 NOS. TOGETHER
        0000
X 03EC  0000
  03ED  03F0    0010          DATA    ADDR1       ADDR.OF FIRST NO.
  03EE  03F1    0011          DATA    ADDR2       ADDR.OF SECOND NO.
  03EF  8002    0012          STA     ANSWER      ANSWER IN REG.A
                0013   .       .                MORE EXECUTABLE
                0014   .       .                  INSTRUCTIONS AND
                0015   .       .                  ASSEMBLER
                0016   .       .                  DIRECTIVES.
  03F0  0007    0017   ADDR1  DATA    7         FIRST NO.
  03F1  0008    0018   ADDR2  DATA    8         SECOND NO.
  03F2          0019   ANSWER BSS     1
                0020          REF     SUB1
        0001    0021   SUB2   DATA    SUB1
X 03F3  0000
        0000    0022   WORD   COMM    6
        0000    0023          DATA    WORD+2
C 03F4  0002
        03E9    0024          END     START
COMMON  0006


 MODEL 980 MAIN PROGRAM                                                   SHEET 0002

Symbol  { ADDR1    03F0   ADDR2   03F1   ANSWER   03F2     BASE   03E8
 Table  { START    03E9   SUB     0000   SUB1     0001   R SUB2   03F3
        { WORD     0000
  0000 ERRORS
```

NOTES:

1. The symbol table is not generated by SAP733
2. In the left column, P = Program counter relocatable
   X = External reference
   C = Common (to programs)
3. In the symbol table, R = Unreferenced symbol
   U = Undefined (error)
   M = Multidefined
   Q = Multidefined unreferenced
4. A,B,C, and D references at top of page are explained in paragraph 4.2

Figure 4-3. Assembled Main Program

```
                                                              SHEET 0001

              0001          IDT    SUB        6 CHARS.FOR OBJECT
              0002          DEF    SUB        DEFINE ENTRY POINT FOR
         0000 0003  A       EQU    0            LINKING
         0005 0004  L       EQU    5          GIVE REGISTERS SYMBOLIC
         0007 0005  P       EQU    7            NAMES.
P 0000        0006  POINT   BSS    2          RESERVE LOCATIONS.
P 0002        0007  HERE    BSS    2
  0004   C550 0008  SUB     RMO    L,A        L POINTS TO FIRST DATA
  0005   80FA 0009          STA    POINT        WORD AFTER @BRL
  0006   C300 0010          RIN    A,A        POINTER TO SECOND DATA
  0007   80F9 0011          STA    POINT+1      WORD AFTER @BRL
  0008   04F7 0012          LDA    *POINT     GET ADDR1
  0009   80F8 0013          STA    HERE       STORE ADDRESS IN THIS
  000A   04F6 0014          LDA    *POINT+1     SUBROUTINE
  000B   80F7 0015          STA    HERE+1     GET AND SAVE ADDR2
  000C   04F5 0016          LDA    *HERE      PICK UP FIRST NO.
  000D   24F5 0017          ADD    *HERE+1    ADD SECOND NO.
  000E   C355 0018          RIN    L,L        MOVE POINTER PAST DATA
  000F   C357 0019          RIN    L,P          WORDS AND RETURN.
         0004 0020          END    SUB
                                                            SHEET 0002

Symbol { A       0000     HERE   0002     L     0005     P     0007
 Table { POINT   0000     SUB    0004
       { 0000 ERRORS
```

NOTE:

Refer to NOTES in figure 4-3.

Figure 4-4. Assembled Subroutine

required by the object program. When a source statement contains a name, the current setting of the location counter is assigned to the name. Each name and the address assigned to it is placed in the assembler's symbol table. During the second pass, the symbol table is used to complete the assembly, and to produce the object with its assembly listing. If bulk storage is available, SAPG will copy the source to bulk storage during pass one. Since the output from the first pass is used as input data for the second pass, this eliminates the requirement to manually enter the source data twice. SAP733 automatically repositions the cassette source file before entering pass 2 to eliminate any manual repositioning.

4.2.1 SAP CODING LINE FORMAT

The symbolic input line accepted by the assembler may contain a label field, operation field, operand field, and a comment field; or the entire line may be a comment. An input line is the first 64 characters read from a card, or in the case of cassette or paper tape, an input line is a string of characters terminated with a special end-of-line sequence. The Model 980 Computer Basic System Use and Operation manual describes the paper tape end-of-line characters. The end-of-line sequence for cassette consists of a carriage return (CR), line feed (LF), X-OFF (press the CTRL and S keys at the same time), and rub out. The input line may exceed 64 characters, not including the end-of-line characters in the cassette and paper tape case, but only 64 characters are processed and only 59 are printed on the listing to the right of the line number. The input line is free form within the limits listed in the following paragraphs.

4.2.1.1 COMMENT LINES. Comment lines provide the user with the ability to annotate program listings. They are indicated by an initial character which is either a period (.) or an asterisk (*). The remaining characters are arbitrary. The comment line in no way affects the assembly process. The line is merely reproduced in the printed output.

4.2.1.2 LABEL FIELD. Labels (also called symbols or names) are provided for symbolic references to instructions, values, and data. A label is composed of from one to six characters. The first character of a label must be a letter. The remaining may be any characters except the following:

+ Plus	* Asterisk	(Left Paren.	>Greater Than
- Minus	/ Slash	) Right Paren.	, Comma

If a label is used, the first character must begin the input line. The label is terminated by the first space.

At assembly time, the labels are stored as variable length data. One or two character labels require one word of memory, three or four character labels take two words, and five or six characters require three words. Therefore, if the symbol overflow error occurs during assembly, labels should be shortened or omitted.

4.2.1.3 OPERATION FIELD. The operation field describes the required action. It may be an instruction mnemonic or an assembler directive. The field consists of from one to four characters followed by a space or the end-of-line characters. The first character of the operation field must be preceded by at least one space.

4.2.1.4 OPERAND FIELD. The operand field consists of a sequence of expressions separated by commas, and is terminated by a space or the end-of-line characters.

exp_1, exp_2, exp_3

If two commas appear successively, the value of the missing expression is understood to be zero. If the currency symbol ($) appears as an element in an expression, the current value of the assembler's location counter is used as its numeric equivalent.

Expressions may be strings of items separated by arithmetic operators and terminated by a space, comma, or end-of-line characters. The arithmetic operators are:

- Addition +
- Subtraction -
- Multiplication *
- Division /

If two operators appear in succession, a zero item is assumed.

An item consists of a symbolic address, dollar sign ($), or a numeric value. If the first character of an item is not numeric, $, or > , it is assumed to be symbolic. Numeric items may be octal, decimal, or hexadecimal. An octal item is a string of octal characters (0 to 7), the first of which is zero. A decimal item is a string of numeric characters (0 to 9), the first of which is non-zero. A hexadecimal item is a greater than symbol (>) followed by a string of hexadecimal digits (0 to 9 and A to F). When using paper tape input, the back slash (\) may be used in place of > to indicate hexadecimal.

Expressions are evaluated left to right using normal arithmetic precedence; i.e., all multiplications and divisions are performed first in order of occurrence followed by additions and subtractions performed in order of occurrence. All quantities are treated as integers. In division only the quotient is retained and any remainder is discarded. Division by zero is performed as division by one and is not considered as an error. Sample expressions are:

```
JOE+TOM*3/BOB
$+5
LEA-6
5034
XYZ+>F4
```

NOTE

All expressions are acceptable in absolute assemblies, but multiplication and division involving labels is not allowed in relocatable assemblies. Hence, the first sample would cause a relocation error in a relocatable program.

4.2.1.5 COMMENT FIELD. Comments may optionally be written on any line. Any characters that appear between the space that terminates the operand field and the end-of-line characters or card column 64 are treated as commentary. The comment field has no effect on the assembly process.

4.2.2 SEGMENTED SOURCE PROGRAMS

SAPG provides the capability of storing a single source program on more than one physical section of the storage medium, enabling long programs to be conveniently stored on cassette or paper tape. (Segmenting cannot be done to disc files.) To segment a source program, divide it and add the flag record (=) as follows:

```
*       first line of program          )
            .                          )
            .                          )  first segment
            .                          )
*       last line of first segment     )
=                                      )
/*                                     )

*       first line of next segment     )
*       immediately follows last line  )
*       of preceding segment           )
            .                          )  intermediate segment
            .                          )
            .                          )
=                                      )
/*                                     )

            .
            .
            .
        additional intermediate
        segments as needed
            .                          )
            .                          )
            .                          )  last segment
            .                          )
            .                          )
            .                          )
          END                          )
        /*                             )
```

4.2.3 SAP OBJECT FORMAT

The object program output by the assembler is in the form of standard object records used by all system programs in the Basic System. Details of the object records are covered in the Model 980 Computer Basic System Use and Operation manual. Information from the IDT and ORG assembler directives is used to generate the header data. Entry point records, external reference records, and common symbols records are constructed as specified in the DEF, REF, and COML assembler directives, respectively. The required text records are created by the assembler, and the end record is generated from the END directive. No block data records are output by the assembler.

4.2.4 SAP ERROR MESSAGES

The two versions of the assembler (SAPG and SAP733) may detect certain syntax errors in the source program. When an error occurs, a diagnostic message (SAPG) or the message number (SAP733) is printed in the assembly listing adjacent to the line in question. These messages (listed in table 4-1) apply only to the assemblers that operate in the Model 980 Computer. Error messages are printed anyway if the UNL directive is in effect.

4.3 ASSEMBLER DIRECTIVES

In addition to the instruction set presented in Section III of this manual, SAP will accept 22 different assembler directives. The assembler directive formats (name, operand, operation, and comment fields) are similar to the symbolic instructions, but the directives do not directly cause code generation as do the instructions. Instead, the directives are commands to the assembler used to provide for storage allocation, program identification, format control, and other such functions. If labels are used with directives, they are assigned the current location counter value unless otherwise specified in the following paragraphs. The assembler directives are covered in detail in alphabetical order under the paragraph numbers listed in table 4-2. The assembly language coding format accompanying each directive description uses symbols from table 3-2.

Table 4-1. SAP Error Messages

Message Number	Message	Meaning (and Corrective Action)
1	FIELD SZ	Address beyond reach (use @ for extended format)
2	UNDF OP	Undefined operation code (check list of valid of codes)
3	LONG SYM	Symbol > 6 characters
4	MDF O/F	OPD or FRM multiply defined (rename label)
5	FRM > 16	FRM fields contain more than 16 bits
6	CAD > 10	Address expression has > 10 elements
7	UNDF SYM	Symbol not defined (label probably omitted)
8	MDF SYM	Symbol multiply defined (rename labels)
9	RELOC	A relocation error (use only one relocatable label in arithmetic expression, or ORG statement can use only one relocatable label)
10	SYM OVF	Too many symbols have been defined (cut out symbols or divide program)
11	BAD NUM	Numeric element not valid (properly define item in label or address field)
12	IMP R/D	A REF or DEF symbol has been used improperly (REF symbol defined inside and outside the program, DEF symbol not defined in the program)
13	X RF USE	A REF symbol has appeared invalidly in an unrelocatable expression
14	IXB ERR	Address mode error (improper use of IXB field)
15	OPD ERR	No such format number (OPD format numbers 0 to 8)
16	ADR MODE	Illegal addressing mode (improperly written address)

Table 4-2. Model 980 Computer SAP Assembler Directives

Directive Mnemonic	Description	Paragraph No.
BES*	Block Ending Symbol	4.3.1
BRR	Base Register Reset	4.3.2
BRS	Base Register Set	4.3.3
BSS	Block Starting Symbol	4.3.4
BYTE	Generate Byte Address	4.3.5
COMM*	Common Storage	4.3.6
COML*	Labelled Common Name	4.3.6
DATA	Generate Word Address or Data	4.3.7
DEF	Define Entry Point Symbol	4.3.8
END	End of Source	4.3.9
EQU	Equate	4.3.10
FLAG	Flag Bit Address	4.3.11
FRM*	Format a New Instruction	4.3.12
HED*	Page Heading	4.3.13
IDT	Object Identifier	4.3.14
IF*	Conditional Assembly	4.3.15
LIS	Start Listing	4.3.16
OPD	Operation Define	4.3.17
ORG	Origin	4.3.18
PEJ*	Page Eject	4.3.19
REF	Referenced External Symbols	4.3.20
UNL	Stop Listing	4.3.21

*are not supported by SAP733

4.3.1 BLOCK ENDING SYMBOL (BES)

The BES directive evaluates the operand field and advances the location counter by that amount. If a label is present, it is assigned to the new value of the location counter. BES is similar to BSS, except the label is applied to

the first location past the reserved area. The assembly language coding format for the BES directive is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	BES	ƀ	exp	ƀ	[comment]

where "exp" is typically a decimal number specifying the reserved area in words. If "exp" involves a symbol, it must be previously defined as an absolute quantity.

The following example reserves 50 words with TEN associated with the first word following the reserved area.

Label	Operation	Operand
TEN	BES	50

4.3.2 BASE REGISTER RESET (BRR)

The BRR directive informs the assembler that the base register is not available to the assembler for addressing purposes. The programmer can still specify base register addressing with the mode field. The BRR directive informs the assembler to use the base register for addressing purposes only in the event the mode field specifies that type of addressing. (This is the initial condition of assembly.) Under BRR directive control, if D is the unsigned displacement in register-memory instructions, then $0 \leqslant D \leqslant 255$ when the mode field contains B=1, or else a field size error occurs. The assembly language coding format for the BRR directive is as follows:

Label		Operation		Operand	Comment
[label]	ƀ	BRR	ƀ		[comment]

4.3.3 BASE REGISTER SET (BRS)

The BRS directive informs the assembler of the value the base register will contain at run time. The operand field of the BRS directive defines a 16-bit value that will be placed in the B register by the programmer. When the BRS is used and the assembler encounters subsequent register-memory format instructions that would produce field size errors if program counter relative, the assembler will attempt to generate these base register relative. In this case, if D is an unsigned 16-bit evaluation of the displacement expression and B is the value assumed in the base register, then $0 \leqslant D-B \leqslant 255$ or else a field size error occurs. The assembly language coding format for the BRS directive is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	BRS	ƀ	exp	ƀ	[comment]

where "exp" is the symbol for a 16-bit base value to be used. An example of BRS usage follows:

Label	Operation	Operand	Comment
	BRS	CAT	DEFINE BASE VALUE TO ASSEMBLER
	.		
	.		
	.		
	@LDA	=CAT	PUT ADDRESS OF CAT IN BASE
	RMO	A, B	REGISTER
	.		
	.		
	.		
CAT	BES	350	CAT IS DEFINED OUT OF
	BSS	10	PROGRAM COUNTER REL. RANGE

4.3.4 BLOCK STARTING SYMBOL (BSS)

The BSS directive reserves an area of memory. The first location in the reserved area is associated with the label in the name field of the BSS directive. The location of the area reserved is that defined by the location counter, which is then advanced past the reserved area. Note that no object code is generated by the BSS directive. If the programmer desires some value(s) to be assembled in the reserved area, he must do so by other means. The assembly language coding format for the BSS directive is as follows:

Label	Operation	Operand	Comment
[label] ƀ	BSS ƀ	exp ƀ	[comment]

where "exp" is typically a decimal number specifying the reserved area in words. If "exp" involves a symbol, it must be previously defined as an absolute quantity. An example of the BSS directive follows:

Location Counter	Label	Operation	Operand	Comments
03AA		BRU	TOM	BRANCH AROUND AREA
03AB	AREA	BSS	40	RESERVE AREA
03D3	TOM	LDA	AREA	REFERENCE AREA

A common usage of symbols in a BSS operand is an expression which defines the length of a reserved area. In the following example, if the length of TABA is likely to change, but TABB must always be the same length as TABA, it may be symbolically stated as follows:

Label	Operation	Operand	Comments
TABA	BSS	50	MIGHT CHANGE
TABB	BSS	TABB-TABA	ALWAYS SAME AS TABA

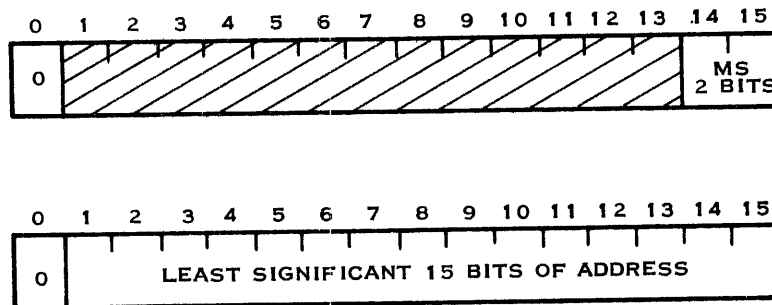
4.3.5 GENERATE BYTE ADDRESS (BYTE)

When using the byte string manipulation instructions, MVC and CLC, it is necessary to address data using byte rather than word addresses. The BYTE directive may be used to generate these byte addresses. Its usage is similar to that of the DATA directive when generating word addresses. The assembly language coding format for the BYTE directive is as follows:

Label	Operation	Operand	Comment
[label] ƀ	BYTE	ƀ $exp_1, exp_2, .. exp_n$	ƀ [comment]

where "$exp_1, exp_2, .. exp_n$" are evaluated and assigned to successive pairs of memory words. If a label is used, it is assigned to the first word of the first byte address.

Each byte address requires two words in the following format:



An expression in a BYTE operand field is evaluated as a word address and then multiplied by two to obtain the byte address. If the expression is preceded by a colon (:), the byte address is also incremented by one. The assembly listing in figure 4-5 shows the BYTE evaluation process.

4.3.6 REFERENCING COMMON STORAGE

4.3.6.1 NAMED COMMON LABEL (COML). The COML directive is used to start a new labeled common block. The label field must be used and gives the name of the new block. Storage reservation (given by the COMM directive) is started at zero for the new common block; all COMM directives following any given COML directive, up to the next COML directive, cause storage to be reserved in that common block. The assembler generates no entry in the common table if no COMM directives appear for a COML directive. Every assembly begins with an implicit COML directive in effect giving the name of FORTRAN blank common, 'ƀBLANK', and the occurrance of the END directive automatically terminates the immediately preceding COML block. The length of a COML block is determined by the sum of the sizes given on all COMM directives appearing under that COML directive. See

paragraph 4.3.7 for examples. The assembly language coding format for the COML directive is as follows:

```
label   ƀ   COML   ƀ   comment
```

NOTE

COML is supported in revisions *E and later of SAPG, part number 943253.

4.3.6.2 RESERVE COMMON STORAGE (COMM). The COMM directive reserves the given number of words in the currently active common block. If a label appears, it is assigned a value corresponding to the first word of the block, relative to the beginning of the currently active block. The assembly language coding format for the COMM directive is as follows:

```
[label]   ƀ   COMM   ƀ   exp   ƀ   [comment]
```

Several examples of the use of COML and COMM follow. In all cases, assume that there are no COML and COMM directives in the program besides those explicitly given.

Example 1: referencing FORTRAN blank common.

```
              .
              .
              .
X       COMM    30
Y       COMM    10
J       COMM     1
              .
              .
              .
              END
```

Blank common is 41 words long, and it is the only common block present.

Example 2: referencing labeled common only.

```
              .
              .
              .
COM1    COML
X       COMM    30
J       COMM     2
              .
              .
              .
              END
```

Common block COM1 is 32 words long, and the name 'COM1' is defined for the linking loader. Note that since no COMM entries occurred prior to the COM1 COML statement, blank common has length zero and hence is not entered.

Example 3: Referencing blank and labeled common.

```
        .
        .
        .
A       COMM    20   in blank common
B       COMM    10   in blank common
        .
        .
        .
X       COML         blank common is terminated at 30 words, and
                     a new common block started, named X.
        .
        .
        .
C       COMM    5    in block X
        .
        .
        .
D       COMM    7    in block X
        .
        .
        .
Y       COML         block X is 12 words long, and a new block
                     started, named Y.
        .
        .
        .
        END          block Y has no COMM directives in it, so has
                     length 0. This is most likely an inadvertent
                     error, but must be detected by noticing that Y
                     fails to appear in the common summary.
```

A common name may appear in an address field, and will address the first word of the common block. However, it may be used in this way only after at least one COMM directive has appeared in it.

COMM is used in a manner similar to FORTRAN COMMON. If a FORTRAN program and assembly language program are merged via link edit, any references in the FORTRAN program to labeled COMMON and references in the assembly language program to COMM defined storage are references to the same area of memory. In many applications this simplifies communications between the two programs. The following COMM directive would be used by a program requiring use of 12 words of common storage referenced as WORD.

Label	Operation	Operand
WORD	COMM	12

4.3.7 GENERATE WORD ADDRESS OR DATA (DATA)

The DATA directive is used for data generation. The assembly language coding format for the DATA directive is as follows:

Label	Operation	Operand	Comment
[label] ƀ	DATA ƀ	$exp_1, exp_2, .. exp_n$ ƀ	[comment]

where "$exp_1, exp_2, .. exp_n$" are expressions or strings that are evaluated and assigned to successive memory locations.

The DATA statement is used to define alphanumeric strings using the following format:

Label	Operation	Operand
CAT	DATA	'STRING'

STRING is a string of characters enclosed in single quotes. The string will be produced in ASCII code, two characters per word, packed left to right. If there is an odd number of characters in the string, the last word contains a delete code in the last character position. If a label is used, it is assigned to the first memory location involved. Figure 4-5 contains examples of several types of operands that may be used in a DATA statement.

4.3.8 DEFINE ENTRY POINT SYMBOL (DEF)

The program-linking assembler directives DEF and REF allow the programmer to symbolically link independently assembled programs that are to be loaded and executed together. Symbolic linkages between programs are created by means of symbols defined in one program and used as operands in another program. Such symbols are termed linkage symbols. A linkage symbol is called a defined entry point symbol in the program in which it is defined; it is a referenced external symbol in the program in which it is used as an operand. Every linkage symbol must be properly identified as such in the source program. A linkage symbol used as an external symbol is identified in each using program by the REF directive. A linkage symbol used

```
Location Code Line   Label   Operation   Operand
    00FF C8C9 0013           DATA        'HI'
P   0100 0107 0014   THERE   DATA        HERE+2,THERE-6,>100,100,0100
P   0101 00FA
    0102 0100
    0103 0064
    0104 0040
P   0105 0105 0015   HERE    DATA        HERE,THERE,>100+104,THERE-HERE
P   0106 0100
    0107 0168
    0108 FFFB
P   0109 0000 0016   HERE1   BYTE        HERE1
    010A 0212
P   010B 0000 0017           BYTE        :HERE1
    010C 0213
P   010D 0000 0018           BYTE        :HERE+6
    010E 0217
P   010F 0000 0019           BYTE        HERE1,:HERE1,:HERE1+6,>100
    0110 0212
P   0111 0000
    0112 0213
P   0113 0000
    0114 021F
    0115 0000
    0116 0200
    0117 0000 0020           BYTE        >100+>104,THERE-HERE
    0118 0408
    0119 FFFF
    011A FFF6
P   011B 0000 0021           BYTE        HERE+2,THERE-6,:>100,100,:0100
    011C 020E
P   011D 0000
    011E 01F4
    011F 0000
    0120 0201
    0121 0000
    0122 00C8
    0123 0000
    0124 0081
```

Figure 4-5. Example of BYTE and DATA Usage

as an entry point must be identified in the defining program by the DEF directive. The assembly language coding format for the DEF directive is as follows:

Label	Operation	Operand	Comment
[label] ƀ	DEF	ƀ $sym_1, sym_2, .. sym_n$	ƀ [comment]

where "$sym_1, sym_2, .. sym_n$" are symbols defined elsewhere in the program that may be used as entry points by other programs. A referenced symbol that is not defined in the program is flagged in the listing as an error.

In the following sequence, SQRT is identified as an entry-point symbol.

Label	Operation	Operand
SUBRO	BSS	10
	DEF	SQRT
	.	
	.	
	.	
SQRT	STA	SAVE

4.3.9 END OF SOURCE (END)

The END directive terminates the assembly of a program. It also supplies a point in the program to which control is transferred after the program is loaded. The END directive must always be the last statement in the source program. The assembly language coding format for the END directive is as follows:

Label	Operation	Operand	Comment
[label] ƀ	END	ƀ [exp]	ƀ [comment]

where "exp" specifies the point to which control is transferred when loading is complete. If the operand field is invalid, the statement is flagged as a possible error. If the operand field is blank, no program entry address is defined.

The point to which control usually is transferred is the first instruction in the program, as shown in the following sequence:

Location Counter	Label	Operation	Operand
		ORG	>2000
2000	AREA	BSS	50
2032	BEGIN	LDA	=3
		.	
		.	
		.	
		END	BEGIN

Here control will be transferred to BEGIN at location 2032_{16}. If the operand field were blank, control would be transferred to location 0000_{16}, a point outside of this program. When several object programs are joined by link editing, one is specified as the main program. Its transfer point is taken as the transfer point for the link edited program.

4.3.10 EQUATE (EQU)

The EQU directive is used to define a symbol in the label field by assigning to it the value of an expression in the operand field. The assembly language coding format for the EQU directive is as follows:

Label		Operation		Operand		Comment
sym	ƀ	EQU	ƀ	exp	ƀ	[comment]

where "sym" in the label field is given the same value as "exp" in the operand field. The expression in the operand field can be relocatable or absolute, and the symbol is similarly defined. Any symbols in the expression must be previously defined.

If the expression in the operand field or the symbol in the label field, or both, are invalid, or are not present, the EQU statement is flagged as an error in the listing and is not used. The EQU directive is the usual way of equating symbols to register numbers, input/output unit numbers, immediate data, actual addresses, and other arbitrary values. The examples below illustrate how this might be done:

Label	Operation	Operand	Comments
REGX	EQU	2	REGISTER X
IO125	EQU	125	INPUT/OUTPUT DATA
TEST	EQU	>3F	IMMEDIATE DATA
TIMER	EQU	80	ACTUAL ADDRESS

To reduce programming time, the programmer can equate symbols to frequently used compound expressions and then use the symbols as operands in place of the expressions. Thus in the statement:

Label	Operation	Operand
FIELD	EQU	ALPHA-BETA+GAMMA

FIELD is defined as ALPHA-BETA+GAMMA and may be used in place of it. Note, however, that ALPHA, BETA, and GAMMA must all be previously defined and only one may be a relocatable value. FIELD can be used anywhere in the program.

4.3.11 FLAG BIT ADDRESS (FLAG)

The FLAG directive is used by the assembler to specify a relative starting address for memory bit-referencing instructions (SMBO, SMBZ, TMBO,

and TMBZ). The FLAG directive may be used at any time, but until it is used, the starting memory address for the memory bit-referencing instructions is 0000_{16}. The assembly language coding format for the FLAG directive is as follows:

Label	Operation	Operand	Comment
[label] ƀ	FLAG ƀ	exp ƀ	[comment]

where "exp" is an expression that evaluates as the 16-bit memory word address used in conjunction with memory bit-referencing instructions.

The following example zeros bit 5 of location ABC with the use of the FLAG directive.

```
FLAG  ABC
SMBZ  5
```

4.3.12 FORMAT A NEW INSTRUCTION (FRM)

The FRM directive is used to create an instruction. The label field of the FRM directive is referenced as an op-code and the operand field of the FRM directive breaks the created instruction down into fields. The assembly language coding format for the FRM directive is as follows:

Label	Operation	Operand	Comment
label ƀ	FRM ƀ	$exp_1, exp_2, \ldots exp_n$ ƀ	[comment]

where "label" is the expression representing the op-code (must be one to four characters) and "$exp_1, exp_2, \ldots exp_n$" are expressions for positive values whose sum is 16.

When the label is used as an op-code, n fields of the associated operand field are evaluated, truncated to the length specified by the corresponding exp in the FRM directive, and placed in the output word. The following example illustrates use of the FRM directive.

```
                    Label   Operation   Operand
            0010    ABC       FRM       5, 5, 6
                               .
                               .
                               .
1000  F846  0020              ABC       >1F, 1, 6
```

In the first line of this example, ABC is defined to have three fields of 5, 5, and 6 bits, respectively. When ABC is subsequently used as an operation code, the assembler puts $1F_{16}$ in the first 5 bits, 1 in the next 5 bits, and 6 in the last 6 bits of the instruction. Thus, the second line in this example shows the assembled instruction $1111\ 1000\ 0100\ 0110_2$, or $F846_{16}$.

4.3.13 PAGE HEADING (HED)

The remaining characters in the line containing the HED directive are printed as page headings on the output listing. The first HED is used as the heading of all pages up to and including the page containing the second HED. Subsequent HED directives appear as page headings on the first page following the one on which the HED appears, and subsequent pages, until another HED is encountered. The assembly language coding format for the HED directive is as follows:

Label		Operation		Comment
[label]	ƀ	HED	ƀ	comment

The program in figure 4-3 makes use of the HED directive.

4.3.14 OBJECT IDENTIFIER (IDT)

The IDT directive reproduces the symbol appearing in the operand field as the program name in the object program. Names less than six characters have trailing blanks. If the name has more than six characters, the output will be truncated, and the name will consist of the first six characters. If the IDT directive is not present, the name will consist of six asterisks. The assembly language coding format for the IDT directive is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	IDT	ƀ	sym	ƀ	[comment]

where "sym" is the symbol for the program name.

4.3.15 CONDITIONAL ASSEMBLY (IF)

The IF directive alters the assembly process in accordance with the results of a conditional test. The operand field of the IF directive consists of two expressions and an optional symbol. The two expressions are evaluated and compared. If they are not equal, the assembly process continues with the next line. If the values are equal, the assembly process is suspended under the influence of the optional symbol. If the symbol is not present, assembly is suspended for one line. If the symbol is present, assembly is suspended until the input line with the same symbol in its label field is found.

All lines suspended from the assembly process are treated as comments; i.e., they are printed but no code is generated. Two or more IF statements may have overlapping ranges. This directive allows assembly-time modification of a program.

NOTE

Mathematical expressions cannot be used in the third (optional symbol) field of the operand.

The assembly language coding format for the IF directive is as follows:

Label	Operation	Operand	Comment
[label] b̸	IF	b̸ exp_1, exp_2, [sym] b̸	[comment]

where "exp_1, exp_2" are the two expressions to be evaluated and compared and "sym" is the optional symbol.

The following example illustrates usage of the IF directive.

Label	Operation	Operand	Comment
TTYVAL	EQU	2	TEST ASSUMES ONE DATA TERMINAL AT STANDARD ADDRESS
	.		
	.		
	.		
ASR	EQU	2	TTY1 - ASSUMED ASR AT STANDARD ADDRESS
TIP	EQU	3	TTY2 - ASSUMED TIP AT STANDARD ADDRESS
	.		
	.		
	.		
TYPE1	CRA	3	ROTATE 50 CHARS PRINT OK
	IF	TTYVAL, ASR	IF TTYVAL=2, REF DATA TERM 2
	WDS	TIP	
	IF	TTYVAL, TIP	IF TTYVAL=1, REF DATA TERM 1
	WDS	ASR	
	DATA	BIT8ON	
	BRU	$-2	

4.3.16 START LISTING (LIS)

The LIS directive initiates printing of the assembly listing. Printing continues until the UNL directive is encountered. If a complete assembly listing is desired, no LIS directive is required. The assembly language coding format for the LIS directive is as follows:

Label	Operation	Comment
[label] b̸	LIS	b̸ [comment]

4.3.17 OPERATION DEFINE (OPD)

The OPD directive is used to define an operation code. The label field of the OPD directive is referenced as the defined op-code mnemonic and the operand field of the OPD directive establishes the op-code bit settings and format type of the defined op-code. The first item in the operand field is evaluated as a 16-bit number and stored as the op-code. The second item in the operand field indicates the format type for the defined instruction. When

the label in the name field of the OPD directive appears as an op-code mnemonic, the accompanying operand field is OR'ed in with the defined op-code bit settings in accordance with the defined format type to assemble the instruction in the object program. Any op-code defined with the OPD directive takes precedence over the standard symbolic op-code. The assembly language coding format for the OPD directive is as follows:

Label	Operation	Operand	Comment
label ƀ	OPD	ƀ bits, n	ƀ [comment]

where "bits" is the hexadecimal representation of the defined op-code, "label" is the expression for the defined op-code mnemonic (must be one to four characters), and "n" defines the format type as follows:

SPACE -	Register-Memory	Identical Formats
0 -	Register-Memory	
1 -	Register-Memory	
2 -	Register-Register	
3 -	Register Shift and IDLE	
4 -	Register Skip	
5 -	Status Indicator Skip	
6 -	Data Bus Input/Output	
7 -	Sense Switch Skip and Register Bit	
8 -	Direct Memory Access Channel and Auxiliary Processor	

The final merging of the operation code and the operand fields is performed using a logical OR. Thus the operation code may be used to force setting of any bit to one. For example:

			Label	Operation	Operand	Comments
		1009	XYZ	OPD	>9800, 1	FORMAT TYPE 1
0A0C	9AFF	1010	JOE	XYZ	JOE, 2	COMMENT

In the first line, XYZ is defined to be the mnemonic of an operation code. The first part of the operand specifies the machine operation code (9800_{16} or $1001\ 1000\ 0000\ 0000_2$) and the second part of the operand specifies format type 1, or a register-memory format.

In this example, the 5-bit operation code ($1001\ 1_2$) for a hardware multiplication instruction (>9800=MPY) is specified. Line two shows the assembled result when the defined operation is subsequently used. Format type 1 causes the assembler to look for an optional label, a required operation code, a required first operand field, and an optional second operand field. The operation code (9800) is OR'ed with the IXB tag (2) to produce $1001\ 1010_2$ or $9A_{16}$. The B bit is not set; therefore, the operand is program counter relative. Since the program counter is pointing to the instruction in location $0A0D_{16}$, the program counter relative address of JOE ($0A0D_{16}$ - 0001_{16} = $0A0C_{16}$) is minus one, or FF_{16}. The OR'ed result produces the machine instruction $9AFF_{16}$.

Similarly, a new multiply instruction may be defined that is always base register relative by setting the B bit in the first field of the OPD operand as follows:

Label	Operation	Operand
MPB	OPD	>9900, 1

4.3.18 ORIGIN (ORG)

The ORG directive sets the value of the location counter to the value of the expression in the operand field. Any symbol in the expression must be previously defined. If the operand field is invalid, the ORG directive is not used. The ORG directive is commonly used to force loading of a program in specified memory locations. The assembly language coding format for the ORG directive is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	ORG	ƀ	exp	ƀ	[comment]

where "exp" is typically a decimal number specifying the location counter setting. If "exp" involves a symbol, it must be previously defined.

The following example shows how the ORG directive can be used for other purposes.

Operation	Operand
ORG	$+500

This ORG directive increases the location counter by 500. Therefore, in this case the directive provides an alternate way to reserve storage areas.

NOTE

> If the operand field of any ORG contains an absolute value instead of a relocatable expression, an absolute object is output; otherwise, a relocatable object is output.

4.3.19 PAGE EJECT (PEJ)

The PEJ directive ejects the remainder of the current assembly listing page. The assembler begins a new page with the heading from the current HED directive and the PEJ itself is printed as the first line on the new page. The assembly language coding format for the PEJ directive is as follows:

Label		Operation		Comment
[label]	ƀ	PEJ	ƀ	[comment]

4.3.20 REFERENCED EXTERNAL SYMBOLS (REF)

The REF directive identifies a linkage symbol as an external symbol that is referenced in the program using the REF directive. Each such external symbol must be identified in a REF directive. The assembly language coding format for the REF directive is as follows:

Label		Operation		Operand		Comment
[label]	ƀ	REF	ƀ	$sym_1, sym_2, .. sym_n$	ƀ	[comment]

where "$sym_1 sym_2, .. sym_n$" are symbols that must be defined in another program and identified in that program as an entry-point symbol with the DEF directive.

As an example, if MTPLY is an entry point symbol in another program, the using program identifies it as an external symbol as follows:

Operation	Operand
REF	MTPLY

The only way an external symbol may be referenced is as a full 16-bit address. The SAP assembler allows an external symbol to be used in an arithmetic calculation. For example, use of MTPLY+2 is allowed. To link to a program named SINE, the following coding might be used:

Label	Operation	Operand
PROGA	BSS	2
	REF	SINE
	.	
	.	
	.	
ADSINE	@BRL	SINE

4.3.21 STOP LISTING (UNL)

The UNL directive terminates the assembly listing process until an LIS directive is encountered. However, error messages are still printed. The assembly language coding format for the UNL directive is as follows:

Label		Operation		Comment
[label]	ƀ	UNL	ƀ	[comment]

APPENDIX A
INSTRUCTION EXECUTION TIMES

APPENDIX A

INSTRUCTION EXECUTION TIMES (IN MICROSECONDS)

This appendix groups the instructions by format type to facilitate presentation of the execution times.

REGISTER-MEMORY INSTRUCTIONS

Mnemonic	Name	Memory-Referencing*	Immediate Addressing
ADD	Add to Register A	1.75	0.75
AND	Logical AND with Register A	1.75	0.75
BIX	Branch on Incremented Index	1.25	1.25
BRL	Branch and Link	1.50	1.50
BRU	Branch Unconditional	1.25	1.00
CPA	Compare Algebraic	1.75	0.75
CPL	Compare Logical	1.75	0.75
DAD	Double Length Add	2.75	1.0
DIV	Divide	2.5 ⟶ 7.75	1.50 ⟶ 6.75
DLD	Double Load Registers A and E	2.75	1.0
DMT	Decrement Memory and Test	2.75	2.75
DSB	Double Length Subtract	2.75	1.0
DST	Double Store Registers A and E	2.75	2.75
IMO	Increment Memory by One	2.75	2.75
IOR	Logical OR with Register A	1.75	0.75
LDA	Load Register A	1.75	0.75
LDE	Load Register E	1.75	0.75
LDM	Load Register M	1.75	0.75
LDX	Load Register X	1.75	0.75
MPY	Multiply	2.25 ⟶ 6.25	1.25 ⟶ 5.25
STA	Store Register A	2.00	2.0
STE	Store Register E	2.00	2.0
STX	Store Register X	2.00	2.0
SUB	Subtract from Register A	1.75	0.75

*Add the following to execution times, when applicable: 0.25 microseconds for indexing, 0.75 microseconds for indirect addressing, and 0.25 microseconds for DAD, DLD, DST, and DSB extended format.

REGISTER SHIFT INSTRUCTIONS

Mnemonic	Name	Execution Time
ALA	Arithmetic Left Shift A	0.75+SC*/4
ALD	Arithmetic Left Shift Double	1.00+
ARA	Arithmetic Right Shift A	0.75+
ARD	Arithmetic Right Shift Double	1.00+
CLD	Circular Left Shift Double	0.75+
CRA	Circular Right Shift A	
CRB	Circular Right Shift B	
CRD	Circular Right Shift Double	
CRE	Circular Right Shift E	
CRL	Circular Right Shift L	
CRM	Circular Right Shift M	
CRS	Circular Right Shift S	
CRX	Circular Right Shift X	
LLA	Logical Left Shift A	
LLD	Logical Left Shift Double	
LRA	Logical Right Shift A	
LRD	Logical Right Shift Double	0.75+
LTO	Left Test for Ones	1.00+
LTZ	Left Test for Zeros	1.00+
RTO	Right Test for Ones	1.00+
RTZ	Right Test for Zeros	1.00+SC/4

*SC=Shift Count

REGISTER TO REGISTER INSTRUCTIONS

Mnemonic	Name	Execution Time
RAD	Register ADD	1.25
RAN	Register AND	1.25
RCA	Register Compare Algebraic	1.25
RCL	Register Compare Logical	1.25
RCO	Register Complement	1.00
RDE	Register Decrement	1.00
REO	Register Exclusive OR	1.25
REX	Register Exchange	1.50
RIN	Register Increment	1.00
RIV	Register Invert	1.00
RMO	Register Move	1.00
ROR	Register OR	1.25
RSU	Register Subtract	1.25

REGISTER SKIP INSTRUCTIONS

Mnemonic	Name	Execution Time
SEV	Skip on Even	1.00
SMI	Skip on Minus	↑
SNO	Skip on Not All Ones	
SNZ	Skip on Not All Zeros	
SOD	Skip on Odd	
SOO	Skip on All Ones	
SPL	Skip on Plus	↓
SZE	Skip on Zero	1.00

INDICATOR SKIP INSTRUCTIONS

Mnemonic	Name	Execution Time
SEQ	Skip on Equal	1.00
SGE	Skip on Greater Than or Equal	↑
SGT	Skip on Greater Than	
SLE	Skip on Less Than or Equal	
SLT	Skip on Less Than	
SNC	Skip on No Carry	
SNE	Skip on Not Equal	
SNV	Skip on No Overflow	
SOC	Skip on Carry	↓
SOV	Skip on Overflow	1.00

SENSE SKIP INSTRUCTIONS

Mnemonic	Name	Execution Time
SSE	Skip on Sense Switch Equal	1.00
SSN	Skip on Sense Switch Not Equal	1.00

MULTI-REGISTER INSTRUCTIONS

Mnemonic	Name	Execution Time
LRF	Load Register File	7.00
LSB	Load Status Block and Branch	3.25
LSR	Load Status Block, Reset Interrupt, and Branch	3.25
SRF	Store Register File	7.00
SSB	Store Status Block and Branch	3.25

BYTE MANIPULATION INSTRUCTIONS

Mnemonic	Name	Execution Time
CLC	Compare Logical Character String	5.00+2.25/ Byte
MVC	Move Character String	4.75+2.75/ Byte

MEMORY BIT MANIPULATION INSTRUCTIONS

Mnemonic	Name	Execution Time
SMBO	Set Memory Bit to One	3.25
SMBZ	Set Memory Bit to Zero	3.25
TMBO	Test Memory Bit for One	2.75
TMBZ	Test Memory Bit for Zero	2.75

REGISTER BIT MANIPULATION INSTRUCTIONS

Mnemonic	Name	Execution Time
SABO	Set Register A Bit to One	1.00
SABZ	Set Register A Bit to Zero	1.00
TABO	Test Register A Bit for One	1.25
TABZ	Test Register A Bit for Zero	1.25

MISCELLANEOUS

Mnemonic	Name	Execution Time
API	Auxiliary Processor Initiate	AP Controller Dependent
ATI	Automatic Transfer Initiate	2.50
IDL	Idle	1.00
NRM	Normalize	1.00→8.75
RDS	Read Direct Single	3.00→4.75
WDS	Write Direct Single	3.00→5.00

APPENDIX B

ALPHABETICAL AND HEXADECIMAL INSTRUCTION INDEXES

APPENDIX B

ALPHABETICAL INSTRUCTION INDEX

Mnemonic	Hexadecimal Code	Name	Paragraph
ADD	2000	Add to Register A	3.5.1
ALA	C880	Arithmetic Left Shift A	3.8.1
ALD	C8A0	Arithmetic Left Shift Double	3.8.2
AND	3800	Logical AND with Register A	3.9.1
API	DD00	Auxiliary Processor Initiate	3.12.1
ARA	C800	Arithmetic Right Shift A	3.8.3
ARD	C820	Arithmetic Right Shift Double	3.8.4
*ATI	D900	Automatic Transfer Initiate	3.12.2
BIX	4000	Branch on Incremented Index	3.4.1
BRL	7000	Branch and Link	3.4.2
BRU	7800	Branch Unconditional	3.4.3
CLC	DF80	Compare Logical Character String	3.6.1
CLD	CB80	Circular Left Shift Double	3.8.5
CPA	6800	Compare Algebraic	3.6.2
CPL	6000	Compare Logical	3.6.3
CRA	CA00	Circular Right Shift A	3.8.6
CRB	CB60	Circular Right Shift B	3.8.7
CRD	CBC0	Circular Right Shift Double	3.8.8
CRE	CA20	Circular Right Shift E	3.8.9
CRL	CB40	Circular Right Shift L	3.8.10
CRM	CA60	Circular Right Shift M	3.8.11
CRS	CB20	Circular Right Shift S	3.8.12
CRX	CA40	Circular Right Shift X	3.8.13
DAD	B800	Double Length Add	3.5.2
DIV	5800	Divide	3.5.3
DLD	B000	Double Load Registers A and E	3.2.1
DMT	4800	Decrement Memory and Test	3.7.1
DSB	A800	Double Length Subtract	3.5.4
DST	A000	Double Store Registers A and E	3.3.1
*IDL	CE00	Idle	3.4.4
IMO	5000	Increment Memory by One	3.5.5
IOR	3000	Logical OR with Register A	3.9.2
LDA	0000	Load Register A	3.2.2
LDE	0800	Load Register E	3.2.3
LDM	1800	Load Register M	3.2.4
LDX	1000	Load Register X	3.2.5
LLA	C8C0	Logical Left Shift A	3.8.14
LLD	C8E0	Logical Left Shift Double	3.8.15
LRA	C840	Logical Right Shift A	3.8.16

*Privileged instructions

ALPHABETICAL INSTRUCTION INDEX (Continued)

Mnemonic	Hexadecimal Code	Name	Paragraph
LRD	C860	Logical Right Shift Double	3.8.17
LRF	D8A0	Load Register File	3.2.6
*LSB	D880	Load Status Block and Branch	3.4.5
*LSR	D890	Load Status Block, Reset Interrupt, and Branch	3.4.6
LTO	C980	Left Test for Ones	3.8.18
LTZ	C9C0	Left Test for Zeros	3.8.19
MPY	9800	Multiply	3.5.6
MVC	DF00	Move Character String	3.11.1
NRM	CA9F	Normalize	3.8.20
**RAD	C080	Register Add	3.5.7
**RAN	C680	Register AND	3.9.3
**RCA	C400	Register Compare Algebraic	3.6.4
**RCL	C600	Register Compare Logical	3.6.5
**RCO	C100	Register Complement	3.5.8
**RDE	C700	Register Decrement	3.5.9
*RDS	D800	Read Direct Single	3.12.3
**REO	C280	Register Exclusive OR	3.9.4
**REX	C780	Register Exchange	3.11.2
**RIN	C300	Register Increment	3.5.10
**RIV	C200	Register Invert	3.5.11
**RMO	C500	Register Move	3.11.3
**ROR	C480	Register OR	3.9.5
**RSU	C000	Register Subtract	3.5.12
RTO	C900	Right Test for Ones	3.8.21
RTZ	C940	Right Test for Zeros	3.8.22
SABO	DB50	Set Register A Bit to One	3.10.1
SABZ	DB40	Set Register A Bit to Zero	3.10.2
SEQ	CD20	Skip on Equal	3.7.2
SEV	CCC0	Skip on Even	3.7.3
SGE	CD80	Skip on Greater Than or Equal	3.7.4
SGT	CD40	Skip on Greater Than	3.7.5
SLE	CDC0	Skip on Less Than or Equal	3.7.6
SLT	CD00	Skip on Less Than	3.7.7
SMBO	DB70	Set Memory Bit to One	3.10.3
SMBZ	DB60	Set Memory Bit to Zero	3.10.4
SMI	CC60	Skip on Minus	3.7.8
SNC	CFE0	Skip on No Carry	3.7.9

*Privileged instructions
**Privileged instructions when status register is the destination register.

ALPHABETICAL INSTRUCTION INDEX (Continued)

Mnemonic	Hexadecimal Code	Name	Paragraph
SNE	CDA0	Skip on Not Equal	3.7.10
SNO	CCA0	Skip on Not All Ones	3.7.11
SNV	CDE0	Skip on No Overflow	3.7.12
SNZ	CC80	Skip on Not All Zeros	3.7.13
SOC	CF60	Skip on Carry	3.7.14
SOD	CC40	Skip on Odd	3.7.15
SOO	CC20	Skip on All Ones	3.7.16
SOV	CD60	Skip on Overflow	3.7.17
SPL	CCE0	Skip on Plus	3.7.18
SRF	D8E0	Store Register File	3.3.2
SSB	D8C0	Store Status Block and Branch	3.4.7
SSE	CC10	Skip on Sense Switch Equal	3.7.19
SSN	CC90	Skip on Sense Switch Not Equal	3.7.20
STA	8000	Store Register A	3.3.3
STE	8800	Store Register E	3.3.4
STX	9000	Store Register X	3.3.5
SUB	2800	Subtract from Register A	3.5.13
SZE	CC00	Skip on Zero	3.7.21
TABO	DB10	Test Register A Bit for One	3.10.5
TABZ	DB00	Test Register A Bit for Zero	3.10.6
TMBO	DB30	Test Memory Bit for One	3.10.7
TMBZ	DB20	Test Memory Bit for Zero	3.10.8
*WDS	D820	Write Direct Single	3.12.4

HEXADECIMAL INSTRUCTION INDEX

Hexadecimal Code	Mnemonic	Name	Paragraph
0000	LDA	Load Register A	3.2.2
0800	LDE	Load Register E	3.2.3
1000	LDX	Load Register X	3.2.5
1800	LDM	Load Register M	3.2.4
2000	ADD	Add to Register A	3.5.1
2800	SUB	Subtract from Register A	3.5.13
3000	IOR	Logical OR with Register A	3.9.2
3800	AND	Logical AND with Register A	3.9.1
4000	BIX	Branch on Incremented Index	3.4.1
4800	DMT	Decrement Memory and Test	3.7.1

*Privileged Instructions

HEXADECIMAL INSTRUCTION INDEX (Continued)

Hexadecimal Code	Mnemonic	Name	Paragraph
5000	IMO	Increment Memory by One	3.5.5
5800	DIV	Divide	3.5.3
6000	CPL	Compare Logical	3.6.3
6800	CPA	Compare Algebraic	3.6.2
7000	BRL	Branch and Link	3.4.2
7800	BRU	Branch Unconditional	3.4.3
8000	STA	Store Register A	3.3.3
8800	STE	Store Register E	3.3.4
9000	STX	Store Register X	3.3.5
9800	MPY	Mulitply	3.5.6
A000	DST	Double Store Registers A and E	3.3.1
A800	DSB	Double Length Subtract	3.5.4
B000	DLD	Double Load Registers A and E	3.2.1
B800	DAD	Double Length Add	3.5.2
**C000	RSU	Register Subtract	3.5.12
**C080	RAD	Register Add	3.5.7
**C100	RCO	Register Complement	3.5.8
**C200	RIV	Register Invert	3.5.11
**C280	REO	Register Exclusive OR	3.9.4
**C300	RIN	Register Increment	3.5.10
**C400	RCA	Register Compare Algebraic	3.6.4
**C480	ROR	Register OR	3.9.5
**C500	RMO	Register Move	3.11.3
**C600	RCL	Register Compare Logical	3.6.5
**C680	RAN	Register AND	3.9.3
**C700	RDE	Register Decrement	3.5.9
**C780	REX	Register Exchange	3.11.2
C800	ARA	Arithmetic Right Shift A	3.8.3
C820	ARD	Arithmetic Right Shift Double	3.8.4
C840	LRA	Logical Right Shift A	3.8.16
C860	LRD	Logical Right Shift Double	3.8.17
C880	ALA	Arithmetic Left Shift A	3.8.1
C8A0	ALD	Arithmetic Left Shift Double	3.8.2
C8C0	LLA	Logical Left Shift A	3.8.14
C8E0	LLD	Logical Left Shift Double	3.8.15
C900	RTO	Right Test for Ones	3.8.21
C940	RTZ	Right Test for Zeros	3.8.22
C980	LTO	Left Test for Ones	3.8.18
C9C0	LTZ	Left Test for Zeros	3.8.19
CA00	CRA	Circular Right Shift A	3.8.6

**Privileged instructions when the status register is the destination register.

HEXADECIMAL INSTRUCTION INDEX (Continued)

Hexadecimal Code	Mnemonic	Name	Paragraph
CA20	CRE	Circular Right Shift E	3.8.9
CA40	CRX	Circular Right Shift X	3.8.13
CA60	CRM	Circular Right Shift M	3.8.11
CA9F	NRM	Normalize	3.8.20
CB20	CRS	Circular Right Shift S	3.8.12
CB40	CRL	Circular Right Shift L	3.8.10
CB60	CRB	Circular Right Shift B	3.8.7
CB80	CLD	Circular Left Shift Double	3.8.5
CBC0	CRD	Circular Right Shift Double	3.8.8
CC00	SZE	Skip to Zero	3.7.21
CC10	SSE	Skip on Sense Switch Equal	3.7.19
CC20	SOO	Skip on All Ones	3.7.16
CC40	SOD	Skip on Odd	3.7.15
CC60	SMI	Skip on Minus	3.7.8
CC80	SNZ	Skip on Not All Zeros	3.7.13
CC90	SSN	Skip on Sense Switch Not Equal	3.7.20
CCA0	SNO	Skip on Not All Ones	3.7.11
CCC0	SEV	Skip on Even	3.7.3
CCE0	SPL	Skip on Plus	3.7.18
CD00	SLT	Skip on Less Than	3.7.7
CD20	SEQ	Skip on Equal	3.7.2
CD40	SGT	Skip on Greater Than	3.7.5
CD60	SOV	Skip on Overflow	3.7.17
CD80	SGE	Skip on Greater Than or Equal	3.7.4
CDA0	SNE	Skip on Not Equal	3.7.10
CDC0	SLE	Skip on Less Than or Equal	3.7.6
CDE0	SNV	Skip on No Overflow	3.7.12
*CE00	IDL	Idle	3.4.4
CF60	SOC	Skip on Carry	3.7.14
CFE0	SNC	Skip on No Carry	3.7.9
*D800	RDS	Read Direct Single	3.12.3
*D820	WDS	Write Direct Single	3.12.4
*D880	LSB	Load Status Block and Branch	3.4.5
*D890	LSR	Load Status Block, Reset Interrupt, and Branch	3.4.6
D8A0	LRF	Load Register File	3.2.6
D8C0	SSB	Store Status Block and Branch	3.4.7
D8E0	SRF	Store Register File	3.3.2
*D900	ATI	Automatic Transfer Initiate	3.12.2
DB00	TABZ	Test Register A Bit for Zero	3.10.6

*Privileged instructions

HEXADECIMAL INSTRUCTION INDEX (Continued)

Hexadecimal Code	Mnemonic	Name	Paragraph
DB10	TABO	Test Register A Bit for One	3.10.5
DB20	TMBZ	Test Memory Bit for Zero	3.10.8
DB30	TMBO	Test Memory Bit for One	3.10.7
DB40	SABZ	Set Register A Bit to Zero	3.10.2
DB50	SABO	Set Register A Bit to One	3.10.1
DB60	SMBZ	Set Memory Bit to Zero	3.10.4
DB70	SMBO	Set Memory Bit to One	3.10.3
DD00	API	Auxiliary Processor Initiate	3.12.1
DF00	MVC	Move Character String	3.11.1
DF80	CLC	Compare Logical Character String	3.6.1

APPENDIX C
ILLEGAL INSTRUCTION OPERATION CODES

APPENDIX C

ILLEGAL INSTRUCTION OPERATION CODES

When the op-code of an instruction is other than one of the 99 standard op-codes, it is considered illegal. Table C-1 lists the instruction bit-patterns that are detected as illegal.

Table C-1. Illegal Instruction Codes

Instruction Bits															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	0	0	X	X	X	X	X	X	X	1	X	X	X
1	1	0	0	0	0	X	1	1	X	X	X	X	X	X	X
1	1	0	0	0	1	0	1	1	X	X	X	X	X	X	X
1	1	0	0	1	0	0	1	X	X	1	X	X	X	X	X
1	1	0	0	1	0	1	0	1	0	1	X	X	X	X	X
1	1	0	0	1	0	1	0	1	1	X	X	X	X	X	X
1	1	0	0	1	0	1	1	0	0	0	X	X	X	X	X
1	1	0	0	1	0	1	1	1	X	1	X	X	X	X	X
1	1	0	0	1	1	1	0	0	0	0	1	X	X	X	X
1	1	0	0	1	1	1	0	0	0	1	X	X	X	X	X
1	1	0	0	1	1	1	0	0	1	X	X	X	X	X	X
1	1	0	0	1	1	1	0	1	X	X	X	X	X	X	X
1	1	0	0	1	1	1	1	X	0	X	X	X	X	X	X
1	1	0	0	1	1	1	1	0	1	0	X	X	X	X	X
1	1	0	1	0	X	X	X	X	X	X	X	X	X	X	X
1	1	0	1	1	0	1	X	1	X	X	X	X	X	X	X
1	1	0	1	1	1	X	0	X	X	X	X	X	X	X	X
1	1	1	X	X	X	X	X	X	X	X	X	X	X	X	X

X = DON'T CARE (0 or 1)

USER'S RESPONSE SHEET

Manual Title: Model 980 Computer Assembly Language Programmer's Reference Manual (943013-9701)

Date of Manual: 1 March 1976 Date of This Letter: ________

User: ________ Office/Dept. No.: ________

Company: ________

Street Address: ________

City/State/Zip: ________

Please list any discrepancy found in this manual by page, paragraph, figure, or table number in the following space. If there are any other suggestions that you wish to make, feel free to include them. Thank you.

Location in Manual	Comment/Suggestion

CUT ALONG LINE

NO POSTAGE NECESSARY IF MAILED IN U.S.A.
FOLD ON TWO LINES (LOCATED ON REVERSE SIDE), STAPLE AND MAIL

First Class
PERMIT NO. 3135
Austin, Texas

BUSINESS REPLY MAIL
No Postage Necessary if Mailed in the United States

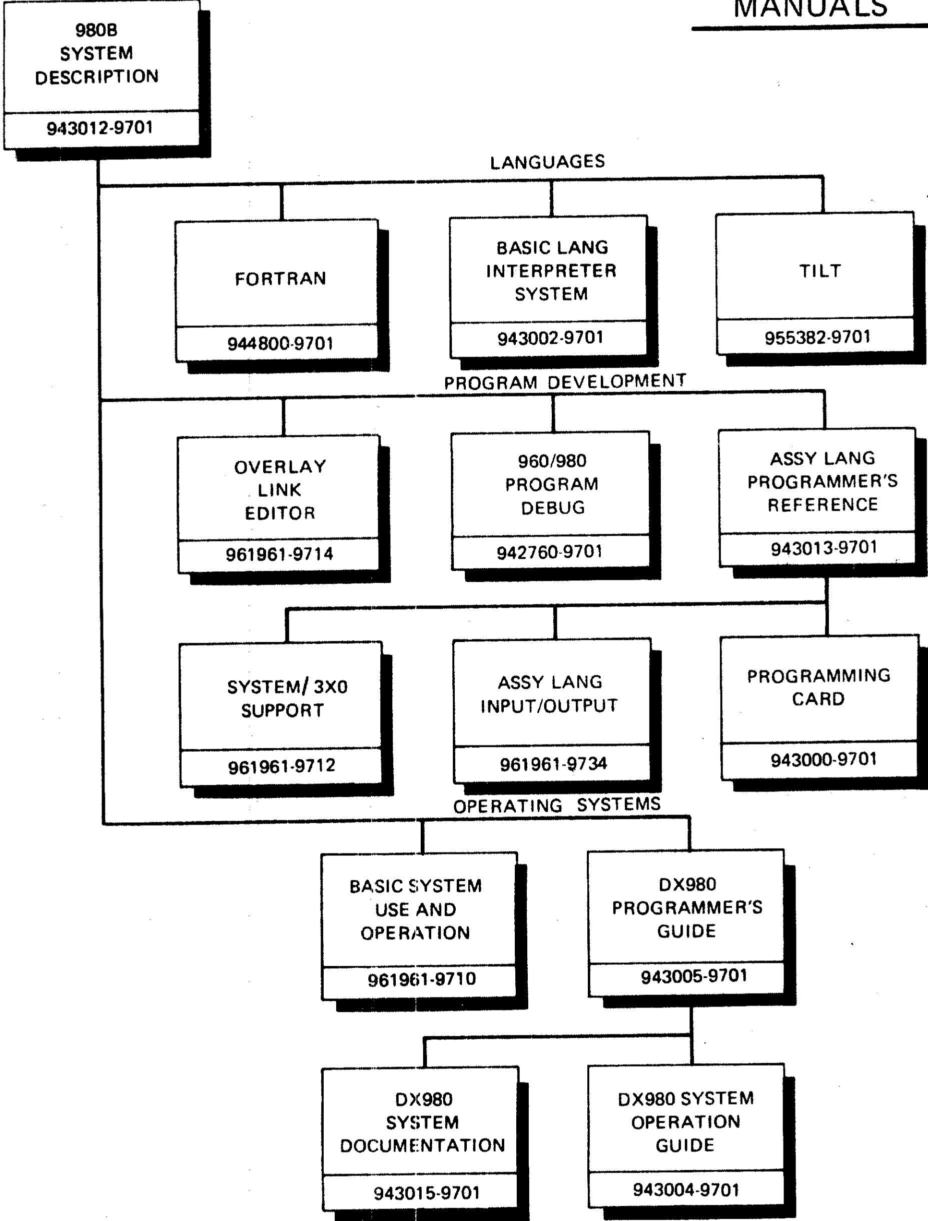
Postage Will Be Paid by

TEXAS INSTRUMENTS INCORPORATED
DIGITAL SYSTEMS DIVISION

P.O. BOX 2909 · AUSTIN, TEXAS 78767

Attn: TECHNICAL PUBLICATIONS, MS 2146

980 COMPUTER SYSTEM SOFTWARE MANUALS



TEXAS INSTRUMENTS
INCORPORATED
DIGITAL SYSTEMS DIVISION
POST OFFICE BOX 2909 • AUSTIN, TEXAS 78767