



© Texas Instruments Incorporated 1981, 1982

All Rights Reserved, Printed in U.S.A.

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.

## MANUAL REVISION HISTORY

990/99000 Assembly Language Reference Manual (2270509-9701)

Original Issue .....	15 January 1981
Revision .....	15 November 1982

The total number of pages in this publication is 528.

# Preface

This manual describes the assembly language for Model 990/10, 990/10A, and 990/12 Computers as implemented by the SDSMAC macro assembler. Programs written in the 990 assembly language may run under control of an operating system, DX10 for example, or they may be written to run without the services provided by an operating system. This manual describes the 990 assembly language and the SDSMAC assembler only. For more information on the operating systems, see the manuals in the reference list following. Programs that are written to run without the services of an operating system are called stand-alone programs and are discussed in Section 4.

## NOTE

S300 processors use the 990/10A assembly language set.

This manual is intended for system programmers and analysts. It is assumed that the reader is familiar with an assembly language and with at least one of the applicable operating systems. This manual is intended for use by an experienced assembly language programmer who is learning the 990 instruction set or who has questions about the operation of a specific instruction.

The sections and appendixes of this manual are organized as follows:

### Section

- 1 Introduction — Provides an introduction to the Model 990/10, 990/10A, and 990/12 Computers and describes the status register and memory organization of the 990 computer.
- 2 General Programming Information — Introduces the 990/10, 990/10A, and 990/12 instruction sets, then describes the assembly language source statement format and the data types used in the assembly language.
- 3 Assembly Instructions — Describes the addressing modes and instruction formats, then discusses the individual assembly instructions in detail in alphabetical order by mnemonic.
- 4 Application Notes — Provides information about implementation and control of subroutines, provides information for upgrading a program from a 990/10 system to a 990/12 system, describes reentrant programming techniques, and describes factors that must be considered when writing a program that does not run under the control of an operating system.
- 5 Assembler Directives — Describes the directives that control the assembly process.
- 6 Pseudo-Instructions — Defines the three assembly pseudo-instructions.

- 7 Macro Language — Describes the macro language available with the SDSMAC assembler and defines the verbs used by the macro language.
- 8 Relocatability and Program Linking — Describes the relocation capability of assembly language programs and the methods by which modules are linked with shared symbols across the modules.
- 9 Assembler Output — Describes the error messages and object code formats produced by the SDSMAC assembler.

**Appendixes**

- A Character Set — Lists the characters recognized by the SDSMAC assembler.
- B Instruction Tables — Lists the instruction set by functional group, showing syntax, results, and status information.
- C Program Organization — Describes the data, workspace, and procedure areas of a program for the 990 computer.
- D Hexadecimal Instruction Table — Lists the instructions by order of hexadecimal opcode.
- E Assembler Directive Table — Lists the assembler directives and gives the required syntax.
- F Macro Language Table — Lists the macro language verbs and gives the required syntax.
- G Example Program — Gives a general assembly language programming example.
- H Numerical Tables — Gives commonly referenced numeric terms and decimal-hexadecimal conversion charts.
- I Instruction Usage Cross-Reference Table — Lists all of the instructions that perform a type of operation and all instructions that operate on a data type.
- J Illegal Opcodes — Lists the opcodes that cause an illegal operation interrupt.

The following documents contain additional information that may be helpful to the assembly language programmer.

Title	Part Number
<i>DX10 Operating System Concepts and Facilities (Volume I)</i>	946250-9701
<i>DX10 Operating System Operations Guide (Volume II)</i>	946250-9702
<i>DX10 Operating System Application Programming Guide (Volume III)</i>	946250-9703



<b>Title</b>	<b>Part Number</b>
<i>DX10 Operating System Text Editor Manual (Volume IV)</i>	946250-9704
<i>DX10 Operating System Systems Programming Guide (Volume V)</i>	946250-9705
<i>DX10 Operating System Error Reporting and Recovery Manual (Volume VI)</i>	946250-9706
<i>DX10 Link Editor Reference Manual</i>	949617-9701
<i>Model 990 Computer Reference Manual</i>	943442-9701
<i>Model 990/10 Computer System Field Hardware Reference Manual</i>	945417-9701
<i>Model 990/10A Computer General Description</i>	2302633-9701
<i>Model 990/12 Computer Hardware User's Guide</i>	2264446-9701
<i>DNOS Master Index to Operating System Manuals</i>	2270500-9701
<i>DNOS Concepts and Facilities</i>	2270501-9701
<i>DNOS Operations Guide</i>	2270502-9701
<i>DNOS System Command Interpreter (SCI) Reference Manual</i>	2270503-9701
<i>DNOS Text Editor Reference Manual</i>	2270504-9701
<i>DNOS Reference Handbook</i>	2270505-9701
<i>DNOS Messages and Codes</i>	2270506-9701
<i>DNOS Supervisor Call (SVC) Reference Manual</i>	2270507-9701
<i>DNOS Assembly Language Programmer's Guide</i>	2270508-9701
<i>DNOS Link Editor Reference Manual</i>	2270522-9701

# Contents

Paragraph	Title	Page
<b>1 — Introduction</b>		
1.1	Introduction	1-1
1.2	990/10 and 990/10A Computer Characteristics	1-1
1.2.1	Byte Organization	1-1
1.2.2	Word Organization	1-2
1.2.3	Memory Organization	1-2
1.2.4	Transfer Vectors	1-4
1.2.5	Interrupts	1-4
1.2.5.1	General Interrupt Structure	1-5
1.2.5.2	Interrupt Sequence	1-5
1.2.6	Workspace	1-5
1.2.7	Registers	1-7
1.3	990/12 Computer Characteristics	1-7
1.4	Status Register	1-8
1.4.1	Logical Greater Than	1-8
1.4.2	Arithmetic Greater Than	1-8
1.4.3	Equal	1-9
1.4.4	Carry	1-9
1.4.5	Overflow	1-9
1.4.6	Odd Parity	1-10
1.4.7	Extended Operation	1-10
1.4.8	Privileged Mode	1-10
1.4.9	Map File Select	1-10
1.4.10	Memory Management And Protection Enabled	1-10
1.4.11	Overflow Interrupt Enable	1-10
1.4.12	Writable Control Store	1-10
1.4.13	Interrupt Mask	1-11

## 2 — General Programming Information

2.1	990 Assembly Language	2-1
2.2	990/10 and 990/10A Instruction Sets	2-1
2.2.1	Arithmetic Instructions	2-2
2.2.2	Branch and Jump Instructions	2-2
2.2.3	Compare Instructions	2-4
2.2.4	Control and CRU Instructions	2-4
2.2.5	Load and Move Instructions	2-5
2.2.6	Logical Instructions	2-6
2.2.7	Shift Instructions	2-6

Paragraph	Title	Page
2.2.8	Extended Operation Instruction . . . . .	2-6
2.2.9	Long-Distance Addressing Instructions . . . . .	2-7
2.3	990/12 Instruction Set . . . . .	2-7
2.3.1	Arithmetic Instructions . . . . .	2-7
2.3.2	Jump Instructions . . . . .	2-8
2.3.3	Compare Instructions . . . . .	2-8
2.3.4	Control and CRU Instructions . . . . .	2-9
2.3.5	Load and Move Instructions . . . . .	2-9
2.3.6	Logical Instructions . . . . .	2-10
2.3.7	Shift Instructions . . . . .	2-11
2.3.8	Extended Operation and Long-Distance Addressing Instructions . . . . .	2-11
2.3.9	Conversion Instructions . . . . .	2-11
2.4	Source Statement Format . . . . .	2-12
2.4.1	Character Set . . . . .	2-14
2.4.2	Label Field . . . . .	2-14
2.4.3	Operation Field . . . . .	2-14
2.4.4	Operand Field . . . . .	2-14
2.4.5	Comment Field . . . . .	2-14
2.5	Constants . . . . .	2-15
2.5.1	Decimal Integer Constants . . . . .	2-15
2.5.2	Hexadecimal Integer Constants . . . . .	2-15
2.5.3	Character Constants . . . . .	2-15
2.5.4	Assembly-Time Constants . . . . .	2-15
2.6	Symbols . . . . .	2-16
2.6.1	Predefined Symbols . . . . .	2-16
2.6.2	Character Strings . . . . .	2-17
2.7	Expressions . . . . .	2-17
2.7.1	Use of External Referenced Symbols in Expressions . . . . .	2-18
2.7.2	Arithmetic Operators in Expressions . . . . .	2-19
2.7.3	Logical Operators in Expressions . . . . .	2-20
2.7.4	Relational Operators in Expressions . . . . .	2-20
2.7.5	Use of Parentheses in Expressions . . . . .	2-21
2.8	990/12 Data Types . . . . .	2-22
2.8.1	Extended Integers . . . . .	2-22
2.8.2	Multiple Precision Integers . . . . .	2-22
2.8.3	Byte Strings . . . . .	2-23
2.8.4	Stacks . . . . .	2-24
2.8.5	Lists . . . . .	2-24
2.8.6	Single Precision Real Numbers . . . . .	2-26
2.8.7	Double Precision Real Numbers . . . . .	2-28
2.8.8	Floating Point Accumulator (FPA) . . . . .	2-28
2.9	Reexecutable Instructions . . . . .	2-28

### 3 — Assembly Instructions

3.1	General . . . . .	3-1
3.2	Addressing Modes . . . . .	3-1

Paragraph	Title	Page
3.2.1	Workspace Register Addressing .....	3-1
3.2.2	Workspace Register Indirect Addressing .....	3-1
3.2.3	Workspace Register Indirect Autoincrement Addressing .....	3-2
3.2.4	Symbolic Memory Addressing .....	3-3
3.2.5	Indexed Memory Addressing .....	3-3
3.2.6	Program Counter Relative Addressing .....	3-4
3.2.7	CRU Bit Addressing .....	3-4
3.2.8	Immediate Addressing .....	3-4
3.3	Addressing Summary .....	3-5
3.4	Instruction Formats .....	3-5
3.4.1	Format 1 — Two Address Instructions .....	3-9
3.4.2	Format 2 — Jump Instructions .....	3-10
3.4.3	Format 2 — Bit I/O Instructions .....	3-11
3.4.4	Format 3 — Logical Instructions .....	3-11
3.4.5	Format 4 — CRU Instructions .....	3-12
3.4.6	Format 5 — Register Shift Instructions .....	3-13
3.4.7	Format 6 — Single Address Instructions .....	3-14
3.4.8	Format 7 — Instructions Without Operands .....	3-15
3.4.9	Format 8 — Immediate Instructions .....	3-15
3.4.10	Format 9 — Extended Operations Instruction .....	3-16
3.4.11	Format 9 — Multiply and Divide Instructions .....	3-17
3.4.12	Format 10 — Memory Map File Instruction .....	3-18
3.4.13	Format 11 — Multiple Precision Instructions .....	3-18
3.4.14	Format 12 — String Instructions .....	3-19
3.4.15	Format 13 — Multiple Precision Shift Instructions .....	3-21
3.4.16	Format 14 — Bit Testing Instructions .....	3-22
3.4.17	Format 15 — Invert Order of Field Instruction .....	3-23
3.4.18	Format 16 — Field Instructions .....	3-24
3.4.19	Format 17 — Alter Registers and Jump Instructions .....	3-25
3.4.20	Format 18 — Single Register Operand Instructions .....	3-25
3.4.21	Format 19 — Move Address Instruction .....	3-26
3.4.22	Format 20 — List Search Instructions .....	3-27
3.4.23	Format 21 — Extend Precision Instruction .....	3-28
3.5	Instruction Descriptions .....	3-29
3.5.1	Opcode .....	3-29
3.5.2	Instruction Format .....	3-29
3.5.3	Applicability .....	3-29
3.5.4	Syntax Definition .....	3-29
3.5.5	Instruction Example .....	3-31
3.5.6	Operation Definition .....	3-31
3.5.7	Status Bits Affected .....	3-31
3.5.8	Execution Results .....	3-31
3.5.9	Application Notes .....	3-31
3.6	Add Words — A .....	3-31
3.7	Add Bytes — AB .....	3-32
3.8	Absolute Value — ABS .....	3-34
3.9	Add Double Precision Real — AD .....	3-36
3.10	Add Immediate — AI .....	3-37
3.11	Add Multiple Precision Integer — AM .....	3-38

Paragraph	Title	Page
3.12	AND Immediate — ANDI .....	3-39
3.13	AND Multiple Precision — ANDM .....	3-41
3.14	Add Real — AR .....	3-43
3.15	Add to Register and Jump — ARJ .....	3-44
3.16	Branch — B .....	3-46
3.17	Binary to Decimal ASCII Conversion — BDC .....	3-47
3.18	Branch Indirect — BIND .....	3-49
3.19	Branch and Link — BL .....	3-51
3.20	Branch Immediate and Push Link to Stack — BLSK .....	3-52
3.21	Branch and Load Workspace Pointer — BLWP .....	3-53
3.22	Compare Words — C .....	3-54
3.23	Compare Bytes — CB .....	3-55
3.24	Convert Double Precision Real to Extended Integer — CDE .....	3-57
3.25	Convert Double Precision Real to Integer — CDI .....	3-58
3.26	Convert Extended Integer to Double Precision Real — CED .....	3-59
3.27	Convert Extended Integer to Real — CER .....	3-61
3.28	Compare Immediate — CI .....	3-62
3.29	Convert Integer to Double Precision Real — CID .....	3-63
3.30	Convert Integer to Real — CIR .....	3-65
3.31	Clock Off — CKOF .....	3-66
3.32	Clock On — CKON .....	3-67
3.33	Clear — CLR .....	3-67
3.34	Count Ones — CNTO .....	3-68
3.35	Compare Ones Corresponding — COC .....	3-69
3.36	Cyclic Redundancy Code Calculation — CRC .....	3-70
3.37	Convert Real to Extended Integer — CRE .....	3-73
3.38	Convert Real to Integer — CRI .....	3-74
3.39	Compare Strings — CS .....	3-76
3.39.1	Reexecutable Instructions .....	3-78
3.40	Compare Zeros Corresponding — CZC .....	3-80
3.41	Decimal ASCII to Binary Conversion — DBC .....	3-81
3.42	Divide Double Precision Real — DD .....	3-82
3.43	Decrement — DEC .....	3-84
3.43.1	Decrement Instruction Example .....	3-85
3.44	Decrement by Two — DECT .....	3-86
3.44.1	Decrement By Two Instruction Example .....	3-87
3.45	Disable Interrupts — DINT .....	3-88
3.46	Divide — DIV .....	3-89
3.47	Divide Signed — DIVS .....	3-91
3.48	Divide Real — DR .....	3-92
3.49	Enable Interrupts — EINT .....	3-94
3.50	Execute Micro-Diagnostic — EMD .....	3-94
3.51	Extend Precision — EP .....	3-96
3.52	Idle — IDLE .....	3-97
3.53	Increment — INC .....	3-98
3.53.1	Increment Instruction Example .....	3-99
3.54	Increment by Two — INCT .....	3-100
3.55	Insert Field — INSF .....	3-101
3.56	Invert — INV .....	3-102

Paragraph	Title	Page
3.57	Invert Order of Field — IOF	3-103
3.58	Jump if Equal — JEQ	3-104
3.59	Jump if Greater Than — JGT	3-105
3.60	Jump if Logical High — JH	3-106
3.61	Jump if High or Equal — JHE	3-107
3.62	Jump if Logical Low — JL	3-108
3.63	Jump if Low or Equal — JLE	3-109
3.64	Jump if Less Than — JLT	3-110
3.65	Unconditional Jump — JMP	3-111
3.66	Jump if No Carry — JNC	3-111
3.67	Jump if Not Equal — JNE	3-112
3.68	Jump if No Overflow — JNO	3-113
3.69	Jump on Carry — JOC	3-114
3.70	Jump if Odd Parity — JOP	3-115
3.71	Load Writable Control Store — LCS	3-116
3.72	Load Double Precision Real — LD	3-117
3.73	Load CRU — LDCR	3-119
3.74	Long Distance Destination — LDD	3-120
3.74.1	Memory Mapping	3-121
3.75	Long Distance Source — LDS	3-122
3.76	Load Immediate — LI	3-124
3.77	Load Interrupt Mask — LIM	3-125
3.78	Load Interrupt Mask Immediate — LIM1	3-126
3.79	Load Memory Map File — LMF	3-127
3.80	Load Real — LR	3-130
3.81	Load or Restart Execution — LREX	3-131
3.82	Load Status Register — LST	3-132
3.83	Left Test for Ones — LTO	3-133
3.84	Load Workspace Pointer — LWP	3-135
3.85	Load Workspace Pointer Immediate — LWPI	3-136
3.86	Multiply Double Precision Real — MD	3-136
3.87	Move Word — MOV	3-138
3.88	Move Address — MOVA	3-140
3.89	Move Byte — MOVB	3-141
3.90	Move String — MOVS	3-142
3.91	Multiply — MPY	3-145
3.92	Multiply Signed — MPYS	3-146
3.93	Multiply Real — MR	3-148
3.94	Move String From Stack — MVSK	3-149
3.95	Move String Reverse — MVSR	3-152
3.96	Negate — NEG	3-154
3.97	Negate Double Precision Real — NEGD	3-155
3.98	Negate Real — NEGR	3-156
3.99	Normalize — NRM	3-158
3.100	OR Immediate — ORI	3-159
3.101	OR Multiple Precision — ORM	3-161
3.102	Pop String From Stack — POPS	3-163
3.103	Push String to Stack — PSHS	3-166
3.104	Reset — RSET	3-170

Paragraph	Title	Page
3.105	Right Test For One — RTO	3-170
3.106	Return With Workspace Pointer — RTWP	3-172
3.107	Subtract Words — S	3-173
3.108	Subtract Bytes — SB	3-174
3.109	Set CRU Bit to Logic One — SBO	3-175
3.110	Set CRU Bit to Logic Zero — SBZ	3-176
3.111	Subtract Double Precision Real — SD	3-177
3.112	Search String For Equal Byte — SEQB	3-179
3.113	Set to One — SETO	3-182
3.114	Shift Left Arithmetic — SLA	3-182
3.115	Shift Left Arithmetic Multiple Precision — SLAM	3-183
3.116	Search List Logical Address — SLSL	3-185
3.117	Search List Physical Address — SLSP	3-188
3.118	Subtract Multiple Precision Integer — SM	3-190
3.119	Search String For Not Equal Byte — SNEB	3-192
3.120	Set Ones Corresponding — SOC	3-194
3.121	Set Ones Corresponding, Byte — SOCB	3-195
3.122	Subtract Real — SR	3-196
3.123	Shift Right Arithmetic — SRA	3-198
3.124	Shift Right Arithmetic Multiple Precision — SRAM	3-199
3.125	Shift Right Circular — SRC	3-200
3.126	Subtract From Register and Jump — SRJ	3-201
3.127	Shift Right Logical — SRL	3-203
3.128	Store CRU — STCR	3-204
3.129	Store Double Precision Real — STD	3-205
3.130	Store Program Counter — STPC	3-206
3.131	Store Real — STR	3-207
3.132	Store Status — STST	3-208
3.133	Store Workspace Pointer — STWP	3-209
3.134	Swap Bytes — SWPB	3-209
3.135	Swap Multiple Precision — SWPM	3-210
3.136	Set Zeros Corresponding — SZC	3-212
3.137	Set Zeros Corresponding, Byte — SZCB	3-213
3.138	Test Bit — TB	3-214
3.139	Test and Clear Memory Bit — TCMB	3-215
3.140	Test Memory Bit — TMB	3-217
3.141	Translate String — TS	3-218
3.142	Test and Set Memory Bit — TSMB	3-221
3.143	Execute — X	3-223
3.143.1	X Applications	3-224
3.144	Extract Field — XF	3-224
3.145	Exit From Floating Point Interpreter — XIT	3-226
3.146	Extended Operation — XOP	3-227
3.146.1	Extended Operations	3-228
3.147	Exclusive OR — XOR	3-232
3.148	Exclusive OR Multiple Precision — XORM	3-233
3.149	Extract Value — XV	3-235

Paragraph	Title	Page
<b>4 — Application Notes</b>		
4.1	General . . . . .	4-1
4.2	Subroutines . . . . .	4-1
4.2.1	BL Instruction Common Workspace Subroutine Example . . . . .	4-1
4.2.2	BLSK Common Workspace Subroutine Example . . . . .	4-3
4.2.3	Context Switch Subroutine Example . . . . .	4-4
4.2.4	Passing Data to Subroutines . . . . .	4-8
4.2.5	Non-Reentrant Subroutine Control . . . . .	4-10
4.2.5.1	ABS Instruction . . . . .	4-10
4.2.5.2	TSMB and TCMB Instructions . . . . .	4-11
4.3	990/10 to 990/12 Upgrade Considerations . . . . .	4-12
4.3.1	Cache Usage . . . . .	4-13
4.3.2	Execution Differences . . . . .	4-14
4.3.2.1	ABS Instruction . . . . .	4-14
4.3.2.2	Second Word Modification . . . . .	4-14
4.3.2.3	Illegal Opcodes . . . . .	4-14
4.3.2.4	Workspace Crossing Map Segment Boundaries . . . . .	4-15
4.3.2.5	Deferred Mapping Error . . . . .	4-15
4.3.2.6	Error Status Register . . . . .	4-15
4.3.2.7	990/12 CPU Status Register . . . . .	4-15
4.3.2.8	Map Diagnostic Hardware . . . . .	4-15
4.3.2.9	TILINE Access to Workspace Cache . . . . .	4-15
4.3.3	Performance Differences . . . . .	4-15
4.3.3.1	Timing Loops . . . . .	4-16
4.3.3.2	Slower Instructions on the 990/12 . . . . .	4-16
4.3.3.3	Workspace Register Addressing . . . . .	4-16
4.3.3.4	Instruction Execution from Workspace Registers . . . . .	4-17
4.3.3.5	LDS Instruction . . . . .	4-17
4.3.3.6	User Device Service Routines . . . . .	4-17
4.4	Reentrant Programming . . . . .	4-17
4.5	Writing Stand-Alone Programs . . . . .	4-20
4.5.1	Interrupts . . . . .	4-20
4.5.1.1	Predefined Interrupts . . . . .	4-20
4.5.1.2	System Error Interrupt . . . . .	4-20
4.5.1.3	Error Interrupt Trace Memory . . . . .	4-22
4.5.1.4	Breakpoint System . . . . .	4-24
4.5.1.5	Twelve Millisecond Test Clock . . . . .	4-25
4.5.1.6	Forced Interrupts . . . . .	4-25
4.5.1.7	Forced Memory Errors . . . . .	4-25
4.5.2	CRU Input/Output . . . . .	4-26
4.5.2.1	CRU I/O Instructions . . . . .	4-26
4.5.2.2	CRU Base Address Development . . . . .	4-26
4.5.2.3	SBO Example . . . . .	4-27
4.5.2.4	SBZ Example . . . . .	4-27
4.5.2.5	TB Example . . . . .	4-28
4.5.2.6	LDCR Example . . . . .	4-29
4.5.2.7	STCR Example . . . . .	4-29



Paragraph	Title	Page
4.5.3	CRU Interface Example .....	4-29
4.5.3.1	Software Interface Requirements .....	4-32
4.5.3.2	Software Routines Required .....	4-32
4.5.3.3	Initialization .....	4-33
4.5.3.4	Character Transfer .....	4-34
4.5.3.5	End-of-Data Reporting .....	4-35
4.5.3.6	Interrupt Routine .....	4-35
4.5.3.7	Programming Notes .....	4-35
4.5.4	TILINE Input/Output .....	4-35
4.5.4.1	TILINE Interface Example .....	4-36
4.5.4.2	Peripheral Controller Application .....	4-37

## 5 — Assembler Directives

5.1	General .....	5-1
5.2	SDSMAC Assembler .....	5-1
5.3	Assembler Directives .....	5-2
5.3.1	Directives That Affect the Location Counter .....	5-2
5.3.1.1	Absolute Origin — AORG .....	5-3
5.3.1.2	Relocatable Origin — RORG .....	5-3
5.3.1.3	Dummy Origin — DORG .....	5-5
5.3.1.4	Block Starting with Symbol — BSS .....	5-6
5.3.1.5	Block Ending with Symbol — BES .....	5-6
5.3.1.6	Word Boundary — EVEN .....	5-7
5.3.1.7	Data Segment — DSEG .....	5-7
5.3.1.8	Data Segment End — DEND .....	5-8
5.3.1.9	Common Segment — CSEG .....	5-9
5.3.1.10	Common Segment End — CEND .....	5-10
5.3.1.11	Program Segment — PSEG .....	5-11
5.3.1.12	Program Segment End — PEND .....	5-12
5.3.2	Directives That Affect the Assembler Output .....	5-12
5.3.2.1	Output Options — OPTION .....	5-12
5.3.2.2	Program Identifier — IDT .....	5-13
5.3.2.3	Page Title — TITL .....	5-14
5.3.2.4	List Source — LIST .....	5-14
5.3.2.5	No Source List — UNL .....	5-15
5.3.2.6	Page Eject — PAGE .....	5-15
5.3.3	Directives That Initialize Constants .....	5-16
5.3.3.1	Initialize Byte — BYTE .....	5-16
5.3.3.2	Initialize Word — DATA .....	5-16
5.3.3.3	Initialize Text — TEXT .....	5-17
5.3.3.4	Define Assembly-Time Constant — EQU .....	5-17
5.3.3.5	Checkpoint Register — CKPT .....	5-18
5.3.3.6	Workspace Pointer — WPNT .....	5-19
5.3.4	Directives That Provide Linkage Between Programs .....	5-20
5.3.4.1	External Definition — DEF .....	5-20
5.3.4.2	External Reference — REF .....	5-20
5.3.4.3	Secondary External Reference — SREF .....	5-21

Paragraph	Title	Page
5.3.4.4	Force Load — LOAD . . . . .	5-22
5.3.5	Miscellaneous Directives . . . . .	5-23
5.3.5.1	Define Extended Operation — DXOP . . . . .	5-23
5.3.5.2	Program End — END . . . . .	5-24
5.3.5.3	Copy Source File — COPY . . . . .	5-24
5.3.5.4	Conditional Assembly Directives — ASMIF, ASMELS,ASMEND . . . . .	5-25
5.3.5.5	Define Operation — DFOP . . . . .	5-27
5.3.5.6	Set Right Margin — SETRM . . . . .	5-28
5.3.5.7	Set Maximum Macro Nesting Level — SETMNL . . . . .	5-28
5.4	Symbolic Addressing Techniques . . . . .	5-29

## 6 — Pseudo-Instructions

6.1	General . . . . .	6-1
6.2	No Operation — NOP . . . . .	6-1
6.3	Return — RT . . . . .	6-1
6.4	Transfer Vector — XVEC . . . . .	6-2

## 7 — Macro Language

7.1	General . . . . .	7-1
7.2	Processing of Macros . . . . .	7-1
7.3	Macro Library . . . . .	7-2
7.4	Macro Language Elements . . . . .	7-3
7.4.1	Labels . . . . .	7-3
7.4.2	Strings . . . . .	7-3
7.4.3	Constants and Operators . . . . .	7-4
7.4.4	Variables . . . . .	7-4
7.4.5	Parameters . . . . .	7-4
7.4.5.1	Macro Symbol Table . . . . .	7-4
7.4.5.2	Referencing Variables . . . . .	7-6
7.4.5.3	Variable Qualifiers . . . . .	7-7
7.4.5.4	Concatenation . . . . .	7-8
7.4.6	Symbols . . . . .	7-8
7.4.6.1	Linked Symbols . . . . .	7-9
7.4.7	Keywords . . . . .	7-11
7.4.7.1	Symbol Attribute Component Keywords . . . . .	7-11
7.4.7.2	Parameter Attribute Keywords . . . . .	7-12
7.5	Verbs . . . . .	7-13
7.5.1	\$Macro . . . . .	7-13
7.5.2	\$VAR . . . . .	7-16
7.5.3	\$ASG . . . . .	7-17
7.5.4	\$SBSTG . . . . .	7-18
7.5.5	\$NAME . . . . .	7-19
7.5.6	\$GOTO . . . . .	7-20
7.5.7	\$EXIT . . . . .	7-20
7.5.8	\$CALL . . . . .	7-20

Paragraph	Title	Page
7.5.9	\$IF .....	7-21
7.5.10	\$ELSE .....	7-22
7.5.11	\$ENDIF .....	7-22
7.5.12	\$END .....	7-22
7.5.13	Model Statements .....	7-23
7.6	Assembler Directives to Support Macro Libraries .....	7-24
7.6.1	LIBOUT Directive .....	7-24
7.6.2	LIBIN Directive .....	7-25
7.6.3	Macro Library Management .....	7-25
7.7	Macro Examples .....	7-26
7.7.1	Macro GOSUB .....	7-26
7.7.2	Macro EXIT .....	7-27
7.7.3	Macro ID .....	7-28
7.7.4	Macro UNIQUE .....	7-30
7.7.5	Macro GENCMT .....	7-31
7.7.6	Macro LOAD .....	7-31
7.7.7	Macro TABLE .....	7-33
7.7.8	Macro LISTS .....	7-34

## 8 — Relocatability and Program Linking

8.1	Introduction .....	8-1
8.2	Relocation Capability .....	8-1
8.2.1	Relocatability of Source Statement Elements .....	8-1
8.3	Program Linking .....	8-2
8.3.1	External Reference Directives .....	8-2
8.3.2	External Definition Directive .....	8-2
8.3.3	Program Identifier Directive .....	8-2
8.3.4	Linking Program Modules .....	8-2

## 9 — Assembler Output

9.1	Introduction .....	9-1
9.2	Source Listing .....	9-1
9.3	Assembler Error Messages .....	9-3
9.3.1	Abnormal Completion Error Messages .....	9-3
9.3.2	Assembly Listing Errors .....	9-4
9.4	Cross-Reference Listing .....	9-11
9.5	Object Code .....	9-12
9.5.1	Object Code Format .....	9-13
9.5.2	Machine Language Format .....	9-18
9.5.3	Symbol Table .....	9-18
9.5.4	Changing Object Code .....	9-18

## Appendixes

Appendix	Title	Page
A	Character Set .....	A-1
B	Instruction Tables .....	B-1
C	Program Organization .....	C-1
D	Hexadecimal Instruction Table .....	D-1
E	Assembler Directive Table .....	E-1
F	Macro Language Table .....	F-1
G	Example Program .....	G-1
H	Numerical Tables .....	H-1
I	Instruction Usage Cross-Reference Table .....	I-1
J	Illegal Opcodes .....	J-1

## Index

## Illustrations

Figure	Title	Page
1-1	Memory Byte .....	1-2
1-2	Memory Word .....	1-3
1-3	990 Computer Memory Map .....	1-3
1-4	Workspace Pointer and Registers .....	1-7
1-5	990 Computer Status Register .....	1-9
2-1	Source Statement Formats .....	2-13
2-2	Stacks .....	2-25
2-3	Lists .....	2-27
2-4	Memory Representation of Single Precision Real Numbers .....	2-27
2-5	Memory Representation of Double Precision Real Numbers .....	2-28
3-1	Mapping Limit Register .....	3-121
3-2	Address Development, Model 990 Mapping .....	3-123
3-3	PSHS or POPS Representation .....	3-169
3-4	Extended Operation Example .....	3-230
3-5	Extended Operation Example after Context Switch .....	3-231
4-1	Common Workspace Subroutine Example .....	4-2
4-2	PC Contents After BL Instruction Execution .....	4-2
4-3	Before Execution of BLSK Instruction .....	4-3
4-4	After Execution of BLSK Instruction .....	4-4
4-5	Before Execution of BLWP Instruction .....	4-5
4-6	After Execution of BLWP Instruction .....	4-6

Figure	Title	Page
4-7	After Execution of RTWP Instruction .....	4-7
4-8	Reentrant Procedure for Process Control .....	4-18
4-9	Error Interrupt Handling Routine .....	4-22
4-10	Breakpoint Register Interrupts .....	4-25
4-11	LDCR/STCR Data Transfer .....	4-28
4-12	LDCR Byte Transfer .....	4-29
4-13	LDCR Word Transfer .....	4-30
4-14	STCR Byte Transfer .....	4-30
4-15	STCR Word Transfer .....	4-31
4-16	CRU Bit Assignments .....	4-33
4-17	TILINE Device Controller Example .....	4-36
7-1	Macro Assembler Block Diagram .....	7-2
9-1	Cross-Reference Listing Format .....	9-12
9-2	Object Code Example .....	9-12
9-3	External Reference Example .....	9-16
9-4	Machine Instruction Formats .....	9-19

## Tables

Table	Title	Page
1-1	Dedicated Workspace Registers .....	1-6
2-1	Status Bits Tested by Instructions .....	2-3
3-1	Addressing Modes .....	3-2
3-2	Instruction Addressing .....	3-6
3-3	CRC Byte String Format .....	3-72
3-4	Memory Protection Control .....	3-121
3-5	SEQB/SNEB Status Bit Conditions .....	3-180
3-6	Search Termination Conditions .....	3-186
3-7	XOP Vectors .....	3-229
4-1	Error Interrupt Status Register (CRU Base 1FC0 <sub>16</sub> ) .....	4-21
4-2	Error Interrupt Trace Memory Data Word Bit Functions .....	4-23
4-3	CRU Output Bit Assignments for Error Interrupt Trace Control and Map Control (CRU Base Address 1FA0 <sub>16</sub> ) .....	4-23
4-4	Medium-Speed Line Printer Characteristics .....	4-31
4-5	Printer Control and Response Signals .....	4-32
7-1	Variable Qualifiers .....	7-7
7-2	Variable Qualifiers for Symbol Components .....	7-10
7-3	Symbol Attribute Keywords .....	7-11
7-4	Parameter Attribute Keywords .....	7-12
9-1	Abnormal Completion Messages .....	9-4
9-2	Assembly Listing Errors .....	9-5
9-3	Symbol Attributes .....	9-12
9-4	Object Record Format and Tags .....	9-14

# Introduction

---

## 1.1 INTRODUCTION

This manual provides detailed reference information on the 990/99000 assembly language instruction sets.

The 990/10 and 990/10A instruction sets are subsets of the 990/12 instruction set. The 990/10 instruction set consists of 72 instructions, and the 990/10A instruction set consists of 77 instructions. Both of these instruction sets can be executed on a 990/12 computer. The 990/12 instruction set consists of 144 instructions, 67 of which may execute only on a 990/12 computer. The instructions are implemented on all 990/99000 computer systems by the Software Development System Macro (SDSMAC) assembler.

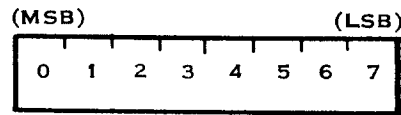
Programs written in the 990 assembly language may run under the control of an operating system, DX10 for example, or the programs may be written to run without the services provided by an operating system. The standard TI operating systems are disk-based, general-purpose, multi-tasking operating systems that are oriented toward interactive, online applications, although each supports a batch mode. Each operating system provides for the management of files, tasks, and interrupts and supports a wide variety of peripheral devices. An assembly language program written to run under the control of an operating system is called a task. A program that does not execute under control of an operating system is called a stand-alone program. Such a program must handle for itself the required services that an operating system normally provides. Considerations in writing stand-alone programs are discussed in Section 4.

## 1.2 990/10 AND 990/10A COMPUTER CHARACTERISTICS

The 990/10 minicomputer is a general purpose computer available in two chassis configurations with a wide variety of optional memory boards, power supplies, and I/O interface boards. All statements about the 990/10 and 990/10A apply equally to the 990/12 computer. The memory data word in the 990 computer is 16 bits long. Each word is also defined as two bytes of eight bits each. The maximum memory size in the 990/10 computer without the memory mapping option is 64,000 bytes. With the addition of the memory mapping option, the 990/10 memory may be expanded to a maximum of 2,048,000 bytes. The 990/10A computer can hold a maximum of 512,000 bytes.

### 1.2.1 Byte Organization

Memory for the Model 990 Computer uses byte addresses. A byte consists of eight bits of memory, as shown in Figure 1-1. The bits may represent the states of eight independent two-valued quantities or the configuration of a character code used for input, output, or data transmission. The bits also may represent a number which is interpreted either as a signed number in the range of -128 through +127 or as an unsigned number in the range of zero through 255. The 990 computer implements signed integer numbers in two's complement form.



**Figure 1-1. Memory Byte**

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

### 1.2.2 Word Organization

A word in the memory of the Model 990 Computer consists of 16 bits, a byte at an even address and a following byte at an odd address. As shown in Figure 1-2, the MSB of a memory word is designated bit zero, and the LSB is designated bit 15. A word may contain a computer instruction in machine language, a memory address, the bit configurations of two characters, or a number. When a word contains a number, the number may be interpreted as a signed number in the range of -32,768 through + 32,767 or as an unsigned number in the range of zero through 65,535. (Signed integer numbers are implemented in two's complement form.)

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

### NOTE

All instructions must begin on word boundaries. Instructions are one, two, three, or four words long.

### 1.2.3 Memory Organization

Figure 1-3 shows a generalized memory map for the 990/99000 systems without the memory mapping option. Memory mapping is described in paragraph 3.74.1. The area of low-order memory from physical addresses 0 through  $>7F$  is used for interrupt and extended operation transfer vectors. Transfer vectors are explained in paragraph 1.2.4.

The area of memory from physical addresses  $>80$  through  $>9E$  is reserved as a workspace for the front panel routine.

The area of memory from physical addresses  $>A0$  through  $>F7FE$  is available for workspaces, instructions, and data. Part of this area is used by the operating system.

When map file zero is active, addresses  $>F800$  through  $>FBFE$  are reserved for TILINE communication with peripheral devices. These addresses may be assigned to registers in controllers for direct memory access devices. Direct input/output from or to these devices is performed using any instruction that may be used to access memory. For I/O, the address in the instruction must be the TILINE address assigned to the appropriate controller register. TILINE I/O to standard 990

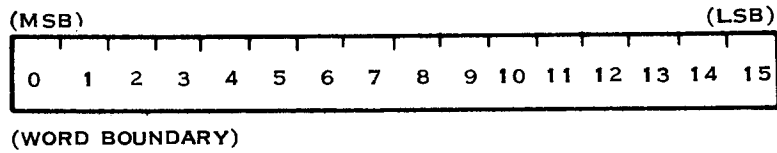
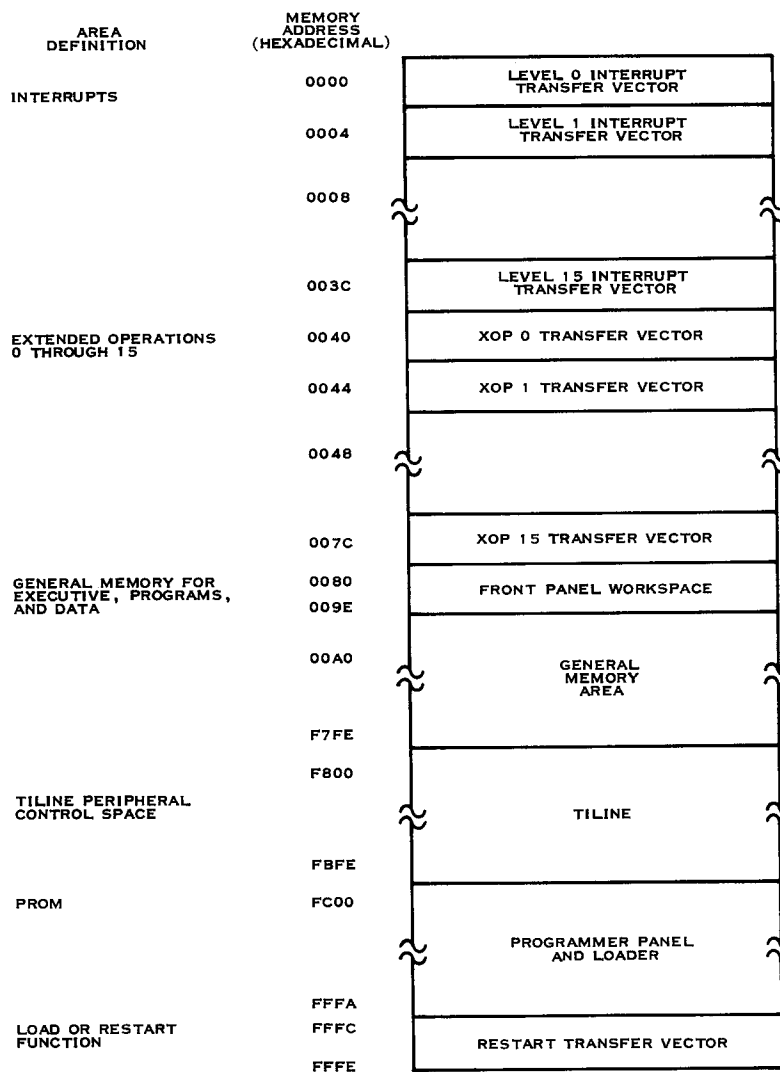


Figure 1-2. Memory Word



(A)132200A

Figure 1-3. 990 Computer Memory Map



peripheral devices is handled by the operating systems. An example of TILINE interface is shown in Section 4.

Where map file zero is active, addresses >FC00 through >FFFE are reserved for programmed read-only memory (PROM) which contains the programmer panel program, the loader program, and the self-test program.

A context switch using the transfer vector at memory location >FFFC loads the workspace pointer with the value >80, loads the program counter register with the address of a subroutine which can load or restart the computer or read the front panel, and transfers control to that subroutine. Execution of a Load or Restart Execution (LREX) instruction or activation of a switch in the control panel initiates the context switch.

#### 1.2.4 Transfer Vectors

A transfer vector is a pair of memory addresses in two consecutive words of memory. The first word contains the address of a 32-byte area of memory called a workspace. The second word contains the address of a subroutine entry point. The 990 computer uses the transfer vector in a type of transfer of control called a context switch. A context switch places the contents of the first word of a transfer vector in the workspace pointer (WP) register, making the workspace addressed by that word the active workspace. The 16 words of the active workspace become workspace registers 0 through 15, which are available for use as general purpose registers, address registers, or index registers. A context switch places the contents of the second word of a transfer vector in the program counter (PC), causing the instruction at that address to be executed next.

A context switch transfers control to an interrupt subroutine whenever an interrupt occurs. The transfer vectors for interrupt levels 0 through 15 are located in memory locations >0000 through >003E, as shown in Figure 1-3. The address of the first byte of the vector for an interrupt level is the product of the level number times four.

The 990 computer supports extended operations implemented by subroutines. These extended operations are effectively additional instructions that may perform user-defined functions. Up to 16 extended operations may be implemented. An extended operation (XOP) machine instruction results in a context switch to the specified extended operation subroutine. The transfer vectors for extended operations 0 through 15 are located in memory locations >0040 through >007E, as shown in Figure 1-3. The address of the first byte of the vector for an extended operation is the product of the extended operation number times four, plus >40.

A context switch to a user subroutine is performed by the Branch and Load Workspace (BLWP) instruction. The transfer vector must have previously been placed at a user-defined location in memory.

#### 1.2.5 Interrupts

Sixteen priority-vectored interrupt levels are implemented in the 990 computer. The contents of the interrupt mask in the status register define the interrupt level that will be recognized. Low-order memory addresses 0 through >3F are reserved for transfer vectors used by the interrupts, as detailed in Figure 1-3. When an interrupt request at an enabled level occurs, the contents of the transfer vector corresponding to the level are used to enter a subroutine to service the interrupt.

**1.2.5.1 General Interrupt Structure.** The interrupt levels, numbered 0 through 15, determine the interrupt priority. Level 0 has the highest priority and level 15 the lowest. The contents of the interrupt mask, bits 12 through 15 of the status (ST) register, determine the enabled interrupt levels. The value in the interrupt mask enables the corresponding interrupt level and all higher priority levels. For example, if bits 12 through 15 of the status register contain the value >A, interrupt levels 0 through 10 are enabled. Note that level 0 interrupts cannot be disabled, since the level contained in the mask is always enabled.

**1.2.5.2 Interrupt Sequence.** The level of the highest priority pending interrupt request is continually compared with the interrupt mask contents. When the level of the pending request is equal to or less than the mask contents (equal or higher priority), the interrupt is taken after the currently executing instruction has completed or has reached a point where it can be interrupted (interruptible instructions).

The workspace defined for the interrupt subroutine becomes active and the entry point is placed in the program counter. The CPU also stores the previous contents of the WP register in the new workspace register 13, the previous contents of the program counter in the new workspace register 14, and the previous contents of the ST register in the new workspace register 15. This series of operations preserves the program environment existing when the interrupt is taken. No additional interrupt is taken until the first instruction of the interrupt subroutine is completed. Thereafter, interrupts of higher priority can interrupt processing of the current interrupt.

After storing the ST register contents, the CPU subtracts one from the level of the interrupt taken and places the result in the interrupt mask. This operation disables all interrupts of priority equal to or less than the one taken. Higher priority interrupts will be processed. If a higher priority interrupt is taken, upon completion the previous interrupt routine is returned to at the point at which it was interrupted. If the interrupt request for that routine is still active, it is ignored and status bits 7 through 11 are reset. For more information on interrupts, see Section 4.

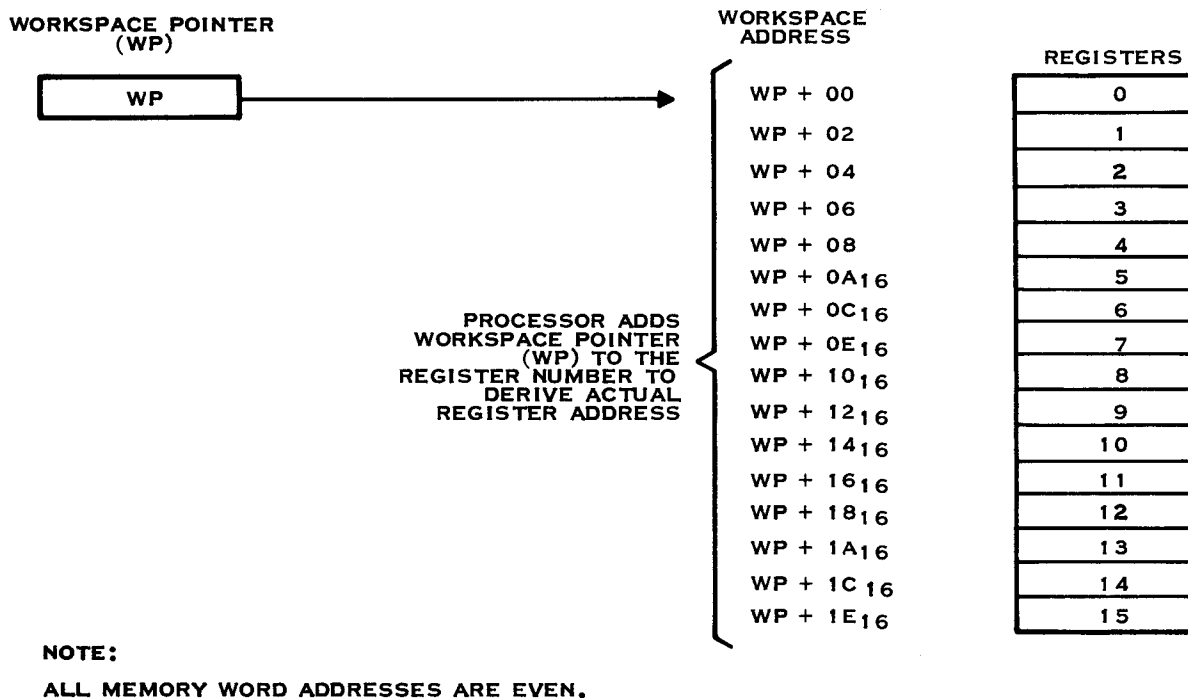
## 1.2.6 Workspace

The 990 computer employs an advanced memory-to-memory architecture. Blocks of memory designated as workspace replace internal hardware registers as program data registers. Each workspace register may hold data or an address and may function as an operand register, accumulator, address register, or index register. This architecture allows for rapid context switching without having to load and store a series of registers. The workspace concept is particularly valuable during operations that require a context switch (a change from one program to another or to a subroutine, as in the case of an interrupt). Such an operation using a conventional multiregister arrangement requires that at least part of the contents of the register file be stored and reloaded. A memory cycle is required to store or fetch each word. By exchanging the contents of the program counter, status register, and workspace pointer, the processor accomplishes a complete context switch with only three store cycles and three fetch cycles. After the switch, the workspace pointer contains the starting address of a new 32-byte workspace in memory for use in the new routine. The contents of the workspace pointer, status register, and program counter registers from the previous routine have been saved. A corresponding saving in time occurs when the original context is restored.

Any 32-byte area of general memory beginning on an even byte address may be assigned as a workspace and becomes the active workspace when the address of the first byte of the area is placed in the workspace register. Figure 1-4 shows a workspace pointer and associated registers. The registers that have assigned functions are identified in Table 1-1.

**Table 1-1. Dedicated Workspace Registers**

Register	Contents	Used During
0 (bits 12-15)	Shift count (optional)	Shift instructions (SLA,SRA, SRC,SRL)
0	Count field (optional)	Byte string instructions (990/12 only - CRC,CS,MOVS,MVSK,MVSR, POPS,PSHS,SEQB,SNEB,TCMB,TMB, TS,TSMB)
0 (bits 4-7)	Count field (optional)	(990/12 only — INSF,IOF,SLAM, SRAM,XF,XV)
0 (bits 12-15)	Count field (optional)	Multiple precision instructions (990/12 only - AM,ANDM,BDC,CNTO, DBC,INSF,IOF,LTO,NRM,ORM,RTO, SLAM,SM,SRAM,SWPM,XF,XORM,XV)
0	Floating point accumulator	Conversion instructions (990/12 only - CDI,CRI)
0-1	Floating point accumulator	Floating point instructions (990/12 only - AR,CDE,CER,CIR, CRE,CRI,DR,LR,MR,NEGR,SR,STR)
0-1	Destination operand	Signed instructions (990/12 and 990/10A only - DIVS,MPYS)
0-3	Floating point accumulator	Floating point instructions (990/12 only - AD,CDE,CDI,CED, CID,DD,LD,MD,NEGD,SD,STD)
11	Return address	Branch and link (BL) instruction
11	Effective address	Software implemented XOP instruction
12 (bits 3-14)	CRU base address	CRU instructions (SBO,SBZ,TB, LDCR,STCR)
13	Saved WP register	Context switching (BLWP,RTWP, software XOP,recognized interrupt,LOAD,RSET)
14	Saved PC register	Context switching (BLWP,RTWP, software XOP,recognized interrupt,LOAD,RSET)
15	Saved ST register	Context switching (BLWP,RTWP, software XOP,recognized interrupt,LOAD,RSET)



(A)133119

**Figure 1-4. Workspace Pointer and Registers**

The assembly language programmer should not place an incomplete workspace at the end of the user segment. Certain conditions can result in memory beyond the user's area being illegally modified with no report of a mapping violation. This typically causes a >21 system crash or a system hang.

### 1.2.7 Registers

Three internal hardware registers are accessible to the user. The program counter (PC) register contains the address of the instruction following the current instruction being executed. This register is referenced by the processor to fetch the next instruction from memory and is then automatically updated. The workspace pointer (WP) register contains the address of the first byte in the currently active set of workspace registers. A workspace register file occupies 32 contiguous memory bytes in the general memory area. The address of the first byte must be even. The status (ST) register contains the interrupt mask level and status information pertaining to the instruction operation. Each bit position in the register signifies a particular function or condition that exists in the processor. Some instructions use the status register to check for a prerequisite condition, others affect the values of the bits in the register, and others load the entire status register with a new set of parameters. The status register is described in detail in paragraph 1.4.

## 1.3 990/12 COMPUTER CHARACTERISTICS

The 990/12 minicomputer is a full-function minicomputer incorporating all of the features of the 990/10 computer plus floating-point arithmetic, byte string operations, bit-array instructions, and multiprecision integer and decimal instructions. The 990/12 is implemented with Schottky TTL

and low-power Schottky TTL technology, using high-speed workspace register and memory caches to achieve an average factor of two increase in speed over the 990/10 computer. The maximum memory size available in the 990/12 computer without the memory mapping option is 64,000 8-bit bytes. Memory may be expanded through the memory mapping option to a maximum of 2,048,000 8-bit bytes. Other features of the 990/12 include an expanded version of the 990/10 instruction set, memory cache controller, workspace cache memory, and a writable control store. The memory cache controller decreases the execution time of memory-intensive programs by automatically storing the words most recently fetched from regular memory in the high-speed cache, then always looking for the next words needed in cache memory first, thereby decreasing memory access time. The workspace cache is a 16-bit, high-speed bipolar memory. In this cache, data accessed as workspace registers is stored as it is fetched from regular memory. Subsequent accesses to the same data are made to the cache, alleviating the need for a memory cycle and decreasing the amount of time required for processing.

**NOTE**

Execution of instructions from the workspace causes the cache to be disabled with accompanying degradation of performance.

The writable control store is a 1024 by 16-bit memory containing microcode instructions. The writable control store feature allows the user to code his own assembly language instructions through microcode programming. For more information on microcode programming, refer to the *Model 990 Computer MDS-990 Microcode Development System Programmer's Guide*.

**1.4 STATUS REGISTER**

The status register of the Model 990 Computer is shown in Figure 1-5. Bits 0 through 7 are identical on both the 990/10 and 990/12. Bit 8 is used on 990/99000 systems that have the memory mapping option. Bits 9, 10, and 11 are not used on a 990/10 or 990/10A. Bits 12 through 15 contain the interrupt mask for both computers.

**1.4.1 Logical Greater Than**

The logical greater than bit (zero) of the status register contains the result of a comparison of bytes, words, real numbers, or strings as unsigned binary numbers. Bit zero set to one indicates logically greater than.

**1.4.2 Arithmetic Greater Than**

The arithmetic greater than bit (one) of the status register contains the result of a comparison of bytes, words, real numbers, or strings as two's complement signed numbers. In this comparison, the most significant bits of the operands being compared represent the sign of the number: zero for positive, one for negative. For positive integers, the remaining bits represent the binary value. For negative integers, the remaining bits represent the two's complement of the binary value. For real numbers, the remaining bits represent the exponent and the unsigned digits of the binary value. Bit one set to one indicates arithmetic greater than.

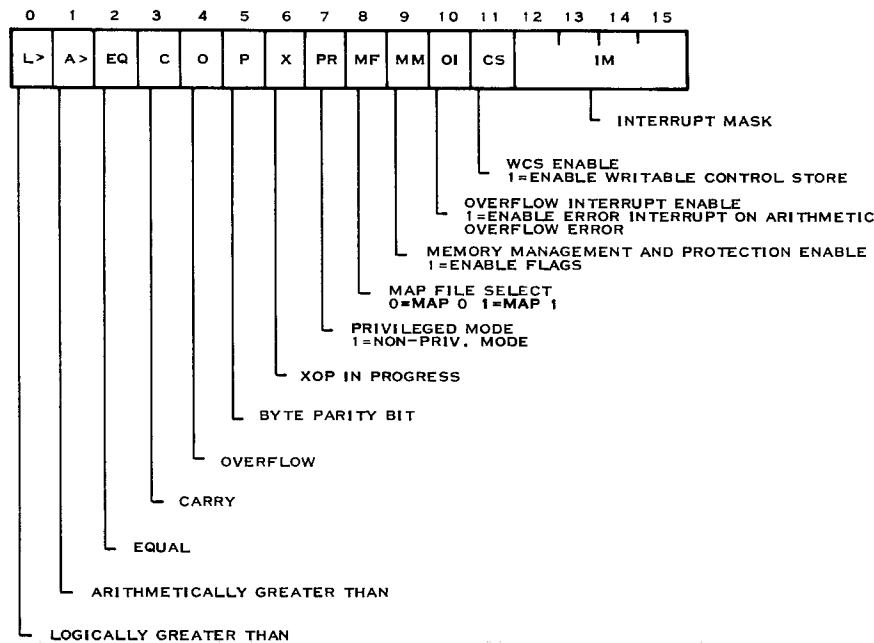


Figure 1-5. 990 Computer Status Register

### 1.4.3 Equal

The equal bit (two) of the status register is set when two bytes, words, real numbers, or strings being compared are equal. It is also set to indicate the value of a bit under certain conditions. Whether the comparison is that of unsigned binary numbers or two's complement numbers the significance of equality is the same. Bit two set to one indicates equality.

### 1.4.4 Carry

The carry bit (three) of the status register is set by a carry out of a bit of an operand during arithmetic operations. The carry bit is used by the shift operations to store the last bit shifted out of the workspace register being shifted. The carry bit is used by floating point operations to distinguish between underflow and overflow. When bit 3 is one and bit 4 is zero, a carry occurred. When both bits 3 and 4 are one, an overflow condition occurred. When bit 3 is zero and bit 4 is one, underflow occurred.

### 1.4.5 Overflow

The overflow bit (four) of the status register is set when the magnitude of the result of an arithmetic operation is too large to be correctly represented in two's complement representation. In integer addition operations, the overflow bit is set when the most significant bits of the operands are equal and the most significant bit of the result is not equal to the most significant bit of the destination operand. In integer subtraction operations, the overflow bit is set when the most significant bits of the operands are not equal, and the most significant bit of the result is not equal to the most significant bit of the destination operand. For an integer divide operation, the overflow bit is set when the most significant 16 bits of the dividend are greater than or equal to the divisor. In floating point arithmetic operations (add, subtract, multiply, and divide), overflow is set if the magnitude of the exponent cannot be represented in seven bits. For an arithmetic left shift, the overflow bit is set if the most significant bit of the operand being shifted changes value. For the absolute value and negate instructions, the overflow bit is set when the source operand is the maximum negative value,  $8000_{16}$ . Bit four set to one indicates that an overflow has occurred.

#### **1.4.6 Odd Parity**

The odd parity bit (five) of the status register is set in byte operations when the parity of the result is odd and is reset when the parity is even. The parity of a byte is odd when the number of bits having values of one is odd; when the number of bits having values of one is even, the parity of the byte is even. The odd parity bit is equal to the least significant bit of the sum of the bits in the byte. Bit five set to one indicates odd parity.

#### **1.4.7 Extended Operation**

The extended operation bit (six) of the status register is set to one when a software-implemented extended operation is initiated. An extended operation initiates a context switch using the transfer vector for the specified extended operation. After the WP and PC have been set to the values in the transfer vector, the extended operation bit is set. Bit six set to one indicates that an extended operation is in progress.

#### **1.4.8 Privileged Mode**

The privileged mode bit (seven) of the status register is set to one to inhibit execution of privileged instructions. When execution of a privileged instruction is attempted with the PR bit set to one, a privileged instruction error occurs. Bit seven must be reset to zero for execution of privileged instructions. Privileged instructions perform operating system functions not appropriate to user programs. The specific privileged instructions are identified in Section 3. The computer is placed in privileged mode and the map file select is set to map file zero when power is applied, when an interrupt occurs, or when an XOP instruction is executed.

#### **1.4.9 Map File Select**

The memory file bit (eight) of the status register provides access to memory addresses outside of the range of addresses (32K words) of the address portions of the instructions. When bit eight is set to zero, the six mapping registers of map zero are active. When bit eight is set to one, the six mapping registers of map one are active. This bit has no meaning on a 990/10 computer without the memory mapping option.

#### **1.4.10 Memory Management And Protection Enabled**

The memory management and protection enable bit (nine) of the status register is set to one to enable the management and protection flags in the mapping limit register. The memory management and protection enable bit is set or cleared by loading the status register. This bit has no meaning on a 990/10 or a 990/10A computer.

#### **1.4.11 Overflow Interrupt Enable**

The overflow interrupt enable bit (10) of the status register is set to one to allow error interrupts on arithmetic errors. An arithmetic error is any condition that would set the overflow bit (four). If the overflow interrupt bit is set to one and an overflow error occurs due to an arithmetic instruction, an error interrupt (level two) occurs and bit four of the error interrupt register is set. The overflow interrupt enable bit is set to one (enabled) or reset to zero (disabled) by loading the status register. This bit has no meaning on a 990/10 or a 990/10A computer.

#### **1.4.12 Writable Control Store**

If the writable control store bit (11) of the status register is set to one, the XOP instructions vector into the writable control store. If the bit is set to zero, XOP operates as explained in paragraph 1.2.4. The writable control store bit is set to one (enabled) or reset to zero (disabled) by loading the status register. This bit has no meaning on a 990/10 or a 990/10A computer.

### **1.4.13 Interrupt Mask**

The interrupt mask (bits 12–15) indicates the lowest priority interrupt that can occur. When an interrupt occurs, the interrupt mask is loaded with the value of the next highest priority interrupt, masking the active level and all levels below it. The interrupt mask can also be loaded under program control by the LIM1 or LIM instructions.



# General Programming Information

---

## 2.1 990 ASSEMBLY LANGUAGE

The 990 assembly language is a powerful set of instructions consisting of mnemonic operation codes (sometimes called mnemonics or opcodes) that correspond directly with binary machine instructions. The assembly language program as coded by the programmer is called a source program and must be processed by an assembler to obtain a machine language program before it can be executed by the computer. Processing of a source program is called assembling. It consists of assembling the binary values that correspond to the mnemonic operation code with the binary address information to form the machine language instruction. Assembler directives, discussed in Section 5, control the process of making a machine language program from the assembly language program, place data in the program, and assign values to symbols to be used in the program.

## 2.2 990/10 AND 990/10A INSTRUCTION SETS

The 990/10 assembly language is a set of 72 instructions that provide for the input, output, manipulation, and comparison of integer and ASCII character data. The 72 instructions are a subset of the 990/12 instruction set and may execute on any 990/99000 system. The 990/10A instruction set adds five instructions to the 990/10 instruction set. All five are applicable to the 990/12 instruction set. The 990/10 instruction set provides five modes for addressing memory. Each instruction must be stated in one of the ten instruction formats for the 990/10 and 990/10A instruction sets described in paragraph 3.4. The instructions in the 990/10 and 990/10A assembly language are introduced according to logical function in the paragraphs that follow and are described in detail in the referenced paragraph in Section 3. The instructions are divided into the following functional categories:

- Arithmetic instructions
- Branch and Jump instructions
- Compare instructions
- Control and CRU instructions
- Load and Move instructions
- Logical instructions
- Shift instructions
- Extended operation instruction
- Long-distance addressing instructions

### 2.2.1 Arithmetic Instructions

The 990/10 arithmetic instructions operate on integer values only and are listed below. Detailed descriptions are found in the referenced paragraph in Section 3.

Instruction	Mnemonic	Paragraph
Add Words	A	3.6
Add Bytes	AB	3.7
Absolute Value	ABS	3.8
Add Immediate	AI	3.10
Decrement	DEC	3.43
Decrement by Two	DECT	3.44
Divide	DIV	3.46
Divide Signed*	DIVS	3.47
Increment	INC	3.53
Increment by Two	INCT	3.54
Multiply	MPY	3.91
Multiply Signed*	MPYS	3.92
Negate	NEG	3.96
Subtract Words	S	3.107
Subtract Bytes	SB	3.108

\*Does not apply to 990/10.

### 2.2.2 Branch and Jump Instructions

The branch instructions transfer control to the specified address, which may be anywhere within the 64K byte program space. Some branch instructions also store a return address. The jump instructions also transfer control, but they are limited to a displacement of up to + 127 or – 128 words relative to the current PC. The jump instructions transfer control either unconditionally, or conditionally according to the state of one or more of the bits in the status register. Table 2-1 lists the conditional jump instructions and shows the status bit or bits tested.

Table 2-1. Status Bits Tested by Instructions

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

**Note:**

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

For all jump instructions, a displacement of zero words results in execution of the next instruction in sequence. A displacement of minus one word results in execution of the same jump instruction (a single-instruction loop). The jump instructions are:

Instruction	Mnemonic	Paragraph
Branch	B	3.16
Branch Indirect*	BIND	3.18
Branch and Link	BL	3.19
Branch and Load WP	BLWP	3.21
Jump if Equal	JEQ	3.58
Jump if Greater Than	JGT	3.59
Jump if Logical High	JH	3.60
Jump if High or Equal	JHE	3.61
Jump if Logical Low	JL	3.62
Jump if Low or Equal	JLE	3.63
Jump if Less Than	JLT	3.64
Unconditional Jump	JMP	3.65
Jump if No Carry	JNC	3.66

\* Does not apply to 990/10.

Instruction	Mnemonic	Paragraph
Jump if Not Equal	JNE	3.67
Jump if No Overflow	JNO	3.68
Jump on Carry	JOC	3.69
Jump if Odd Parity	JOP	3.70
Return with WP	RTWP	3.106
Execute	X	3.143

### 2.2.3 Compare Instructions

The compare instructions set or reset bits in the status register, usually in preparation for a jump instruction. The compare instructions perform both arithmetic and logical comparisons. The arithmetic comparison is of the two operands as two's complement values and the logical comparison is of the two operands as unsigned magnitude values. The compare instructions are as follows:

Instruction	Mnemonic	Paragraph
Compare Words	C	3.22
Compare Bytes	CB	3.23
Compare Immediate	CI	3.28
Compare Ones Corresponding	COC	3.35
Compare Zeros Corresponding	CZC	3.40

### 2.2.4 Control And CRU Instructions

The control instructions affect the operation of the Arithmetic Unit (AU) and the associated portions of the computer. CRU instructions affect the interface boards connected to the Communications Register Unit. These instructions perform functions normally handled by the operating system and are not applicable to most user tasks running under an operating system. Most of the control and CRU instructions are privileged instructions and require that the privileged bit (bit 7) of the status register be set to zero before they can execute. Instructions marked by an asterisk (\*) are privileged only if the effective CRU address is greater than >0E00. The control and CRU instructions are as follows:

Instruction	Mnemonic	Paragraph
Clock Off	CKOF	3.31
Clock On	CKON	3.32
Idle	IDLE	3.52

<b>Instruction</b>	<b>Mnemonic</b>	<b>Paragraph</b>
Load CRU	LDCR*	3.73
Load or Restart Execution	LREX	3.81
Reset	RSET	3.104
Set CRU Bit to Logic One	SBO*	3.109
Set CRU Bit to Logic Zero	SBZ*	3.110
Store CRU	STCR*	3.128
Test Bit	TB*	3.138

\* These instructions are privileged if the effective CRU address is greater than >E00.

### 2.2.5 Load and Move Instructions

The load and move instructions manipulate data between two memory locations or between hardware registers and memory locations. The LIMI and LMF instructions are privileged. The instructions are as follows:

<b>Instruction</b>	<b>Mnemonic</b>	<b>Paragraph</b>
Load Immediate	LI	3.76
Load Interrupt Mask Immediate	LIMI	3.78
Load Memory Map File	LMF	3.79
Load Status Register*	LST	3.82
Load Workspace Pointer*	LWP	3.84
Load Workspace Pointer Immediate	LWPI	3.85
Move Word	MOV	3.87
Move Byte	MOVB	3.89
Store Status	STST	3.132
Store Workspace Pointer	STWP	3.133
Swap Bytes	SWPB	3.134

\* Does not apply to 990/10.

### 2.2.6 Logical Instructions

The logical instructions provide the capability to perform various logical operations on memory locations and/or workspace registers. The instructions are as follows:

Instruction	Mnemonic	Paragraph
AND Immediate	ANDI	3.12
Clear	CLR	3.33
Invert	INV	3.56
OR Immediate	ORI	3.100
Set to One	SETO	3.113
Set Ones Corresponding (OR)	SOC	3.120
Set Ones Corresponding, Byte (OR)	SOCB	3.121
Set Zeros Corresponding	SZC	3.136
Set Zeros Corresponding, Byte	SZCB	3.137
Exclusive OR	XOR	3.147

### 2.2.7 Shift Instructions

The shift instructions permit the shifting of the contents of a specified workspace register from 1 to 16 bits. For each of these instructions, if the shift count in the instruction is zero, the shift count is taken from bits 12 through 15 of workspace register 0. If the four bits of workspace register 0 are equal to zero, the shift count is 16 bit positions. The value of the last bit shifted out of the workspace register is placed in the carry bit of the status register. The resulting value is compared to zero and the results of that comparison are shown in the logical greater than, arithmetic greater than, and equal bits (bits 0 through 2) of the status register. If a shift count greater than 15 is specified, the assembler fills in the four-bit field with the least significant four bits of the shift count and issues a warning message. The shifting instructions are:

Instruction	Mnemonic	Paragraph
Shift Left Arithmetic	SLA	3.114
Shift Right Arithmetic	SRA	3.123
Shift Right Circular	SRC	3.125
Shift Right Logical	SRL	3.127

### 2.2.8 Extended Operation Instruction

The extended operation (XOP) instruction permits the extension of the existing instruction set to include additional instructions and provides an interface between a user program and standard operating system subroutines. The additional instructions may be coded in 990 assembly

language, in which case the XOP instruction functions like a subroutine call, or they may be coded in 990 microcode. For more information on using 990 microcode, refer to the *Model 990/12 Computer MDS-990 Microcode Development System Programmer's Guide*. For more information on using the XOP instruction as a call to a user-written or operating system subroutine, refer to the manuals describing the applicable operating system.

<b>Instruction</b>	<b>Mnemonic</b>	<b>Paragraph</b>
Extended Operation	XOP	3.146

### **2.2.9 Long-Distance Addressing Instructions**

These instructions are applicable only to 990 computers equipped with the memory mapping option. The long-distance addressing instructions enable access to memory locations outside of the current memory map. These instructions require that the map enable bit (bit 4 at CRU base address >1FA0) be set to one. The instructions are as follows:

<b>Instruction</b>	<b>Mnemonic</b>	<b>Paragraph</b>
Long-Distance Destination	LDD	3.74
Long-Distance Source	LDS	3.75

## **2.3 990/12 INSTRUCTION SET**

The 990/12 instruction set includes all of the instructions listed in the preceding paragraphs and 72 additional instructions. The additional instructions provide the capability to manipulate stacks, lists, strings, individual memory bits, multiple-precision integers, and single- and double-precision real numbers, to enable or disable interrupts, to execute microcode diagnostics, to load the writable control store, and to convert one type of data to another. The paragraphs that follow list the 990/12 extensions to the nine functional groups of instructions described for the 990/10 instruction set and introduce a tenth functional group of conversion instructions.

### **2.3.1 Arithmetic Instructions**

The additional 990/12 arithmetic instructions provide the capability to manipulate, in hardware, multiple-precision integer, single- and double-precision real, and signed numbers. The additional instructions are as follows:

<b>Instruction</b>	<b>Mnemonic</b>	<b>Paragraph</b>
Add Double-Precision Real	AD	3.9
Add Multiple-Precision Integer	AM	3.11
Add Real	AR	3.14
Count Ones	CNTO	3.34
Divide Double-Precision Real	DD	3.42

<b>Instruction</b>	<b>Mnemonic</b>	<b>Paragraph</b>
Divide Signed	DIVS	3.47
Divide Real	DR	3.48
Multiply Double-Precision Real	MD	3.86
Multiply Signed	MPYS	3.92
Multiply Real	MR	3.93
Negate Double-Precision Real	NEGD	3.97
Negate Real	NEGR	3.98
Subtract Double-Precision Real	SD	3.111
Subtract Multiple-Precision Integer	SM	3.118
Subtract Real	SR	3.122

### 2.3.2 Jump Instructions

The additional 990/12 jump instructions enable the manipulation of a register and jumping in a single instruction, the use of a stack to maintain subroutine linkage, and a branch indirect. The instructions are as follows:

<b>Instruction</b>	<b>Mnemonic</b>	<b>Paragraph</b>
Add to Register and Jump	ARJ	3.15
Branch Indirect	BIND	3.18
Branch Immediate and Push Link to Stack	BLSK	3.20
Subtract from Register and Jump	SRJ	3.126

### 2.3.3 Compare Instructions

The additional 990/12 compare instructions enable the comparison and searching of strings, testing for the presence of ones, and the testing and setting of individual memory bits. The instructions are as follows:

<b>Instruction</b>	<b>Mnemonic</b>	<b>Paragraph</b>
Compare Strings	CS	3.39
Left Test for Ones	LTO	3.83
Right Test for Ones	RTO	3.105



<b>Instruction</b>	<b>Mnemonic</b>	<b>Paragraph</b>
Search String for Equal Byte	SEQB	3.112
Search List for Logical Address	SLSL	3.116
Search List for Physical Address	SLSP	3.117
Search String for Not Equal Byte	SNEB	3.119
Test Memory Bit	TMB	3.140

### 2.3.4 Control And CRU Instructions

The additional 990/12 control and CRU instructions can enable or disable interrupts, execute a microcode diagnostic routine, and load the writable control store. The DINT, EINT, LCS, and EMD instructions are privileged. The 990/12 control and CRU instructions are as follows:

<b>Instruction</b>	<b>Mnemonic</b>	<b>Paragraph</b>
Disable Interrupts	DINT	3.45
Enable Interrupts	EINT	3.49
Execute Micro Diagnostic	EMD	3.50
Load Writable Control Store	LCS	3.71

### 2.3.5 Load and Move Instructions

The additional 990/12 load and move instructions enable the manipulation of real numbers, strings, stacks, lists, and fields. The instructions are as follows:

<b>Instruction</b>	<b>Mnemonic</b>	<b>Paragraph</b>
Insert Field	INSF	3.55
Load Double-Precision Real	LD	3.72
Load Interrupt Mask	LIM	3.77
Load Real	LR	3.80
Load Status Register	LST	3.82
Load Workspace Pointer	LWP	3.84
Move Address	MOVA	3.88
Move String	MOVS	3.90

<b>Instruction</b>	<b>Mnemonic</b>	<b>Paragraph</b>
Move String from Stack	MVSK	3.94
Move String Reverse	MVSR	3.95
Pop String From Stack	POPS	3.102
Push String to Stack	PSHS	3.103
Store Double-Precision Real	STD	3.129
Store Program Counter	STPC	3.130
Store Real	STR	3.131
Swap Multiple-Precision	SWPM	3.135
Extract Field	XF	3.144
Extract Value	XV	3.149

### 2.3.6 Logical Instructions

The additional 990/12 logical instructions enable the manipulation of multiple precision values and single memory bits. The instructions are as follows:

<b>Instruction</b>	<b>Mnemonic</b>	<b>Paragraph</b>
AND Multiple-Precision	ANDM	3.13
Invert Order of Field	IOF	3.57
OR Multiple-Precision	ORM	3.101
Test and Clear Memory Bit	TCMB	3.139
Test and Set Memory Bit	TSMB	3.142
Exit from Floating Point Interpreter	XIT	3.145
Exclusive OR Multiple-Precision	XORM	3.148

### 2.3.7 Shift Instructions

The additional 990/12 shift instructions enable the shifting and normalization of multiple-precision values. The additional instructions are as follows:

Instruction	Mnemonic	Paragraph
Normalize	NRM	3.99
Shift Left Arithmetic Multiple-Precision	SLAM	3.115
Shift Right Arithmetic Multiple-Precision	SRAM	3.124

### 2.3.8 Extended Operation and Long-Distance Addressing Instructions

No extensions have been added to the 990/12 instruction set for these two groups. Any instruction in these two groups may be executed on either computer. The long-distance addressing instructions require that the memory mapping option be installed.

### 2.3.9 Conversion Instructions

The 990/12 conversion instructions enable the conversion of one data type to another and the calculation of a cyclic redundancy code. The instructions are as follows:

Instruction	Mnemonic	Paragraph
Binary to Decimal ASCII Conversion	BDC	3.17
Convert Double-Precision Real to Extended Integer	CDE	3.24
Convert Double-Precision Real to Integer	CDI	3.25
Convert Extended Integer to Double- Precision Real	CED	3.26
Convert Extended Integer to Real	CER	3.27
Convert Integer to Double- Precision Real	CID	3.29
Convert Integer to Real	CIR	3.30
Cyclic Redundancy Code Calculation	CRC	3.36
Convert Real to Extended Integer	CRE	3.37
Convert Real to Integer	CRI	3.38
Decimal ASCII to Binary Conversion	DBC	3.41

Instruction	Mnemonic	Paragraph
Extend Precision	EP	3.51
Translate String	TS	3.141

## 2.4 SOURCE STATEMENT FORMAT

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

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

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

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

```
[<label>]b...opcodeb...[<operand>]...b...[<comment>]
```

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

Comment statements are recognized by an asterisk (\*) in the first character position. Comment statements are listed in the source portion of the assembly listing and have no other effect on the assembly.

The length of source records to be scanned by the assembler is set by the set right margin (SETRM) assembler directive. The default value is 60 characters. Using the SETRM directive, the length to be scanned may range from ten to 80 characters. Assembler directives are described in Section 5. Only the first 52 characters are printed on a Model 733 ASR data terminal. The last



### 2.4.1 Character Set

The assembler for the Model 990 Computer recognizes the ASCII character set and special characters that are undefined in ASCII. Appendix A contains tables that list the 990 character set with the ASCII and Hollerith codes.

### 2.4.2 Label Field

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

### 2.4.3 Operation Field

The operation (opcode) field begins following the blank that terminates the label field, or in the first nonblank character after the first character position when the label is omitted. The operation field is terminated by one or more blanks and may not extend past the right margin as specified in the SETRM assembler directive or character position 60 if SETRM has not been used. The operation field contains one of the following types of opcodes:

- Mnemonic operation code of a machine instruction.
- Assembler directive operation code.
- Symbol assigned to be an extended operation by a DXOP directive.
- Symbol assigned to be a previously defined operation by a DFOP directive.
- Pseudo-instruction operation code.
- Macro name.

### 2.4.4 Operand Field

The operand field begins following the blank that terminates the operation field and may not extend past the right margin of the source record. The operand field may contain one or more expressions or constants, according to the requirements of the operation. The operand field is terminated by one or more blanks.

### 2.4.5 Comment Field

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

## 2.5 CONSTANTS

The assembler recognizes four types of constants: decimal integer constants, hexadecimal integer constants, character constants, and assembly-time constants.

### 2.5.1 Decimal Integer Constants

A decimal integer constant is written as a string of numerals. The range of values of decimal integers is  $-32,768$  to  $+32,767$ . Positive decimal integer constants in the range  $32,768$  to  $65,535$  are considered negative when interpreted as two's complement values.

The following are valid decimal constants:

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

### 2.5.2 Hexadecimal Integer Constants

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

The following are valid decimal constants:

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

### 2.5.3 Character Constants

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

The following are valid character constants:

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

### 2.5.4 Assembly-Time Constants

An assembly-time constant is a symbol given a value by an EQU directive, described in a subsequent paragraph. The value of the symbol is determined at assembly time and is considered to be absolute or relocatable according to the relocatability of the expression, not according to the relocatability of the location counter value.

## 2.6 SYMBOLS

Symbols are used in the label field, the operator field, and the operand field. A symbol is a string of alphanumeric characters, (A through Z, zero through nine ‘;’ and ‘\$’), the first of which must be an alphabetic character (A through Z), ‘;’ or ‘\$’ and none of which may be blank. When more than six characters are used in a symbol, the assembler prints all the characters, but accepts only the first six characters for processing. The assembler also prints a warning indicating that the symbol has been truncated. Therefore symbols must be unique in the first six characters. User-defined symbols are valid only during the assembly in which they are defined.

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

Symbols used in the operator field must be either predefined instruction mnemonics, assembler directives, defined in a DFOP or DXOP, or defined as a macro name.

Symbols used in the operand field must be defined in the assembly, usually by appearing in the label field of a statement or in the operand field of a REF or SREF directive.

The following are examples of valid symbols:

START	Assigned the value of the location at which it appears in the label field.
AI	Assigned the value of the location at which it appears in the label field.
OPERATION	OPERAT is assigned the value of the location where it appears in the label field.

### 2.6.1 Predefined Symbols

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

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



The following is an example of a valid predefined symbol:

\$ Represents the current location.

### 2.6.2 Character Strings

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

The following are valid character strings:

'SAMPLE PROGRAM'	Defines a 14-character string consisting of SAMPLE PROGRAM
'PLAN\$C''	Defines an 8-character string consisting of PLAN\$C'
'OPERATOR MESSAGE*PRESS START SWITCH'	Defines a 37-character string consisting of the expression enclosed in single quotes

## 2.7 EXPRESSIONS

Expressions are used in the operand fields of assembler directives and machine instructions. An expression is a constant, a symbol, or a series of constants, a series of symbols, or a series of constants and symbols separated by arithmetic operators. Each constant or symbol may be preceded by a minus sign (unary minus) or a plus sign (unary plus). An expression may not contain embedded blanks. Symbols that are defined as external references may be operands of arithmetic instructions within certain limits, described in the next paragraph.

An expression may contain more than one symbol that is not previously defined. When the symbols in an expression have values which are absolute, they may also be operands of multiplication or division operations within an expression. An expression that contains a relocatable symbol or relocatable constant immediately following a multiplication or division operator is an illegal expression. Also, when the result of evaluating an expression up to a multiplication or division operator is relocatable, the expression is illegal. An expression in which the number of relocatable symbols or constants added to the expression exceeds the number of relocatable symbols or constants subtracted from the expression by more than one is an illegal expression.

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

Then if

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

An expression containing relocatable symbols or constants of several different relocation types (see Section 5) is absolute if it is absolute with respect to all relocation types. If it is relocatable with respect to one relocation type and absolute with respect to all other relocation types, then the expression is relocatable.

An expression is represented in the syntax definitions by <exp>.

Two other types of expressions are used in the operand field:

- Well-defined expressions
- Nibble expressions

For an expression to be well-defined, any symbols or assembly-time constants must have been previously defined. Also, the evaluation of a well-defined expression must be absolute. A well-defined expression is represented in the syntax definitions by <wd-exp>.

A nibble expression is an expression which evaluates to an absolute number in the range 0 through 15. This expression's result will be one hexadecimal digit (four bits). A nibble expression may be represented in the syntax definition by several conventions (see Syntax Definition in Section 3).

The following are examples of valid expressions:

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

### 2.7.1 Use of External Referenced Symbols in Expressions

The assembler allows external referenced symbols (defined in REF and SREF directives) in expressions under the following conditions:

1. Only one externally referenced symbol may be used in an expression.
2. The character preceding the referenced symbol must be a plus sign, a blank, or a comma (at sign "@" is not considered). The portion of the expression preceding the symbol, if any, must be added to the symbol.

3. The portion of the expression following the referenced symbol must not include multiplication, division, or logical operations on the symbol (same as for a relocatable symbol, paragraph 2.7).
4. The remainder of the expression except the referenced symbol must be absolute (relocatable symbols may be used).

As a result of this feature, the assembler limits the user to a total of 255 external referenced symbols per module. Modules using more than 255 external symbols must be broken into smaller modules for assembly and linked using the link editor.

### 2.7.2 Arithmetic Operators in Expressions

The arithmetic operators in expressions are as follows:

- + for addition
- - for subtraction
- \* for multiplication
- / for signed division
- // for logical right shift.

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

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

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

#### Examples:

$8000 // 1 = 4000$

$AAAB // 1 = 5555$

$FFFF // 0 = FFFF$

$FFFF // 16 = 0000$

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

$4000 * 2$      $7FFF + 1$      $- 1 * > 8000$

$8000 * 2$      $8000 - 1$      $- 2 * > 8001$

### 2.7.3 Logical Operators in Expressions

The assembler supports logical operations in expressions that are the bit-by-bit logical operations between the values of the symbols and/or constants. The logical operators are as follows:

- & for AND
- && for exclusive OR
- + + for OR
- # for NOT (logical complement)

The order of evaluation of expressions that contain logical operators is similar to that of expressions that contain only arithmetic operators. Like the unary minus, the logical complement takes precedence over other operations, regardless of position, except as altered by parentheses.

The following are examples of expressions that contain logical operators:

BLUE&&255	Specifies the result of an exclusive OR operation between the bits of the value of symbol BLUE and the bits of constant value 255.
GREEN + + 15	Specifies the result of an OR operation between the bits of the value of symbol GREEN and the bits of constant value 15.
RED&#255	Specifies the result of an AND operation between the bits of the value of symbol RED and the inversion of the bits of constant value 255.
RED&#255 + + (BLUE&255)	ANDs the value of BLUE with the constant 255. ANDs the value of RED with one's complement of 255. ORs the two AND results to get the value of the expression.

### 2.7.4 Relational Operators in Expressions

The assembler supports six relational operators that represent the relationship between two constants and/or symbols, that is, the result of comparing the constants and/or symbols. When the relationship corresponding to the operator exists (is true), the value of the combination is one. When the relationship corresponding to the operator does not exist (is not true), the value of the combination is zero. The result may be used as an arithmetic value or as a logical value. The relational operators are as follows:

- = for equal
- < for less than
- > for greater than
- <= for less than or equal

- `>=` for greater than or equal
- `#=` for not equal.

### NOTE

The greater than character (`>`) is also used to identify hexadecimal constants. The context determines the meaning of the greater than character in each statement.

The following are examples of expressions that contain relational operators:

<code>BLUE# = GREEN</code>	Compares the value of symbol <code>BLUE</code> to the value of symbol <code>GREEN</code> . When the values are not equal, the combination has a value of one. When the values are equal, the combination has a value of zero.
<code>WHITE&lt;BLACK</code>	Compares the value of symbol <code>WHITE</code> to the value of symbol <code>BLACK</code> . When the value of <code>WHITE</code> is less than the value of <code>BLACK</code> , the combination has a value of one. Otherwise, the value of the combination is zero.
<code>RED*(GREEN = 0)</code>	Compares the value of symbol <code>GREEN</code> to zero. When <code>GREEN</code> equals zero, the value of symbol <code>RED</code> is multiplied by one, and the value of the expression is that of symbol <code>RED</code> . When <code>GREEN</code> is not equal to zero, the multiplier is zero, and the value of the expression is zero.
<code>BLUE&gt;= RED</code>	Compares the value of symbol <code>BLUE</code> to the value of symbol <code>RED</code> . When <code>BLUE</code> is greater than or equal to <code>RED</code> , the combination is equal to one. When <code>BLUE</code> is less than <code>RED</code> , the combination is equal to zero.

#### 2.7.5 Use of Parentheses in Expressions

The assembler supports the use of parentheses in expressions to alter the order of evaluation of the expression. Nesting of pairs of parentheses within expressions is also supported. When parentheses are used, the portion of the expression within the innermost parentheses is evaluated first. Then the portion of the expression within the next-innermost pair is evaluated. When evaluation of the portions of the expression within all parentheses has been completed, the evaluation is completed from left to right. Evaluation of portions of an expression within parentheses at the same nesting level may be considered to be simultaneous.

For example, the use of parentheses in the expression `LAB1 + ((4 + 3)*7)` would result in the addition of four and three. The result, seven, would be multiplied by seven, giving 49. The complete

evaluation would be the value of LAB1 plus 49. Without parentheses, four would have been added to the value of LAB1, three would have been added to the sum, and the value of the second addition would have been multiplied by seven if LAB1 had an absolute value. If LAB1 had a relocatable value, the expression would have been illegal without the parentheses.

## 2.8 990/12 DATA TYPES

In addition to the data types described in the preceding paragraphs, which are applicable to both computers, there are seven other data types, applicable only to the 990/12 instruction set, which are listed below and explained in the paragraphs that follow.

- Extended Integers
- Multiple Precision Integers
- Byte Strings
- Stacks
- Lists
- Single Precision Real Numbers
- Double Precision Real Numbers

### 2.8.1 Extended Integers

An extended integer represents an integer value in the range  $-2^{31}$  to  $+(2^{31}-1)$ . The extended integer uses two consecutive 16-bit words in memory. The value is right-justified in the double word. Extended integers are represented in two's complement form.

### 2.8.2 Multiple Precision Integers

A multiple precision integer is a series of one to 16 consecutive bytes. A length may be specified in one of two ways using the 990/12 multiple precision integer instructions:

- A length of one to 15 bytes may be specified in the instruction.
- If a zero is specified in the instruction, then the length is fetched from the four LSBs of workspace register R0. If the four LSBs are zero, then the length is 16 bytes.

When using the multiple precision instructions to manipulate these integers, there is no way to specify an integer of zero precision. Instructions operating on two multiple precision operands operate only on operands of the same precision.

Multiple precision integers also may be manipulated using the byte string instructions such as MOVS or CS, which are described in Section 3. Note that the method of specifying a length of 16 bytes using byte string instructions is slightly different than multiple precision instructions. Multiple precision integers use the four LSBs of workspace register zero; strings use all 16 bits of register zero.

Multiple precision instruction:

CLR R0	CLEAR R0 SO THAT 4 LSBs EQUAL 0
AM @INTEGER,@ACCUM,0	ADD MULTIPLE PRECISION INTEGERS OF LENGTH 16 (R0 BITS 12-15) = 0)

Byte string instruction:

SET0 R7	INITIALIZE CHECKPOINT REGISTER
LI R0,16	LOAD R0 WITH 16
MOVS @INTEGER,@ACCUM,0,R7	MOVE STRING OF LENGTH 16

### 2.8.3 Byte Strings

A byte string is a group of consecutive bytes that have a specified general address and length. The length of the byte string may be specified in one of three ways.

- If the byte string length is from 1 to 15 bytes, the length can be specified in the instruction.
- If the byte string length is from 0 to  $FFFE_{16}$  bytes, length may be specified in workspace register zero.
- If R0 is equal to  $FFFF_{16}$ , the byte string length is in the first byte of the string. This type of string is referred to as a tagged string, and the length specified is from 1 to 256. The tag byte is the most significant byte of the string and is included in the string length. A tag value of zero indicates a string length of 256.

When an instruction is encountered with a byte string operand, the length is searched for in the following order:

- in the instruction;
- in R0, if instruction specifies length of zero;
- in the tag, if R0 equals  $FFFF_{16}$ .

A zero length string is specified by a length field in the instruction equal to zero and workspace register zero equal to zero.

The following examples illustrate these methods of specifying the string length in these three respective ways using the Move String instruction:

- MOVS                      A,B,4, <ckpt >      A length of 4 explicitly in the instruction

- LI R0,80 A length of 80 in workspace register zero  
MOVS A,B,,<ckpt>
- SETO R0 As a tagged string  
MOVS A,B,,<ckpt>

#### 2.8.4 Stacks

A stack, as illustrated by Figure 2-2, is an area of accessible consecutive memory that is used for storing, calculating, and manipulating byte strings of information. The stack is addressed using a three-word control block. A stack may be addressed also with a register that contains the top of stack through direct workspace register addressing.

The control block contains the following information:

- Word 1 — the address of the first byte containing data (the top of stack [TOS])
- Word 2 — the lowest address of the stack (the stack limit)
- Word 3 — the highest address of the stack + 1 (the bottom of the stack)

The stack grows from high addresses to low addresses. An empty stack is described with (Word 1) = (Word 3)

#### 2.8.5 Lists

A list is a group of data blocks that are linked together by linkage words. Each data block contains at least one linkage word and an arbitrary number (possibly zero) of data words. A List Search Control Block (LSCB) is used in searching the list. The LSCB is five words long and contains the following information:

- Word 0 — Signed byte displacement to link word (the LSB is ignored)
- Word 1 — Signed byte displacement to compare word (the LSB is ignored)
- Word 2 — Test value to be used
- Word 3 — Test mask to be used
- Word 4 — Terminal link value

The List Search Control Block is located at the source address of the list instructions. The destination address specifies a two-word block, as follows:

- Word 0 — Pointer to the beginning of the list (or the first element of the list involved in the search)
- Word 1 — Pointer to the previous element in the list

Word 0 of the destination address can point to any place in the data block; the signed byte displacement to the link word (word zero of the LSCB) is added to the pointer to access the next linkage word. When a data block is accessed, and the list search is incomplete, word zero of the



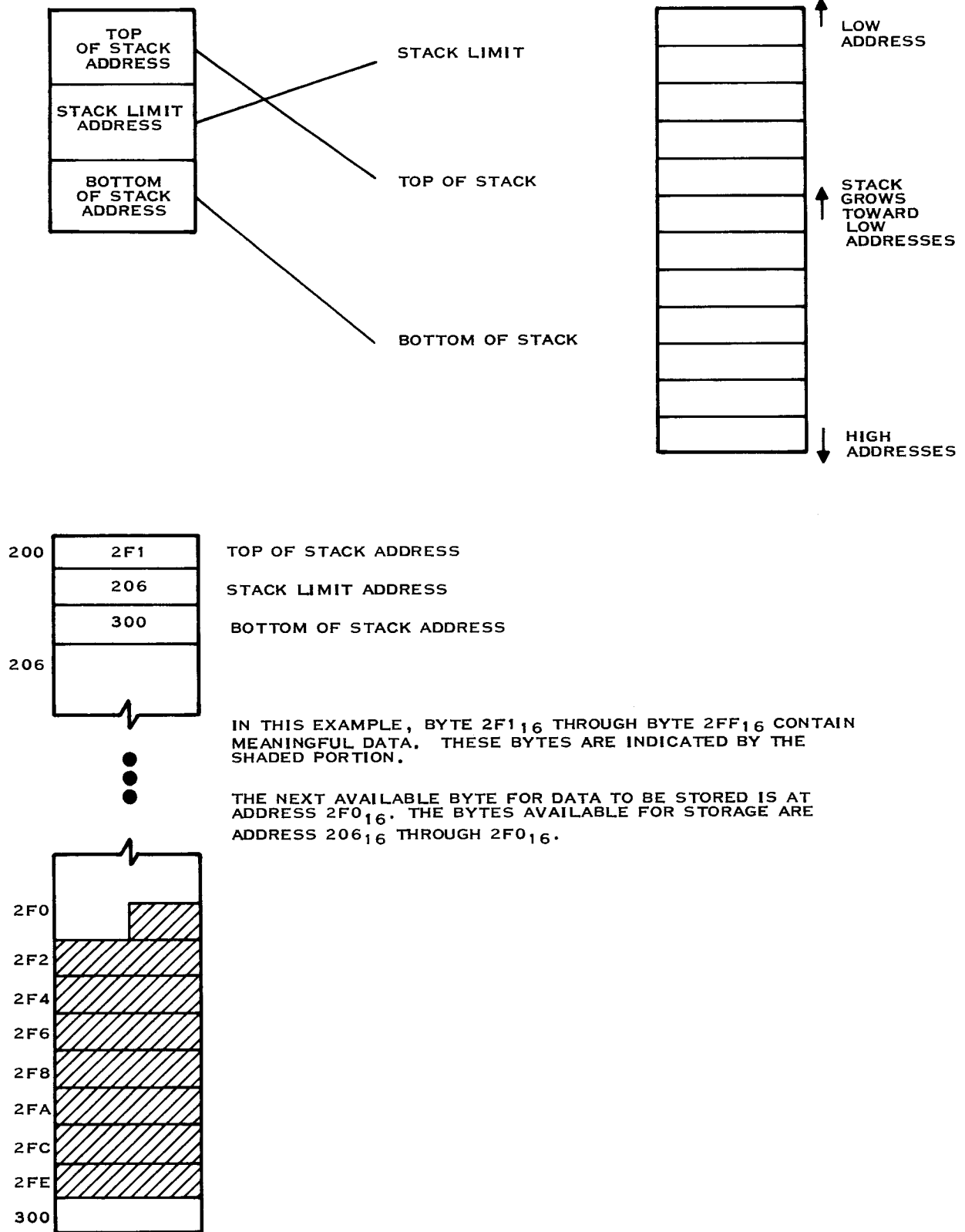


Figure 2-2. Stacks

destination address is moved to word one, and the linkage word in the data block is moved into word zero. The linkage word in each data block must be located at the same displacement from the previous pointer. The compare word in each data block must also be the same displacement from the pointer. Figure 2-3 illustrates the list data structure and the control blocks used in list instructions.

### 2.8.6 Single Precision Real Numbers

Single precision real numbers (floating point numbers) represent any value within the approximate range  $10^{-78}$  to  $10^{75}$ , including zero. Single precision real numbers are stored in memory in two 16-bit words as illustrated in Figure 2-4. The number consists of a normalized hexadecimal fraction, a corresponding hexadecimal exponent, and a sign bit.

The fractional portion of the number (mantissa) is normalized; that is, it is shifted to the left to eliminate leading zeros between the radix point and the first significant digit of the fraction. Normalization is by hexadecimal digits, not by bits. Each digit position shift in the normalization process produces a corresponding change in the exponent portion of the number to maintain the correct magnitude of the number. When completely normalized, the hexadecimal mantissa is stored in bits 8 through 15 of the first memory word and in the entire second memory word. The radix point for the fraction is assumed to be positioned between bits seven and eight of the first memory word (at the start of the hexadecimal fraction).

The exponent portion of the number is a hexadecimal exponent. The exponent is biased by  $40_{16}$  (excess 64 notation), so that an exponent for the number  $16^0$  ( $.1X16^1$ ) is represented in memory by  $41_{16}$ . Exponents of zero are represented by  $40_{16}$ , except for the number zero. The number zero is represented with the exponent and mantissa both as 0. Positive exponents, therefore, are represented by numbers greater than  $40_{16}$ , and negative exponents are represented by numbers less than  $40_{16}$ . For example, a normalized  $16^{-8}$  is represented in the exponent field by a value of  $39_{16}$ . The exponent may be any value from  $00_{16}$  to  $7F_{16}$ . Using the  $40_{16}$  bias value, these numbers represent exponent values from  $-40_{16}$  to  $+3F_{16}$  ( $16^{-64}$  to  $16^{63}$ ). The seven exponent bits are stored in bits one through seven of the first memory word.

Bit 0 of the first memory word is used for a sign bit. When this bit is a zero, the number is positive; when this bit is one, the number is negative.

#### Single Precision Examples:

Base Ten Number	Hexadecimal Contents of Memory Words	
	Word 1	Word 2
1.0	4110	0000
0.5	4080	0000
101.0	4265	0000
.03125 (1/32)	3F80	0000
- 1.0	C110	0000

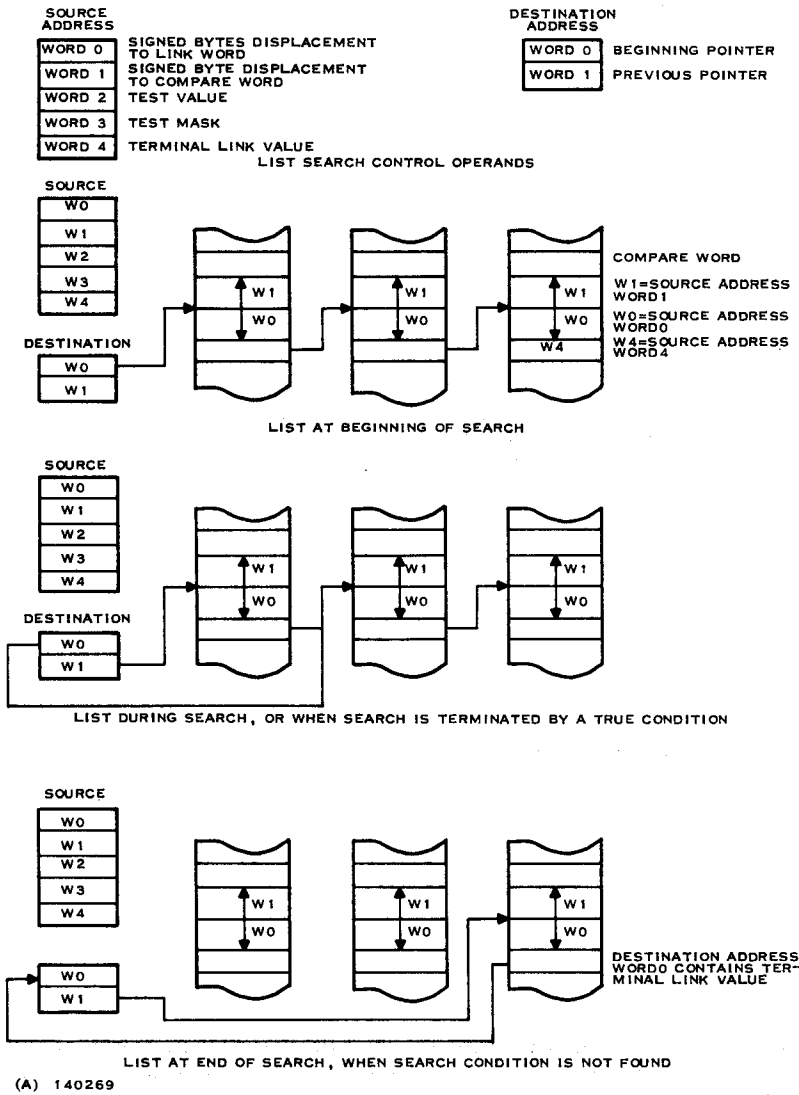


Figure 2-3. Lists

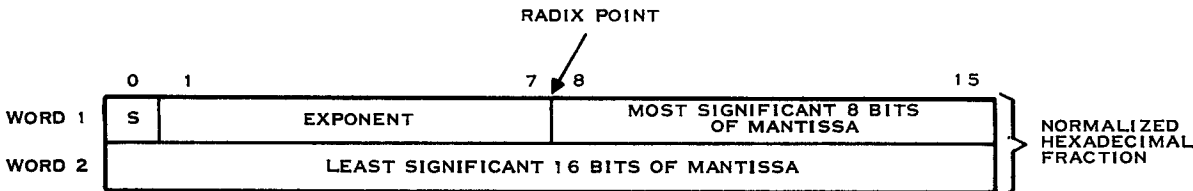


Figure 2-4. Memory Representation of Single Precision Real Numbers

**2.8.7 Double Precision Real Numbers**

Double precision real numbers are similar to single precision real numbers, except that they occupy two more memory words and provide a 56-bit mantissa instead of the 24 bits available with single precision real numbers. Double precision real numbers have values from  $10^{-78}$  to  $10^{75}$ , including zero.

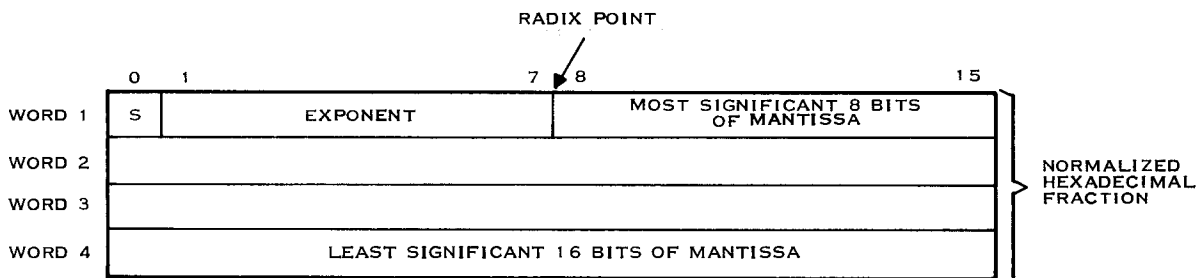
Double precision real numbers are stored in memory in four 16-bit words as illustrated in Figure 2-5. The most significant bit of the first word is a sign bit for the mantissa; zero if the number is positive and one if it is negative. Bits one through seven of the first word are the exponent. The exponent follows the same form as for single-precision real number exponents. The remaining bits of the first word of the other three words contain the significant digits of the double precision mantissa normalized in the same manner as a single precision floating point.

**2.8.8 Floating Point Accumulator (FPA)**

If a floating point operation requires two operands, such as multiply or subtract, then the second operand is assumed to exist in an implicit "accumulator" register created by the results of a load instruction or a previous calculation. The implicit accumulator acts as a single register that participates in all floating operations as either an operand or result, or both. The outcome of all floating point operations (except the store operations), is placed in the implicit accumulator. Single precision real instructions use R0 and R1 as the FPA, leaving R2 and R3 unaltered. Double precision real instructions use R0, R1, R2, and R3 as the FPA.

**2.9 REEXECUTABLE INSTRUCTIONS**

Certain instructions (CS, SEQB, SNEB) can be reexecuted to continue a search or comparison from the point the instruction ended. For example, the search equal byte (SEQB) instruction can be executed to find the first equal byte of a byte string, the second, and so on. This is made possible by the checkpoint register: when the instruction ends on the proper search termination condition, the checkpoint register returns a displacement into the string where the byte meeting that condition is located. By executing a jump that goes back to that instruction, or by recoding the instruction with the same checkpoint register (unmodified), the search can be made to continue from the point it ended, which is the value in the checkpoint register. Also by making the initial value of the checkpoint register greater than -1, the search can be made to begin at some point inside the string. String and stack instructions use the value of the checkpoint register plus one as the initial index into the string. This is true for the other string instructions also (MOVS, MVSR, PSHS, MVSK, TS, and CRC). For more information on reexecutable instructions, see paragraph 3.39.1.



**Figure 2-5. Memory Representation of Double Precision Real Numbers**

# Assembly Instructions

## 3.1 GENERAL

This section describes the mnemonic instructions of the 990 assembly language. Detailed assembly instruction descriptions follow descriptions of the addressing modes used in the assembly language and the instruction formats of the assembly instructions. The section also includes examples of programming the various instructions.

## 3.2 ADDRESSING MODES

Eight addressing modes are featured in the 990 assembly language. Three of these modes, program counter relative addressing, CRU bit addressing, and immediate addressing, are special purpose addressing modes discussed in paragraphs 3.2.6, 3.2.7, and 3.2.8, respectively. The remaining five modes are used in the instructions that specify a general address for the source or destination operand. Table 3-1 lists these modes and shows how each is used in the assembly language. Each of the modes is described in a subsequent paragraph.

### 3.2.1 Workspace Register Addressing

Workspace register addressing specifies a workspace register that contains the operand. A workspace register address is written as an expression having an absolute value of 0 through 15.

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

**MOV R4,R8**

Copy the contents of workspace register four into workspace register eight.

**COC R15,R10**

Compare to one the bits of workspace register ten that correspond to the bits set to one in workspace register 15.

### 3.2.2 Workspace Register Indirect Addressing

Workspace register indirect addressing specifies a workspace register that contains the address of the operand. An indirect workspace register address is written as an expression having an absolute value of 0 through 15 preceded by an asterisk (\*). The following examples show the coding of instructions that have workspace register indirect addresses.

**A \*R7,\*R2**

Add the contents of the word at the address in workspace register seven to the contents of the word at the address in workspace register two, and place the sum in the word at the address in workspace register two.

MOV \*R7,R0

Copy the contents of the word at the address in workspace register seven into workspace register zero.

**Table 3-1. Addressing Modes**

Addressing Mode	T Field Value <sup>1</sup>	Example
Workspace Register	0	R5
Workspace Register Indirect	1	*R7
Workspace Register Indirect Autoincrement	3	*R7 +
Symbolic Memory <sup>2,3</sup>	2	@LABEL
Indexed Memory <sup>2,4</sup>	2	@LABEL(5)

**Notes:**

- <sup>1</sup> The T field is described in the addressing format descriptions as T, and T<sub>a</sub>.
- <sup>2</sup> The instruction requires an additional word for each T field value of two. This word contains a memory address.
- <sup>3</sup> The S or D field is set to zero by the assembler.
- <sup>4</sup> Workspace register zero cannot be used for indexing.

**3.2.3 Workspace Register Indirect Autoincrement Addressing**

Workspace register indirect autoincrement addressing specifies a workspace register that contains the address of the operand. After the address is obtained from the workspace register, the workspace register is incremented by one for a byte instruction or by two for a word instruction. Increment values for other data types are defined in the instruction that uses those data types. A workspace register autoincrement address is written as an expression having an absolute value of 0 through 15 preceded by an asterisk and followed by a plus sign (+). The following are coding examples of instructions that have workspace register indirect autoincrement addresses:

S \*R3+,R2

Fetch the word at the address contained in register 3, increment register 3 by two, subtract the fetched word from the contents of register 2, and place the result in register 2.

C R5,\*R6+

Fetch the word at the address contained in register 6, increment register 6 by two, and compare the fetched word to the contents of register 5.

#### NOTE

The autoincrement is performed after the operand is fetched and before the operation takes place.

### 3.2.4 Symbolic Memory Addressing

Symbolic memory addressing specifies the memory address that contains the operand. A symbolic memory address is written as an expression preceded by an 'at' sign (@). The following are coding examples of instructions which have symbolic memory addresses:

S @TABLE1,@LIST4

Subtract the contents of the word at location TABLE1 from the contents of the word at location LIST4, and place the remainder in the word at location LIST4.

C R0,@STORE

Compare the contents of workspace register zero with the contents of the word at location STORE.

MOV @>C,@>7C

Copy the word at address 000C<sub>16</sub> into location 007C<sub>16</sub>.

#### NOTE

Symbols previously defined as having relocatable values or values greater than 15 need not have @. The at (@) sign is used to distinguish a register from an undefined memory storage label.

### 3.2.5 Indexed Memory Addressing

Indexed memory addressing specifies the memory address that contains the operand. The address is computed by adding the contents of the specified index register to an expression. An indexed memory address is written as an expression preceded by an at sign (@) and followed by a symbol specifying a workspace register enclosed in parentheses. The workspace register specified by the symbol within the parentheses is the index register. Workspace register 0 may not be specified as an index register. The following are coding examples of instructions that have indexed memory addresses:

A @2(R7),R6

Add the contents of workspace register six to the contents of the word at the address computed by adding 0002<sub>16</sub> and the contents of workspace register seven. Store the sum in workspace register six.

MOV R7,@LIST4-6(R5)	Copy the contents of workspace register seven into a word of memory. The address of the word of memory is the sum of the contents of workspace register five and the value of symbol LIST4 minus six.
---------------------	---

### 3.2.6 Program Counter Relative Addressing

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

JMP THERE	Jumps unconditionally to location THERE
-----------	---

### 3.2.7 CRU Bit Addressing

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

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

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

### 3.2.8 Immediate Addressing

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

LIMI 5	Places five in the interrupt mask, enabling interrupt levels 0 through 5
--------	--

LI R5,>1000	Places $1000_{16}$ into workspace register 5
-------------	--

#### NOTE

An @ sign may precede an immediate operand, but has no effect.



### 3.3 ADDRESSING SUMMARY

Table 3-2 shows the addressing required for each instruction. The first column lists the instruction mnemonics. The other columns specify the required address as follows:

G	—	General address: Workspace register address Indirect workspace register address Indirect workspace register autoincrement address Symbolic memory address Indexed memory address
WR	—	Workspace register address
PC	—	Program counter relative address
CRU	—	CRU bit address
I	—	Immediate value
N	—	Nibble
*	—	The address into which the result is placed when two operands are required.
CNT	—	Count
CKPT	—	Checkpoint
POS	—	Position
WID	—	Width
COND	—	Condition

The Processor set explains the group of processors (990/10, S300, and so on) for which the instruction is valid. See paragraph 3.5.3 for further details.

### 3.4 INSTRUCTION FORMATS

The required addressing previously described relates to the 21 instruction formats of the Model 990 Computer. These formats are described in the following paragraphs.

Table 3-2. Instruction Addressing

Mnemonic	Processor Set	First Operand	Second Operand	Third Operand	Fourth Operand
A	A	G	G*	—	—
AB	A	G	G*	—	—
ABS	A	G	—	—	—
AD	C	G	—	—	—
AI	A	WR*	I	—	—
AM	C	G	G*	CNT	—
ANDI	A	WR*	I	—	—
ANDM	C	G	G*	CNT	—
AR	C	G	—	—	—
ARJ	C	PC	CNT	WR	—
B	A	G	—	—	—
BDC	C	G	G*	CNT	—
BIND	B	G	—	—	—
BL	A	G	—	—	—
BLSK	C	WR*	I	—	—
BLWP	A	G	—	—	—
C	A	G	G	—	—
CB	A	G	G	—	—
CDE	C	—	—	—	—
CDI	C	—	—	—	—
CED	C	—	—	—	—
CER	C	—	—	—	—
CI	A	WR	I	—	—
CID	C	G	—	—	—
CIR	C	G	—	—	—
CKOF	A	—	—	—	—
CKON	A	—	—	—	—
CLR	A	G	—	—	—
CNTO	C	G	G*	CNT	—
COC	A	G	WR	—	—
CRC	C	G	G*	CNT	CKPT
CRE	C	—	—	—	—
CRI	C	—	—	—	—
CS	C	G	G	CNT	CKPT
CZC	A	G	WR	—	—
DBC	C	G	G*	CNT	—
DD	C	G	—	—	—
DEC	A	G	—	—	—
DECT	A	G	—	—	—
DINT	C	—	—	—	—
DIV	A	G	WR*	—	—
DIVS	B	G	—	—	—
DR	C	G	—	—	—
EINT	C	—	—	—	—
EMD	C	—	—	—	—
EP	C	G	G*	CNT	—
IDLE	A	—	—	—	—
INC	A	G	—	—	—
INCT	A	G	—	—	—
INSF	C	G	G*	POS	WID

Table 3-2. Instruction Addressing (Continued)

Mnemonic	Processor Set	First Operand	Second Operand	Third Operand	Fourth Operand
INV	A	G	—	—	—
IOF	C	G	—	—	—
JEQ	A	PC	—	—	—
JGT	A	PC	—	—	—
JH	A	PC	—	—	—
JHE	A	PC	—	—	—
JL	A	PC	—	—	—
JLE	A	PC	—	—	—
JLT	A	PC	—	—	—
JMP	A	PC	—	—	—
JNC	A	PC	—	—	—
JNE	A	PC	—	—	—
JNO	A	PC	—	—	—
JOC	A	PC	—	—	—
JOP	A	PC	—	—	—
LCS	C	WR	—	—	—
LD	C	G	—	—	—
LDCR	A	G	NOTE 1	—	—
LDD	A	G	—	—	—
LDS	A	G	—	—	—
LI	A	WR*	I	—	—
LIM	C	WR	—	—	—
LIMI	A	I	—	—	—
LMF	A	WR*	NOTE 2	—	—
LR	C	G	—	—	—
LREX	A	—	—	—	—
LST	B	WR	—	—	—
LTO	C	G	G*	CNT	—
LWP	B	WR	—	—	—
LWPI	A	I	—	—	—
MD	C	G	—	—	—
MOV	A	G	G*	—	—
MOVA	C	G	G*	—	—
MOVB	A	G	G*	—	—
MOVS	C	G	G*	CNT	CKPT
MPY	A	G	WR*	—	—
MPYS	B	G	—	—	—
MR	C	G	—	—	—
MVSK	C	G	G*	CNT	CKPT
MVSR	C	G	G*	CNT	CKPT
NEG	A	G	—	—	—
NEGD	C	—	—	—	—
NEGR	C	—	—	—	—
NRM	C	G	G*	CNT	—

**Notes:**

1. The second operand is the number of bits to be transferred, 1 through 15, 0 = 16.
2. The second operand specifies a memory map file, zero or one.

Table 3-2. Instruction Addressing (Continued)

Mnemonic	Processor Set	First Operand	Second Operand	Third Operand	Fourth Operand
ORI	A	WR*	I	—	—
ORM	C	G	G*	CNT	—
POPS	C	G	G*	CNT	CKPT
PSHS	C	G	G*	CNT	CKPT
RSET	A	—	—	—	—
RTO	C	G	G*	CNT	—
RTWP	A	—	—	—	—
S	A	G	G*	—	—
SB	A	G	G*	—	—
SBO	A	CRU	G*	—	—
SBZ	A	CRU	—	—	—
SD	C	G	—	—	—
SEQB	C	G	G	CNT	CKPT
SETO	A	G	—	—	—
SLA	A	WR*	NOTE 3	—	—
SLAM	C	G	G*	CNT	—
SLSL	C	COND	G	G*	—
SLSP	C	COND	G	G*	—
SM	C	G	G*	CNT	—
SNEB	C	G	G*	CNT	CKPT
SOC	A	G	G*	—	—
SOCB	A	G	G*	—	—
SR	C	G	—	—	—
SRA	A	WR*	NOTE 3	—	—
SRAM	C	G	G*	CNT	—
SRC	A	WR*	NOTE 3	—	—
SRJ	C	PC	CNT	WR*	—
SRL	A	WR*	NOTE 3	—	—
STCR	A	G*	NOTE 1	—	—
STD	C	G	—	—	—
STPC	C	WR*	—	—	—
STR	C	G	—	—	—
STST	A	WR	—	—	—
STWP	A	WR	—	—	—
SWPB	A	G	—	—	—
SWPM	C	G	G*	CNT	—
SZC	A	G	G*	—	—
SZCB	A	G	G*	—	—

**Notes:**

1. The second operand is the number of bits to be transferred, 1 through 15, 0 = 16.
3. The second operand is the shift count, 0 through 15. Zero means count is in bits 12 through 15 of workspace register zero. When count = zero and bits 12 through 15 of workspace register zero equal zero, count is 16.

**Table 3-2. Instruction Addressing (Continued)**

Mnemonic	Processor Set	First Operand	Second Operand	Third Operand	Fourth Operand
TB	A	CRU	—	—	—
TCMB	C	G	POS	—	—
TMB	C	G	POS	—	—
TS	C	G	G*	CNT	CKPT
TSMB	C	G	POS	—	—
X	A	G	—	—	—
XF	C	G	G*	POS	WID
XIT	C	—	—	—	—
XOP	A	G	NOTE 4	—	—
XOR	A	G	WR*	—	—
XORM	C	G	G*	CNT	—
XV	C	G	G*	POS	WID

**Notes:**

4. Second operand specifies the extended operation, 0 through 15. Disposition of result may or may not be in the first operand address, determined by the user.

**3.4.1 Format 1 — Two Address Instructions**

The operand field of Format 1 instructions contains two general addresses separated by a comma. The first address is the source address; the second is the destination address. The following mnemonic operation codes use Format 1.

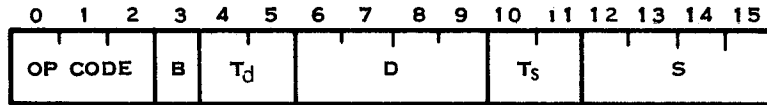
A	MOV	SOC
AB	MOVB	SOCB
C	S	SZC
CB	SB	SZCB

The following example shows a source statement for a Format 1 instruction:

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

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

The assembler assembles Format 1 instructions as follows:



The bit fields are as follows:

- Opcode — Three bits that define the machine operation
- B — Byte indicator, one for byte instructions, zero for word instructions
- T<sub>d</sub> — Address mode (Table 3-1) for destination
- D — Destination workspace register
- T<sub>s</sub> — Addressing mode (Table 3-1) for source
- S — Source workspace register

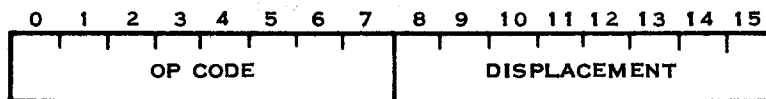
When T<sub>s</sub> or T<sub>d</sub> is equal to 2, the instruction occupies two words of memory, and the second word contains a memory address used with S or D, respectively, in developing the effective address. When both T<sub>s</sub> and T<sub>d</sub> are equal to 2, the instruction occupies three words of memory. The second word contains the memory address for the source operand, and the third word contains the memory address for the destination operand.

### 3.4.2 Format 2 — Jump Instructions

Format 2 instructions use program counter relative addresses which are coded as expressions that correspond to instruction locations on word boundaries. The following mnemonic operation codes are Format 2 jump instructions:

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

The following is an example of a source statement for a Format 2 jump instruction:



NOW JMP BEGIN

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

The assembler assembles Format 2 instruction as follows:

The bit fields are as follows:

- Opcode — Eight bits that define the machine operation
- Displacement — Signed displacement value.

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

### 3.4.3 Format 2 — Bit I/O Instructions

The operand field of Format 2 CRU bit I/O instructions contains a well-defined expression. It is a CRU bit address, relative to the contents of workspace register 12. The following mnemonic operation codes are Format 2 CRU bit I/O instructions:

SBO            SBZ            TB

The following example shows a source statement for a Format 2 CRU bit I/O instruction:

```
SBO 5                                Sets a CRU bit to one. If workspace register 12 contains 1016, CRU bit 13 is set by this instruction.
```

The format assembled for Format 2 instructions is shown and described in the preceding paragraph. For CRU bit instructions the signed displacement is shifted one bit position to the left and added to the contents of workspace register 12. In other words, it is a displacement in bits from the contents of bits 3 through 14 of workspace register 12.

### 3.4.4 Format 3 — Logical Instructions

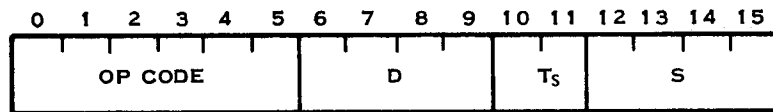
The operand field of Format 3 instructions contains a general address followed by a comma and a workspace register address. The general address is the source address, and the workspace register address is the destination address. The following mnemonic operation codes use **Format 3**:

COC            CZC            XOR

The following example shows a source statement for a Format 3 instruction:

```
COMP XOR @LABEL8(R3), R5        Perform an exclusive OR operation of the contents of a memory word and the contents of workspace register five, and place the result in workspace register five. The address of the memory word is the sum of the contents of workspace register three and the value of symbol LABEL8.
```

The assembler assembles Format 3 instructions as follows:



The bit fields are as follows:

- Opcode — Six bits that define the machine operation.
- D — Destination workspace register
- T<sub>s</sub> — Addressing mode (Table 3-1) for source
- S — Source workspace register

When T<sub>s</sub> is equal to 2, the instruction occupies two words of memory. The second word contains the memory address for the source operand.

### 3.4.5 Format 4 — CRU Instructions

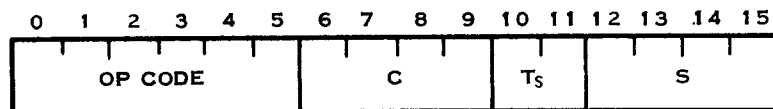
The operand field of Format 4 instructions contains a general address followed by a comma and a well-defined expression. The general address is the memory address from which bits will be transferred. The CRU address for the transfer is the contents of bits 3 through 14 of workspace register 12. The well-defined expression is the number of bits to be transferred, a value of 0 through 15 (a 0 value transfers 16 bits). For eight or fewer bits the effective address is a byte address. For nine or more bits the effective address is a word address. The following mnemonic operation codes use Format 4:

LDCR      STCR

The following example shows a source statement for a Format 4 instruction:

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

The assembler assembles Format 4 instructions as follows:







### 3.4.7 Format 6 — Single Address Instructions

The operand field of Format 6 instructions contains a general address. The following mnemonic operation codes use Format 6:

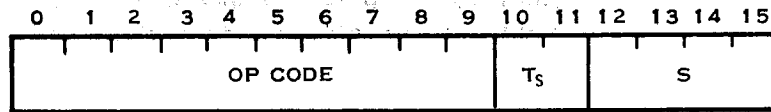
ABS	DEC	MD
AD	DECT	MPYS
AR	DIVS	MR
B	DR	NEG
BIND	INC	SETO
BL	INCT	SD
BLWP	INV	SR
CID	LD	STD
CIR	LDD	STR
CLR	LDS	SWPB
DD	LR	X

The following example shows a source statement for a Format 6 instruction.

CNT INC R7

Add one to the contents of workspace register seven, and place the sum in workspace register seven. CNT is the label of the location into which the instruction is placed.

The assembler assembles Format 6 instructions as follows:



The bit fields are as follows:

- Opcode — Ten bits that define the machine operation
- T<sub>s</sub> — Addressing mode for source
- S — Source workspace register

When T<sub>s</sub> is equal to 2, the instruction occupies two words of memory. The second word contains the memory address for the source operand.

### 3.4.8 Format 7 — Instructions Without Operands

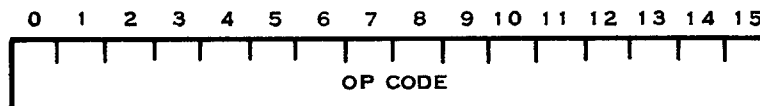
Format 7 instructions require no operand field. The following operation codes use Format 7:

CDE	CRE	LREX
CDI	CRI	NEGD
CED	DINT	NEGR
CER	EINT	RSET
CKOF	EMD	RTWP
CKON	IDLE	XIT

The following example shows a source statement for a Format 7 instruction:

RTWP	Returns control to the calling program and restores the context of the calling program by placing the contents of workspace registers 13, 14, and 15 into the workspace pointer, the program counter, and the status register, respectively.
------	--

The assemblers assemble Format 7 instructions as follows:



The opcode field contains 16 bits that define the machine operation.

### 3.4.9. Format 8 — Immediate Instructions

The operand field of Format 8 instructions contains a workspace register address followed by a comma and an expression. The workspace register is the destination address, and the expression is the immediate operand. The following mnemonic operation codes use Format 8:

AI	BLSK	LI
ANDI	CI	ORI

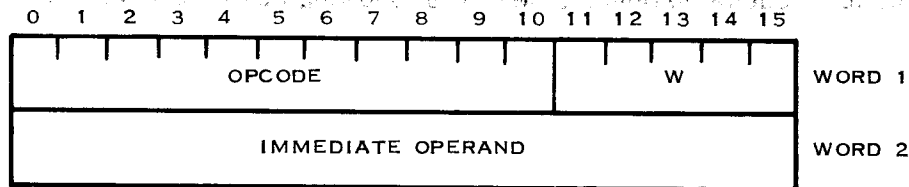
There are two additional Format 8 instructions that require only an expression in the operand field. The expression is the immediate operand. The destination is implied in the name of the instruction. The following mnemonic operation codes use this modified Format 8:

LIMI	LWPI
------	------

The following are examples of source statements for Format 8 instructions:

ANDI R4,>000F	Perform an AND operation on the contents of workspace register four and immediate operand 000F <sub>16</sub> . Place the result in workspace register four.
LWPI WRK1	Place the address defined for the symbol WRK1 into the WP register.

The assembler assembles Format 8 instructions as follows:



The bit fields are as follows:

- Opcode — Twelve bits that define the machine operation
- W — Workspace register operand
- Immediate operand — Sixteen bits which are the immediate operand

The instructions that have no workspace register operand place zeros in the W field. The instructions that have immediate operands place the operands in the word following the word that contains the opcode, that is, these instructions occupy two words each.

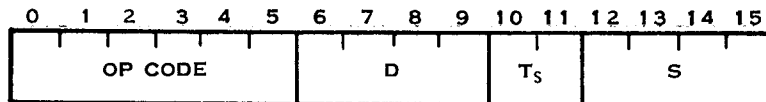
### 3.4.10 Format 9 — Extended Operation Instruction

The operand field of a Format 9, extended operation, instruction contains a general address and a well-defined expression. The general address is the address of the operand for the extended operation. The well-defined expression specifies the extended operation to be performed and must be in the range of 0 to 15. The mnemonic operation code is XOP.

The following example shows a source statement for a Format 9 extended operation instruction:

```
XOP @LABEL(R4),12      Perform the extended operation 12 using the address
                        computed by adding the value of symbol
                        LABEL to the contents of workspace register four.
```

The assembler assembles Format 9 instructions as follows:



The bit fields are as follows:

- **Opcode** — Six bits that define the machine operation
- **D** — Four bits that define the extended operation
- **T<sub>s</sub>** — Addressing mode (Table 3-1) for source
- **S** — Source workspace register

When T<sub>s</sub> is equal to 2, the instruction occupies two words of memory. The second word contains the memory address for the source operand.

### 3.4.11 Format 9 — Multiply and Divide Instructions

The operand field of Format 9 multiply and divide instructions contains a general address followed by a comma and a workspace register address. The general address is the address of the multiplier or divisor, and the workspace register address is the address of the workspace register that contains the multiplicand or dividend. The workspace register address is also the address of the first of two workspace registers to contain the result. The mnemonic operation codes are MPY and DIV.

The following example shows a source statement for a Format 9 Multiply instruction:

**MPY @ACC,R9**

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

The format assembled for the Format 9 multiply and divide instructions is shown and described in the preceding paragraph.

Field	Bits	Value
Opcode	15-10	000000
D	9-6	0000
T <sub>s</sub>	5	0000
S	4	0000

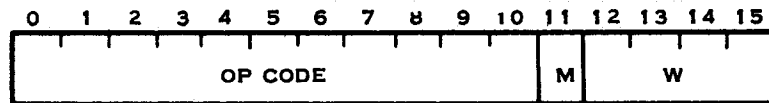
### 3.4.12 Format 10 — Memory Map File Instruction

The operand field of a Format 10, memory map file, instruction contains a workspace register address followed by a comma and a well-defined expression which evaluates to either zero or a one. The workspace register address specifies a workspace register that contains the address of a six-word area of memory that contains the map file data. The well-defined expression specifies the map file into which the data is to be loaded. The mnemonic operation code is LMF.

The following example shows a source statement for a Format 10 memory map file instruction:

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

The assembler assembles a Format 10 instruction as follows:



The bit fields are as follows:

- Opcode — Eleven bits that define the machine operation
- M — A single bit that specifies a memory map file, zero or one
- W — Workspace register operand

### 3.4.13 Format 11 — Multiple Precision Instructions

The operand field of the Format 11 instructions contains two general addresses and a nibble operand separated by commas. The first operand is the source address; the second is the destination address. The nibble operand is the number of bytes addressed by the operands. All Format 11 instructions are applicable to the 990/12 only.

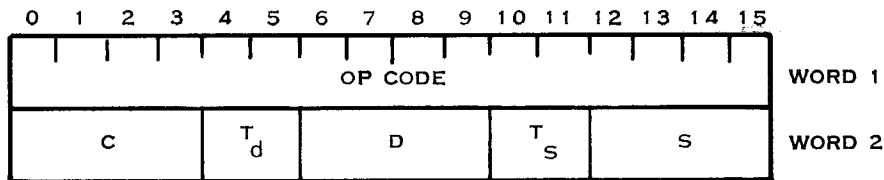
The following mnemonic operation codes use Format 11:

AM	DBC	RTO
ANDM	LTO	SM
BDC	NRM	SWPM
CNTO	ORM	XORM

The following example shows a source statement for a Format 11 instruction:

```
AM *R1, @RCN(R2),3  Add the contents of the three-byte operand at the ad-
                    dress in workspace register one to the contents of
                    the three-byte operand whose address is the sum of
                    the contents of workspace register two and the
                    value of symbol RCN, and place the result in the
                    three bytes beginning at the address specified by
                    the second general address.
```

The assembler assembles Format 11 instructions as follows:



The bit fields are as follows:

- Opcode — Sixteen bits that define the machine operation
- C — Four bits that contain the byte count
- $T_d$  — Addressing mode for destination
- D — Destination workspace register
- $T_s$  — Addressing mode for source
- S — Source workspace register

The nibble operand is optional. If it is missing, the assembler will supply a zero in the C field.

If  $T_s$  or  $T_d$  is equal to 2, the instruction occupies three words of memory. When both  $T_s$  and  $T_d$  are equal to 2, the instruction occupies four words of memory.

#### 3.4.14 Format 12 — String Instructions

The operand field of the Format 12 instructions contains two general addresses, a nibble operand and a checkpoint register separated by commas. The first operand is the source address; the second is the destination address. The nibble operand is the number of bytes addressed by the operands. When the byte count is not specified, a zero is provided by the assembler. This indicates that the byte count is in R0, bits 0 through 15. The checkpoint register is a multipurpose register whose specific purpose depends on the instruction being executed. When the checkpoint register is not specified, the implied checkpoint register is used. This is specified by the CKPT directive. All Format 12 instructions are applicable to the 990/12 only.

#### NOTE

The assembler will not automatically set the CKPT register.

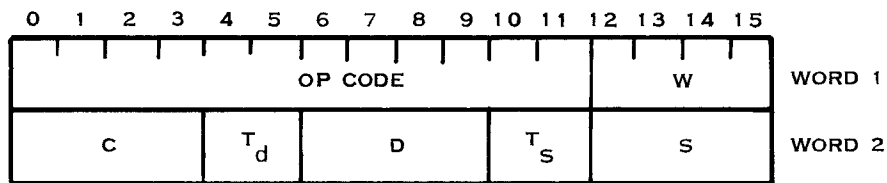
The following mnemonic operation codes are Format 12 instructions:

CRC	POPS
CS	PSHS
MOVS	SEQB
MVSK	SNEB
MVSR	TS

The following is an example of a source statement using a Format 12 instruction.

<p>CS @INPUT, @CORE</p>	<p>Compare the string starting at location INPUT for the length specified in workspace register zero to the string starting at location CORE for the length specified in workspace register zero using the implied checkpoint directive.</p>
-------------------------	--

The assembler assembles Format 12 instructions as follows:



The bit fields are as follows:

- Opcode — Twelve bits that define the machine operation
- W — Defines the checkpoint register
- C — Four bits that contain the byte count
- $T_d$  — Addressing mode for destination
- D — Destination workspace register
- $T_s$  — Addressing mode for source
- S — Source workspace register

When  $T_s$  or  $T_d$  is equal to 2, the instruction occupies three words of memory. When  $T_s$  and  $T_d$  are both equal to 2, the instruction occupies four words of memory.



**3.4.15 Format 13 — Multiple Precision Shift Instructions**

The operand field of Format 13 instructions contains a general address and two nibble operands. The general address is the source address. The second operand contains the byte count for the first operand. The third operand is the number of bits to shift the operand. All Format 13 instructions are applicable to the 990/12 only.

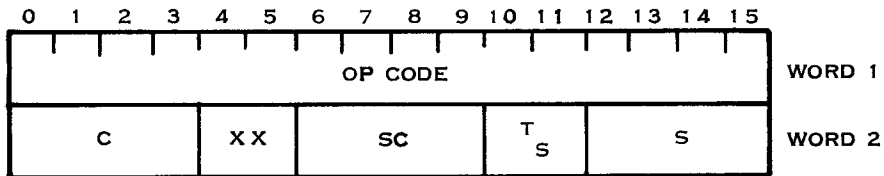
The following mnemonic operation codes are Format 13 instructions:

SLAM  
SRAM

The following examples are source statements using Format 13 instructions.

SLAM @BIT,6,8                      Shift the six-byte string BIT eight bits to the left.  
SRAM @BIT,6,8                      Shift the six-byte string BIT eight bits to the right.

The assembler assembles Format 13 instruction as follows:



The bit fields are as follows:

- Opcode — Sixteen bits that define the machine operation
- C — Four bits that define the length of the field
- SC — Four bits that define the length the field is to be shifted
- T<sub>s</sub> — Addressing mode for source
- S — Source workspace register

When the T<sub>s</sub> field is equal to 2, the instruction occupies three words of memory.

When the C field is zero, the four LSBs of workspace register zero are used. When determining a byte count, if the four LSBs of the workspace register are zero, then the byte count is 16. When the SC field is zero, bits 4 through 7 of workspace register zero are used. If bits 4 through 7 are zero, the shift is zero.

### 3.4.16 Format 14 — Bit Testing Instructions

The operand field of Format 14 instructions contains a general address and an expression defining the position of the bit separated by a comma. If the position is not defined, the default value  $3FF_{16}$  is used. The following mnemonic operation codes are Format 14 instructions and execute on the 990/12 only:

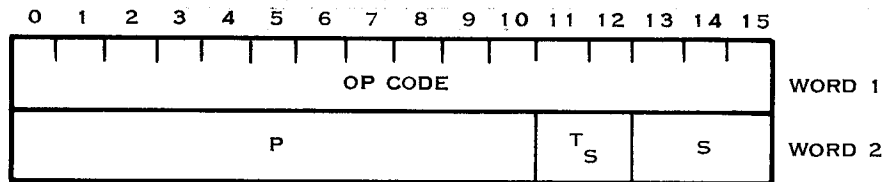
**TCMB**  
**TMB**  
**TSMB**

The following is an example of a source statement using the Format 14 instruction:

**TSMB @BITMAP(R3),6**      Test and set bit 6 at the location specified by adding the value of BITMAP to the contents of workspace register three.

Bit positions are defined such that the most significant bit is position zero.

The assembler assembles Format 14 instructions as follows:



The bit fields are as follows:

- **Opcode** — Sixteen bits that define the machine operation
- **P** — Ten bits that define the position of the bit to be tested
- **T<sub>s</sub>** — Addressing mode for source
- **S** — Source workspace register

When the T<sub>s</sub> field is equal to 2, the instruction occupies three words of memory.

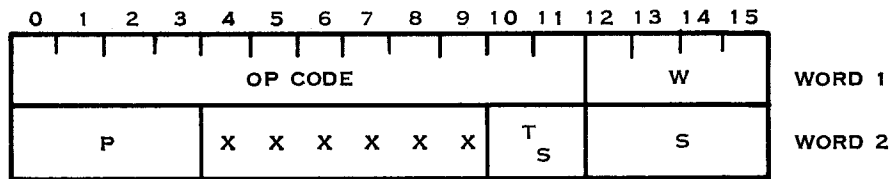
### 3.4.17 Format 15 — Invert Order of Field Instruction

The operand field of the Format 15 instruction contains a general address and the position and width of the source in parentheses separated by commas. The IOF mnemonic operation code is the only Format 15 instruction. This instruction executes on the 990/12 only.

The following is an example of a source statement using the Format 15 instruction:

IOF @WORD,(1,8)                      Invert (reverse) the order of the bits in the word at location WORD beginning at bit position 1 for a bit-field width of eight bits.

The assembler assembles the Format 15 instruction as follows:



The bit fields are as follows

- Opcode — Twelve bits that define the machine operation
- W — Four bits that define the width of the source
- P — Four bits that define the position of the source
- XXXXXX — These bits are not used
- T, — Addressing mode of the source
- S — Source workspace register

When T, is equal to 2, the instruction occupies three words of memory.

### 3.4.18 Format 16 — Field Instructions

The operand field of Format 16 instructions contains two general addresses and two nibble operands. The first address defines the source; the second defines the destination. The first nibble defines the position of the field; the second nibble defines the width of the field.

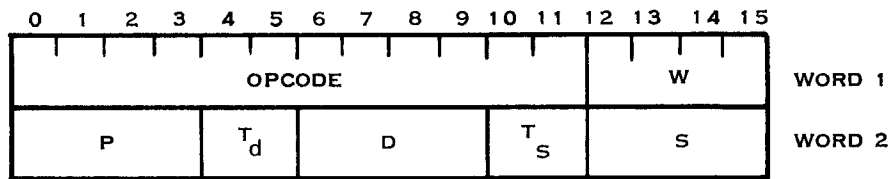
The following mnemonic operation codes are Format 16 instructions and execute on the 990/12 only:

INSF  
XF  
XV

The following is an example of a source statement using the Format 16 instruction:

XF @CORE, @OUTPUT, (3,6)      Place the six-bit value at location CORE, starting at bit position 3, into the word at location OUTPUT.

The assembler assembles Format 16 instructions as follows:



The bit fields are as follows:

- Opcode — Twelve bits that define the machine operation
- W — Four bits that define the width of a field
- P — Four bits that define the position of a field
- $T_d$  — Addressing mode for the destination
- D — Destination workspace register
- $T_s$  — Addressing mode for the source
- S — Source workspace register

If  $T_s$  or  $T_d$  is equal to 2, the instruction takes up three words of memory. If  $T_s$  and  $T_d$  are equal to 2, the instruction takes up four words of memory.

**3.4.19 Format 17 — Alter Registers and Jump Instructions**

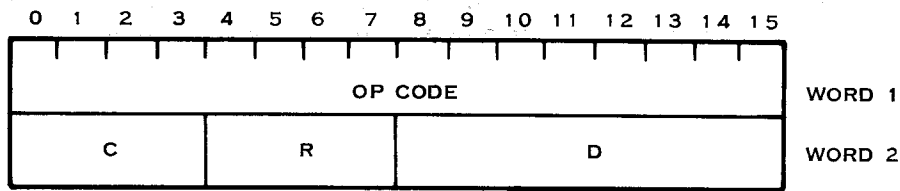
The operand field of Format 17 instructions contains an expression, a nibble operand, and a workspace register. If the nibble operand is not present, a default of one is assumed. The following mnemonic operation codes are Format 17 instructions and execute on the 990/12 only:

ARJ  
SRJ

The following are examples of source statements using Format 17 instructions:

ARJ @BEGIN, , R3	Increment workspace register three and jump to the contents of the word at location BEGIN.
ARJ @BEGIN, 12, R3	Add 12 to workspace register three and jump to the contents of the word at location BEGIN.

The assembler assembles Format 17 instructions as follows:



The bit fields are as follows:

- Opcode — Sixteen bits that define the machine operation
- C — Four bits that define the constant to be added or subtracted
- R — Four bits that define the workspace register to be altered
- D — Eight bit signed displacement

**3.4.20. Format 18 — Single Register Operand Instructions**

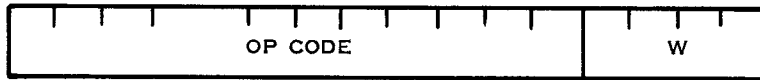
The operand field of Format 18 instructions contains a workspace register address. The following mnemonic operation codes are Format 18 instructions:

LCS	STPC
LIM	STST
LST	STWP
LWP	

The following is an example of a source statement using the Format 18 instruction:

LCS R4                                    Load the writable control store using the control block addressed by workspace register four.

The assembler assembles Format 18 instructions as follows:



The bit fields are as follows:

- Opcode — Twelve bits that define the machine operation
- W — Four bits that define a workspace register address

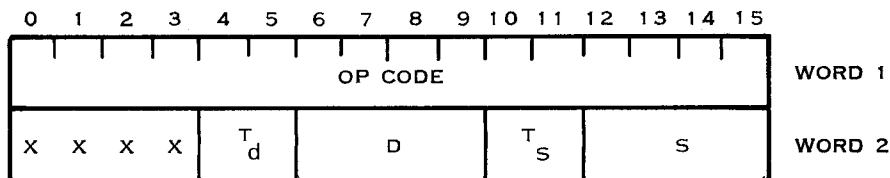
**3.4.21 Format 19 — Move Address Instruction**

The operand field of the Format 19 instruction contains two general addresses separated by a comma. The first address defines the source address; the second is the destination address. The mnemonic operation code, MOVA, is the Format 19 instruction and executes on the 990/12 only.

The following is an example of a source statement using the Format 19 instruction:

MOVA @CHAR(R6), \*R7            Move the sum of the value of CHAR and the contents of workspace register six to the address in workspace register seven.

The assembler assembles the Format 19 instruction as follows:



The bit fields are as follows:

- Opcode — Sixteen bits that define the machine operation
- XXXX — These bits are not used
- T<sub>d</sub> — Addressing mode for the destination
- D — Destination workspace register
- T<sub>s</sub> — Addressing mode for the source
- S — Source workspace register

If  $T_s$  or  $T_d$  is equal to 2, the instruction occupies three words. If both  $T_s$  and  $T_d$  are equal to 2, the instruction occupies four words.

### 3.4.22 Format 20 — List Search Instructions

The operand field of Format 20 instructions contains a condition and two general addresses.

The condition statement is one of the following:

EQ	equal	0
NE	not equal	1
HE	high or equal	2
L	low	3
GE	greater than or equal	4
LT	less than	5
LE	low or equal	6
H	high	7
LTE	less than or equal	8
GT	greater than	9

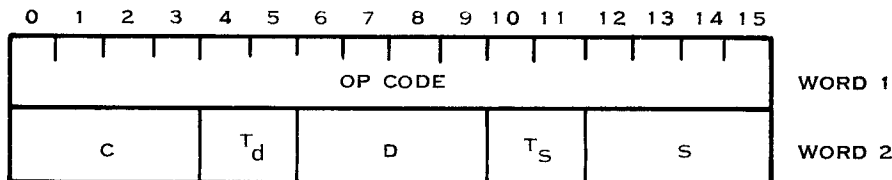
The first general address is the source address; the second is the destination address. The following mnemonic operation codes are format 20 instructions and execute on the 990/12 only.

SLSL  
SLSP

The following is an example of a source statement using a Format 20 instruction:

SLSL EQ,@CONTROL,@MATCH      Search the list addressed by the contents of the word at location MATCH until an equal condition is met using the contents of the five-word control block addressed by the contents of the word at location CONTROL.

The assembler assembles the Format 20 instruction as follows:



The bit fields are as follows:

- Opcode — Sixteen bits that define the machine operation
- C — Four bits that define the condition searched for
- $T_d$  — Addressing mode for the destination

- D — Destination workspace register
- $T_s$  — Addressing mode for the source
- S — Source workspace register

If  $T_s$  or  $T_d$  is equal to 2, then the instruction is three words long. If both  $T_s$  and  $T_d$  are equal to 2, then the instruction is four words long.

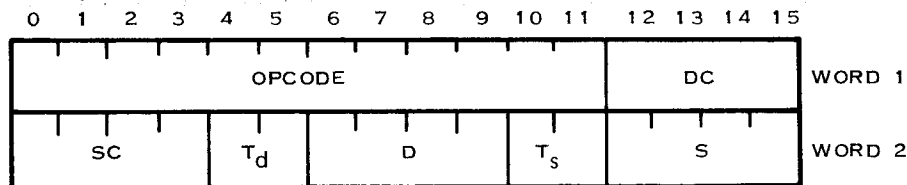
### 3.4.23 Format 21 — Extend Precision Instruction

The operand field of the Format 21 instruction contains two general addresses and two nibble operands separated by commas. The first general address is the source address; the second general address is the destination address. The first nibble operand specifies the byte count (length) of the source address; the second nibble operand specifies the byte count (length) of the destination address. The mnemonic operation code, EP, is the Format 21 instruction. EP executes only on the 990/12.

The following is an example of a source statement using the Format 21 instruction:

```
LABEL EP @NUMBER, @NEWNUM, 6,10 Extend the six-byte long contents of the word at
                                location NUMBER to 10 bytes and place it in
                                NEWNUM.
```

The assembler assembles the Format 21 instruction as follows:



The bit fields are as follows:

- Opcode — Twelve bits that define the machine operation
- DC — Four bits that define the length of the destination field
- SC — Four bits that define the length of the source field
- $T_d$  — Addressing mode for destination
- D — Destination workspace register
- $T_s$  — Addressing mode for source
- S — Source workspace register

If  $T_s$  or  $T_d$  is equal to 2, the instruction is three words long. If both  $T_s$  and  $T_d$  are equal to 2, the instruction is four words long.



### 3.5 INSTRUCTION DESCRIPTIONS

The instruction descriptions include the following information:

- The instruction's opcode
- The instruction's assembled format
- The syntax definition
- An example
- A definition of the instruction's operation
- The status bits affected
- The execution results
- The application notes

#### 3.5.1 Opcode

The opcode is the four-digit hexadecimal number that defines the instruction to be executed.

#### 3.5.2 Instruction Format

The instruction format lists the format (1 through 21) in which the instruction is assembled. A block diagram gives the machine language format of the instruction after it is assembled. An X in the block diagram indicates an unused bit. The assembler generates zeros for bits shown as X.

#### 3.5.3 Applicability

The applicability defines the processor set for which the instruction is valid, and specifies whether the instruction is privileged or requires the memory mapping option. The sets are:

- Processor set A — 990/10, 990/10A, 990/12, and S300
- Processor set B — 990/10A, 990/12, and S300
- Processor set C — 990/12 only

#### 3.5.4 Syntax Definition

The syntax definition for each instruction is shown, using the following conventions:

- Items in capital letters and special characters must be entered as shown.
- Items within angle brackets (< >) are defined by the user.
- Items in lowercase letters are classes (generic names) of items. The generic names used are described in the next paragraph.

- Items within brackets ( [ ] ) are optional.
- Items within braces ( { } ) are alternative items; one must be entered.
- All ellipses ( . . . ) indicate that the preceding item may be repeated.
- The symbol **b** represents a blank or a space

The generic names used in these definitions are as follows:

- $ga_s$  — General address of source operand
- $ga_d$  — General address of destination operand
- $wa$  — Workspace register address
- $iop$  — Immediate operand
- $wa_d$  — Destination workspace register address
- $disp$  — Displacement of CRU lines from the CRU base register or signed word displacement from the current location counter
- $exp$  — Expression that represents an instruction location
- $cnt$  — Bit or byte count of another operand (specific type of nibble)
- $m$  — Memory map file
- $scnt$  — Shift count (nibble value)
- $op$  — Number (0 through 15) of extended operation (nibble value)
- $ckpt$  — Checkpoint register (specific type of  $wa$ )
- $pos$  — Bit position (nibble value)
- $wid$  — Bit width (nibble value)
- $cond$  — Matching condition to search for (nibble value)

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

**3.5.5 Instruction Example**

An executable example is given for each instruction. Across from the example is a brief description of the operation taking place in the example.

**3.5.6 Operation Definition**

The operation definition describes the function of each of the operands in the operand field.

**3.5.7 Status Bits Affected**

The status bits affected by the execution of the instruction are listed.

**3.5.8 Execution Results**

The execution results uses a relational expression to describe the execution results. The following conventions are used in the expression:

- ( ) — indicates “the contents of”.
- → — indicates “replaces”.
- :: — indicates the absolute value.

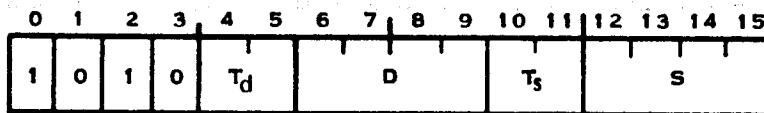
**3.5.9 Application Notes**

The application notes expand on the operation definition to give the user a more complete explanation of the use of the instruction from an application point of view.

**3.6 ADD WORDS — A**

*Opcode:* A000

*Instruction Format:* 1



*Applicability:* Processor set A

*Syntax definition:*

[<label>]b..Ab..<ga>,<ga>b..[<comment>]

## A, AB

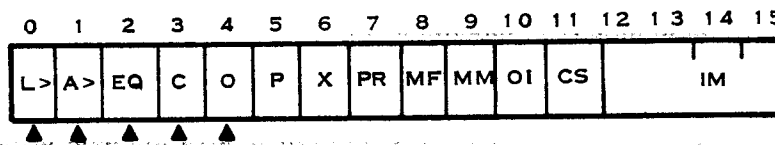
**Example:**

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

Add the contents of the word at the location ADDR1 plus workspace register two to the contents of the word at the location ADDR2 plus workspace register three and store the results in the location ADDR2 plus workspace register three.

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

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



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

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

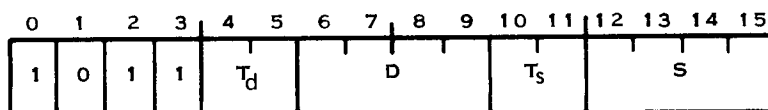
A R5, @TABLE

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

### 3.7 ADD BYTES — AB

**Opcode:** B000

**Instruction Format:** 1



**Applicability:** Processor set A

# AB

**Syntax definition:**

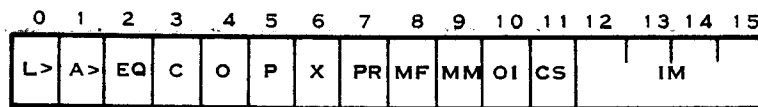
[<label>]b..ABb..<ga<sub>s</sub>>,<ga<sub>d</sub>>b..[<comment>]

**Example:**

LABEL AB R3, R2                      Add the contents of workspace register three (byte) to the contents of workspace register two (byte) and place the result in workspace register two. The most significant bytes are used as operands.

**Definition:** Add a copy of the source operand (byte) to the destination operand (byte), and replace the destination operand with the sum. When operands are addressed in the workspace register mode, only the leftmost bytes (bits 0 through 7) of the addressed workspace registers are used. The AU compares the sum to zero and sets/resets the status bits to indicate the results of the comparison. When there is a carry out of the most significant bit of the byte, the carry status bit sets. The odd parity bit sets when the bits in the sum (destination operand) establish odd parity and resets when the bits in the sum establish even parity.

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



**Execution results:** (ga<sub>s</sub>) + (ga<sub>d</sub>) →(ga<sub>d</sub>)

**Application notes:** AB is used to add signed integer bytes. For example, if the contents of workspace register three is 7400<sub>16</sub>, the contents of memory location 2122<sub>16</sub> is F318<sub>16</sub>, and the contents of workspace register two is 2123<sub>16</sub>, then the instruction

AB R3,\*R2

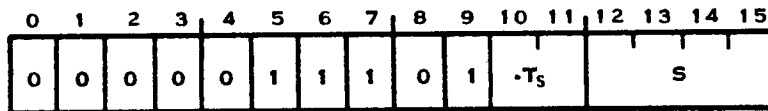
changes the contents of memory location 2122<sub>16</sub> to F38C<sub>16</sub> and the contents of workspace register two to 2124<sub>16</sub>, while the contents of workspace register three remain unchanged. The logical greater than, overflow, and odd parity status bits set, while the arithmetic greater than, equal, and carry status bits reset.

# ABS

## 3.8 ABSOLUTE VALUE — ABS

Opcode: 0740

Instruction Format: 6



Applicability: Processor set A

Syntax definition:

[<label>]b...ABSb...<ga,>b...[<comment>]

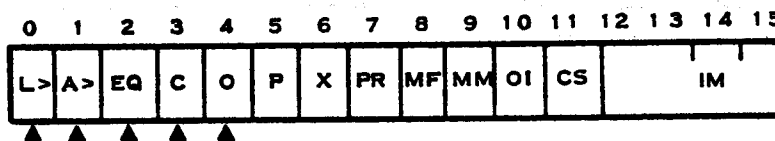
Example:

LABEL ABS \*R2

Replace the contents of the memory word addressed by workspace register two with its absolute value.

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

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



Execution results: :(ga.): → (ga.)

**ABS**

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

ABS @LIST (R7)

changes the contents of the fourth word in array LIST to  $00C4_{16}$ . The logical greater than status bit sets while the arithmetic greater than and equal status bits reset. The carry bit is unconditionally reset. The overflow bit is set if the operand is  $8000_{16}$ ; otherwise, it is reset. Refer to Section 4 for additional application notes.

*Multiple CPU Systems:* Several 990/12 CPUs can be connected together to create a multiple CPU system. In these systems, the CPUs must share a common memory. Simultaneous access attempts to memory by more than one CPU can result in a loss of data. To prevent this conflict, software "memory busy" flags in memory can be used. When a processor desires access to memory, it must first check the flag to determine if any other processor is actively using memory. If memory is not busy, the processor sets the busy flag to lock out other processors and begins its memory transfers. When the processor is finished with memory, it clears the busy flag to allow access from other processors.

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

**NOTE**

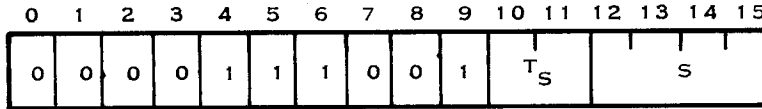
When workspace registers are cached, ABS in direct register addressing will not detect a flag changed in the corresponding memory location by another processor. Therefore, ABS can only be used with indirect, indirect autoincrement, indexed, and symbolic addressing modes when used to control common memory in multiple CPU systems.

# AD

## 3.9 ADD DOUBLE PRECISION REAL — AD

Opcode: 0E40

Instruction Format: 6



Applicability: Processor set C

Syntax definition:

[<label>]b..ADb<ga>b..[<comment>]

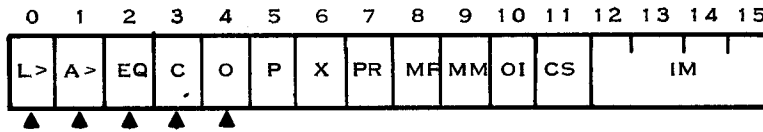
Example:

LABEL AD R6

Add the contents of workspace registers six through nine to the FPA (R0-R3)

Definition: The contents of the source address are added to the FPA (R0-R3).

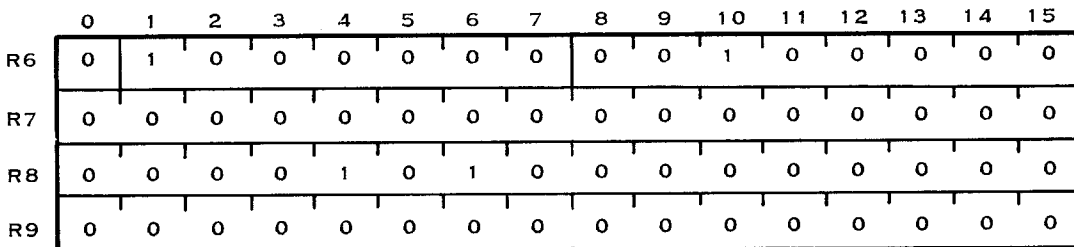
Status bits affected: Logical greater than, arithmetic greater than, equal, carry, overflow



Execution results: (ga) + FPA → FPA

Application notes: The results of the AD instruction are compared to zero and status register bits 0, 1, and 2 reflect the comparison. If status register bits 3 and 4 are set to zero and one, respectively, underflow has occurred. If both are set to one, overflow has occurred. If T<sub>S</sub> is equal to three, the indicated register is incremented by eight.

The following is an example of the add double precision real instruction. Workspace registers six, seven, eight, and nine contain, after normalization, the value .2000000A<sub>16</sub>, as shown figuratively:





## AD, AI

The double precision FPA (R0-R3) contains, after normalization, the value 0400770AB<sub>16</sub>, as shown figuratively:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R0	0	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0
R1	0	0	0	0	0	1	1	1	0	1	1	1	0	0	0	0
R2	1	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0
R3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The instruction

LABEL AD R6

Adds the contents R6-R9 to the FPA and places the result, .24007714B<sub>16</sub>, in the FPA, figuratively shown:

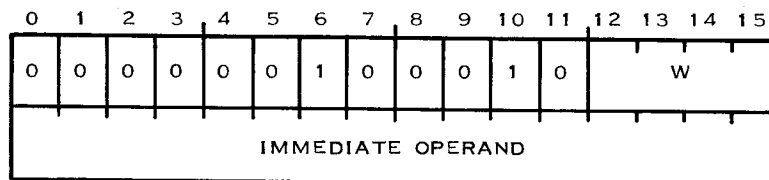
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0
R1	0	0	0	0	0	0	0	0	0	1	1	1	0	1	1	1
R2	0	0	0	1	0	1	0	0	1	0	1	1	0	0	0	0
R3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The logical greater than and arithmetic greater than bits of the status register are set; the equal, carry, and overflow bits of the status register are reset.

### 3.10 ADD IMMEDIATE — AI

Opcode: 0220

Instruction Format: 8



Applicability: Processor set A

Syntax definition:

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

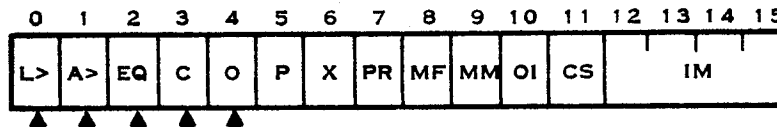
## AI, AM

**Example:**

LABEL AI R2, 7                      Add seven to the contents of workspace register two.

**Definition:** Add a copy of the immediate operand, the contents of the word following the instruction word in memory, to the contents of the workspace register specified in the W field and replace the contents of the workspace register with the results. The AU compares the sum to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry out of bit 0, the carry status bit sets. When there is an overflow (the result cannot be represented within a word as a two's complement value), the overflow status bit sets.

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



**Execution results:** (wa) + iop → (wà)

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

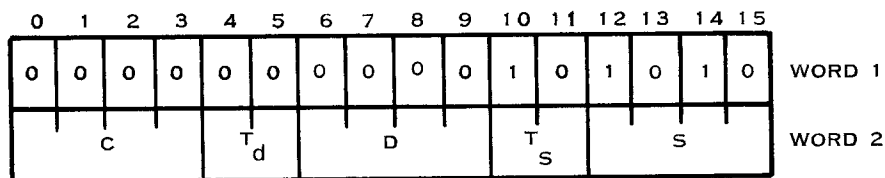
AI R6, >C

changes the contents of workspace register 6 to 000C<sub>16</sub>. The logical greater than and arithmetic greater than status bits set while the equal, carry, and overflow status bits reset.

### 3.11 ADD MULTIPLE PRECISION INTEGER — AM

**Opcode:** 002A

**Instruction Format:** 11



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b.. AMb..<gas>, <gad>[, <cnt>]b.. [<comment>]

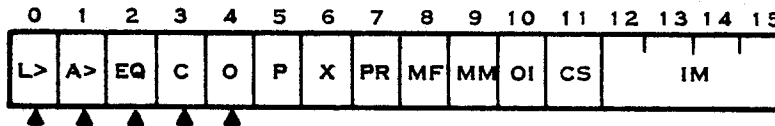
## AM, ANDI

### Example:

**LABEL AM \*R1, @RCN(R2),3** Add the three bytes beginning at the address in workspace register one to the three bytes beginning at the location RCN plus workspace register two. The result is placed in the three bytes beginning at the location RCN plus workspace register two.

**Definition:** The multibyte two's complement integer at the source address is added to the multibyte two's complement integer at the destination address. The sum is placed in the destination address. The <cnt> field is the number of bytes of precision of the integer. If <cnt> equals zero or is not present, the count is taken from the four LSBs of workspace register zero. If the four LSBs of workspace register zero are zero, the count is 16.

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



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

**Application notes:** If T<sub>s</sub> and/or T<sub>d</sub> is equal to three, the indicated register is incremented by the byte count.

The result of the AM instruction is compared to zero, and status register bits 0, 1, and 2 reflect the comparison. Status register bits 3 and 4 indicate the carry and overflow.

The following is an example of Add Multiple Precision. An eight-byte string at location RA contains the values 3124<sub>16</sub>, E008<sub>16</sub>, 6742<sub>16</sub> and 4013<sub>16</sub>, and an eight-byte string at location RCN contains the values 0010<sub>16</sub>, 4135<sub>16</sub>, 000F<sub>16</sub> and 0072<sub>16</sub>. The instruction

**LABEL AM @RA, @RCN, 4**

changes the contents of the first four bytes of RCN to 3135<sub>16</sub> and 213D<sub>16</sub>. The latter four bytes of RCN and the entire contents of RA are unchanged.

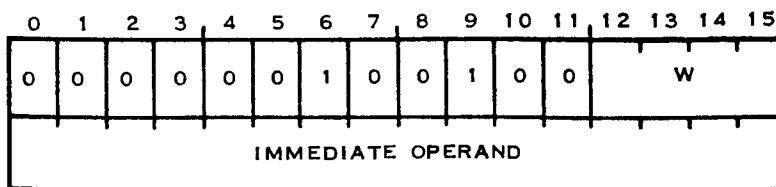
The logical greater than and arithmetic greater than bits of the status register are set; the equal, carry, and overflow bits of the status register are reset.

### 3.12 AND IMMEDIATE — ANDI

**Opcode:** 0240

# ANDI

Instruction Format: 8



Applicability: Processor set A

Syntax definition:

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

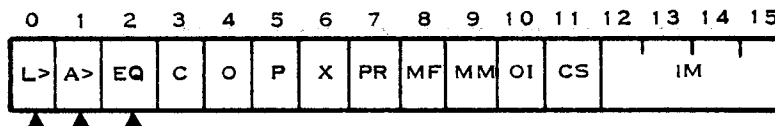
Example:

LABEL ANDI R3,>FFF0

Perform the logical AND of workspace register three and the immediate value FFF0<sub>16</sub>.

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

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



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

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

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

For an example, if workspace register zero contains D2AB<sub>16</sub>, the instruction

ANDI R0,>6D03

## ANDI, ANDM

Results in workspace register zero changing to 4003<sub>16</sub>. This AND operation on a bit-by-bit basis is as follows:

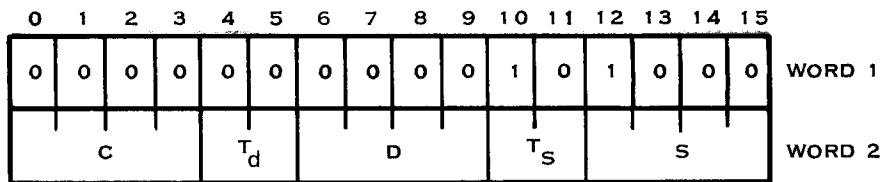
0110110100000011	(Immediate operand)
1101001010101011	(Workspace register 0)
0100000000000011	(Workspace register 0 result)

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

### 3.13 AND MULTIPLE PRECISION — ANDM

Opcode: 0028

Instruction Format: 11



Applicability: Processor set C

Syntax definition:

[<label>]b..ANDMb..<ga, ><ga<sub>d</sub>>[,<cnt>]b..[<comment>]

Example:

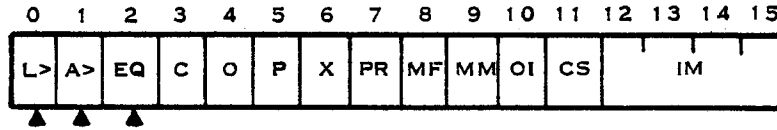
LABEL ANDM \*R1,@RCN(R2),3

Perform the logical AND between the three-byte value starting at the location pointed to by workspace register one and the three-byte value at the location starting at RCN plus workspace register two. The result is placed at the location RCN plus workspace register two.

**Definition:** Perform a bit-by-bit AND operation of the multibyte two's complement integer at the source address with the multibyte two's complement integer at the destination address. The result is placed in the destination address. The <cnt> field is the number of bytes of precision of the integer. If <cnt> equals zero, or is not present, the count is taken from the four LSBs of workspace register zero. If the four LSBs of workspace register zero are zero, the count is 16.

# ANDM

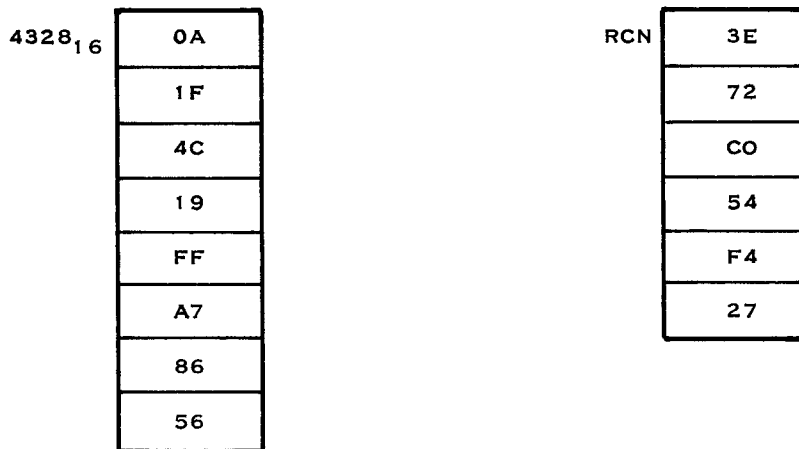
Status bits affected: Logical greater than, arithmetic greater than, and equal



Execution results:  $(ga_r) \text{ AND } (ga_d) \rightarrow (ga_d)$

Application notes: The result of the ANDM instruction is compared to zero, and the status register bits 0, 1, and 2 indicate the results of the comparison. If  $T_r$  and/or  $T_d$  is equal to three, the indicated register is incremented by the byte count.

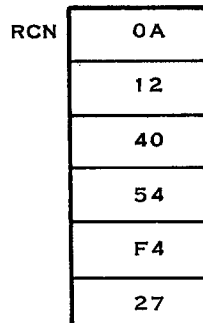
The following is an example of the AND Multiple Precision instruction. Workspace register one contains the address of an eight-byte string at location  $4328_{16}$ , and RCN is the address of a six-byte string:



The instruction

```
LABEL ANDM *R1,@RCN,3
```

ANDs the three bytes beginning at location  $4328_{16}$  with the three bytes beginning at location RCN, and places the results in the three bytes starting at location RCN. The results of this example are shown figuratively:



## ANDM, AR

The logical greater than and arithmetic greater than bits of the status register are set, and the equal bit is reset.

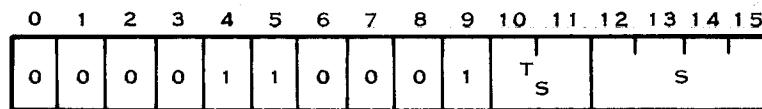
This truth table describes the AND operation between two bits:

Source Bit	Corresponding Destination Bit	Corresponding Result Bit
0	0	0
0	1	0
1	0	0
1	1	1

### 3.14 ADD REAL — AR

Opcode: 0C40

Instruction Format: 6



Applicability: Processor set C

Syntax definition:

[<label>]b..ARb..<ga,>b..[<comment>]

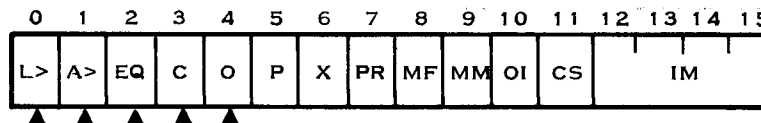
Example:

LABEL AR R5

Add workspace registers five and six to the FPA (R0, R1).

Definition: Add the real number specified by the source address to the FPA and store the result in the FPA (R0,R1).

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



Execution results: FPA + (ga,) → FPA

## AR, ARJ

*Application notes:* If T, is equal to three, the indicated register is incremented by four. The results of the AR instruction are compared to zero and status register bits 0, 1, and 2 reflect the comparison. If status register bits 3 and 4 are set to zero and one, respectively, underflow has occurred. If both are set to ones, overflow has occurred.

The following is an example of an Add Real instruction. Workspace register five and workspace register six, after normalization, contain the value  $.3_{16}$ , as shown figuratively:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R5	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0
R6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The FPA (R0,R1) contains the value  $.00A_{16}$ , after normalization, as shown figuratively:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R0	0	0	1	1	1	1	1	0	1	0	1	0	0	0	0	0
R1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The instruction

`LABEL AR R5`

adds the contents of R5 and R6 to the FPA and places the result,  $.30A_{16}$ , in the FPA, shown figuratively:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0
	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

The logical greater than and arithmetic greater than bits of the status register are set; and the equal, carry, and overflow status bits of the status register are reset.

Refer to Section 2 for a detailed description of normalization and single precision floating point instructions.

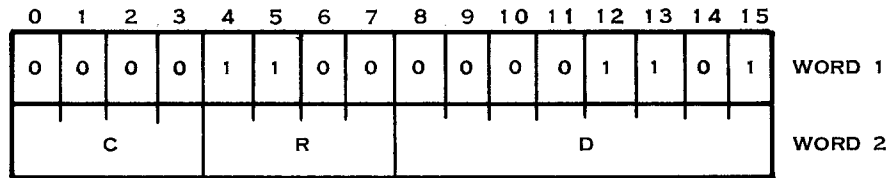
### 3.15 ADD TO REGISTER AND JUMP — ARJ

*Opcode:* 0C0D



**ARJ**

Instruction Format: 17

**Applicability:** Processor set C**Syntax definition:**

```
[<label>]b..ARJb..<exp>,<cnt>,<wa>b..[<comment>]
```

**Example:**

```
LABEL ARJ @BEGIN, 12,R3
```

Add 12 to the contents of workspace register three and jump to BEGIN if the sum of 12 and the contents of workspace register three is not equal to zero or does not pass through zero.

**Definition:** The unsigned positive integer in the <cnt> field is added to the register specified by <wa>. If the <cnt> field is zero, the value to be added is obtained from workspace register zero as a 16-bit unsigned value. If the <cnt> operand is not present, it defaults to one. If <cnt> plus (wa) is not equal to zero, or does not pass through zero (wrap-around), the signed word displacement, <exp>, is added to the program counter.

**Status bits affected:** None

**Execution results:** <cnt> + (wa) → (wa)  
Conditionally (PC) + <exp> → (PC)

If the value of the register is not equal to zero or has not passed through zero, then the value of the program counter plus the displacement is placed in the program counter.

**Application notes:** The ARJ instruction is not useful for writing loops where the ARJ instruction controls loop termination. It lends itself more to being the last instruction in a loop where the loop is exited from some other point. The example that follows shows a “while” loop. A table of text strings is indexed consecutively while the input value is not equal to the current table value.

```
TABLE      EVEN
           TEXT 'JAN'
           BYTE 31
ENTSIZ    EQU $-TABLE
           TEXT 'FEB'
           BYTE 28
           TEXT 'MAR'
           BYTE 31
           TEXT 'APR'
```

# ARJ, B

```

        BYTE 30
        .
        .
        .
TABEND   EQU $
INPUT    BSS 4
    
```

The ARJ instruction could be used to search this table for a specified month as follows:

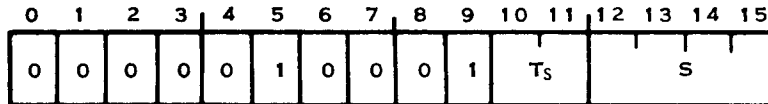
```

        SETO      R0
        LI        R1, TABLE-TABEND
WHILE
        CS        @INPUT, @TABEND(R1), 3, R0
        JEQ      ENDWHL
        ARJ      WHILE, ENTSIZ, R1
NOTFND   MONTH NOT FOUND
ENDWHL   CI      R1, TABEND
        JEQ      NOTFND (INPUT NOT FOUND IN TABLE)
*       MONTH FOUND
    
```

## 3.16 BRANCH — B

Opcode: 0440

Instruction Format: 6



Applicability: Processor set A

Syntax definition:

```
[<label>]b..Bb..<ga, >b..[<comment>]
```

## B, BDC

**Example:**

LABEL B @THERE                      Transfer control to the instruction at location THERE.

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

**Status bits affected:** None

**Execution results:**  $ga_r \rightarrow (PC)$

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

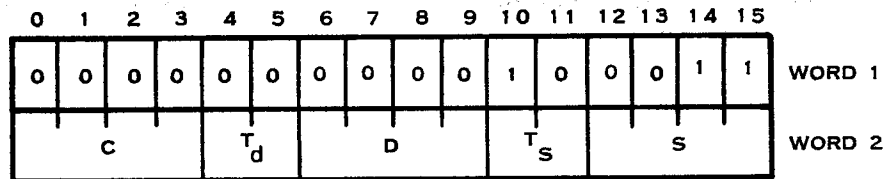
**B \*R3**

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

### 3.17 BINARY TO DECIMAL ASCII CONVERSION — BDC

**Opcode:** 0023

**Instruction Format:** 11



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b...BDCh..<ga,><ga,>,<cnt>]b...[<comment>]

**Example:**

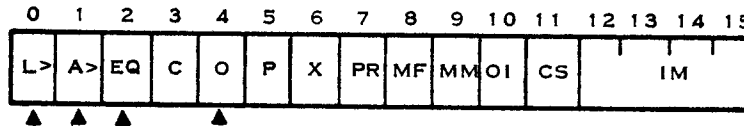
LABEL BDC @BIN,@DEC,9                      The nine-byte binary number starting at location BIN is converted to decimal ASCII and placed at location DEC.

**Definition:** The multibyte binary integer at the source address is converted to a multibyte decimal integer in USASCII format. The number of bytes in the source is specified by the <cnt> field. If <cnt> equals zero or is not present, the count is taken from the four LSBs of workspace

## BDC

register zero. If the four LSBs are zero, the count is 16. The result (of length  $2 \times \text{cnt}$  bytes) is deposited at the destination address. Leading zeros are stored as space characters ( $20_{16}$ ). The sign of the result is indicated by a USASCII plus or minus character after the least significant digit of the result.

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



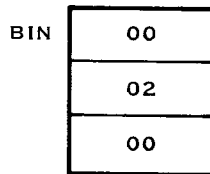
**Execution results:**  $(ga_n) \rightarrow (ga_n)$

**Application notes:** A zero is expressed by spaces followed by a single right-justified zero character and a plus character. If  $T_n$  and/or  $T_d$  is equal to three, the indicated register is incremented by the byte or character count (the byte count or twice the byte count, respectively).

The result of the BDC instruction is compared to zero and status register bits 0, 1, and 2 reflect the comparison. If the result cannot be contained in  $2 \times \text{cnt}$  bytes, status register bit 4 is set to one. In this event, the low order part of the ASCII result will be in the destination operand.

Care must be taken by the programmer to provide adequate space in the destination operand to hold the result. For example, a one-byte binary number has the range  $-128$  to  $+127$ . The ASCII result requires three bytes to represent either of these values, plus one byte for the sign. To ensure the correct number of bytes is allocated in this example, the programmer can perform an extend precision (EP) instruction, extending the precision of the one byte value to two bytes. This causes the destination operand to be four bytes long, enough for the one byte case. However, a general rule for extending the precision of the source operand cannot be devised, since 16 bytes is the maximum precision the EP instruction will extend a multiple precision number.

The following is an example of the binary to decimal ASCII conversion. The three bytes at location BIN contain the binary value of  $200_{16}$ , as shown figuratively:

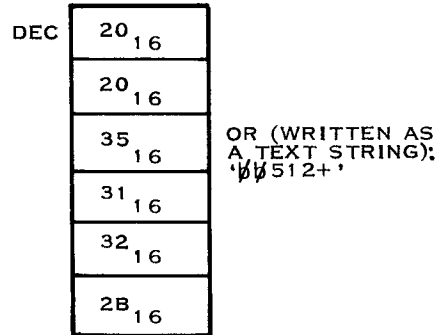


### The instruction

**LABEL BDC @BIN,@DEC,3**

## BDC, BIND

converts the value to a decimal integer, placing the integer in a six-byte string starting at location DEC. The results of this instruction are shown figuratively:

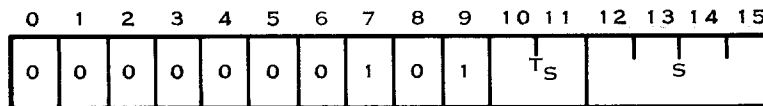


The logical greater than and arithmetic greater than bits of the status register are set; the equal and overflow bits are reset.

### 3.18 BRANCH INDIRECT — BIND

Opcode: 0140

Instruction Format: 6



Applicability: Processor set B

Syntax definition:

[<label>]b..BINDb..<ga, >b..[<comment>]

Example:

LABEL BIND R4

Branch to the instruction addressed by workspace register four.

Definition: The value specified by the source address is loaded into the program counter.

Status bits affected: None

Execution results: (ga,) → (PC)

Application notes: The indirect autoincrement form of the BIND instruction can be used in implementing threaded code. If T, is equal to three, the indicated register is incremented by two.

## BIND

In the following example of the branch indirect instruction, a branch and link to location PROCESS is first executed. At this point, address SUBR, the label of a list containing the addresses of several subroutines, is loaded into workspace register four, and the BIND instruction activates the first subroutine in the list.

The last instruction of each subroutine is a BIND instruction using the address contained in R4, which points to the next subroutine in the list to be executed.

When the return instruction at the end of the EXIT subroutine is encountered, execution resumes at the instruction following the BL @PROCESS instruction.

The sample code listed below shows the execution steps for this example.

BL @PROCESS	1) Branch to location PROCESS
.	
.	
(Executable Instructions)	8) Resume execution
.	
.	
.	
PROCESS	
LI R4,@SUBR	2) Load SUBR address in R4
BIND *R4 +	3) Branch to SUBRA and increment value in R4
.	
.	
(Executable Instructions)	
.	
.	
.	
SUBR	
DATA SUBRA	
DATA SUBRB	
DATA SUBRC	
.	
.	
.	
(Other Subroutines)	
DATA EXIT	
SUBRA	
(Executable Instructions)	
.	
.	
BIND *R4 +	4) Branch to SUBRB and increment value in R4
.	
.	
.	

## BIND, BL

### SUBRB

(Executable Instructions)

5) Branch to SUBRC and increment value in R4

### SUBRC

(Executable Instruction)

6) Branch to EXIT and increment value in R4

### EXIT

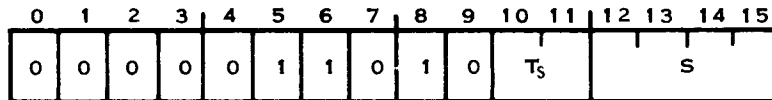
(Executable Instructions)

7) Return to instruction following BL @PROCESS instruction. RT is a pseudo-instruction described in Section 6.

### 3.19 BRANCH AND LINK — BL

Opcode: 0680

Instruction Format: 6



Applicability: Processor set A

Syntax definition:

[<label>]b..BLb..<ga, >b..[<comment>]

## BL, BLSK

**Example:**

LABEL BL @SUBR

Branch to the instruction at location SUBR and continue execution from that point. The current value of the program counter is stored in workspace register 11.

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

**Status bits affected:** None

**Execution results:** (PC) → (workspace register 11)  
 ga<sub>s</sub> → (PC)

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

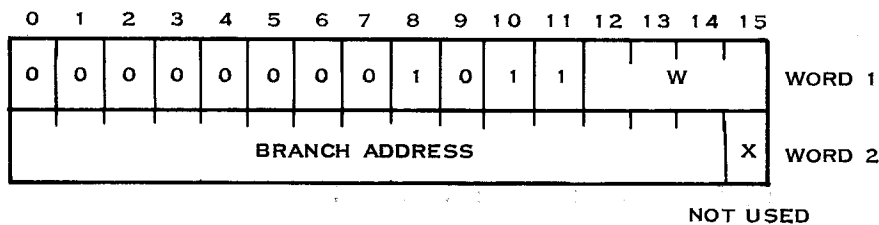
BL @TRAN

is located at memory location (PC count) 04BC<sub>16</sub>, then this instruction has the effect of placing memory location TRAN in the PC and placing the value 04C0<sub>16</sub> in workspace register 11. Refer to Section 4 for additional application notes.

### 3.20 BRANCH IMMEDIATE AND PUSH LINK TO STACK — BLSK

**Opcode:** 00B0

**Instruction Format:** 8



**Applicability:** Processor set C

**Syntax definition:**

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

**Example:**

LABEL BLSK R5,@SUBR

Branch to location SUBR and push the address of the next instruction onto the stack addressed by workspace register five.



## BLSK, BLWP

**Definition:** The first operand is the register containing the TOS (top of stack pointer); the second is the address to which to branch. The address of the next instruction (Program Counter plus four), is pushed onto the stack pointed to by the first operand <wa>. The address being pushed is treated as two bytes, so the TOS may be odd.

**Status bits affected:** None

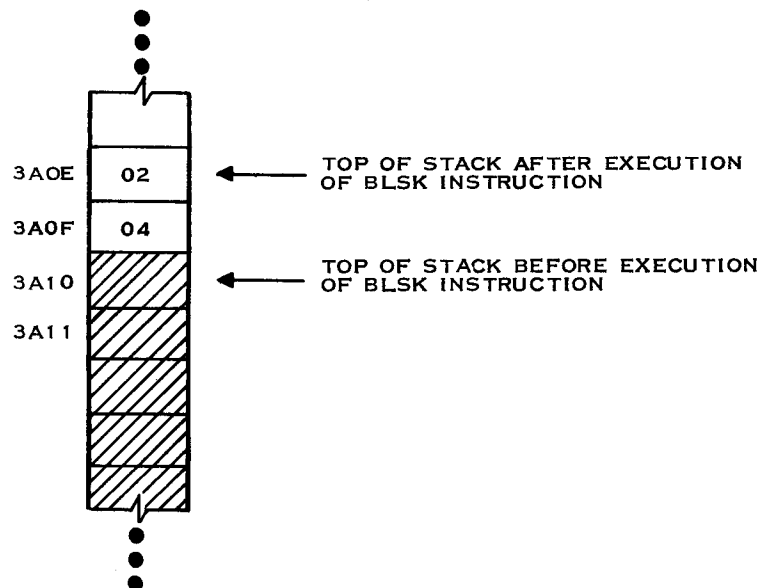
**Execution results:**  $PC \rightarrow (wa) - 1$   $(wa) - 2 \rightarrow (wa)$   $iop \rightarrow PC$

**Application notes:** Limit checking is not performed because the TOS must be specified as a register. Note that the branch address must specify a word address. If an odd branch address is given, it will be rounded down to a word boundary.

The following is an example of the branch immediate and push link to stack instruction. Workspace register five points to location  $3A10_{16}$ , label SUBR is at location  $4236_{16}$ , and the BLSK instruction is at location  $200_{16}$ . The instruction

`LABEL BLSK R5,@SUBR`

updates the PC to point to location  $4236_{16}$ , and updates workspace register five to point to the next available byte in the stack, in this case  $3A0E_{16}$ , shown figuratively:



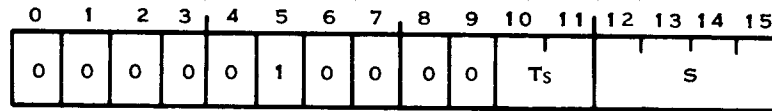
The status register is not affected.

### 3.21 BRANCH AND LOAD WORKSPACE POINTER — BLWP

**Opcode:** 0400

## BLWP, C

Instruction Format: 6



Applicability: Processor set A

Syntax definition:

[<label>]b..BLWPb..<ga,>b..[<comment>]

Example:

LABEL BLWP @VECT

Load the workspace pointer with the contents of the memory word at location VECT. Load the program counter with the contents of the memory word at location VECT plus two. The previous values of the workspace pointer, program counter, and status register are stored in new workspace registers 13, 14, and 15, respectively. The status register is unchanged.

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

Status bits affected: None

Execution results:

- (ga<sub>s</sub>) → (WP)
- (ga<sub>s</sub> + 2) → (PC)
- (old WP) → (Workspace register 13)
- (old PC) → (Workspace register 14)
- (ST) → (Workspace register 15)

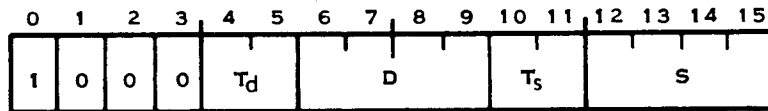
**Application notes:** Use the BLWP instruction for linkage to subroutines, program modules, or other programs that do not necessarily share the calling program workspace. Refer to Section 4 for a detailed explanation and example.

### 3.22 COMPARE WORDS — C

Opcode: 8000

## C, CB

Instruction Format: 1



Applicability: Processor set A

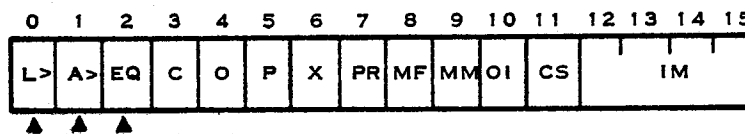
[<label>]b..Cb..<ga>,<ga>b..[<comment>]

Example:

LABEL C R2,R3                      Compare the contents of workspace register two and workspace register three.

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

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



Execution results: (ga<sub>s</sub>) : (ga<sub>d</sub>)

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

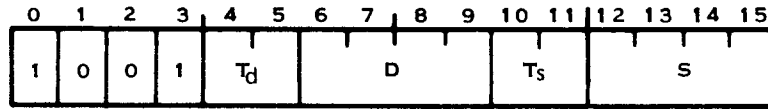
Source	Destination	Logical	Arithmetic	Equal
FFFF	0000	1	0	0
7FFF	0000	1	1	0
8000	0000	1	0	0
8000	7FFF	1	0	0
7FFF	7FFF	0	0	1
7FFF	8000	0	1	0
7FFE	7FFF	0	0	0

### 3.23 COMPARE BYTES — CB

Opcode: 9000

# CB

Instruction Format: 1



Applicability: Processor set A

Syntax definition:

[<label>]b..CBb..<ga>,<ga>b..[<comment>]

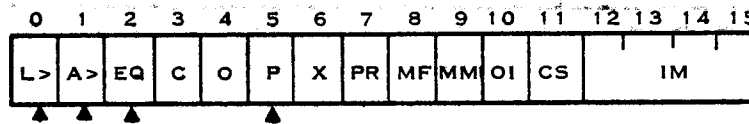
Example:

LABEL CB R2,R3

Compare the leftmost bytes of workspace register two and workspace register three.

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

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



Execution results: (ga<sub>s</sub>):(ga<sub>d</sub>)

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

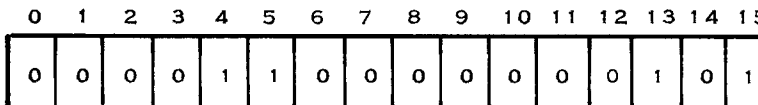
Source	Destination	Logical	Arithmetic	Equal	Odd Parity
FF	00	1	0	0	0
7F	00	1	1	0	1
80	00	1	0	0	1
80	7F	1	0	0	1
7F	7F	0	0	1	1
7F	80	0	1	0	1
7E	7F	0	0	0	0

# CDE

## 3.24 CONVERT DOUBLE PRECISION REAL TO EXTENDED INTEGER — CDE

Opcode: 0C05

Instruction Format: 7



Applicability: Processor set C

Syntax definition:

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

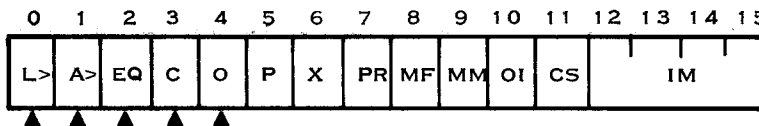
Example:

LABEL CDE

Convert the double precision number in the FPA to an extended integer and place it in the FPA.

Definition: Convert the double precision number in the FPA (R0, R1, R2, R3) to an extended integer in the FPA (R0,R1).

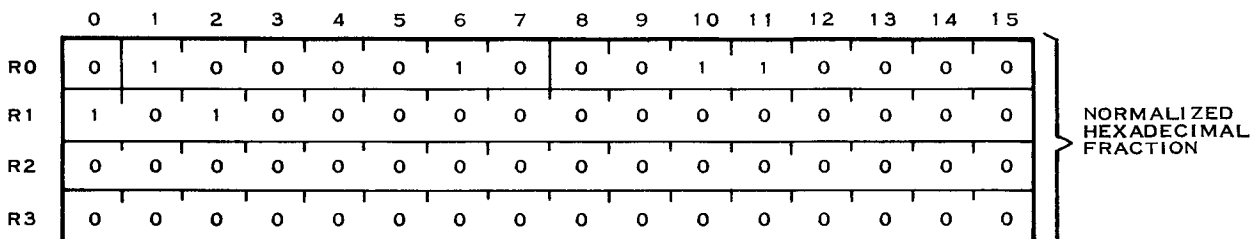
Status bits affected: Logical greater than, arithmetic greater than, equal, carry, and overflow



Execution results: FPA → FPA

Application notes: The result of the CDE instruction is compared to zero, and status register bits 0, 1, and 2 reflect the comparison. Status bit 3 is set to one. If status register bit 4 is set to one, overflow has occurred. Fractions are converted to zero without underflow.

The following is an example of the convert double precision real to extended integer instruction. The double precision real number in the FPA (R0–R3) is the normalized representation of  $30.A_{16}$  as shown figuratively:



## CDE, CDI

The instruction

LABEL CDE

converts the double precision real number in the FPA to an extended integer and places the result,  $30_{16}$ , in the FPA (R0-R1), as shown figuratively:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0

The logical greater than, arithmetic greater than, and carry bits of the status register are set; the equal and overflow bits are reset.

Refer to Section 2 for information concerning normalization and double precision real numbers.

### 3.25 CONVERT DOUBLE PRECISION REAL TO INTEGER — CDI

Opcode: 0C01

Instruction Format: 7

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1

Applicability: Processor set C

Syntax definition:

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

Example:

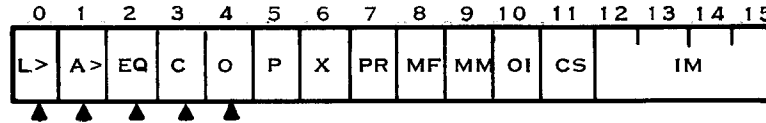
LABEL CDI

Convert the double precision number in the FPA to an integer and place it in the FPA.

Definition: The double precision number in the FPA (R0,R1,R2,R3) is converted to an integer and the result is placed in the FPA (R0).

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

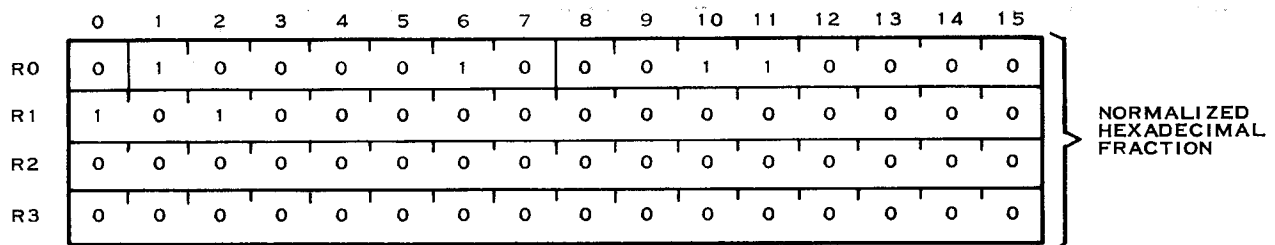
## CDI, CED



**Execution results:** FPA → FPA

**Application notes:** The results of the CDI instruction are compared to zero and status register bits 0, 1, and 2 reflect the results. Bit 3 is set to one. If overflow occurs, bit 4 is set to one. Fractions are converted to zero without underflow.

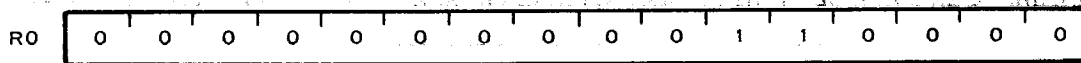
The following is an example of the convert double precision real to integer instruction. The double precision real number in the FPA (R0-R3) is the normalized representation of 30.A<sub>16</sub>, as shown figuratively:



The instruction

LABEL CDI

converts the double precision real number in the FPA to an integer, and places the result, 30<sub>16</sub>, in the FPA (R0), as shown figuratively:



The logical greater than, arithmetic greater than, and carry bits of the status register are set; the equal and overflow bits are reset.

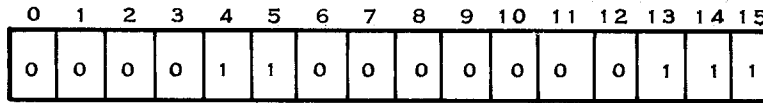
Refer to Section 2 for information concerning normalization and double precision real numbers.

### 3.26 CONVERT EXTENDED INTEGER TO DOUBLE PRECISION REAL — CED

**Opcode:** 0C07

## CED

Instruction Format: 7



Applicability: Processor set C

Syntax definition:

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

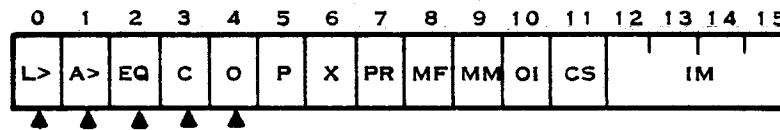
Example:

LABEL CED

Convert the extended integer in the FPA to a double precision real number and place it in the FPA.

**Definition:** The extended integer in the FPA (R0,R1) is converted to a double precision number and placed in the FPA (R0,R1,R2,R3).

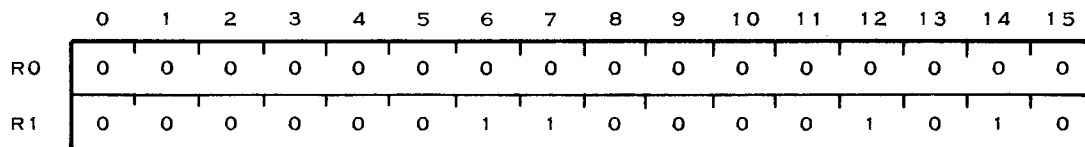
**Status bits affected:** Logical greater than, arithmetic greater than, equal, carry, and overflow



Execution results: FPA → FPA

**Application notes:** The result of the operation is compared to zero and status register bits 0, 1, and 2 reflect the comparison. Status register bits 3 and 4 are set to zero.

The following is an example of the convert extended integer to double precision real instruction. The value in the FPA (R0-R1) is 030A<sub>16</sub>, as shown figuratively:



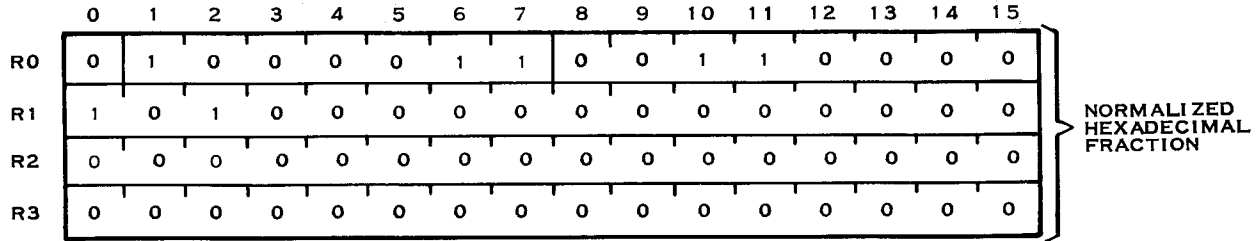
The instruction

LABEL CED



## CED, CER

converts the extended integer in the FPA to a normalized double precision real number and places it in the double precision FPA, as shown figuratively:



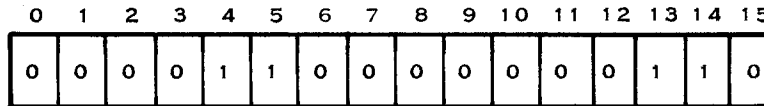
The logical greater than and arithmetic greater than bits of the status register are set; the equal, carry, and overflow bits are reset.

Refer to Section 2 for information concerning normalization and double precision real numbers.

### 3.27 CONVERT EXTENDED INTEGER TO REAL — CER

Opcode: 0C06

Instruction Format: 7



Applicability: Processor set C

Syntax definition:

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

Example:

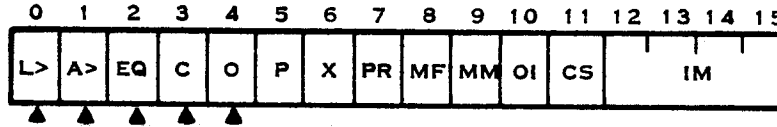
LABEL CER

Convert the extended integer in the FPA to a real number and place it in the FPA.

Definition: The extended integer in the FPA is converted to a real number and placed in the FPA (R0,R1).

## CER, CI

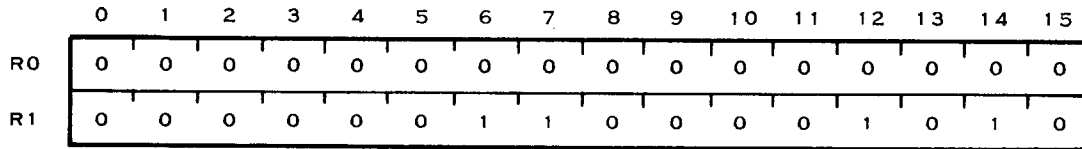
**Status bits affected:** Logical greater than, arithmetic greater than, equal, carry, and overflow



**Execution results:** FPA → FPA

**Application notes:** The result of the CER instruction is compared to zero and status register bits 0, 1, and 2 reflect the comparison. Status register bits 3 and 4 are reset to zero.

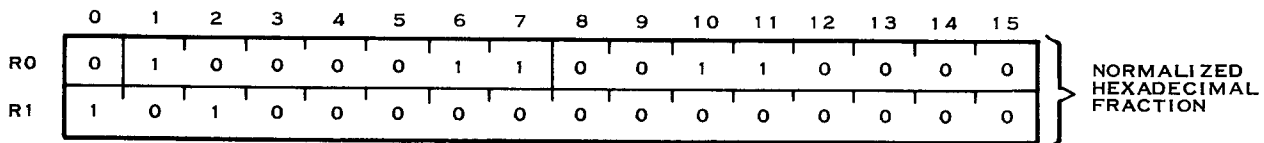
The following is an example of the convert extended integer to real instruction. The value in the FPA (R0,R1) is 030A<sub>16</sub>, as shown figuratively:



The instruction

LABEL CER

converts the extended integer in the FPA to a normalized single precision real number and places it in the single precision FPA, as shown figuratively:



The logical greater than and arithmetic greater than bits of the status register are set; the equal, carry, and overflow bits are reset.

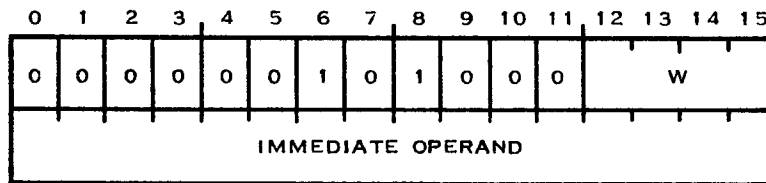
Refer to Section 2 for information concerning normalization and single precision real numbers.

### 3.28 COMPARE IMMEDIATE — CI

**Opcode:** 0280

## CI, CID

**Instruction Format:** 8



**Applicability:** Processor set A

**Syntax definition:**

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

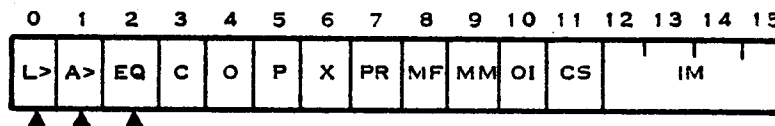
**Example:**

LABEL CI R3,7

Compare the contents of workspace register three to seven.

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

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



**Execution results:** (wa) : iop

**Application notes:** Use the CI instruction to compare the workspace register to an immediate operand. For example, if the contents of workspace register nine is 2183<sub>16</sub>, then the instruction

CI R9,>F330

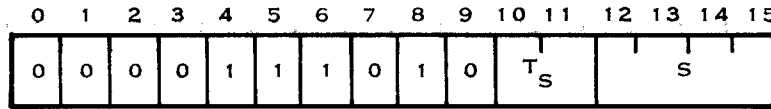
results in the arithmetic greater than status bit set, and the logical greater than and equal status bits reset.

### 3.29 CONVERT INTEGER TO DOUBLE PRECISION REAL — CID

**Opcode:** 0E80

# CID

Instruction Format: 6



Applicability: Processor set C

Syntax definition:

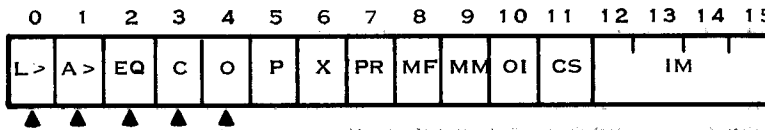
[<label>]b..CIDb..<ga,>b..[<comment>]

Example:

LABEL CID @WORD      Convert the integer at location WORD to a double precision real number and place it in the FPA.

Definition: The integer at the source address (1 word) is converted to double precision and is stored in the floating point accumulator (R0-R3).

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



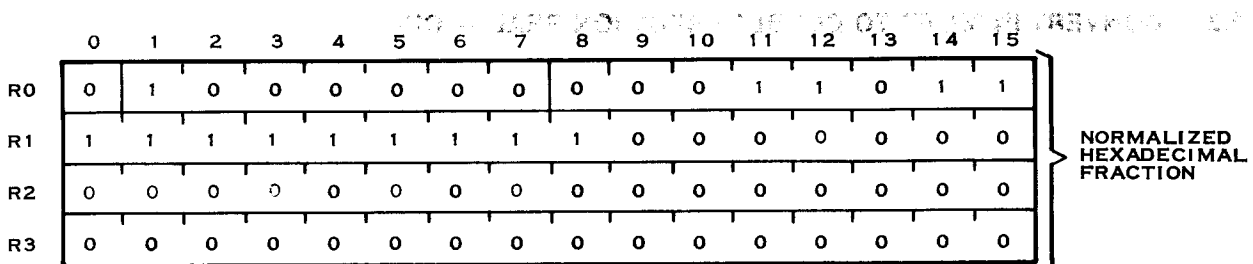
Execution results: (ga,) → FPA

Application notes: If T<sub>s</sub> is equal to three, the indicated register is incremented by two. The results of the CID instruction are compared to zero and status register bits 0, 1, and 2 reflect the results. Status register bits 3 and 4 are reset.

The following is an example of the convert integer to double precision real instruction. The value in WORD is 1BFF<sub>16</sub>. The instruction

LABEL CID @WORD

converts the integer in WORD to a normalized double precision real number and places the value in the double precision FPA, as shown figuratively:



## CID, CIR

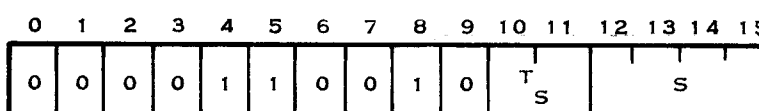
The logical greater than and arithmetic greater than bits of the status register are set; the equal, carry, and overflow bits are reset.

Refer to Section 2 for information on normalization and double precision real number.

### 3.30 CONVERT INTEGER TO REAL — CIR

Opcode: 0C80

Instruction Format: 6



Applicability: Processor set C

Syntax definition:

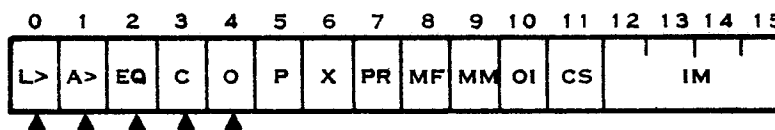
[<label>]b..CIRb..<ga>b..<comment>

Example:

LABEL CIR @WORD      Convert the integer at location WORD to a real number  
and store it in the FPA.

Definition: The integer specified by the source address is converted to a real number and stored in the FPA (R0-R1).

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



Execution results: (ga)<sub>i</sub> → FPA

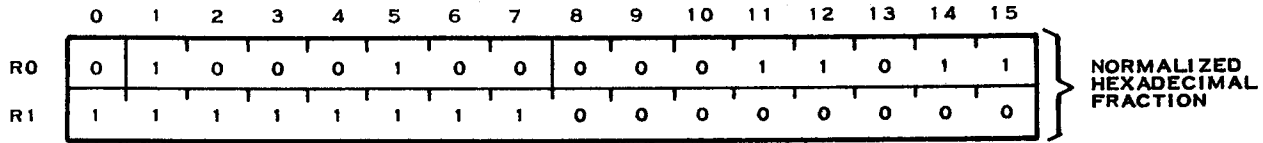
Application notes: The results of the CIR instruction are compared to zero, and status register bits 0, 1, and 2 reflect the comparison. Status register bits 3 and 4 are set to zero. If T<sub>i</sub> is equal to three, the indicated register is incremented by two.

The following is an example of the convert integer to real instruction. Location WORD contains the value 1BFF<sub>16</sub>. The instruction

LABEL CIR @WORD

## CIR, CKOF

converts the integer at location WORD to a normalized single precision real number and places the value in the single precision FPA, as shown figuratively:



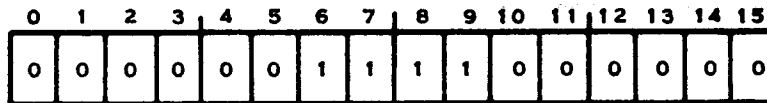
The logical greater than and arithmetic greater than bits of the status register are set; the equal, carry, and overflow bits are reset.

Refer to Section 2 for information concerning normalization and single precision real numbers.

### 3.31 CLOCK OFF — CKOF

Opcode: 03C0

Instruction Format: 7



Applicability: Processor set A, privileged instruction

Syntax definition:

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

Example:

STOCK CKOF                      Stop the clock.

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

When the privileged mode bit (bit 7 of ST register) is set to zero, instruction executes normally. When the privileged mode bit is set to one and execution of a CKOF instruction is attempted, an error interrupt occurs.

Status bits affected: None

Execution results: The line frequency clock is disabled, and the clock interrupt is cleared.

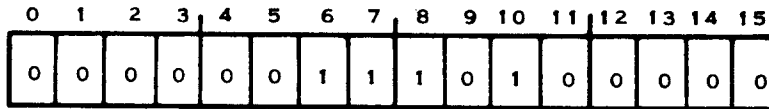
Application notes: Clock applications are described in the application notes in Section 4.

## CKON, CLR

### 3.32 CLOCK ON — CKON

Opcode: 03A0

Instruction Format: 7



**Applicability:** Processor set A, privileged instruction

**Syntax definition:**

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

**Example:**

```

STRTC CKON           Start the clock.
    
```

**Definition:** Enable the line frequency clock. The clock interrupt may be wired as interrupt level five or level 15. If the interrupt is enabled, an interrupt will occur every 8.33 ms after the initial interrupt, which may occur from 1 ms to 8.33 ms after the clock is turned on. The clock interrupt may be enabled/disabled by the interrupt mask as necessary. CKON is a privileged instruction.

When the privileged mode bit (bit 7 of ST register) is set to zero, the instruction executes normally. When the privileged mode bit is set to one and execution of a CKON instruction is attempted, an error interrupt occurs.

**Status bits affected:** None

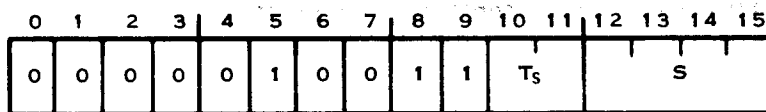
**Execution results:** The line frequency clock is enabled.

**Application notes:** Clock applications are described in the application notes in Section 4.

### 3.33 CLEAR — CLR

Opcode: 04C0

Instruction Format: 6



## CLR, CNTO

**Applicability:** Processor set A

**Syntax definition:**

[<label>]b..CLRb..<ga,>b..[<comment>]

**Example:**

PRELM CLR @BUFF(R2)

Replace the word at the location BUFF plus workspace register two with zeros.

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

**Status bits affected:** None

**Execution results:** 0 → (ga<sub>s</sub>)

**Application notes:** Use the CLR instruction to set a 16-bit memory word to zero. For example, if workspace register 11 contains the value 2001<sub>16</sub>, then the instruction

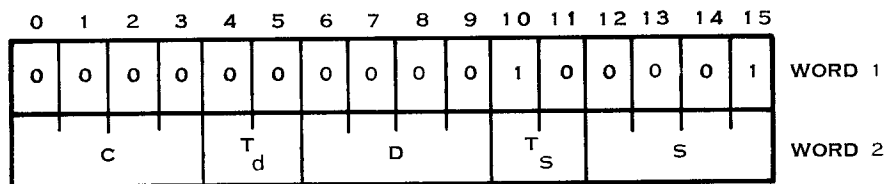
CLR \*R11

results in the contents of memory location 2001<sub>16</sub> being set to zero. Workspace register 11 and the status register are unchanged.

### 3.34 COUNT ONES — CNTO

**Opcode:** 0020

**Instruction Format:** 11



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..CNTOb..<ga,><ga,>[,<cnt>]b..[<comment>]



## CNTO, COC

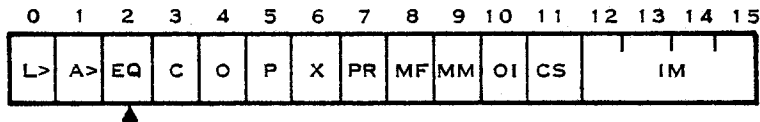
**Example:**

LABEL CNTO R4,R7,3

Count the number of ones in workspace register four and the most significant half of workspace register five and add the count to workspace register seven.

**Definition:** The number of ones in the multibyte value at the source address is counted and the count is added to the word at the destination address. The number of bytes of precision of the source operand is determined by the <cnt> field. If the <cnt> field equals zero or is not present, the count is taken from the four LSBs of workspace register zero. If the four LSBs of workspace register zero are zero, the count is 16.

**Status bits affected:** Equal



**Execution results:**  $(ga_d) + \# \text{ of '1' bits in } (ga_s) \rightarrow (ga_d)$

**Application notes:** If  $T_s$  is equal to three, the indicated register is incremented by the byte count. Status register bit 2 is set if the entire source operand is zero. If  $T_d$  is equal to three, the indicated register is incremented by two.

The following is an example of a count ones instruction. Workspace register four contains the value  $4312_{16}$ , workspace register five contains the value  $1136_{16}$ , and workspace register seven contains the value  $2157_{16}$ . The instruction

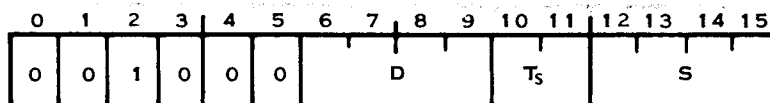
LABEL CNTO R4,R7,3

counts the number of ones in register four and the most significant half of register five and adds the count to register seven. After the execution of this instruction, the contents of register seven changes to the value  $215E_{16}$ . The equal bit of the status register is reset.

### 3.35 COMPARE ONES CORRESPONDING — COC

**Opcode:** 2000

**Instruction Format:** 3



**Applicability:** Processor set A

## COC, CRC

### Syntax definition:

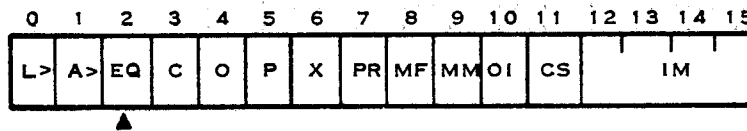
[<label>]b..COCb..<ga,>,<wa,>b..[<comment>]

### Example:

`LABEL COC @MASK,R2` Compare the bits in workspace register two that correspond with the logic one bits in MASK. If they are all equal, set the equal status bit.

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

**Status bits affected:** Equal



**Execution results:** The equal bit sets if all bits of (wa<sub>a</sub>) that correspond to the bits of (ga<sub>a</sub>) that are equal to one are also equal to one.

**Application notes:** Use the COC instruction to test single/multiple bits within a word in a workspace register. For example, if TESTBI contains the word C102<sub>16</sub> and workspace register eight contains the value E306<sub>16</sub>, then instruction

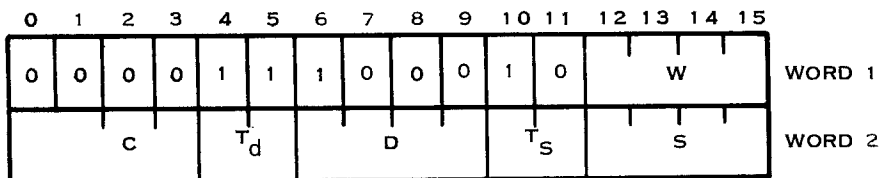
`COC @TESTBI,R8`

results in setting the equal status bit. If workspace register eight were to contain E301<sub>16</sub>, the equal status bit would reset. Use this instruction to determine if a workspace register has ones in the bit positions indicated by ones in a mask.

### 3.36 CYCLIC REDUNDANCY CODE CALCULATION — CRC

**Opcode:** 0E20

**Instruction Format:** 12



# CRC

**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..CRCb..<ga,><ga\_d>,<cnt>],[<ckpt>]b..[<comment>]

Trailing commas in the operand list may be omitted. The checkpoint register may be omitted from the instruction if a default has been specified with the CKPT assembler directive.

**Example:**

CRC @CORE,@TCORE,,R6

Update the value of the word at location TCORE by the value of the byte string starting at location CORE. The length of CORE is specified by R0 or as a tagged string. R6 is the checkpoint register.

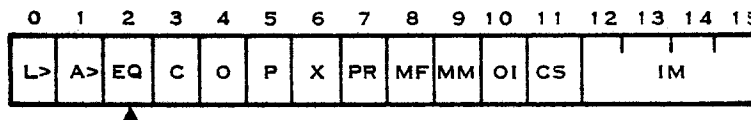
**Definition:** The 16-bit CRC partial sum at the destination address is updated by the byte string at the source address. The string length may be specified in the <cnt> field, in workspace register zero, or as a tagged string (if <cnt> = 0 and R0 = >FFFF). If the <cnt> field is not present, then zero is assumed. If the length of the string is 16 bytes or more, the checkpoint register is used for interrupts. If an interrupt occurs during execution, checkpoint data is stored in the checkpoint register. After the interrupt is serviced, execution continues from the point it was interrupted. Upon completion of the instruction, the checkpoint register is set to -1.

The checkpoint register value plus one is used as an initial index into the string (from the beginning of the string). To access the first byte (lowest address) in the string, the checkpoint register must be set to -1 before the instruction is executed. When the string length is zero, the CRC partial sum is not updated.

## NOTE

1. If no checkpoint register is specified, the default must be defined by the CKPT assembler directive.
2. The register following the checkpoint register must contain the 16-bit polynomial ( $X^{16} = \text{LSB}$ ;  $X^1 = \text{MSB}$ ) for the CRC calculation.

**Status bits affected:** The resultant CRC partial sum is compared to zero and status bit 2 is set accordingly.



**Execution results:** (ga\_s,ga\_d) CRC (ckpt + 1) → (ga\_d)

## CRC

*Application notes:* If  $T_s$  is equal to three, the indicated register is incremented by the string length. The resulting partial sum is compared to zero and status register bit 2 reflects the comparison. If  $T_d$  is equal to three, the indicated register is incremented by two.

Table 3-3 displays the format of the byte string for the CRC instruction.

**Table 3-3. CRC Byte String Format**

OPTIONAL BYTE COUNT								
FIRST BYTE OF STRING	$X^{n-7}$	$X^{n-6}$	$X^{n-5}$	$X^{n-4}$	$X^{n-3}$	$X^{n-2}$	$X^{n-1}$	$X^{n^*}$
SECOND BYTE OF STRING	$X^{n-15}$	$X^{n-14}$	$X^{n-13}$	$X^{n-12}$	$X^{n-11}$	$X^{n-10}$	$X^{n-9}$	$X^{n-8}$
	⋮							
LAST BYTE OF STRING	$X^{17}$	$X^{18}$	$X^{19}$	$X^{20}$	$X^{21}$	$X^{22}$	$X^{23}$	$X^{24}$

\*  $X^n$  IS THE FIRST BIT TRANSMITTED.

EFFECTIVE DESTINATION	$X^9$	$X^{10}$	$X^{11}$	$X^{12}$	$X^{13}$	$X^{14}$	$X^{15}$	$X^{16}$
EFFECTIVE DESTINATION+1	$X^1$	$X^2$	$X^3$	$X^4$	$X^5$	$X^6$	$X^7$	$X^8$

MS BYTE OF WORD AFTER <CKPT> REGISTER	$X^1$	$X^2$	$X^3$	$X^4$	$X^5$	$X^6$	$X^7$	$X^8$
LS BYTE OF WORD AFTER <CKPT> REGISTER	$X^9$	$X^{10}$	$X^{11}$	$X^{12}$	$X^{13}$	$X^{14}$	$X^{15}$	$X^{16}$

## CRC,CRE

The following is an example of the cyclic redundancy code instruction. STRING points to a five-byte string containing the values >9, >8, >7, >6, and >5, R4 contains the value ABCD, and the polynomial located in R11 is >21A5. The instructions

```
SETO R10
LABEL CRC @STRING,R4,5,R10
```

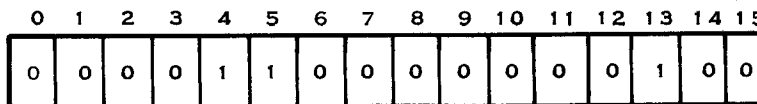
update R4 by the values of the byte string pointed to by STRING. After execution of this instruction, the value of R4 is >D0AD.

The equal bit of the status register is reset.

### 3.37 CONVERT REAL TO EXTENDED INTEGER — CRE

Opcode: 0C04

Instruction Format: 7



Applicability: Processor set C

Syntax definition:

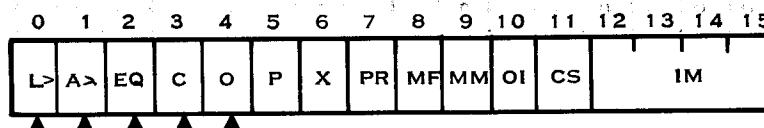
[<label>]b..CREb..[<comment>]

Example:

```
LABEL CRE          Convert the real number in the FPA to an extended
                    integer and place it in the FPA.
```

Definition: The real number in the FPA (R0,R1) is converted to an extended integer in the FPA (R0,R1).

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

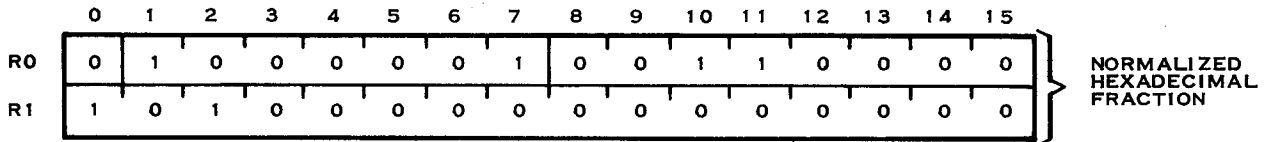


Execution results: FPA → FPA

## CRE, CRI

*Application notes:* The result of the CRE instruction is compared to zero, and status register bits 0, 1, and 2 reflect the comparison. Status bit 3 is set to one. Status bit 4 is set to one if overflow occurs; otherwise, it is set to zero. Fractions convert to zero and underflow does not occur.

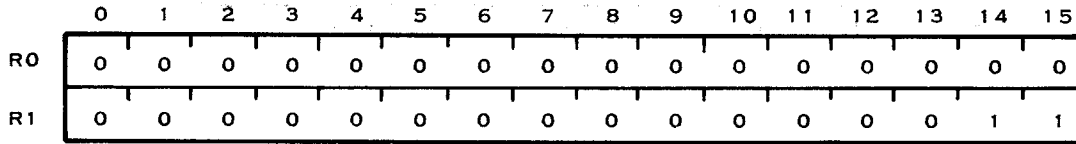
The following is an example of the convert real to extended integer instruction. The real number in the FPA (R0-R1) is the normalized representation of  $3.0A_{16}$ , as shown figuratively.



The instruction

LABEL CRE

converts the real number in the FPA to an extended integer, and places the result,  $3_{16}$ , in the FPA (R0-R1), as shown figuratively:



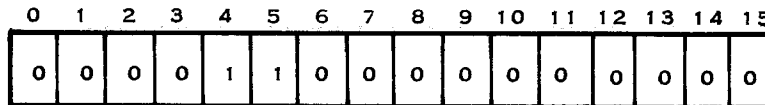
The logical greater than, arithmetic greater than, and carry bits of the status register are set; the equal and overflow bits are reset.

Refer to Section 2 for information concerning normalization and single precision real numbers.

### 3.38 CONVERT REAL TO INTEGER — CRI

*Opcode:* 0C00

*Instruction Format:* 7



*Applicability:* Processor set C

*Syntax definition:*

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

# CRI

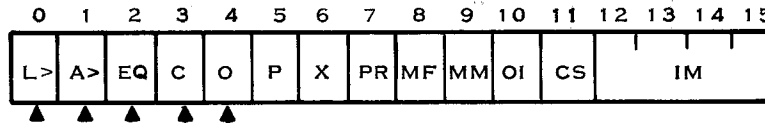
**Example:**

LABEL CRI

Convert the real number in the FPA to an integer and place it in the FPA.

**Definition:** The real number in the FPA (R0,R1) is converted to an integer in the FPA (R0). Fractions convert to zero.

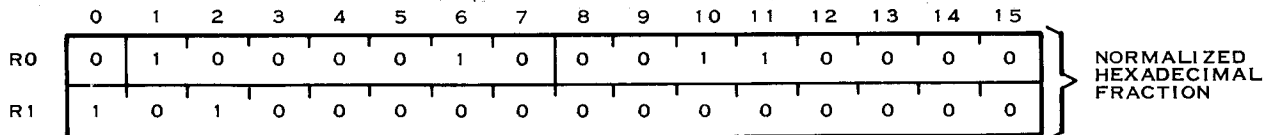
**Status bits affected:** Logical greater than, arithmetic greater than, equal, carry, and overflow



**Execution results:** FPA → FPA

**Application notes:** The result of the CRI instruction is compared to zero and status register bits 0, 1, and 2 reflect the comparison. Status bit 3 is set to one. Status bit 4 is set to one if overflow occurs, otherwise it is set to zero.

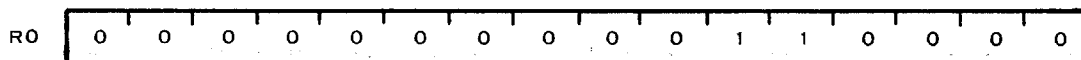
The following is an example of the convert real to integer instruction. The real number in the FPA (R0-R1) is the normalized representation of 30.A<sub>16</sub> as shown figuratively:



The instruction

LABEL CRI

converts the real number in the FPA to an integer, and places the result, 30<sub>16</sub>, in the FPA (R0), as shown figuratively:



The logical greater than, arithmetic greater than, and carry bits of the status register are set; the equal and overflow bits are reset.

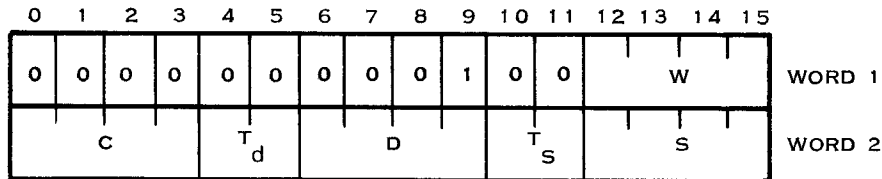
Refer to Section 2 for information concerning normalization and single precision real numbers.

## CS

### 3.39 COMPARE STRINGS — CS

Opcode: 0040

Instruction Format: 12



Applicability: Processor set C

Syntax definition:

[<label>]b...CSb...<ga, >,<ga, >,<cnt>],<ckpt>]b...[<comment>]

Trailing commas on the operand list may be omitted. The checkpoint register may be omitted from the instruction if a default has been specified with the CKPT assembler directive.

Example:

LABEL CS @INPUT,@CORE,,R6

Compare the string, starting at location INPUT, for the length specified in workspace register zero, with the string starting at location CORE, for the length specified in workspace register zero. Index the first nonequal bytes in workspace register six.

**Definition:** The bytes in the string starting at the source address are compared to the bytes in the string starting at the destination address. The comparison reflects any of three conditions: equality/inequality of the strings, equality/inequality of the strings as signed, two's complement integers, and equality/inequality of the strings as unsigned binary numbers. Status bits 0 through 2 are set to reflect the results of the comparison.

When the two strings are compared for equality, the equal status bit (bit 2) reflects the results of the comparison. If the strings are equal, the equal status bit is set to one at the end of the instruction. If the strings are not equal, the equal status bit is set to zero at the end of the instruction.

When the two strings are compared as signed, two's complement values, the arithmetic greater than bit (bit 1) and the equal bit of the status register reflect the results of the comparison. If the strings are equal, the equal status bit is set to one at the end of the instruction. If the strings are not equal, the equal bit is set to zero, and the arithmetic greater than bit reflects the relationship of the source string to the destination string (set to one, the source string is arithmetically greater than the destination string).



**CS**

When the two strings are compared as unsigned binary numbers, the logical greater than bit (bit 0) and the equal bit of the status register reflect the results of the comparison. If the strings are equal, the equal status bit is set to one at the end of the instruction. If the strings are not equal, the equal status bit is set to zero, and the logical greater than bit reflects the relationship of the source string to the destination string (set to one, the source string is greater than the destination string as an unsigned binary number).

There is no difference in the instruction when comparing the strings as different values. The interpretation as strings, two's complement integers, or unsigned binary numbers is performed using bits 0 through 2 of the status register. The equal bit (bit 2) is used in all three cases to determine equality. The logical greater than bit (bit 0) reflects the comparison of the strings as unsigned binary numbers. The arithmetic greater than bit (bit 1) reflects the comparison of the strings as signed, two's complement integers.

The string length may be specified in the <cnt> field, register zero, or as a tagged string (if <cnt> = 0 and R0 = >FFFF).

An index to the first nonequal bytes is returned in the checkpoint register <ckpt> and status bits 0 through 2 reflect the comparison of the strings as binary integers. If the strings are equal, the checkpoint register is set to -1 and status bits 0 through 2 equal zero, zero, and one, respectively.

The checkpoint register value plus one acts as an initial index into the string. To access the beginning of the string it must be set to -1 (>FFFF) before the compare strings instruction is executed. If the checkpoint register is not set to ones before the CS instruction is executed, the initial value of status bit 2 (EQ) determines how the instruction operates:

- If status bit 2 equals one, it is assumed that the bytes skipped are equal. If two unequal bytes are subsequently found, the status bits (0 through 2) are set to reflect the comparison of the strings as binary integers.
- If status bit 2 equals zero, it is assumed that the instruction is being reexecuted with two unequal bytes having already been found. If two more unequal bytes are found, the status is set to reflect the comparison of the two bytes only (not the whole strings).

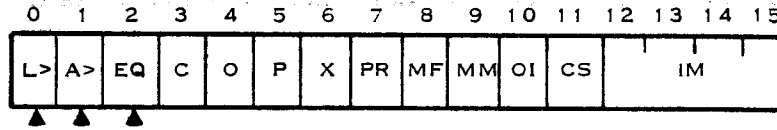
If the string length is zero, no comparison is made, status bits 0 through 2 are set to zero, zero, and one, respectively, and the checkpoint register is set to minus one.

If tagged strings are specified and the tags are equal, the instruction behaves as with untagged strings. The tag values do not affect the setting of the status. If the tags are not equal, the checkpoint register equals zero (pointing to the tag byte) and the status reflects the comparison of the two tags as unsigned integers (a tag of zero is handled as though it were 256). If the instruction is reexecuted with the checkpoint register equal to zero, the strings are compared for the number of bytes in the shortest string.

If the length of the strings is 16 bytes or more, the checkpoint register <ckpt> is used for interrupts. After the interrupt is serviced, the instruction continues the comparison where it left off.

# CS

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



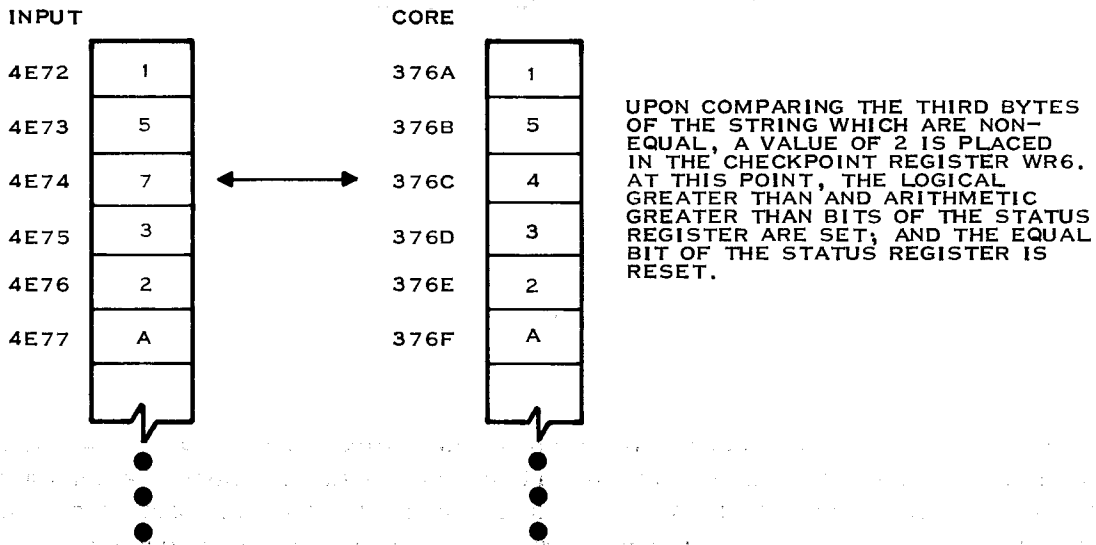
Execution results: (ga<sub>s</sub>) : (ga<sub>d</sub>)

Application notes: If T<sub>s</sub> and/or T<sub>d</sub> is equal to three, the indicated register is incremented by the byte count. The compare strings instruction may be used to compare two extended-precision binary integers.

The following is an example of the compare strings instruction. INPUT addresses a byte string at memory location 4E72<sub>16</sub>, CORE addresses a byte string at memory location 376A<sub>16</sub>, workspace register zero contains the value six, and workspace register six has been set to ones. The instruction

```
LABEL CS @INPUT,@CORE,,R6
```

compares a six-byte string beginning at location INPUT against a six-byte string beginning at location CORE. Workspace register six contains the displacement to the unequal bytes. The results, upon execution of the CS instruction in this example, are shown figuratively:



### 3.39.1 Reexecutable Instructions

The byte string manipulation instructions are reexecutable instructions. There are two types of reexecutable instructions. The interruptible instructions can be stopped during execution to allow the CPU to process hardware interrupts. After the hardware interrupts are serviced, the instruction continues from where it was interrupted. The interruptions are transparent to the program. The program reexecutable instructions are instructions that may be reexecuted under program control. There are three reexecutable instructions of this type: compare strings (CS), search

**CS**

string for equal byte (SEQB), and search string for not equal byte (SNEB). These instructions compare two strings or search a string until the first test condition is met. Under program control, these instructions can be reexecuted to find subsequent occurrences of the test condition.

Reexecutable instructions use a checkpoint register as the index (zero origin) into the string. The checkpoint register acts as the initial index on the first execution of the instruction. After the first execution, the checkpoint register indicates the byte at which the test condition was met. When the instruction is reexecuted, the checkpoint register indexes the next byte after the byte that met the test condition. The checkpoint register is incremented before the comparison is made. Therefore, the checkpoint register must be set to a -1 (FFFF<sub>16</sub>) to index the first byte of the string.

The example below shows the CS instruction used to compare the bytes of two strings. When a nonequal byte is found, a branch is made to an appropriate processing routine. When the processing routine is finished, the CS instruction is reexecuted to test the rest of the string. When the entire string is compared, control passes out of the string test loop.

STRINGA BYTE A0,B0,C0,D0,E0,F0,01,02,03,04,05,06,07,08,09,

STRINGB BYTE A0,B0,C0,FF,E0,F0,01,FF,03,04,05,06,07,08,09

	SETO	R0	INITIALIZE CHECKPOINT REGISTER <sup>1</sup>
LOOP	CS	@STRINGA,@STRINGB,15,R0	COMPARE STRINGS
	JEQ	OUT	JUMP IF STRINGS EQUAL <sup>2</sup>
	BL	@SUBR	BRANCH IF NOT EQUAL
	JMP	LOOP	LOOP IF NOT DONE
OUT	.	.	CONTINUE WITH PROGRAM
	.	.	
	.	.	
SUBR	.	.	BYTE PROCESSING ROUTINE
	.	.	
	RT	.	RETURN FROM ROUTINE

**Notes:**

<sup>1</sup> The set to ones instruction (SETO) initializes the checkpoint register to FFFF<sub>16</sub> for the initial string index.

<sup>2</sup> If the two strings are equal, the equal status bit (bit 2) is set to one. If the two strings are not equal, the equal status bit is set to one if the last bytes compared are equal.

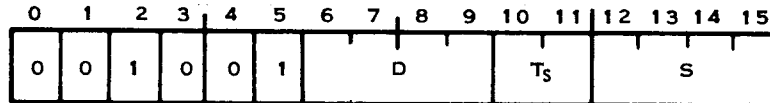
When the CS instruction is first executed, it returns the value three in R0. This indicates that byte three of the two strings are not equal. Upon return from the subroutine (SUBR), the jump to the CS instruction (LOOP) is taken. The CS instruction compares the strings starting at byte four, the next byte after the unequal bytes. The instruction returns the value seven in R0, indicating that byte seven of the two strings are unequal. Upon return from the subroutine, the CS instruction is executed again. The remainder of the strings are equal, which causes the equal status bit to be set. The loop is exited by the jump if equal (JEQ) instruction.

# CZC

## 3.40 COMPARE ZEROS CORRESPONDING — CZC

**Opcode: 2400**

**Instruction Format: 3**



**Applicability:** Processor set A

**Syntax definition:**

[<label>]b..CZCb..<ga,><wa,>b..[<comment>]

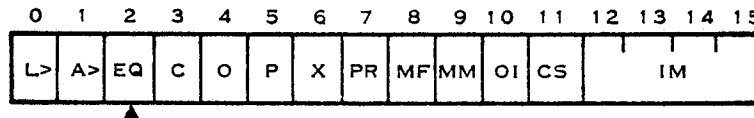
**Example:**

LABEL CZC @MASK,R2

Compare the bits in workspace register two that correspond with the logic one bits in MASK, and if they are all equal to a logic zero, set the equal status bit.

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

**Status bits affected:** Equal



**Execution results:** The equal bit sets if all bits equal to one of (ga<sub>i</sub>) correspond to bits equal to zero in (wa<sub>d</sub>).

**Application notes:** Use the CZC instruction to test single/multiple bits within a word in a workspace register. For example, if the memory location labeled TESTBI contains the value C102<sub>16</sub>, and workspace register eight contains 2301<sub>16</sub>, then the instruction

CZC @TESTBI,R8

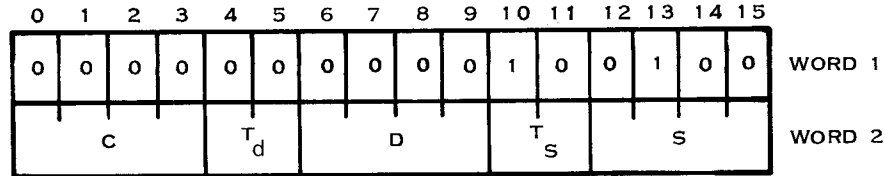
results in the equal status bit reset. If workspace register eight contained the value 2201<sub>16</sub>, then the equal status bit would set. Use this instruction to determine if a workspace register has zeros in the position indicated by ones in a mask.

**DBC**

**3.41 DECIMAL ASCII TO BINARY CONVERSION — DBC**

Opcode: 0024

Instruction Format: 11



*Applicability:* Processor set C

*Syntax definition:*

[<label>]b..DBCb..<ga.>,<ga.>,<cnt>]b..[<comment>]

*Example:*

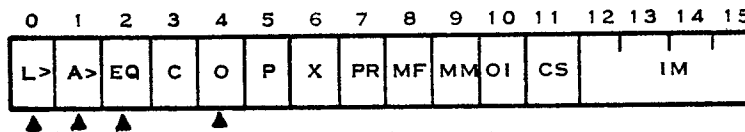
LABEL DBC @DEC,@BIN,11

The 22-byte decimal ASCII value in DEC is converted to an 11-byte binary value and placed in BIN.

*Definition:* The USASCII decimal character string at the source address is converted to a multibyte binary integer and deposited at the destination address. The number of bytes in the source is specified by twice the byte count in the <cnt> field. Any USASCII characters other than 0 through 9 and a minus sign are ignored during the conversion process. If a minus sign is encountered at any point in the string, the result is negative. The length of the result is specified in the <cnt> field.

If <cnt> equals zero or is not present, the count is taken from the four LSBs of workspace register zero. If the four LSBs of workspace register zero are zero, the count is 16.

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



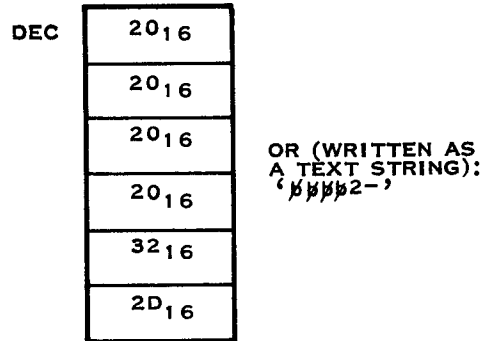
*Execution results:* (ga.) → (ga.) A decimal ASCII value at (ga.) is converted to a binary number at (ga.).

*Application notes:* If T<sub>s</sub> and/or T<sub>d</sub> is equal to three, the indicated register is incremented by the character or byte count, respectively.

The result of the DBC instruction is compared to zero and status register bits 0, 1, and 2 reflect the comparison. Status register bit 4 is set to one if a character other than 0 through 9, minus sign, a blank, or a plus is encountered.

## DBC, DD

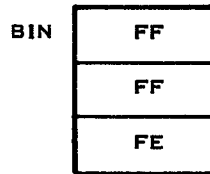
The following is an example of the decimal ASCII to binary conversion instruction. DEC addresses the six-byte decimal value of -2, as shown figuratively:



The instruction

LABEL DBC @DEC,@BIN,3

converts the decimal integer to a binary value and places the binary number in the three-byte string starting at location BIN. The results of this instruction are shown figuratively:

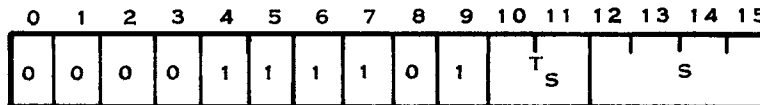


The logical greater than bit of the status register is set, and the arithmetic greater than, equal, and overflow bits are reset.

### 3.42 DIVIDE DOUBLE PRECISION REAL — DD

**Opcode:** 0F40

**Instruction Format:** 6



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..DDb..<ga>b..[<comment>]

# DD

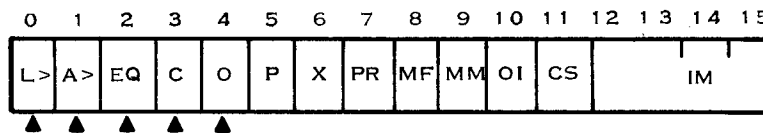
Example:

LABEL DD @WORD

Divide the contents of the FPA by the contents of the word at location WORD and place the result in the FPA.

**Definition:** Divide the FPA by the word at the source address and place the result in the FPA (R0-R3).

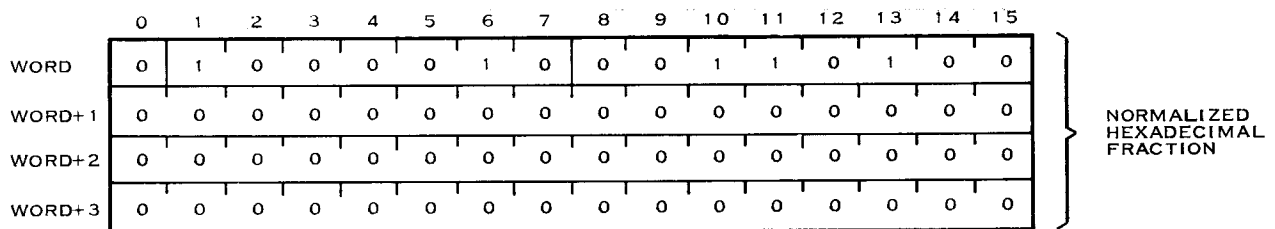
**Status bits affected:** Logical greater than, arithmetic greater than, equal, carry, and overflow



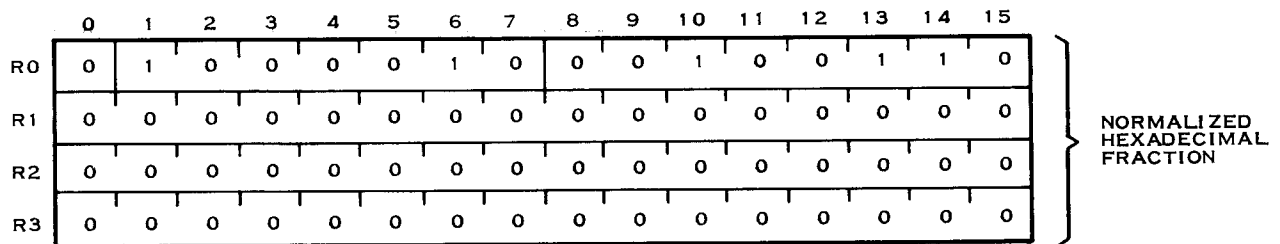
**Execution results:**  $FPA \div (ga.) \rightarrow FPA$

**Application notes:** The results of the DD instruction are compared to zero and status register bits 0, 1, and 2 reflect the comparison. If status register bits 3 and 4 are set to one, overflow has occurred. If status register bits 3 and 4 are set to zero and one, respectively, underflow has occurred. If T<sub>3</sub> is equal to three, the indicated register is incremented by eight.

The following is an example of the divide double precision real instruction. The value starting at location WORD, after normalization, is  $34_{16}$ , shown figuratively:



The value in the double precision FPA (R0-R3), after normalization, is  $26_{16}$ , shown figuratively:

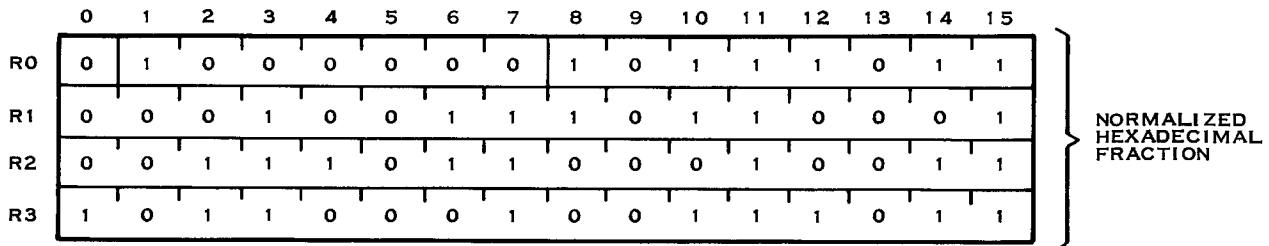


## DD, DEC

The instruction

LABEL DD WORD

divides the value in the FPA by the value starting at location WORD, and places the result, .BB13B13B13B13B<sub>16</sub>, in the FPA, shown figuratively:



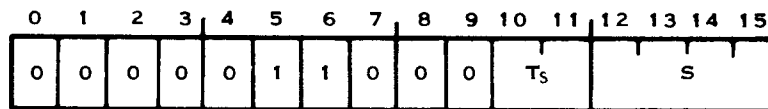
The logical greater than and arithmetic greater than bits of the status register are set; the equal, carry, and overflow bits of the status register are reset.

Refer to Section 2 for a detailed description of normalization and double precision floating point instructions.

### 3.43 DECREMENT — DEC

Opcode: 0600

Instruction Format: 6



Applicability: Processor set A

Syntax definition:

[<label>]b..DECb..<ga<sub>d</sub>>b..[<comment>]

Example:

LABEL DEC R2

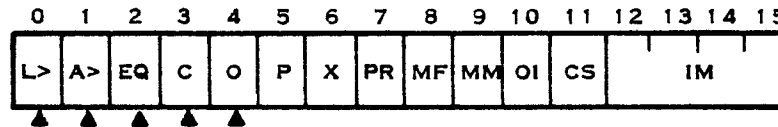
Subtract one from the contents of workspace register two and place the result in workspace register two.

**Definition:** Subtract a value of one from the source operand and replace the source operand with the result. The AU compares the result to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry out of bit 0, the carry status bit sets. When there is an overflow (the difference cannot be represented in a word as a two's complement value), the overflow status bit sets.



## DEC

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



*Execution results:* (ga<sub>n</sub>) - 1 → (ga<sub>n</sub>)

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

DEC @COUNT

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

### 3.43.1 Decrement Instruction Example

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

```
A1          TEXT          'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Array A2 is defined with the BSS statement as follows:

```
A2          BSS          26
```

The sample code for the solution is:

```

                LI        R5,26          COUNTER AND INDEX FOR A1
                LI        R4,A2          ADDRESS OF A2
INVRT          MOVB      @A1(R5),*R4 +  INVERT ARRAY*
                DEC       R5             REDUCE COUNTER
                JGT       INVRT          CONTINUE IF NOT COMPLETE
                .
                .
                .
    
```

**Note:**

\* @A1(5) addresses elements of array A1 in descending order as workspace register five is decremented. \*R4 + addresses array A2 in ascending order as workspace register four is incremented.

## DEC, DECT

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

A2 ZYXWVUTSRQPONMLKJIHGFEDCBA

Even though the result of this sequence of code is trivial, the example use of the MOV<sub>B</sub> instruction, with indexing by workspace register five, and the result incrementally placed into A2 with the autoincrement function can be useful in other applications.

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

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

A1	TEXT	'ABCDEFGHIJKLMNOPQRSTUVWXYZ'	
A2	BSS	26	
	LI	R5,25	COUNTER AND INDEX FOR A1*
	LI	R4,A2	ADDRESS OF A2
INVRT	MOV <sub>B</sub>	@A1(R5),*R4+	INVERT ARRAY
	DEC	R5	REDUCE COUNTER
	JOC	INVRT	CONTINUE IF NOT COMPLETE

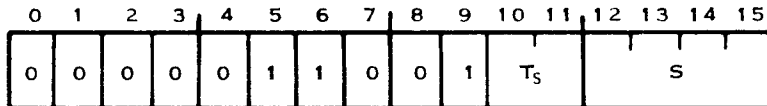
**Note:**

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

### 3.44 DECREMENT BY TWO — DECT

Opcode: 0640

Instruction Format: 6



Applicability: Processor set A

Syntax definition:

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

## DECT

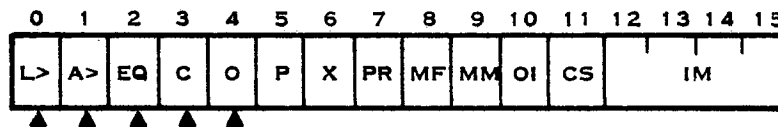
**Example:**

LABEL DECT @ADDR

Subtract two from the contents of location ADDR and replace the contents of location ADDR with the result.

**Definition:** Subtract two from the source operand and replace the source operand with the result. The AU compares the result to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry out of bit 0, the carry status bit sets. When there is an overflow (the result cannot be represented in a word as a two's complement value), the overflow status bit sets.

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



**Execution results:** (ga<sub>s</sub>) - 2 → (ga<sub>s</sub>)

**Application notes:** The DECT instruction is useful in counting and indexing *word* arrays. Also, use the DECT instruction to subtract a value of two from any addressable operand. For example, if workspace register PRT (PRT equals three) contains a value of 2C10<sub>16</sub>, then the instruction

DECT PRT

changes the contents of workspace register three to 2C0E<sub>16</sub>. The logical greater than, arithmetic greater than and carry status bits set while the equal and overflow status bits reset.

### 3.44.1 Decrement By Two Instruction Example

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

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

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

	LI	R4,20	INITIALIZE COUNTER
SUMS	A	@A1-2(R4),@A2-2(R4)	ADD ARRAYS
	DECT	R4	DECREMENT COUNTER BY TWO
	JGT	SUMS	REPEAT ADDITION

### NOTE

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

## DECT, DINT

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

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

There is another method by which this addition process may be accomplished. This method is shown in the following code:

	LI	R4,10	INITIALIZE COUNTER <sup>1</sup>
	LI	R5,A1-2	LOAD ADDRESS OF A1 <sup>2</sup>
	LI	R6,A2-2	LOAD ADDRESS OF A2 <sup>2</sup>
SUMS	A	*R5+,*R6+	ADD ARRAYS <sup>3</sup>
	DEC	R4	DECREMENT COUNTER
	JGT	SUMS	REPEAT ADDITION <sup>4</sup>

**Notes:**

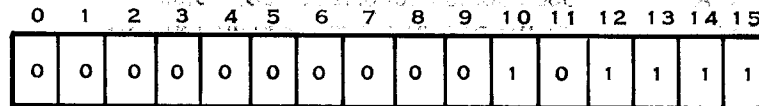
- <sup>1</sup> The counter is preset to ten (the number of elements in the array).
- <sup>2</sup> This address is incremented each time an addition takes place. The increment is via the autoincrement function (+).
- <sup>3</sup> The \* indicates that the contents of the register is to be used as an address, and the + indicates that it is automatically incremented by two each time the instruction is executed.
- <sup>4</sup> Workspace register four will only be greater than zero for ten executions of the DEC instruction, and control will be transferred to SUMS nine times after the initial execution.

The contents of array A2 are the same for this method as for the first.

### 3.45 DISABLE INTERRUPTS — DINT

**Opcode:** 002F

**Instruction Format:** 7



**Applicability:** Processor set C, privileged instruction

**Syntax definition:**

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

**Example:**

LABEL DINT

Disable all interrupts except level zero but do not change the interrupt mask.

## DINT, DIV

**Definition:** DINT disables all interrupts except level zero and the front panel load interrupt without changing the interrupt mask. DINT is a privileged instruction. The interrupts disabled by DINT are enabled only by EINT, RSET, or the power fail/restore sequence.

**Status bits affected:** None

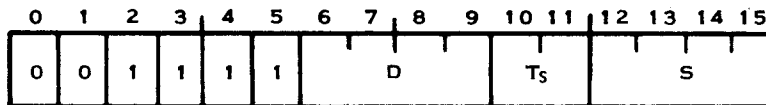
**Execution results:** None

**Application notes:** None

### 3.46 DIVIDE — DIV

**Opcode:** 3C00

**Instruction Format:** 9



**Applicability:** Processor set A

**Syntax definition:**

[<label>]b..DIVb..<ga>,<wa>b..[<comment>]

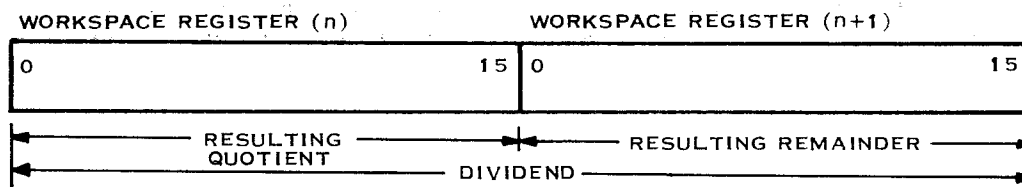
**Example:**

LABEL DIV @ADDR(R2),R3

Divide the contents of workspace registers three and four by the contents of the word at the location ADDR plus workspace register two and store the integer result in workspace register three with the remainder in workspace register four.

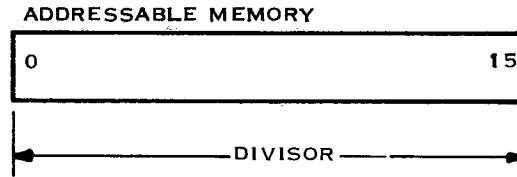
**Definition:** Divide the destination operand (a consecutive two-word area of workspace) by a copy of the source operand (one word) using integer rules, place the quotient in the first word of the two-word destination operand area, and place the remainder in the second word of that same area. This division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

Destination operand workspace registers



## DIV

Source operand

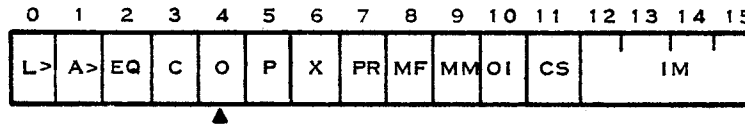


The first of the destination operand workspace registers, shown above, is addressed by the contents of the D field. The dividend is located right-justified in this two-word area. When the division is complete, the quotient (result) is placed in the first workspace register of the destination operand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

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

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

Status bits affected: Overflow



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

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

DIV @LOC,R2

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

# DIVS

## 3.47 DIVIDE SIGNED — DIVS

Opcode: 0180

Instruction Format: 6



2283243

Applicability: Processor set B

Syntax definition:

[<label>]b..DIVb..<ga,>b..[<comment>]

Example:

LABEL DIVS R4

Divide the two's complement of the value in workspace registers zero and one by the two's complement value in workspace register four and place the result in workspace register zero and the remainder in workspace register one.

**Definition:** The signed, double-precision two's complement integer in workspace registers zero and one is divided by the signed two's complement integer at the source address. Algebraic two's complement integer division is performed. The quotient is deposited in workspace register zero and the remainder is deposited in workspace register one. The quotient and remainder are derived so that the following conditions are met:

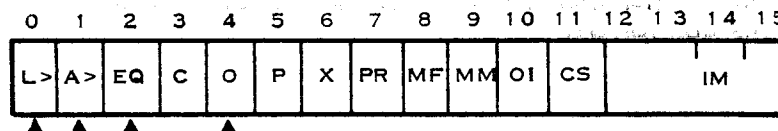
$$\text{DIVISOR} \times \text{QUOTIENT} + \text{REMAINDER} = \text{DIVIDEND.}$$

where the absolute value of the remainder is less than the absolute value of the divisor.

The sign of the remainder is the same as the sign of the dividend. The sign of the quotient is derived by algebraic rules, as shown below.

	DIVIDEND		
	POS.	NEG.	
DIVISOR	POS.	POS.	NEG.
	NEG.	NEG.	POS.

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



## DIVS, DR

*Execution results:* R0,R1 ÷ (ga)<sub>s</sub> = the remainder in R1  
 = the quotient in R0

*Application notes:* The DIVS instruction allows for division of signed numbers. The quotient is compared to zero and status register bits 0, 1, and 2 reflect the comparison. Status register bit 4 is set if the quotient cannot be expressed in 16 bits or if the divisor equals zero. If status bit 4 is set, workspace registers zero and one remain unmodified.

If T<sub>s</sub> is equal to three, the indicated register is incremented by two.

The following is an example of a divide signed instruction. The double precision value contained in workspace register zero and workspace register one is FFFFFFF9F<sub>16</sub>, and the value contained in workspace register four is FFD0<sub>16</sub>. The instruction

LABEL DIVS R4

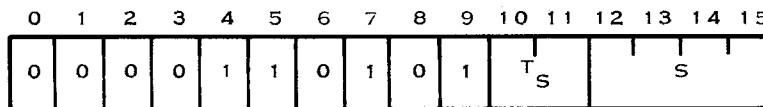
divides the two's complement of the value in R0 and R1, -61<sub>16</sub>, by the two's complement of the value in R4, -30<sub>16</sub>, and places the quotient result, 0002<sub>16</sub>, in R0 and the remainder, FFFF<sub>16</sub>, in R1.

The logical greater than and the arithmetic greater than bits of the status register are set; the equal and overflow bits of the status register are reset.

### 3.48 DIVIDE REAL — DR

*Opcode:* 0D40

*Instruction Format:* 6



*Applicability:* Processor set C

*Syntax definition:*

[<label>]b..DRb...<ga,>b..[<comment>]

*Example:*

LABEL DR R7

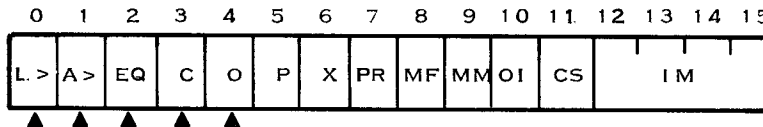
Divide the contents of the FPA (two words) by the contents of workspace registers seven and eight and place the result in the FPA.

*Definition:* The real number specified by the source address is divided into the FPA (R0,R1) and the result is stored in the FPA (R0,R1).



# DR

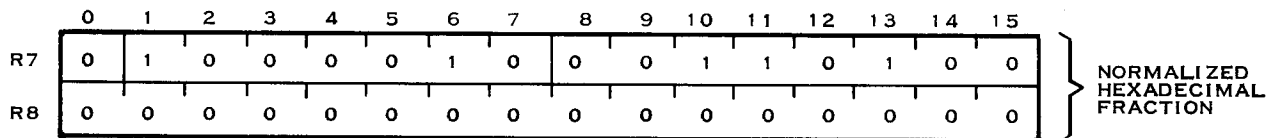
Status bits affected: Logical greater than, arithmetic greater than, equal, carry, and overflow



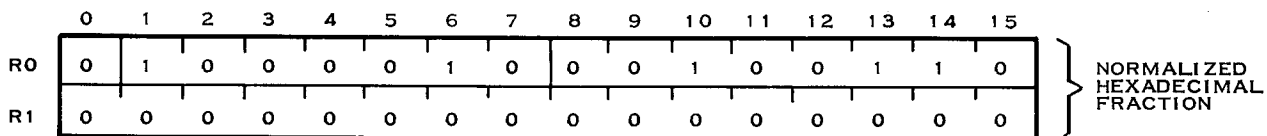
Execution results:  $FPA \div (ga_n) \rightarrow FPA$

Application notes: The result of the DR instruction is compared to zero and status register bits 0, 1, and 2 reflect the comparison. If status register bits 3 and 4 are set to zero and one, respectively, underflow has occurred. If they are set to one, overflow has occurred. If T<sub>i</sub> is equal to three, the indicated register is incremented by four.

The following is an example of the divide real instruction. The value contained in workspace registers seven and eight, after normalization, is  $34_{16}$ , shown figuratively.



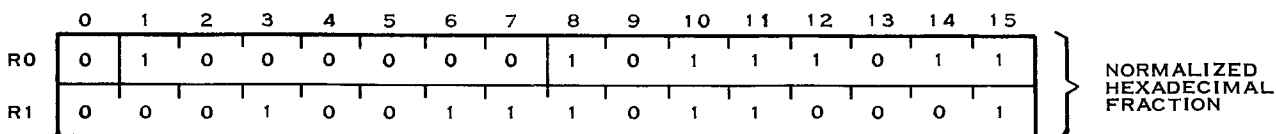
The value contained in the single precision FPA (R0,R1), after normalization, is  $26_{16}$ , shown figuratively:



The instruction

LABEL DR R7

divides the value in the FPA by the value contained in R7 and R8, and places the result,  $.BB13B_{16}$ , in the FPA, shown figuratively:



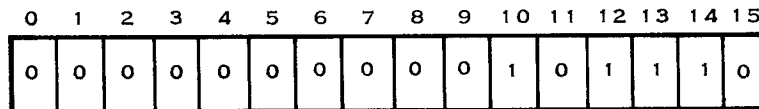
## DR, EINT, EMD

The logical greater than and the arithmetic greater than bits of the status register are set; the equal, carry, and overflow bits of the status register are reset.

### 3.49 ENABLE INTERRUPTS — EINT

Opcode: 002E

Instruction Format: 7



Applicability: Processor set C, privileged instruction

Syntax definition:

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

Example:

LABEL EINT Reenable previously disabled interrupts.

Definition: EINT reenables interrupts previously disabled by DINT, effective after the next instruction. The level of interrupts enabled is given by the current value of the interrupt mask. EINT is a privileged instruction.

Status bits affected: None

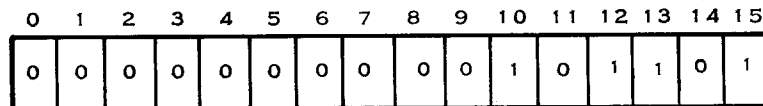
Execution results: None

Application notes: Note that RSET also enables the interrupts disabled by DINT.

### 3.50 EXECUTE MICRO-DIAGNOSTIC — EMD

Opcode: 002D

Instruction Format: 7



Applicability: Processor set C, privileged instruction

Syntax definition:

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



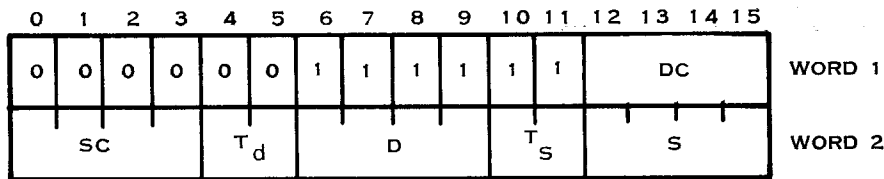
## EMD, EP

EMD loads the last 4K bytes of the loader ROMs into writable control store. The previous contents of the WCS are destroyed.

### 3.51 EXTEND PRECISION — EP

**Opcode:** 03F0

**Instruction Format:** 21



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..EPb..<ga>,<wa>,<scnt>,<dcnt>]b..[<comment>]

Trailing commas in the operand list may be omitted.

**Example:**

**LABEL EP @NUMBER, @NEWNUM, 6, 10**

The six-byte value starting at location NUMBER is extended to 10 bytes and placed in location NEWNUM.

**Definition:** The value specified by the source address (<scnt> bytes long) is extended in precision by placing it right-justified in the destination (<dcnt> bytes long), and appending sign-extension bytes to the left until the precision reaches the value specified by <dcnt>. If <scnt> is greater than <dcnt>, the destination is unchanged and overflow is indicated. If <scnt> is zero or is not present, the source precision is taken from bits 12 through 15 of workspace register zero. If bits 12 through 15 are zero, the source precision is 16 bytes. If <dcnt> is zero or is not present, the destination precision is taken from bits 4 through 7 of workspace register zero. If bits 4 through 7 are zero, the destination precision is 16 bytes.

**Status bits affected:** None

**Execution results:** (ga) → (ga), extended the number of bytes specified by <dcnt>

**Application notes:** If T<sub>s</sub> or T<sub>d</sub> is equal to three, the indicated register is incremented by the source count or the destination count, respectively.

## EP, IDLE

The following is an example of the extend precision instruction. NUMBER addresses a six-byte string, as shown figuratively:

NUMBER	FB
	04
	2C
	18
	AA
	87

The instruction

**LABEL EP @NUMBER,@NUMBR2,6,10**

moves NUMBER to location NUMBR2 and appends sign-extension bytes to the left of NUMBR2 for ten bytes. The result of this instruction is shown figuratively:

NUMBR2	FF
	FF
	FF
	FF
	FF
	FB
	04
	2C
	18
	AA
	87

### 3.52 IDLE — IDLE

Opcode: 0340

Instruction Format: 7

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0



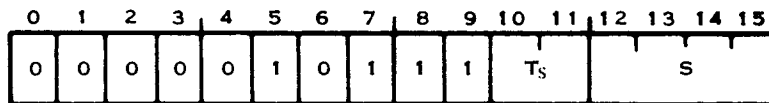


# INCT

## 3.54 INCREMENT BY TWO — INCT

Opcode: 05C0

Instruction Format: 6



Applicability: Processor set A

Syntax definition:

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

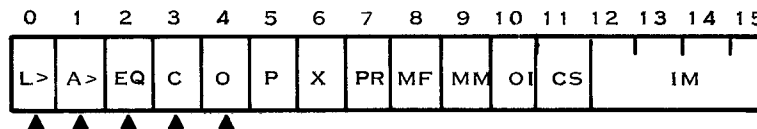
Example:

LABEL INCT R3

Add two to the contents of workspace register three and replace the contents of workspace register three with the result.

**Definition:** Add a value of two to the source operand and replace the source operand with the sum. The AU compares the sum to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry out of bit 0, the carry status bit sets. When there is an overflow, (the sum cannot be represented in a 16-bit word as a two's complement value), the overflow status bit sets.

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



Execution results: (ga<sub>s</sub>) + 2 → (ga<sub>s</sub>)

**Application notes:** Use the INCT instruction to count and index word arrays, and add the value of two to an addressable memory location. For example, if workspace register five contains the address (2100<sub>16</sub>) of the fifteenth word of an array, the instruction

INCT R5

changes workspace register five to 2102<sub>16</sub>, which points to the sixteenth word of the array. The logical greater than and arithmetic greater than status bits are set, while the equal, carry, and overflow status bits are reset.

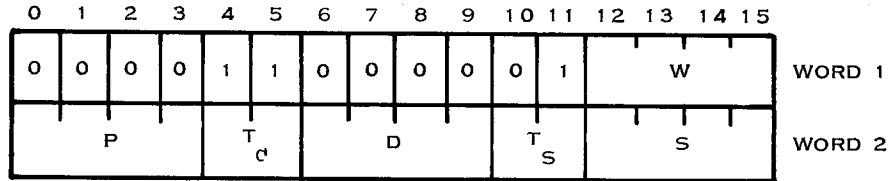


# INSF

## 3.55 INSERT FIELD — INSF

Opcode: 0C10

Instruction Format: 16



Applicability: Processor set C

Syntax definition:

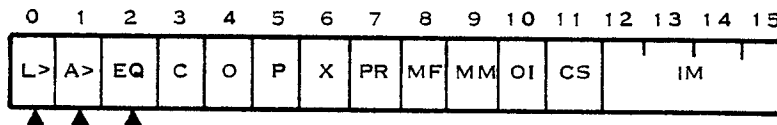
[<label>]b..INSFb..<ga<sub>s</sub>>,<ga<sub>d</sub>>,<pos>,<wid>)b..[<comment>]

Example:

LABEL INSF @ROW,@CORE, (3,6)      The six least-significant bits in the memory word at address ROW are placed in the memory word at address CORE, starting at bit 3.

**Definition:** The right-justified bit field of width (<wid>) in the word at the source address is deposited in the word at the destination address beginning at bit position <pos>. If either <pos> or <wid> is zero, the position or width is taken from workspace register zero. In this case, bits 4 through 7 of workspace register zero determine the position and bits 12 through 15 determine the width. If the four LSBs of register zero are zero when searching for <wid>, the width becomes 16 bits. If bits 4 through 7 are zero, then the position is zero. If <pos> plus <wid> is greater than 16, the remainder of the field is deposited in the next word in memory, starting at the most significant bit. The source and destination operands must start on a word boundary.

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



Execution results: (ga<sub>s</sub>) → (ga<sub>d</sub>)

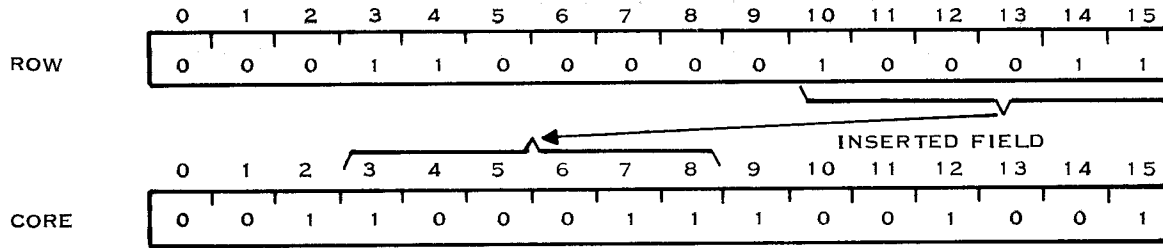
**Application notes:** The resulting field at the destination address of the INSF instruction is compared to zero and status register bits 0, 1, and 2 are set to indicate the results of the comparison. If T<sub>s</sub> or T<sub>d</sub> is equal to three, the indicated register is incremented by two.

## INSF, INV

The following is an example of the insert field instruction. ROW contains the value 1823<sub>16</sub>, and CORE contains the value 3FC9<sub>16</sub>. The instruction

`LABEL INSF @ROW, @CORE, (3, 6)`

places the rightmost six bits in ROW into CORE, starting at bit position 3 of CORE. The new value of CORE is 31C9<sub>16</sub>. The example is shown figuratively:

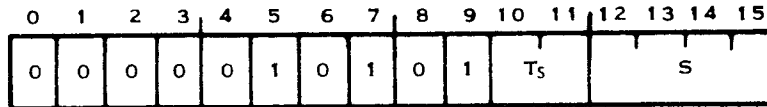


The logical greater than bit of the status register is set, and the arithmetic greater than and equal bits of the status register are reset. Only the inserted bits are compared to zero.

### 3.56 INVERT — INV

*Opcode:* 0540

*Instruction Format:* 6



*Applicability:* Processor set A

*Syntax definition:*

`[<label>]b...INVb...<ga,>b...[<comment>]`

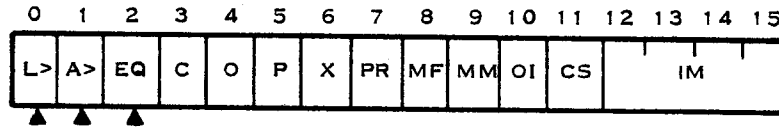
*Example:*

`COMPL INV @BUFF (R2)` Replace the contents of the location BUFF plus workspace register two with the one's complement of the original value.

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

## INV, IOF

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



**Execution results:** The one's complement of (ga<sub>s</sub>) is placed in (ga<sub>d</sub>)

**Application notes:** INV changes each logic zero in the source operand to a logic one and each logic one to a logic zero. For example, if workspace register 11 contains A54B<sub>16</sub>, then the instruction

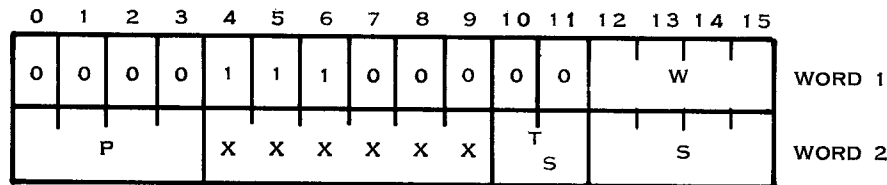
INV R11

changes the contents of workspace register 11 to 5AB4<sub>16</sub>. The logical greater than and arithmetic greater than status bits set, and the equal status bit resets.

### 3.57 INVERT ORDER OF FIELD — IOF

**Opcode:** 0E00

**Instruction Format:** 15



**Applicability:** Processor set C

[<label>]b..IOFb..<ga>,<pos>,<wid>b..[<comment>]

**Example:**

LABEL IOF @WORD, (0, 8) Reverse the order of eight bits of the contents of location WORD beginning at the bit position indicated in workspace register zero.

**Definition:** The order of the bits in the bit field of width <wid> is reversed starting at bit position <pos> in the word at the source address. If either <pos> or <wid> are zero, the position or width is taken from workspace register zero. In this case, bits 4 through 7 of workspace register zero indicate the position and bits 12 through 15 determine the width. If bits 12 through 15 equal zero, then the width is 16. If bits 4 through 7 are zero, then the position is zero. If <pos> plus <wid> is greater than 16, the next word is used for the remainder of the field, starting at the most significant bit. The source operands must start on a word boundary.

## IOF, JEQ

Status bits affected: None

Execution results: The bit field in the word at <ga,> is reversed.

Application notes: The following is an example of the Invert Order of Field instruction. WORD contains the value E72B<sub>16</sub>, and register zero contains 00FF<sub>16</sub>. The instruction

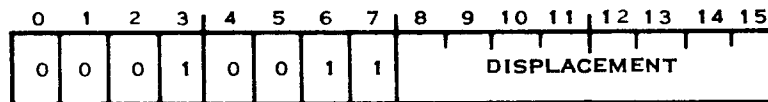
LABEL IOF @WORD, (0, 9)

reverses nine bits, starting a bit 0, of WORD and places the results in WORD. After execution of this instruction, the value in WORD is 73AB<sub>16</sub>.

### 3.58 JUMP IF EQUAL — JEQ

Opcode: 1300

Instruction Format: 2



Applicability: Processor set A

Syntax definition:

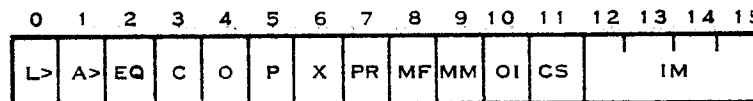
[<label>]b...JEQb...<exp>b...[<comment>]

Example:

LABEL JEQ LOC                      Jump to LOC if EQ = 1.

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

Status bits tested:



Jump if: EQ = 1

Status bits affected: None

## JEQ, JGT

**Execution results:** If the equal bit is equal to one, (PC) + displacement → (PC).

If the equal bit is equal to zero, (PC) → (PC).

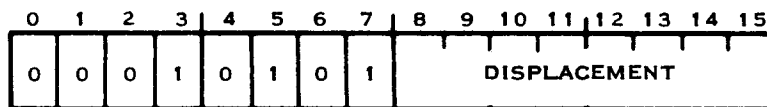
Refer to the explanation of execution in unconditional jump — JMP.

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

### 3.59 JUMP IF GREATER THAN — JGT

**Opcode:** 1500

**Instruction Format:** 2



**Applicability:** Processor set A

**Syntax definition:**

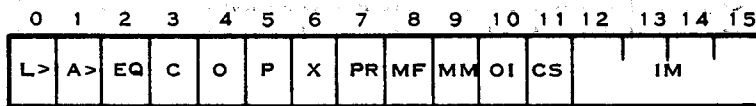
[<label>]b...JGTb...<exp>b...[<comment>]

**Example:**

LABEL JGT THERE                      Jump to THERE if A> = 1.

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

**Status bits tested:**



**Jump if:** A> = 1

**Status bits affected:** None

**Execution results:** If the arithmetic greater than bit is equal to one, (PC) + displacement → (PC).

If the arithmetic greater than bit is equal to zero, (PC) → (PC).

## JGT, JH

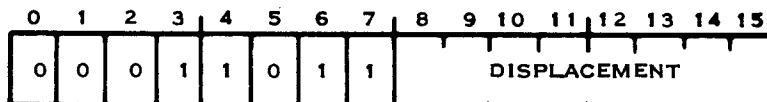
Refer to the explanation of execution in unconditional jump — JMP.

*Application notes:* Use the JGT instruction to transfer control when the arithmetic greater than status bit is set.

### 3.60 JUMP IF LOGICAL HIGH — JH

*Opcode:* 1B00

*Instruction Format:* 2



*Applicability:* Processor set A

*Syntax definition:*

[<label>]h..JHh..<exp>h..[<comment>]

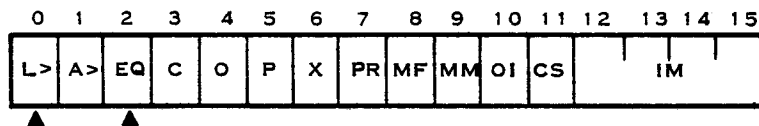
*Example:*

LABEL JH CONT

If the logical greater than bit is equal to one and the equal bit is equal to zero, jump to CONT.

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

*Status bits tested:*



*Jump if:* L> = 1 and EQ = 0

*Status bits affected:* None

*Execution results:* If the logical greater than bit is equal to one and the equal bit is equal to zero, (PC) + displacement → (PC).

If the logical greater than bit is equal to zero or the equal bit is equal to one, (PC) → (PC).

## JH, JHE

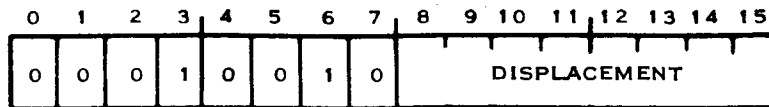
Refer to the explanation of the execution in unconditional jump — JMP.

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

### 3.61 JUMP IF HIGH OR EQUAL — JHE

*Opcode:* 1400

*Instruction Format:* 2



*Applicability:* Processor set A

*Syntax definition:*

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

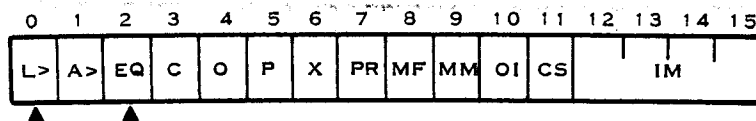
*Example:*

LABEL JHE LABEL1

If L> or EQ equals one, jump to LABEL1.

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

*Status bits tested:*



*Jump if:* L> = 1 or EQ = 1

*Status bits affected:* None

*Execution results:* If the logical greater than bit is equal to one or the equal bit is equal to one, (PC) + displacement → (PC).

If the logical greater than bit and the equal bit are equal to zero, (PC) → (PC).

Refer to the explanation of the execution in unconditional jump — JMP.





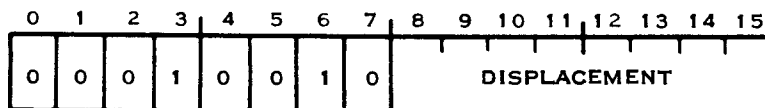
**JL, JLE**

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

**3.63 JUMP IF LOW OR EQUAL — JLE**

*Opcode:* 1200

*Instruction Format:* 2



*Applicability:* Processor set A

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

*Example:*

LABEL JLE THERE

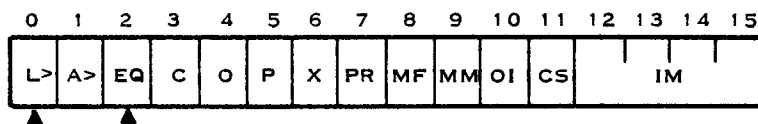
Jump to THERE when EQ = 1 or L> = 0.

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

**NOTE**

JLE is not jump if less than or equal.

*Status bits tested:*



*Jump if:* L> = 0 or EQ = 1

*Status bits affected:* None

*Execution results:* If the logical greater than bit is equal to zero or the equal bit is equal to one, (PC) + displacement → (PC).

If the logical greater than bit is equal to one and the equal bit is equal to zero, (PC) → (PC).

## JLE, FLT

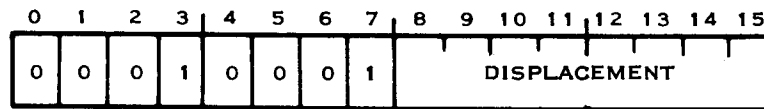
Refer to the explanation of execution in unconditional jump — JMP.

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

### 3.64 JUMP IF LESS THAN — JLT

*Opcode:* 1100

*Instruction Format:* 2



*Applicability:* Processor set A

*Syntax definition:*

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

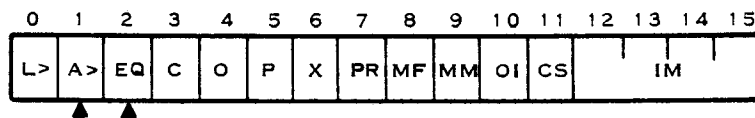
*Example:*

LABEL JLT THERE

Jump to THERE if  $A \geq 0$  and  $EQ = 0$ .

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

*Status bits tested:*



*Jump if:*  $A \geq 0$  and  $EQ = 0$

*Status bits affected:* None

*Execution results:* If the arithmetic greater than bit and the equal bit are equal to zero,  $(PC) + displacement \rightarrow (PC)$ .

If the arithmetic greater than bit is equal to one or the equal bit is equal to one,  $(PC) \rightarrow (PC)$ .

**JLT, JMP, JNC**

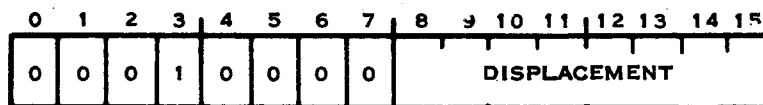
Refer to the explanation of execution in unconditional jump — JMP.

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

**3.65 UNCONDITIONAL JUMP — JMP**

*Opcode:* 1000

*Instruction Format:* 2



*Applicability:* Processor set A

*Syntax definition:*

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

*Example:*

LABEL JMP NXTLBL

Jump to NXTLBL.

*Definition:* Add the signed displacement in the instruction word to the PC and place the sum in the PC.

*Status bits affected:* None

*Execution results:* (PC) + displacement →(PC)

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

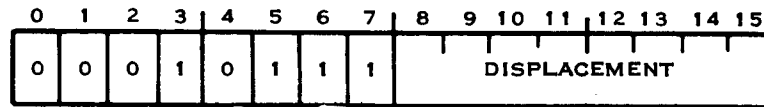
*Application notes:* Use the JMP instruction to transfer control to another section of the program module.

**3.66 JUMP IF NO CARRY — JNC**

*Opcode:* 1700

## JNC, JNE

Instruction Format: 2



Applicability: Processor set A

Syntax definition:

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

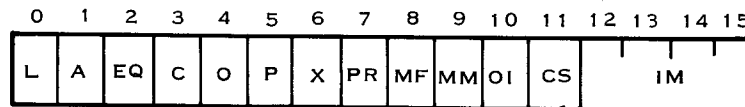
Example:

LABEL JNC NONE

Jump to NONE if C = 0.

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

Status bits tested:



Jump if: C = 0

Status bits affected: None

Execution results: If the carry bit is equal to zero, (PC) + displacement →(PC).

If the carry bit is equal to one, (PC) →(PC).

Refer to the explanation of execution in unconditional jump — JMP.

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

### 3.67 JUMP IF NOT EQUAL — JNE

Opcode: 1600



## JNO, JOC

Syntax definition:

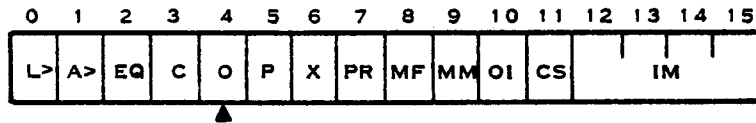
[<label>]b...JNOb...<exp>b...[<comment>]

Example:

LABEL JNO NORML                      Jump to NORML if O = 0.

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

Status bits tested:



Jump if: O = 0

Status bits affected: None

Execution results: If the overflow bit is equal to zero, (PC) + displacement → (PC)

If the overflow bit is equal to one, (PC) → (PC).

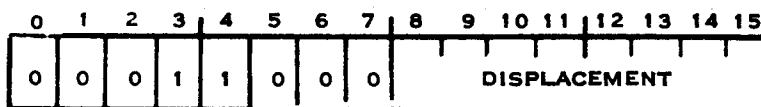
Refer to the explanation of execution in unconditional jump — JMP.

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

### 3.69 JUMP ON CARRY — JOC

Opcode: 1800

Instruction Format: 2



## JOC, JOP

**Applicability:** Processor set A

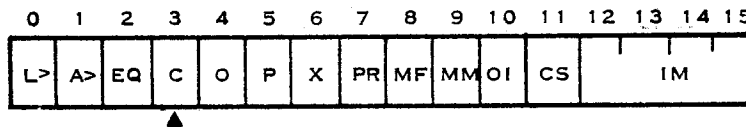
[<label>]b...JOCb...<exp>b...[<comment>]

**Example:**

LABEL JOC PROCED                      If C = 0, jump to PROCED.

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

**Status bits tested:**



**Jump if:** C = 1

**Status bits affected:** None

**Execution results:** If the carry bit is equal to one, (PC) + displacement → (PC).

If the carry bit is equal to zero, (PC) → (PC).

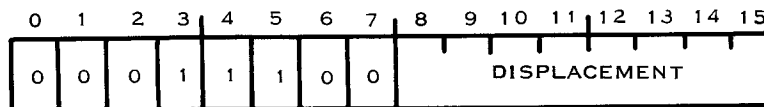
Refer to the explanation of execution in the unconditional jump — JMP.

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

### 3.70 JUMP IF ODD PARITY — JOP

**Opcode:** 1C00

**Instruction Format:** 2



**Applicability:** Processor set A

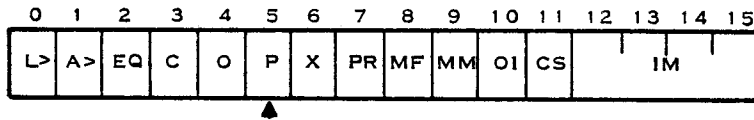
**Syntax definition:**

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

## JOP, LCS

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

**Status bits tested:**



**Jump if:** P = 1

**Status bits affected:** None

**Execution results:** If the odd parity bit is equal to one, (PC) + displacement → (PC).

If the odd parity bit is equal to zero, (PC) → (PC).

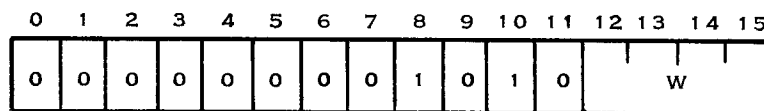
Refer to the explanation of execution in unconditional jump — JMP.

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

### 3.71 LOAD WRITABLE CONTROL STORE — LCS

**Opcode:** 00A0

**Instruction Format:** 18



**Applicability:** Processor set C, privileged instruction

**Syntax definition:**

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

**Example:**

LABEL LCS R4

Load the writable control store using the control block pointed to by R4.



## LCS, LD

**Definition:** The workspace register <wa> contains the starting location of a three-word control block that specifies the memory data to be loaded into the writable control store. The control block has the following format:

Word 1: the number of 64-bit microwords to be loaded.

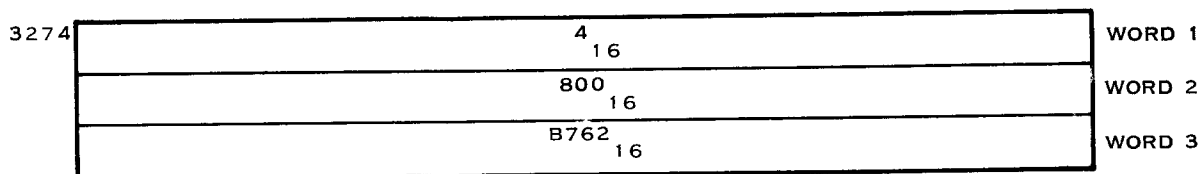
Word 2: the starting microword address.

Word 3: the starting address in memory of the data to be loaded.

**Status bits affected:** None

**Execution results:** ((wa) + 4) → writable control store

The following is an example of the load writable control store instruction. Workspace register four points to a block of memory at location  $3274_{16}$  containing the values as shown:



The instruction

LABEL LCS R4

loads 32 bytes, beginning at location  $B762_{16}$  in memory, into the four 64-bit microwords beginning at location  $0800_{16}$  in the writable control store.

When the privileged mode bit (bit 7) of the ST register is set to zero, the LCS instruction executes normally. When bit 7 is set to one, an error interrupt occurs and the instruction is not executed.

### NOTE

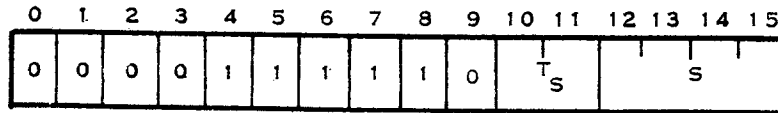
The microcode must be loaded into an address range of  $800_{16}$  through  $BFF_{16}$ . The starting address in word two of the control block must be within this range. Also, the starting address (word two) plus the number of microcode words (word one) cannot exceed the upper limit of  $BFF_{16}$ . An illegal operation interrupt occurs if these conditions are violated.

### 3.72 LOAD DOUBLE PRECISION REAL — LD

**Opcode:** 0F80

# LD

Instruction Format: 6



Applicability: Processor set C

Syntax definition:

[<label>]b..LDb..<ga,>b..[<comment>]

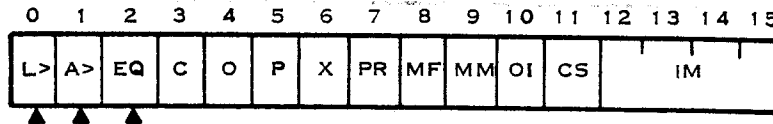
Example:

LABEL LD @WORD

Load the contents of the memory location pointed to by address WORD into the FPA.

Definition: The value specified by the source address is loaded into the FPA (R0-R3).

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



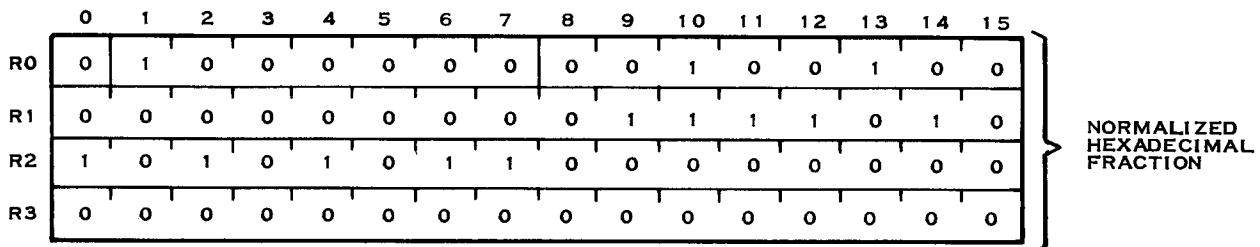
Execution results: (ga,) → FPA

Application notes: The results of the LD instruction are compared to zero and status register bits 0, 1, and 2 reflect the comparison. If T<sub>s</sub> is equal to three, the indicated register is incremented by eight.

The following is an example of the load double precision real instruction. The value contained in the eight bytes pointed to by WORD, after normalization, is .24007AAB<sub>16</sub>. The instruction

LABEL LD @WORD

stores the normalized fraction in the FPA (R0-R3), as shown figuratively:



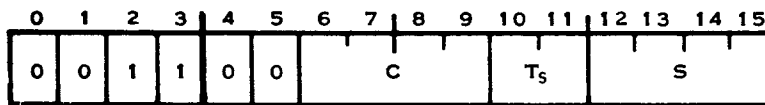
## LD, LDCR

The logical greater than and arithmetic greater than bits of the status register are set; the equal bit is reset.

### 3.73 LOAD CRU — LDCR

Opcode: 3000

Instruction Format: 4



Applicability: Processor set A

Syntax definition:

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

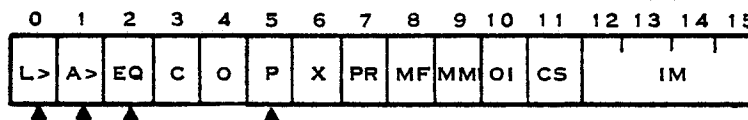
Example:

WRITE LDCR @BUFF,15

Send 15 bits from BUFF to CRU.

**Definition:** Transfer the number of bits specified in the <cnt> field from the source operand to the CRU. The transfer begins with the least significant bit of the source operand. The CRU address is contained in bits 3 through 14 of workspace register 12. When the <cnt> field contains zero, the number of bits transferred is 16. If the number of bits to be transferred is from 1 to 8, the source operand address is a byte address. If the number of bits to be transferred is from 9 to 16, the source operand address is a word address. When the source operand address is odd, the address is truncated to an even address prior to data transfer if more than eight bits are transferred. When the number of bits transferred is a byte or less, the source operand is compared to zero and the status bits are set/reset according to the results of the comparison. The odd parity status bit sets when the bits in a byte (or less) to be transferred establish odd parity.

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



## LDCR, LDD

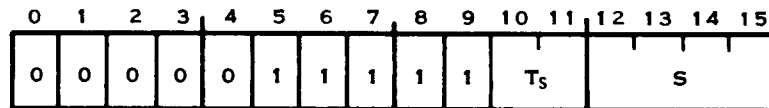
**Execution Results:** The number of bits specified by <cnt> is transferred from memory at address <ga,>to consecutive CRU lines beginning at the address in workspace register 12.

**Application notes:** Use the LDCR instruction to transfer a specific number of bits from memory to the CRU at the address contained in bits 3 through 14 of workspace register 12. Refer to Section 4 for a detailed example and explanation of the LDCR instruction.

### 3.74 LONG DISTANCE DESTINATION — LDD

**Opcode:** 07C0

**Instruction Format:** 6



**Applicability:** Processor set A, privileged instruction, memory mapping option required

**Syntax definition:**

[<label>]b...LDDb...<ga,>b...[<comment>]

**Example:**

LABEL LDD @SIXWD

Place the contents of the six words starting at location SIXWD into map file two and use map file two to develop the destination address of the next instruction.

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

**Status bits affected:** None

**Execution results:** When the privileged mode bit (bit 7 of ST register) is set to zero, the contents of a six-word area at address <ga,> are placed in map file two, and the destination address of the following instruction is mapped with map file two. (If T<sub>a</sub> of the following instruction is equal to zero, or if the destination address is a workspace register address, the new map is not used.) LDD is a privileged instruction.

When the privileged mode bit is set to one, an error interrupt occurs. The LDD instruction will not work unless the map enable bit (bit 4 at CRU base address >1FA0) is set to one.

## LDD

*Application notes:* Use the LDD instruction in the privileged mode to access an address outside of the current map. The contents of the six-word area are placed in the L1, L2, L3, B1, B2, and B3 registers of map file two as shown in paragraph 3.74.1. The address to which the map file applies is the destination address of the next instruction. Placing an LDD instruction prior to an instruction that has no destination operand, or an instruction having a workspace register address for the destination operand does not result in an access outside of the current map.

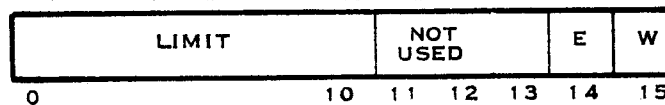
### 3.74.1 Memory Mapping

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

Bits 14 and 15 of the mapping limit registers indicate the status and protection of each segment of mapped memory. Bits E and W (14 and 15) determine the memory protection, listed in Table 3-4. These bits are controlled by software and tested by hardware. The mapping limit register is shown in Figure 3-1.

**Table 3-4. Memory Protection Control**

Mapping Limit Register Bits		Memory Protection in Segment
14(E)	15(W)	
0	0	No protection
0	1	Write protected
1	0	Execute protected
1	1	Execute and write protected



**Figure 3-1. Mapping Limit Register**

## LDD, LDS

When power is applied, the status register clears, selecting map file zero, and the limit and bias registers are set to zero. The map enable bit (bit 4 at CRU base address >1FA0) is also reset so addresses are not mapped. The map enable bit must be set to one to cause memory mapping to occur. When mapping is enabled and all of the map file registers are zero, the limits (one's complement of limit register contents) are >FFFF. This results in all addresses using bias register one, which contains zero, so that all logical addresses are mapped into the numerically equal physical addresses.

### NOTE

Map file zero is used by the operating systems to access the TILINE peripheral control space, execute XOP instructions, and in interrupt processing. Map file zero should not be used by application programs.

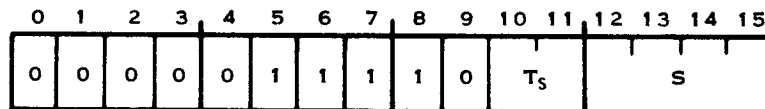
Map file one consists of three limit registers and three bias registers and is intended for application programs. Map file two similarly consists of three limit registers and three bias registers and is used to access a specified address outside of the current map. The LMF instruction loads map files zero and one. The LDD and LDS instructions load map file two.

For example, Figure 3-2 shows a map file and the comparison of processor addresses to limits. Figure 3-2 also shows the addition of a bias register to a processor address. The contents of the map file are chosen in this example so that processor addresses >0000 through >10FF map to addresses >000000 through >0010FF, processor addresses >1100 through >A0FF map to addresses >0322E0 through >03B2DF, and processor addresses >A100 through >F7FF map to addresses >04A100 through >04F7FF. A processor address greater than >F7FF results in an error interrupt. This mapping requires that limit register L1 contain binary 11101111000, the one's complement of the 11 most significant bits of >10FF. Similarly, limit register L2 contains binary 01011111000 (one's complement of 11 most significant bits of >A0FF) and limit register L3 contains binary 00001000000 (one's complement of the 11 most significant bits of >F7FF). Bias register B1 contains >0000, bias register B2 contains >188F, and bias register B3 contains >2000.

### 3.75 LONG DISTANCE SOURCE — LDS

*Opcode:* 0780

*Instruction Format:* 6



*Applicability:* Processor set A, privileged instruction, memory mapping option required

*Syntax definition:*

[<label>]b...LDSb...<ga, >b...[<comment>]

MAP FILE

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L1	1	1	1	0	1	1	1	1	0	0	0	X	X	X	0	0
B1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
L2	0	1	0	1	1	1	1	1	0	0	0	X	X	X	0	0
B2	0	0	0	1	1	0	0	0	1	0	0	0	1	1	1	1
L3	0	0	0	0	1	0	0	0	0	0	0	X	X	X	0	0
B3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

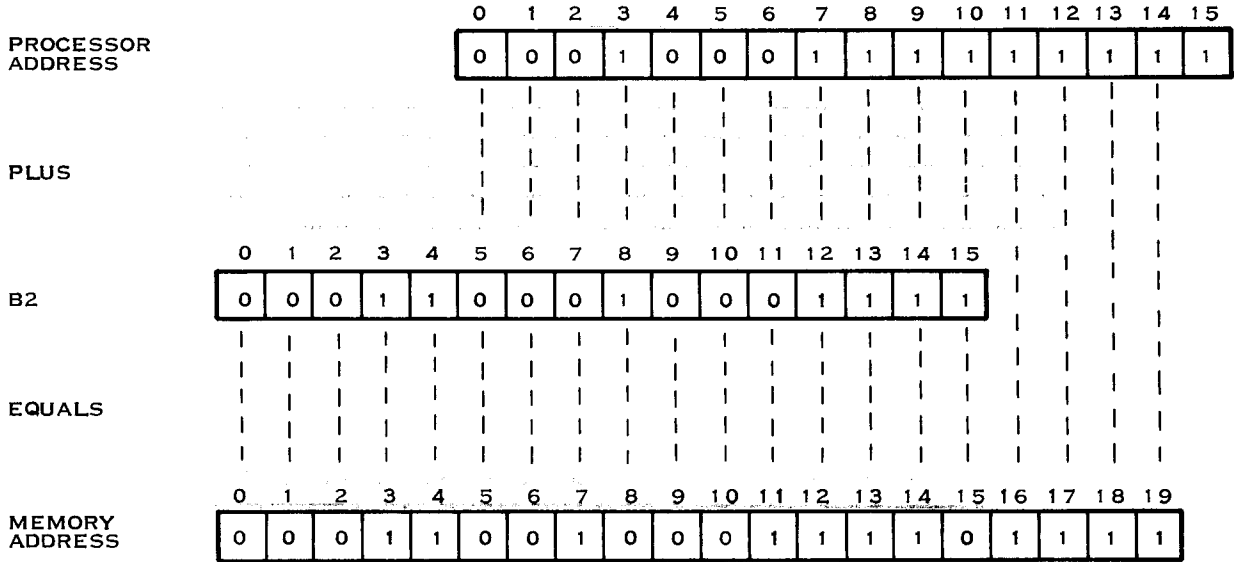
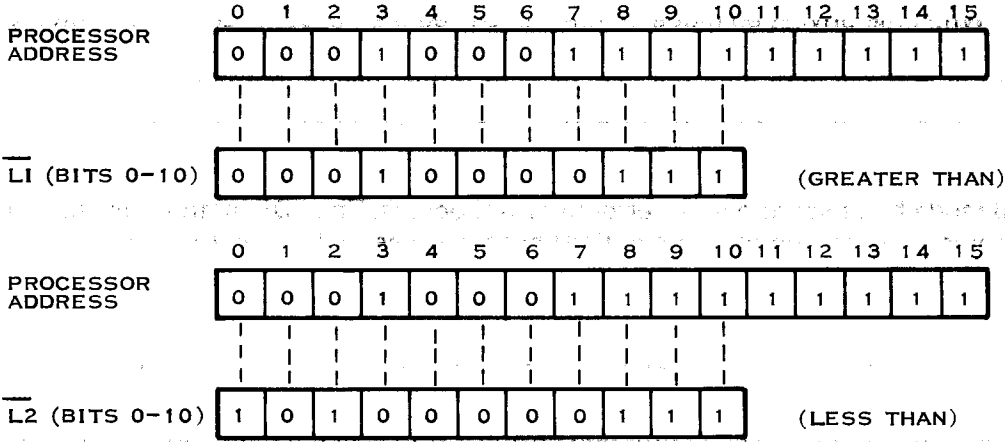


Figure 3-2. Address Development, Model 990 Mapping

## LDS, LI

**Example:**

LABEL LDS @SIXWD

Place the contents of the six words starting at location SIXWD into map file two and use map file two to develop the source address for the next instruction.

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

**Status bits affected:** None

**Execution results:** When the privileged mode bit (bit 7 of ST register) is set to zero, the contents of a six-word area at address <ga,> are placed in map file two and the source address of the following instruction is mapped with map file two. (If T<sub>s</sub> of the following instruction is equal to zero, or if the following instruction is a B, BL, or BLWP instruction, the new map is not used.) LDS is a privileged instruction.

When the privileged mode bit is set to one, an error interrupt occurs. The LDS instruction will not work unless the memory map enable bit (bit 4 at CRU base address 1FA0) is set to one.

**Application notes:** Use the LDS instruction in the privileged mode to access an address outside of the current map. The contents of a six-word area are placed in the L1, L2, L3, B1, B2, and B3 registers of map file two as shown in paragraph 3.74.1. The address to which the map file applies is the source address of the next instruction. Placing an LDS instruction prior to an instruction that has no source operand, or prior to an instruction having a workspace register address for the source operand does not result in an access outside of the current map.

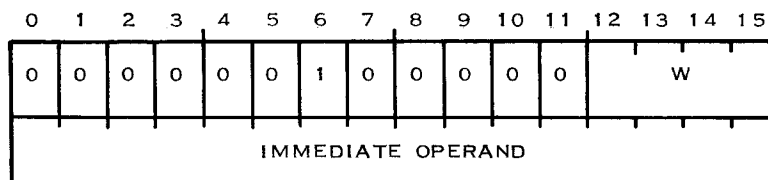
### NOTE

On the 990/12 computer, an LDS instruction immediately addressing workspace register eight or above causes the workspace cache to be dumped into the program starting at the instruction following the LDS. This problem may be avoided by using registers less than eight or by addressing registers greater than seven indirectly.

### 3.76 LOAD IMMEDIATE — LI

**Opcode:** 0200

**Instruction Format:** 8





## LI, LIM

**Applicability:** Processor set A

**Syntax definition:**

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

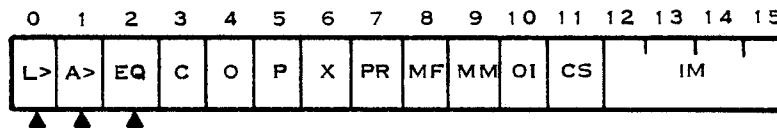
**Example:**

GETIT LI R3,>17

Load workspace register three with 17<sub>16</sub>.

**Definition:** Place the immediate operand (the word of memory immediately following the instruction) in the user-specified workspace register (W field). The immediate operand is not affected by the execution of this instruction. The immediate operand is compared to zero, and status bits 0, 1, and 2 are set or reset according to the result of the comparison.

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



**Execution results:** iop → (wa)

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

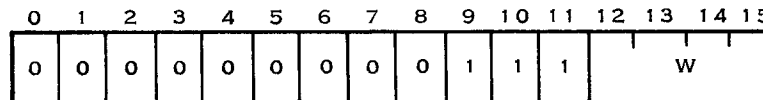
LI R7,5

initializes workspace register seven to the value 0005<sub>16</sub>. The logical greater than and arithmetic greater than status bits are set, while the equal status bit is reset.

### 3.77 LOAD INTERRUPT MASK — LIM

**Opcode:** 0070

**Instruction Format:** 18



**Applicability:** Processor set C, privileged instruction

## LIM, LIM

**Syntax definition:**

[<label>]b..LIMb..<wa<sub>a</sub>>b..[<comment>]

**Example:**

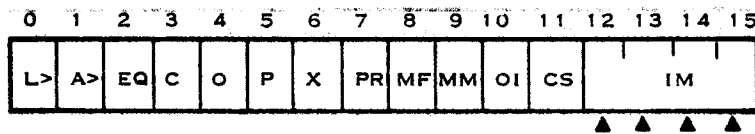
LABEL LIM R3

The four LSBs of workspace register three are loaded into the interrupt mask of the status register.

**Definition:** The four LSBs of <wa<sub>a</sub>> are loaded into the four LSBs of the status register (the interrupt mask). LIM is a privileged instruction.

When the privileged mode bit (bit 7 of ST register) is set to zero, the instruction executes normally. When the privileged mode bit is set to one and execution of an LIM instruction is attempted, an error interrupt occurs and the interrupt mask is not loaded.

**Status bits affected:** Interrupt mask



**Execution results:** The four LSBs of <wa> are loaded into the interrupt mask of the status register.

**Application notes:** The LIM loads the interrupt mask of the status register. Use the load interrupt mask instruction to initialize the interrupt mask for a particular level of interrupt to be accepted. For example, if workspace register four contains the value 3<sub>16</sub>, then the instruction

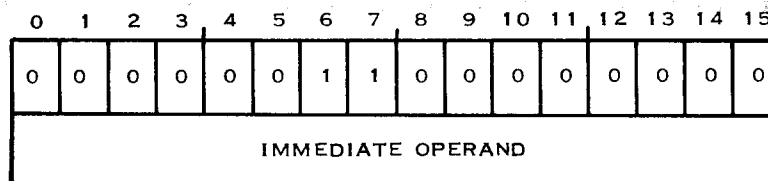
LABEL LIM R4

loads the contents of the four least significant bits of R4, in this case 3<sub>16</sub>, into the four least significant bits of the status register. The interrupt mask of the status register is set to level three and enables interrupts at levels zero, one, two, and three.

### 3.78 LOAD INTERRUPT MASK IMMEDIATE — LIM

**Opcode:** 0300

**Instruction Format:** 8



## LIMI, LMF

**Applicability:** Processor set A, privileged instruction

**Syntax definition:**

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

**Example:**

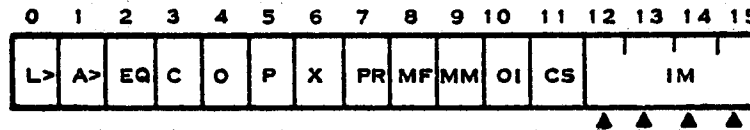
LABEL LIMI 3

Mask interrupt levels 4 through 15.

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

When the privileged mode bit (bit 7 of ST register) is set to zero, the instruction executes normally. When the privileged mode bit is set to one and execution of an LIMI instruction is attempted, an error interrupt occurs and the interrupt mask is not loaded.

**Status bits affected:** Interrupt mask



**Execution results:** Places the four least significant bits of <iop> in the interrupt mask (the four least significant bits of the ST register).

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

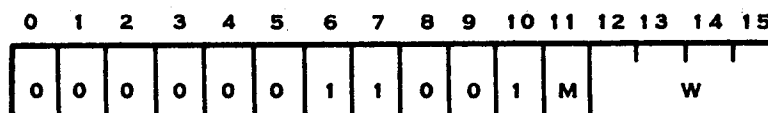
LIMI 3

sets the interrupt mask to level three and enables interrupts at levels zero, one, two, and three.

### 3.79 LOAD MEMORY MAP FILE — LMF

**Opcode:** 0320

**Instruction Format:** 10



# LMF

**Applicability:** Processor set A, privileged instruction, memory mapping option required

**Syntax definition:**

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

**Example:**

NMAP LMF R3,1

Load memory map file 1 with six words of memory, starting at the address specified in workspace register three.

**Definition:** Place the contents of a six-word area of memory at the address in the workspace register specified by <wa> into the memory map file designated by <m>. LMF is a privileged instruction.

When the privileged mode bit (bit 7 of ST register) is set to zero, the instruction executes normally. When the privileged mode bit is set to one and execution of an LMF instruction is attempted, an error interrupt occurs and the mask file is not loaded. memory mapping is explained in paragraph 3.74.1.

**Status bits affected:** None

**Execution results:** When the privileged mode bit (bit 7 of ST register) is set to zero, the contents of a six-word area at the address in <wa> are placed in map file <m>.

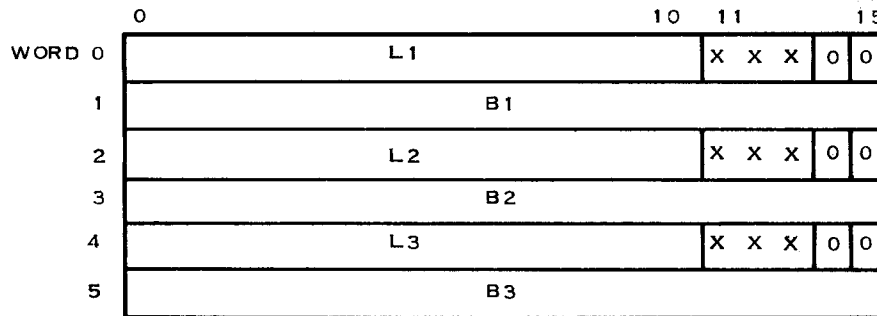
When the privileged mode bit is set to one, an error interrupt occurs.

### NOTE

Do not use the LMF instruction to change the (mapped) address of the workspace pointer. This causes the next context switch to dump cached workspace registers to the wrong memory location.



**Application notes:** Use the LMF instruction to load either map file zero or one (map file two is loaded by the long distance instructions). The map file is a set of six registers that maps the 32K word addresses of the AU into the desired 20-bit addresses of TILINE memory. The six-word area contains the following:



(A) 132204A

Words zero, two, and four contain values that are placed in limit registers L1, L2, and L3.

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

- The 11 most significant bits of each memory word (the limit) are placed in the 11-bit limit registers.
- Bits 11 through 13 may be any value (they are ignored).
- Bits 14 and 15 define the protection of the memory segment (990/12 only). The protection is defined as follows:

Mapping Limit Register Bits		Memory Protection in Segment
14(E)	15(W)	
0	0	No protection
0	1	Write protected
1	0	Execute protected
1	1	Execute and write protected

If one of the three protected options is chosen, status bit 9 must be set for the protection to be enabled. If status bit 9 equals zero, the protection is ignored.

- The one's complement of the limit is placed in the memory word (and in the map file).

The values in words one, three, and five are the 16 most significant bits of the bias register values and are placed in registers B1, B2, and B3.

## LMF, LR

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

- All addresses from zero through limit one are contiguous in memory.
- All addresses greater than limit one and less than or equal to limit two are contiguous in memory.
- All addresses greater than limit two and less than or equal to limit three are contiguous in memory.

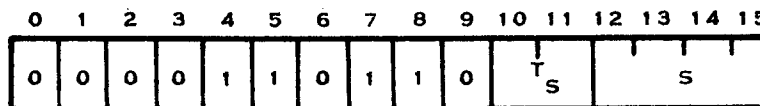
To load the map file, perform the following actions:

- Place the one's complement of the limit values in words zero, two, and four.
- Place the 16 most significant bits of the bias address for the lowest group in the second word.
- Place the 16 most significant bits of the bias address for the next group in the fourth word.
- Place the 16 most significant bits of the bias address for the highest group in the sixth word.

### 3.80 LOAD REAL — LR

Opcode: 0D80

Instruction Format: 6



Applicability: Processor set C

Syntax definition:

[<label>]b..LRb..<ga>b..[<comment>]

Example:

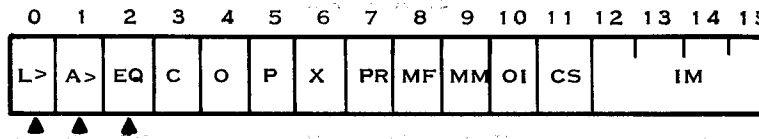
LABEL LR R6

Load workspace registers six and seven into the FPA (R0-R1)

Definition: The real number in the source address is stored in the FPA (R0-R1)

## LR, LREX

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



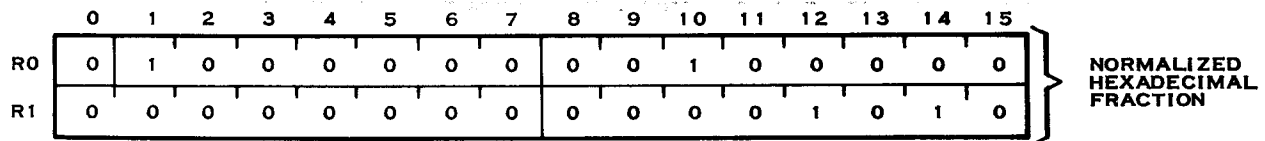
**Execution results:** (ga.) → FPA

**Application notes:** The result of the LR instruction is compared to zero and status register bits 0, 1, and 2 reflect the comparison. If T, is equal to three, the indicated register is incremented by four.

The following is an example of the load real instruction. The value contained in workspace register six and workspace register seven, after normalization, is .20000A<sub>16</sub>. The instruction

LABEL LR R6

stores the normalized fraction in the FPA (R0-R1), shown figuratively:

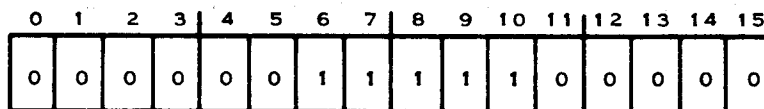


The logical greater than and arithmetic greater than bits of the status register are set; the equal bit is reset.

### 3.81 LOAD OR RESTART EXECUTION — LREX

**Opcode:** 03E0

**Instruction Format:** 7



**Applicability:** Processor set A, privileged instruction

**Syntax definition:**

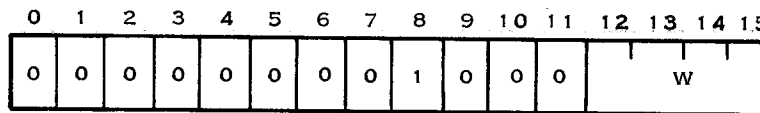
[<label>]b..LREXb..[<comment>]





# LST, LTO

**Instruction Format:** 18



**Applicability:** Processor set B

**Syntax definition:**

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

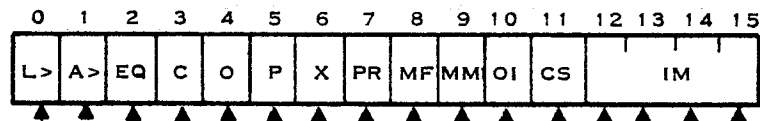
**Example:**

LABEL LST R3

The contents of workspace register three are loaded into the status register.

**Definition:** The contents of <wa<sub>d</sub>> are placed in the status register.

**Status bits affected:** All bits



If the privileged bit is set to one in the current status register, then only bits 0 through 5 and 10 are loaded into the status register.

**Execution results:** (wa<sub>d</sub>) → (ST)

**Application notes:** The LST instruction loads the status register.

The following is an example of the load status register instruction. Workspace register three contains the value 8700<sub>16</sub>, and the value of the status register is 8000<sub>16</sub>. The instruction

LABEL LST R3

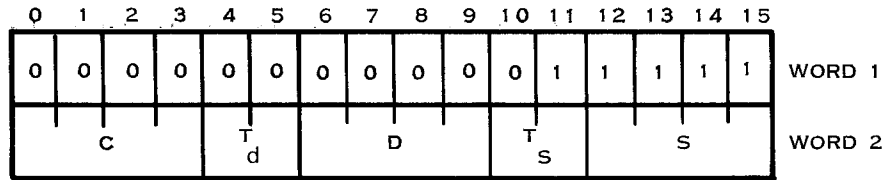
places the value of workspace register three into the status register, in this case 8700<sub>16</sub>. If the privileged bit of the status register had been set, the value of the status register, after execution of this instruction, would be 8400<sub>16</sub>.

### 3.83 LEFT TEST FOR ONES — LTO

**Opcode:** 001F

# LTO

Instruction Format: 11



Applicability: Processor set C

Syntax definition:

[<label>]b...LTOb...<ga<sub>s</sub>>,<ga<sub>d</sub>>[,<cnt>]b...[<comment>]

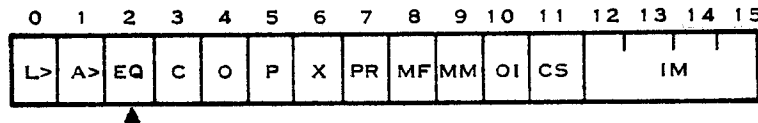
Example:

LABEL LTO @TST,@CNT,4

Locate the leftmost one in the four-byte value starting at location TST and add the bit's position to the word at location CNT.

**Definition:** The multibyte value at the source address is examined for the leftmost one. The bit position value is added to the word at the destination address. If the value at the source address is zero, nothing is added to the destination and status register bit 2 is set to a one; otherwise, status bit 2 is set to zero. The number of bytes of precision of the value at the source address is determined by the <cnt> field. If <cnt> equals zero or is not present, the count is taken from the four LSBs of workspace register zero. If the four LSBs of workspace register zero are zero, the count is 16. If the first bit tested is a one, the word at the destination address remains unchanged.

Status bits affected: Equal



Execution results: (ga<sub>d</sub>) + index to first '1' bit in (ga<sub>s</sub>) → (ga<sub>d</sub>)

Application notes: If T<sub>s</sub> is equal to three, the indicated register is incremented by the byte count.

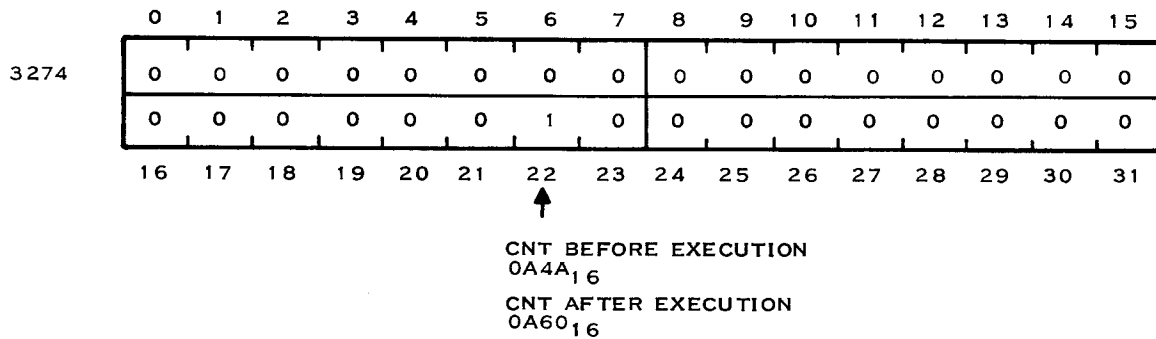
The following is an example of the left test for ones instruction. TST is a pointer to a four-byte string at memory address 3274<sub>16</sub> (the values contained in these bytes are 0<sub>16</sub>, 0<sub>16</sub>, 2<sub>16</sub>, 0<sub>16</sub>, respectively), and CNT contains the value 0A4A<sub>16</sub>. The instruction

LABEL LTO @TST,@CNT,4

checks for the leftmost one, in the four bytes beginning at location TST, and adds the bit position to the value in CNT. The result, in this example, is the value 0A60<sub>16</sub>, being placed in CNT.

## LTO, LWP

This example is shown figuratively:

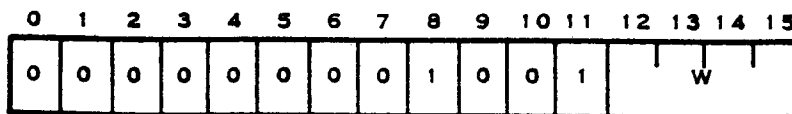


The equal bit of the status register is reset; the other bits of the status register are unaffected.

### 3.84 LOAD WORKSPACE POINTER REGISTER — LWP

Opcode: 0090

Instruction Format: 18



Applicability: Processor set B

Syntax definition:

[<label>]b...LWPb...<wad>b...[<comment>]

Example:

LABEL LWP R5

Place the contents of workspace register five in the workspace pointer register.

**Definition:** The contents of the workspace register <wa<sub>d</sub>> are placed in the workspace pointer register.

Status bits affected: None

Execution results: (wa<sub>d</sub>) → WP

## LWP, LWPI, MD

*Application notes:* The following is an example of the load workspace pointer register instruction. Workspace register five contains the value of  $220_{16}$ . The instruction

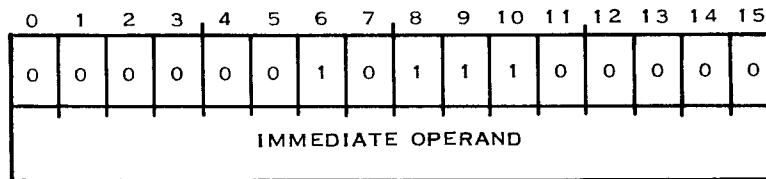
```
LABEL LWP R5
```

places the contents of workspace register five,  $220_{16}$ , into the workspace pointer register.

### 3.85 LOAD WORKSPACE POINTER IMMEDIATE — LWPI

*Opcode:* 02E0

*Instruction Format:* 8



*Applicability:* Processor set A

*Syntax definition:*

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

*Example:*

```
NEWWP LWPI >02F2
```

The workspace pointer is loaded with  $02F2_{16}$ .

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

*Status bits affected:* None

*Execution results:*  $iop \rightarrow (WP)$

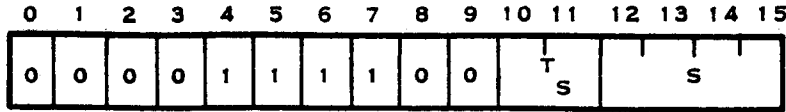
*Application notes:* Use the LWPI instruction to initialize or change the WP register to alter the workspace environment of the program module. The user should use either a BLWP or a LWPI instruction prior to the use of any workspace register in a program module.

### 3.86 MULTIPLY DOUBLE PRECISION REAL — MD

*Opcode:* 0F00

**MD**

**Instruction Format: 6**



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..MDb..<ga,>b..[<comment>]

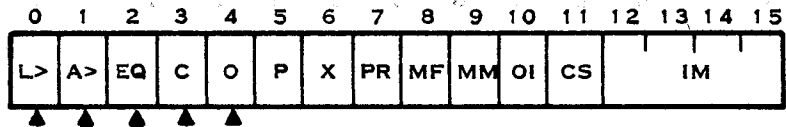
**Example:**

LABEL MD @WORD

Multiply the contents of the FPA by the contents of the word at location WORD and place the result in the FPA.

**Definition:** The contents of the FPA are multiplied by the word at the source address. The result is placed in the FPA (R0-R3).

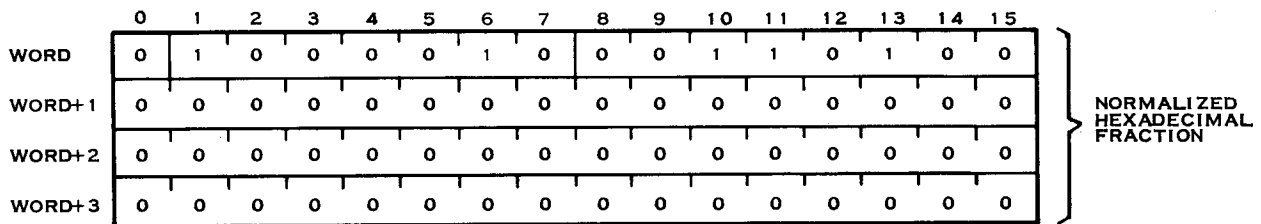
**Status bits affected:** Logical greater than, arithmetic greater than, equal, carry, and overflow



**Execution results:** FPA X (ga,) → FPA

**Application notes:** The results of the MD instruction are compared to zero and status register bits 0, 1, and 2 reflect the comparison. If status register bits 3 and 4 are set to one, overflow has occurred. If status register bits 3 and 4 are set to zero and one, respectively, underflow has occurred. If T<sub>s</sub> is equal to three, the indicated register is incremented by eight.

The following is an example of multiply double precision real. The value starting at location WORD contains, after normalization, the value 34<sub>16</sub>, as shown figuratively:



## MD, MOV

The double precision FPA (R0-R3), after normalization, contains the value  $26_{16}$ , shown figuratively:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R0	0	1	0	0	0	0	1	0	0	0	1	0	0	1	1	0
R1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

} NORMALIZED HEXADECIMAL FRACTION

The instruction

LABEL MD @WORD

multiplies the contents at location WORD by the contents of the FPA, and places the result,  $7B8_{16}$ , in the FPA, shown figuratively:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R0	0	1	0	0	0	0	1	1	0	1	1	1	1	0	1	1
R1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

} NORMALIZED HEXADECIMAL FRACTION

The logical greater than and the arithmetic greater than bits of the status register are set; the equal, carry, and overflow bits of the status register are reset.

Refer to Section 2 for a detailed description of normalization and double precision floating point instructions.

### 3.87 MOVE WORD — MOV

Opcode: C000

Instruction Format: 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	0	$T_d$				D			$T_s$			S	

Applicability: Processor set A

[<label>]b...MOVb...<ga>,<ga>b...[<comment>]

# MOV

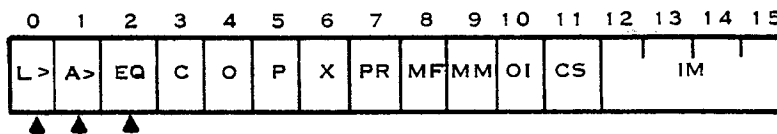
**Example:**

```
GET MOV @WORD,R2
```

Copy the contents of the memory word at location WORD and place the copy in workspace register two.

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

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



**Execution results:** (ga<sub>s</sub>) → (ga<sub>d</sub>)

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

- Memory-to-memory (nonregister).
- Load register (memory-to-register).
- Register-to-register.
- Store register (register-to-memory).

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

```
MOV     R7,R7
JNE     TEST
```

These instructions move register seven to itself and compare the contents of register seven to zero. If the contents are not equal to zero, the equal status bit is reset and control transfers to TEST. Another use of MOV is as follows: if workspace register nine contains 3416<sub>16</sub> and location ONES contains FFFF<sub>16</sub>, the instruction

```
MOV @ONES,R9
```

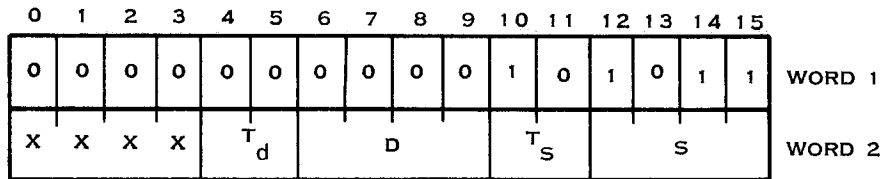
changes the contents of workspace register nine to FFFF<sub>16</sub>, while the contents of location ONES is not changed. For this example, the logical greater than status bit sets and the arithmetic greater than and equal status bits reset.

# MOVA

## 3.88 MOVE ADDRESS — MOVA

Opcode: 002B

Instruction Format: 19



Applicability: Processor set C

Syntax definition:

[<label>]b..MOVAb..<ga<sub>s</sub>>,<ga<sub>d</sub>>b..[<comment>]

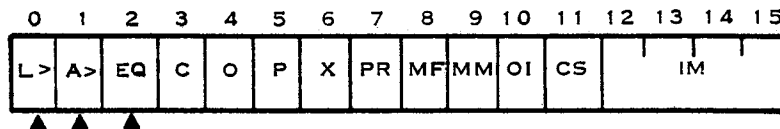
Example:

LABEL MOVA CHAR(R6),R7

Move the value of CHAR plus the contents of workspace register six into workspace register seven.

Definition: The effective address specified by the source address is moved to the destination address.

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



Execution results: ga<sub>s</sub> → (ga<sub>d</sub>)

Application notes: The result of the MOVA instruction is compared to zero and status register bits 0, 1, and 2 indicate the result. If T<sub>s</sub> or T<sub>d</sub> is equal to three, the indicated register is incremented by two.

The following is an example of the move address instruction. TABLE is at location 200<sub>16</sub>, and workspace register two contains the value 6<sub>16</sub>. The instruction

LABEL MOVA TABLE(R2),R7

places the address 206<sub>16</sub> in workspace register seven.

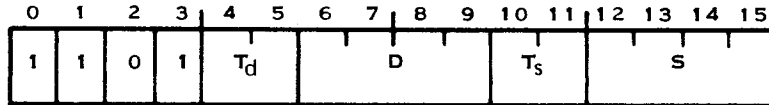


# MOVB

## 3.89 MOVE BYTE — MOVB

Opcode: D000

Instruction Format: 1



Applicability: Processor set A

Syntax definition:

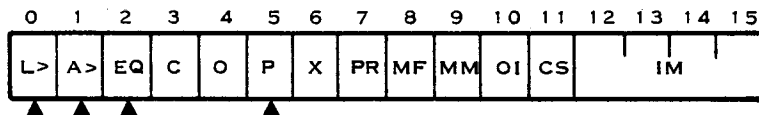
[<label>]b...MOVBb...<ga<sub>s</sub>>,<ga<sub>d</sub>>b...[<comment>]

Example:

NEXT MOVB R2,@BUFFER(R3)      Move the most significant byte of workspace register two to the byte at the location addressed by BUFFER plus workspace register three.

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

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



Execution results: (ga<sub>s</sub>) → (ga<sub>d</sub>)

**Application notes:** MOVB is used to move bytes in the same combinations as the MOV instruction moves words. For example, if memory location TEMP contains a value of 2016<sub>16</sub>, and if workspace register three contains 542B<sub>16</sub>, then the instruction

MOVB @TEMP,R3

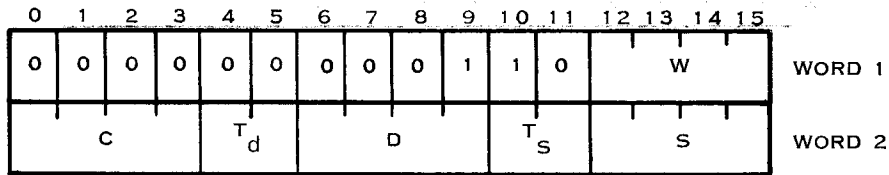
changes the contents of workspace register three to 202B<sub>16</sub>. The logical greater than, arithmetic greater than, and odd parity status bits set while the equal status bit resets.

# MOVS

## 3.90 MOVE STRING — MOVS

Opcode: 0060

Instruction Format: 12



Applicability: Processor set C

Syntax definition:

[<label>]b...MOVSb...<ga<sub>1</sub>>,<ga<sub>2</sub>>,<cnt>][,<ckpt>]b...[<comment>]

Trailing commas may be omitted from the operands. The checkpoint register may be omitted from the instruction if a default has been specified with the CKPT assembler directive.

Example:

```
LABEL MOVS @INPUT,@MOVE,10,R1
```

Move ten bytes beginning at location INPUT to the ten bytes beginning at location MOVE. The checkpoint register is R1.

**Definition:** The byte string, starting at the location specified by the source address, is moved to the destination address. The byte at the source address is moved first. The string length may be specified in the <cnt> field register zero, or as a tagged string (if <cnt> = 0 and R0 = FFFF).

If the string to be moved is 15 bytes or less, the length to be moved may be specified in <cnt>. If <cnt> is zero or is omitted, R0 is assumed to hold the count. If R0 is >FFFF and <cnt> is zero or missing, the string is assumed to be a tagged string and the count is taken from the first byte of the string. The tag byte is moved first.

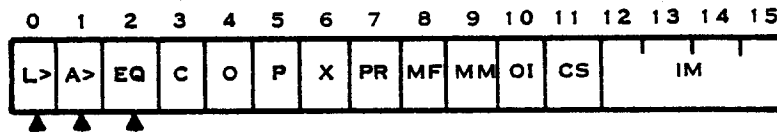
The string is compared to zero (as a signed two's complement value), and bits 0, 1, and 2 of the status register are set accordingly. If the string is tagged, the tag byte is moved, but it is not used in the comparison.

If the length of the string is 16 bytes or more, the checkpoint register <ckpt> is used for interrupts. If an interrupt occurs during execution, checkpoint data is stored in the checkpoint register, the program counter is decremented, and the interrupt trap is taken. After the interrupt is serviced, execution continues from where it stopped. Upon completion of the instruction, the checkpoint register is set to -1.

## MOVS

The checkpoint register value plus one is used as an initial index into the string. To access the beginning of the string, the checkpoint register must be set to -1 before the MOVS instruction is first executed.

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



**Execution results:**  $(ga_1) \rightarrow (ga_2)$  The byte at  $ga_1$  is moved first.

The result of the MOVS instruction is compared to zero and status register bits 0, 1, and 2 reflect the comparison.

**Application notes:** If  $T_1$  and/or  $T_2$  is equal to three, the indicated register is incremented by the string length.

If the checkpoint register is not initially equal to -1, status register bits 0 through 2 are assumed to reflect the correct status for the bits that were skipped.

If a string has a length of zero, no data is moved and status bits 0 through 2 are set to zero. If a tagged string has a length of one, only the tag byte is moved and status bits 0 through 2 are set to zero.

For example, if location ZEROS contains five bytes of zeros and workspace registers R0, R6, R7, and R8 contain >FFFF, >406, >274E, and >764C respectively, then the instruction

```
MOVS @ZEROS,R6,5,R0
```

results in the following register contents:

```
R6 = 0
R7 = 0
R8 = >4C
R0 = FFFF
```

The status register bits affected are the following:

```
logically greater than    = 0
arithmetically greater than = 0
equal                    = 1
```

# MOVS

In the next example, workspace register zero contains the string length, and workspace register one is used as the checkpoint register. Assume that location ALPABT contains the 26-byte long character string consisting of the alphabet. Assume that location BUFFER is completely filled with Xs. The sequence of code

```
LI      R0,26
SETO   R1
MOVS   @ALPABT,@BUFFER,R1
```

results in location BUFFER containing the string ABCDEFGHIJKLMNOPQRSTUVWXYZ. The status bits affected are as follows:

- logically greater than = 1
- arithmetically greater than = 1
- equal = 0

If the MOVS instruction is executed with the value of the checkpoint register something other than -1, the current value of the status register is considered as a partial status result for the instruction. If the checkpoint register is set to a value greater than -1, the bytes moved are taken from inside the string, starting with the byte at <ckpt> + 1.

The MOVS instruction may be used to move overlapping strings where the source address is higher (numerically greater) than the destination address. For example, consider an alphabetic table where a deletion is desired. The instructions to delete the second T in the accompanying figure are as follows:

```
SETO R1
MOVS @TABLE + 3,@TABLE + 2,3,R1
```

TABLE	TABLE
before deletion	after deletion
A	A
R	R
TABLE + 2 T	T
TABLE + 3 T	V
V	Z
Z	Z

## MOVS, MPY

Note that the 'Z' at TABLE + 5 is not affected. MOVS should not be used on overlapping strings when the source address is lower than the destination address. MVSR should be used in this case. If MVSR is substituted for MOVS in the above example, the 'Z' at TABLE + 5 is moved to TABLE + 4 and used in subsequent moves, replicating Zs for the length of the move.

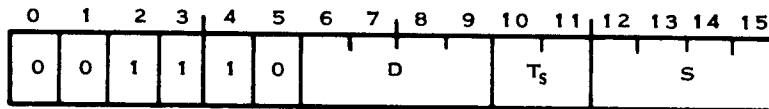
Another aspect of the MOVS instruction is the way in which the use of tagged strings affects the setting of the status register. The following cases demonstrate status register settings for several tagged strings.

SOURCE STRING (BYTES)	STATUS (L>,A>,E)
>FF,0,0,...0	0,0,1
3,>FF,>FF,>FF	1,0,0
4,0,0,1,0	1,1,0
1	0,0,0

### 3.91 MULTIPLY — MPY

Opcode: 3800

Instruction Format: 9



Applicability: Processor set A

Syntax definition:

[<label>]b..MPYb..<ga>,<wa>b..[<comment>]

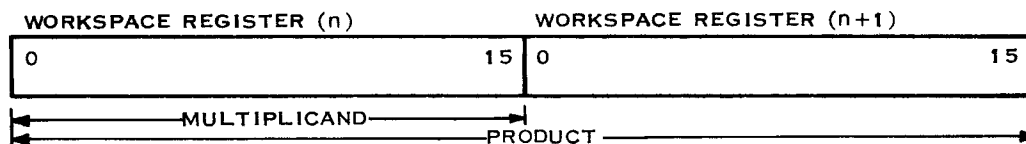
Example:

LABEL MPY @ADDR,R3

Multiply the contents of workspace register three by the contents of the word at location ADDR and place the result, right-justified, in the 32-bits of workspace registers three and four.

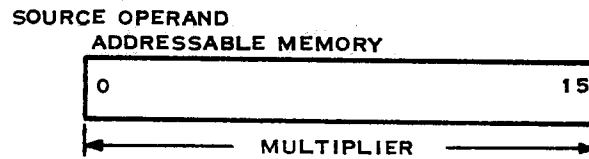
**Definition:** Multiply the first word in the destination operand (a consecutive two-word area in workspace) by a copy of the source operand and replace the two-word destination operand with the result. The multiplication operation may be graphically represented as follows:

Destination operand workspace registers



## MPY, MPYS

Source operand



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

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

*Status bits affected:* None

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

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

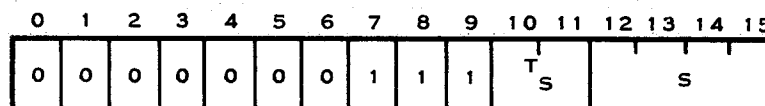
MPY @NEW,R5

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

### 3.92 MULTIPLY SIGNED — MPYS

*Opcode:* 01C0

*Instruction Format:* 6



# MPYS

**Applicability:** Processor set B

**Syntax definition:**

[<label>]b..MPYSb..<ga>b..[<comment>]

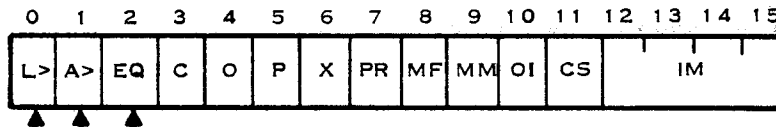
**Example:**

LABEL MPYS R3

Multiply the contents of workspace register zero by the contents of workspace register three and place the results in workspace register zero and one.

**Definition:** The signed, two's complement integer in workspace register zero is multiplied by the signed, two's complement integer at the source address. The product, a signed, two's complement double-length integer, is deposited in workspace registers zero (most-significant half) and one (least significant half).

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



**Execution results:** (R0) X (ga) → (R0 and R1)

**Application notes:** The MPYS instruction allows for multiplication of signed numbers.

The result of the MPYS instruction is compared to zero, and status register bits 0, 1, and 2 reflect the comparison. If T, is equal to three, the indicated register is incremented by two.

The following is an example of the multiply signed instruction. Workspace register zero contains the value FFCC<sub>16</sub>, and workspace register three contains the value FFDA<sub>16</sub>. The instruction

LABEL MPYS R3

multiplies the signed, two's complement value of workspace register zero by the signed, two's complement value of workspace register three, and places the double-length result in workspace register zero and workspace register one. The product is, in this example, R0 = 0 and R1 = 7B8<sub>16</sub>.

The sign of the result follows normal algebraic rules as follows:

- positive X positive = positive
- positive X negative = negative
- negative X negative = positive

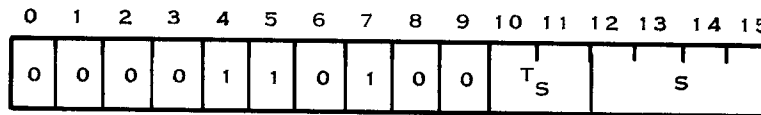
## MPYS, MR

The logical greater than and the arithmetic greater than bits of the status register are set; the equal bit of the status register is reset.

### 3.93 MULTIPLY REAL — MR

Opcode: 0D00

Instruction Format: 6



Applicability: Processor set C

Syntax definition:

[<label>]b...MRb..<ga.>b..[<comment>]

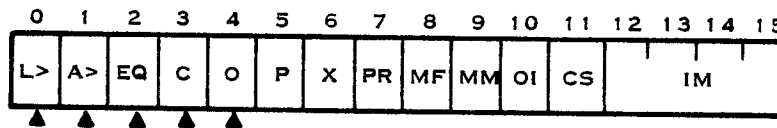
Example:

LABEL MR R5

The contents of the FPA are multiplied by the contents of workspace registers five and six. The result is placed in the FPA.

**Definition:** The FPA (R0,R1) is multiplied by the contents of the two words at the source address, and the result is placed in the FPA (R0,R1).

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



Execution results: FPA X (ga.) → FPA

**Application notes:** The results of the MR instructions are compared to zero and status register bits 0, 1, and 2 reflect the comparison. If status register bits 3 and 4 are set to zero and one, respectively, underflow has occurred. If they are set to ones, overflow has occurred. If T<sub>s</sub> is equal to three, the indicated register is incremented by four.





## MVSK

**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..MVSKb..<ga>,<ga>,<cnt>[,<ckpt>]b..[<comment>]

Trailing commas in the operand list may be omitted. The checkpoint register may be omitted from the instruction if a default has been specified with the CKPT assembler directive.

**Example:**

```
LABEL MVSK @STCK,@INPUT,7,R6
```

Move seven bytes from the top of the stack to the seven bytes beginning at location INPUT. The stack parameters are a three-word block at location STCK. Workspace register six is the checkpoint register.

**Definition:** The source operand is a stack descriptor. The destination is a byte string. The byte string at the top of the stack is moved to the location defined by the destination address. The string length may be specified in the <cnt> field, R0, or as a tagged string (if <cnt> = 0 and R0 = FFFF).

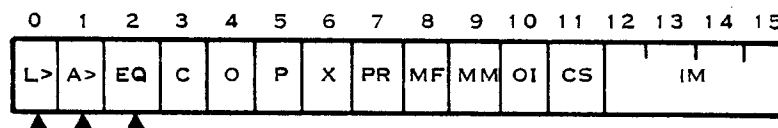
If the top of stack is specified in a workspace register (T<sub>s</sub> is equal to zero), no limit checking is done. Otherwise, before the string is moved, limit checking is done. Limit checking consists of verifying that the string is entirely contained within the stack. If this relationship does not hold, the stack overflow bit in the error interrupt status register (see paragraph 4.5.1.2) is set and a level two interrupt is taken. If the length of the string is 16 bytes or more, the checkpoint register is used for interrupts. If an interrupt occurs during execution, checkpoint data is stored in the checkpoint register. After the interrupt is serviced, execution continues from the point at which it stopped. Upon completion of the instruction, the checkpoint register is set to -1.

The checkpoint register value + 1 is used as an initial index into string. To access the first byte (lowest address) in the string, the checkpoint register must be set to -1 before the instruction is executed.

If a string has a length of zero, no data is moved and status bits 0 through 2 are set to zero. If a tagged string has a length of one, only the tag byte is moved, and status bits 0 through 2 are set to zero.

Stacks are described further in Section 2.

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

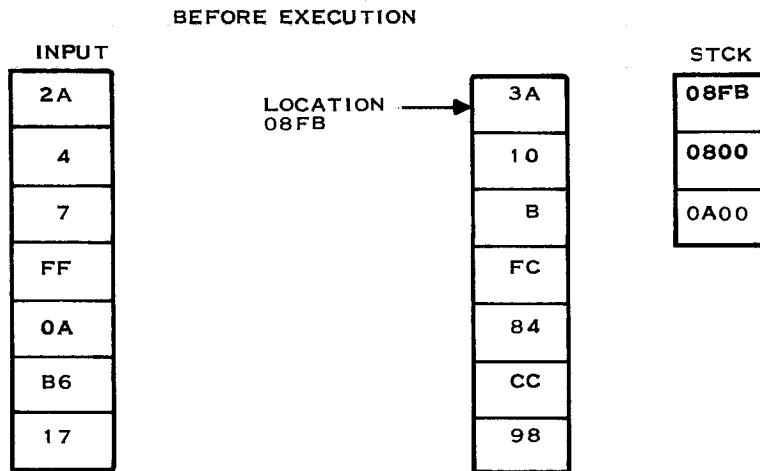


# MVSK

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

Application notes: If  $T_d$  is equal to three, the indicated register is incremented by the byte count. If  $T_s$  is equal to three, the indicated register is incremented by six.

The following is an example of the move string from stack instruction. Locations INPUT and STCK contain the following data:

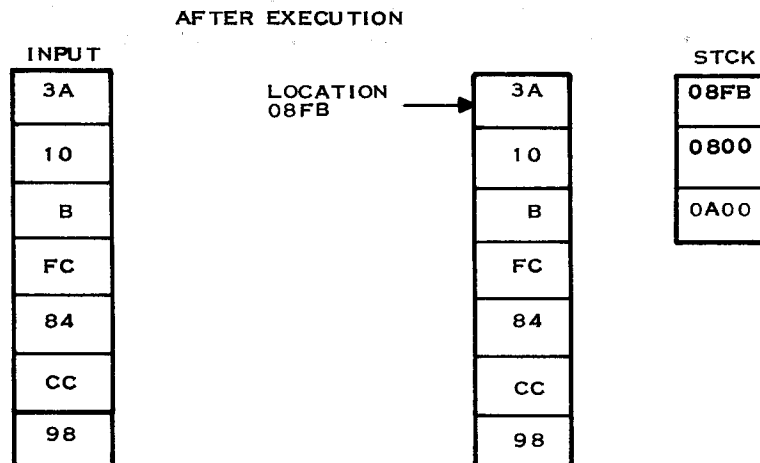


The instructions

```

SETO    R6
LABEL  MVSK    @STCK,@INPUT,7,R6
    
```

move seven bytes starting at the location indicated by location STCK into seven bytes starting at location INPUT, as shown figuratively:

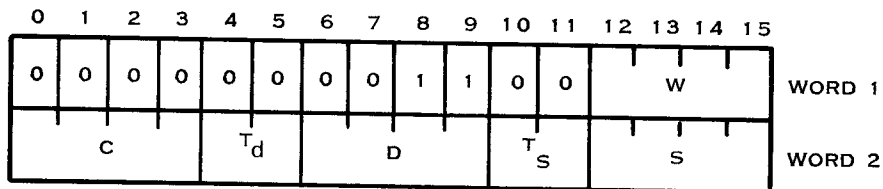


# MVSR

## 3.95 MOVE STRING REVERSE — MVSR

Opcode: 00C0

Instruction Format: 12



Applicability: Processor set C

Syntax definition:

[<label>]b..MVSRb..<ga>,<ga>,<cnt>[,<ckpt>]b..[<comment>]

Trailing commas in the operand list may be omitted. The checkpoint register may be omitted from the instruction if a default has been specified with the CKPT assembler directive.

Example:

LABEL MVSR @INPUT,@MOVE,,R1 Move the byte string beginning at location INPUT to the bytes beginning at location MOVE. The string length is specified in workspace register zero. The last byte of string INPUT is moved first, into the last byte in MOVE.

**Definition:** The byte string starting at the source address is moved to the destination address. The last byte of the source string is moved first. The number of bytes to be moved is specified by the <cnt> field. If the checkpoint register is set to a value greater than -1, the bytes moved are taken from inside the string, starting with the byte at <ckpt> + 1 bytes from the end of the string.

The string length may be specified in the <cnt> field, register zero, or as a tagged string (if <cnt> = 0 and R0 = >FFFF). If the string length is zero, no data is moved.

The string is compared to zero (as a signed two's complement value), and bits 0, 1, and 2 of the status register are set accordingly. If the string is tagged, the tag byte is not used in the comparison.

If the length of the string is 16 bytes or more, the checkpoint register is used for interrupts. If an interrupt occurs during execution, checkpoint data is stored in the checkpoint register. After the interrupt is serviced, execution continues from where it stopped. Upon completion of the instruction, the checkpoint register is set to -1.

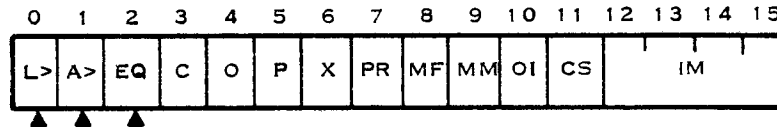
## MVSR

The checkpoint register value + 1 is used as an initial index into the string (from the end of the string). To access the last byte (highest address) in the string, the checkpoint register must be set to -1 before the MVSR instruction is executed.

If the checkpoint register is not initially equal to -1, the value of status bit 2 (EQ) is assumed to have the correct value for the bytes that were skipped.

If a string has a length of zero, no data is moved, and status bits 0 through 2 are set to zero. If a tagged string has a length of one, only the tag byte is moved, and status bits 0 through 2 are set to zero.

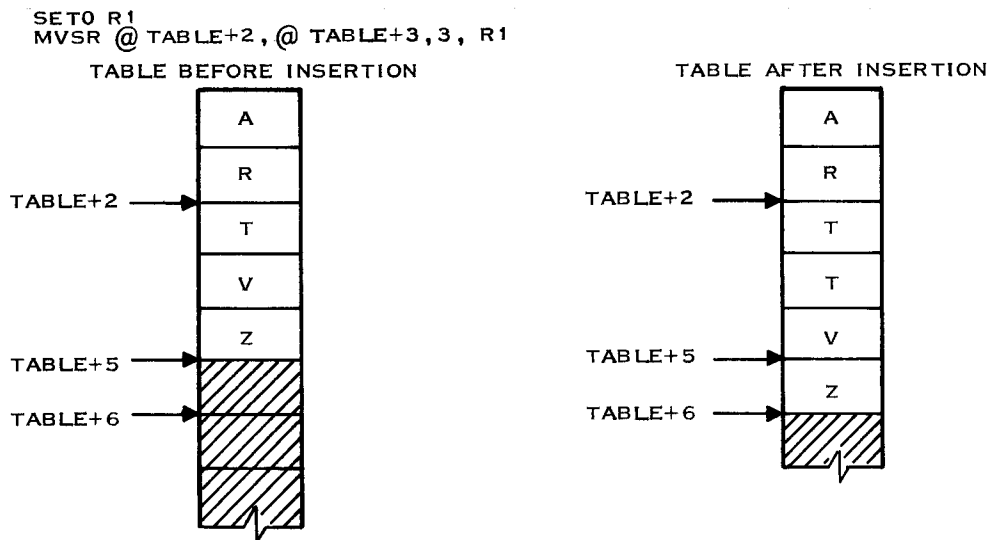
**Status bits affected:** Logical greater than, arithmetic greater than, and equal



**Execution results:**  $(ga_s) \rightarrow (ga_d)$  The last byte of  $(ga_s)$  is moved to the last byte of  $(ga_d)$  first.

**Application notes:** The result of the MVSR instruction is compared to zero, and status register bits 0, 1, and 2 reflect the comparison.

The MVSR instruction may be used to move overlapping strings where the source address is lower than the destination address. An example would be an alphabetical table where an insert is desired. The instructions to make a hole for a T to be inserted in the accompanying figure are as follows:



If the MOVS instruction is used, the first byte moved puts a T on top of the V in the table. Subsequent bytes being moved would also be T.

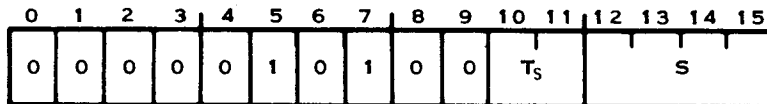
## MVSR, NEG

MVSR should not be used on overlapping strings when the source address is greater than the destination address. MOVS should be used in this case.

### 3.96 NEGATE — NEG

Opcode: 0500

Instruction Format: 6



Applicability: Processor set A

Syntax definition:

[<label>]b..NEGb..<ga,>b..[<comment>]

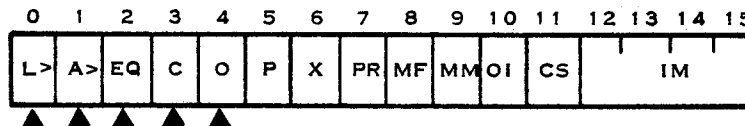
Example:

LABEL NEG R2

Replace the contents of workspace register two with its additive inverse.

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

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



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

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

NEG R5

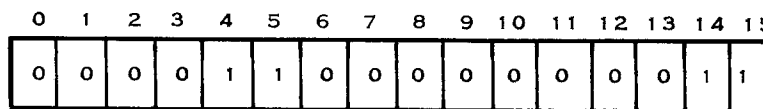
## NEG, NEGD

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

### 3.97 NEGATE DOUBLE PRECISION REAL — NEGD

*Opcode:* 0C03

*Instruction Format:* 7



*Applicability:* Processor set C

*Syntax definition:*

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

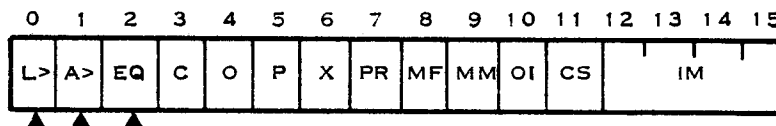
*Example:*

LABEL NEGDb

Negate the double precision number in the FPA and place the results in the FPA.

*Definition:* Negate the double precision number in the FPA (R0-R3) and place the results in the FPA (R0-R3).

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



*Execution results:* -FPA → FPA

*Application notes:* The results of the NEGDb instruction are compared to zero and status register bits 0, 1, and 2 reflect the comparison.

## NEGD, NEGR

The following is an example of the negate double precision real instruction. The value in the double precision FPA (R0-R3), after normalization, is .BB13B13B13B13B<sub>16</sub>, shown figuratively:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R0	0	1	0	0	0	0	0	0	1	0	1	1	1	0	1	1
R1	0	0	0	1	0	0	1	1	1	0	1	1	0	0	0	1
R2	0	0	1	1	1	0	1	1	0	0	0	1	0	0	1	1
R3	1	0	1	1	0	0	0	1	0	0	1	1	1	0	1	1

} NORMALIZED  
HEXADECIMAL  
FRACTION

The instruction

LABEL NEGD

negates (additive inverse) the double precision value in the FPA and places the result, -.BB13B13B13B13B<sub>16</sub>, in the FPA as shown figuratively:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R0	1	1	0	0	0	0	0	0	1	0	1	1	1	0	1	1
R1	0	0	0	1	0	0	1	1	1	0	1	1	0	0	0	1
R2	0	0	1	1	1	0	1	1	0	0	0	1	0	0	1	1
R3	1	0	1	1	0	0	0	1	0	0	1	1	1	0	1	1

} NORMALIZED  
HEXADECIMAL  
FRACTION

The logical greater than bit of the status register is set; the arithmetic greater than and equal bits of the status register are reset.

### 3.98 NEGATE REAL — NEGR

Opcode: 0C02

Instruction Format: 7

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0

Applicability: Processor set C

Syntax definition:

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



# NEGR

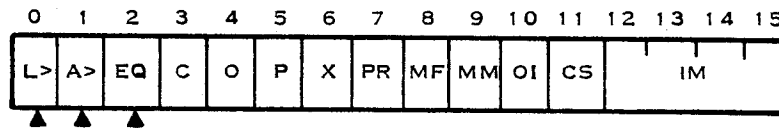
**Example:**

LABEL NEGR

Negate the real number in the FPA and place the result in the FPA.

**Definition:** The real number in the FPA (R0,R1) is negated, and the result is placed in the FPA (R0,R1).

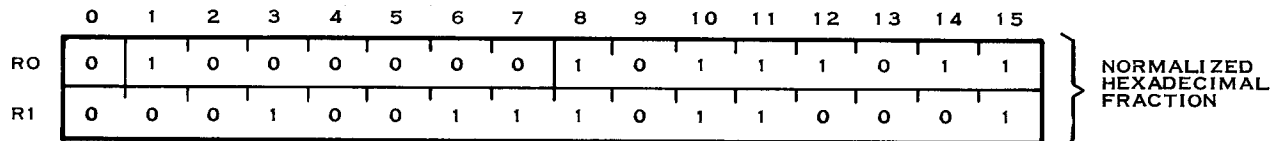
**Status bits affected:** Logical greater than, arithmetic greater than, and equal.



**Execution results:**  $-(FPA) \rightarrow (FPA)$

**Application notes:** The results of the NEGR instruction are compared to zero, and status register bits 0, 1, and 2 reflect the results.

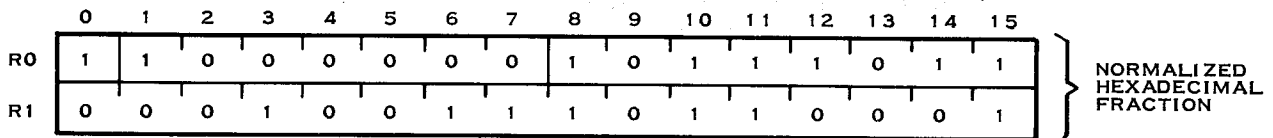
The following is an example of the negate real instruction. The value in the single precision FPA (R0-R1), after normalization, is  $.BB13B1_{16}$ , shown figuratively:



The instruction

LABEL NEGR

negates (additive inverse) the single precision value in the FPA, and places the result,  $-.BB13B1_{16}$ , in the FPA, as shown figuratively:



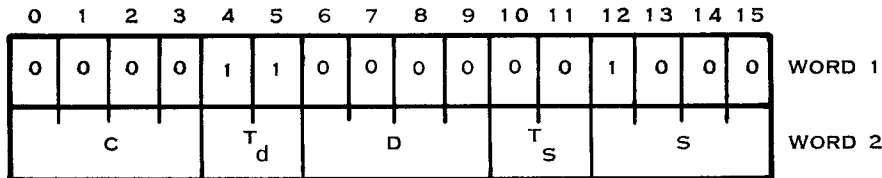
The logical greater than bit of the status register is set, and the arithmetic greater than and equal bits of the status register are reset.

# NRM

## 3.99 NORMALIZE — NRM

Opcode: 0C08

Instruction Format: 11



Applicability: Processor set C

Syntax definition:

[<label>]b..NRMb..<ga<sub>s</sub>>,<ga<sub>d</sub>>[,<cnt>]b..[<comment>]

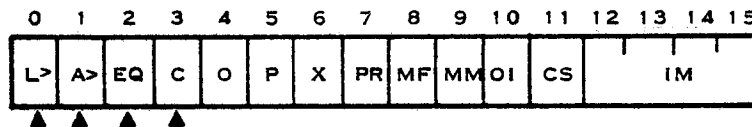
Example:

LABEL NRM \*R4,R5,6

Shift the two's complement of the six-byte-long string addressed by the contents of workspace register four to the left until the two leftmost bits are different. Add the number of positions shifted to workspace register five.

**Definition:** The multibyte two's complement integer at the source address is shifted left until the two leftmost bits are different. The <cnt> field specifies the byte count. If zero or not present, the count is taken from the four LSBs of workspace register zero. If the four LSBs are zero, the count is 16. The bits at the LSB end are filled with zeros. The number of positions shifted is added to the destination address contents. If the integer is zero, no shifting is performed.

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



Execution results: Number of sign extension bits shifted equals N

$$(ga_s) * 2^N \rightarrow (ga_s)$$

$$(ga_d) + N \rightarrow (ga_d)$$

**Application notes:** If T<sub>d</sub> is equal to three, the indicated register is incremented by two. If T<sub>s</sub> is equal to three, the indicated register is incremented by the byte count.

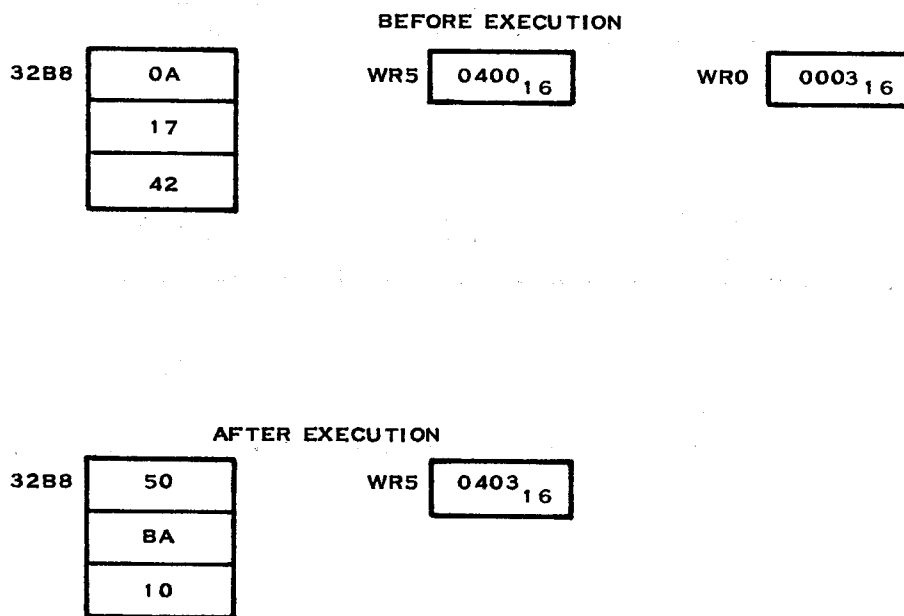
## NRM, ORI

The result of the NRM instruction is compared to zero, and status register bits 0, 1, and 2 reflect the comparison. Status register bit 3 is a copy of the resultant sign bit.

The following is an example of the normalize instruction. Workspace register four points to a three-byte string at location 32B8<sub>16</sub>, workspace register five contains the value 0440<sub>16</sub>, and workspace register zero contains the value 3<sub>16</sub>. The instruction

**LABEL NRM \*R4,R5**

shifts the multibyte two's complement integer, beginning at location 32B8<sub>16</sub> until the two leftmost bits are different, and adds the number of bit positions shifted to register five. This example is shown figuratively:

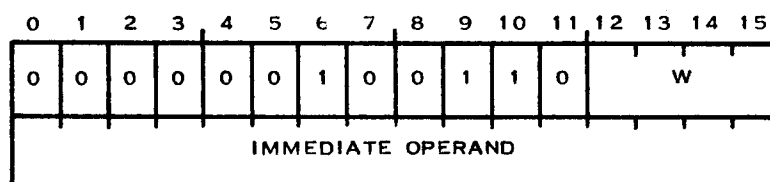


The logical greater than and arithmetic greater than bits of the status register are set, and the equal and carry bits are reset.

### 3.100 OR IMMEDIATE — ORI

*Opcode:* 0260

*Instruction Format:* 8



# ORI

**Applicability:** Processor set A

**Syntax definition:**

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

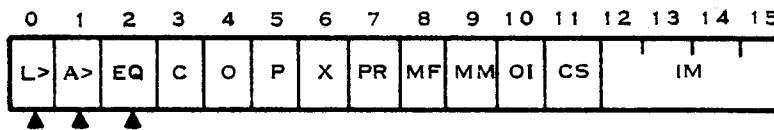
**Example:**

LABEL ORI R3,>F000

Perform the logical OR of workspace register three and the immediate value F000<sub>16</sub>.

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

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



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

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

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

For example, if workspace register five contains D2AB<sub>16</sub>, then the instruction

ORI R5,>6D03

## ORI, ORM

results in workspace register 5 changing to FFAB<sub>16</sub>. This OR operation on a bit-by-bit basis is as follows:

```

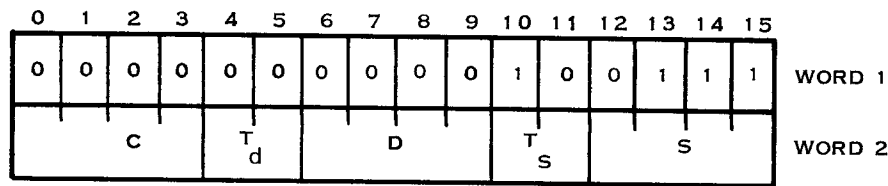
0 1 1 0 1 1 0 1 0 0 0 0 0 1 1 (Immediate operand)
1 1 0 1 0 0 1 0 1 0 1 0 1 0 1 1 (Workspace register 5)
-----
1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 (Workspace register 5 result)
    
```

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

### 3.101 OR MULTIPLE PRECISION — ORM

Opcode: 0027

Instruction Format: 11



Applicability: Processor set C

Syntax definition:

[<label>]b..ORMb..<ga, >,<ga, >,<cnt>]b..[<comment>]

**Example:**

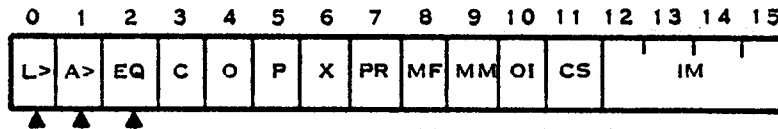
LABEL ORM \*R4,@TAB(R2),4

Perform the logical OR of the four bytes beginning at the address in workspace register four and the four bytes beginning at the location TAB plus workspace register two. The result is placed in the four bytes at location TAB plus workspace register two.

**Definition:** A bit-by-bit OR operation is performed between the bits of the multibyte two's complement integer at the source address and the corresponding bits of the multibyte two's complement integer at the destination address. The result is placed at the destination address. The <cnt> is the number of bytes of precision of the integer. If <cnt> equals zero or is not present, the count is taken from the four LSBs of workspace register zero. If the four LSBs of workspace register zero are zero, the count is 16.

# ORM

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



**Execution results:**  $(ga_n) \text{ OR } (ga_d) \rightarrow (ga_d)$

**Application notes:** The result of the ORM-instruction is compared to zero and the status register bits 0, 1, and 2 indicate the result of the comparison. If  $T_n$  and/or  $T_d$  is equal to three, the indicated register is incremented by the byte count.

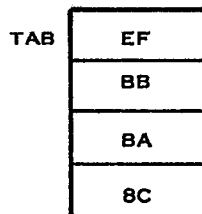
The following is an example of the OR multiple precision instruction. Workspace register four contains the value  $37A4_{16}$ , which points to a four-byte string with the values  $4533_{16}$  and  $328C_{16}$ , and TAB is the address of a four-byte string with the values  $AAAA_{16}$  and  $AA00_{16}$  as shown figuratively:



The instruction

```
LABEL ORM *R4,@TAB,4
```

performs a bit-by-bit OR operation between the four bytes starting at  $37A4_{16}$  and the four bytes starting at location TAB, placing the results in TAB. The results of this instruction, after execution, are shown figuratively:



The logical greater than bit of the status register is set, and the arithmetic greater than and equal bits are reset.

## ORM, POPS

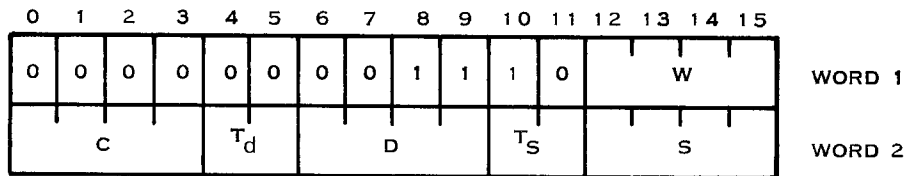
This truth table describes the OR operation between two bits:

Source Bit	Corresponding Destination Bit	Corresponding Resulting Bit
0	0	0
1	0	1
0	1	1
1	1	1

### 3.102 POP STRING FROM STACK — POPS

Opcode: 00E0

Instruction Format: 12



Applicability: Processor set C

Syntax definition:

[<label>]b..POPSb..<ga, ><ga, >,<cnt>],[<ckpt>]b..[<comment>]

Trailing commas in the operand list may be omitted. The checkpoint register may be omitted from the instruction if a default has been specified with the CKPT directive.

Example:

```
LABEL POPS @STACK,@OUTPUT,10,R6
```

Pop ten bytes from the stack and place them starting at location OUTPUT. The stack parameters are a three-word block at location STACK. Workspace register six is the checkpoint register.

**Definition:** The source operand is a stack descriptor. The destination operand is a byte string. The byte string at the top of the stack is moved to the location defined by the destination and the TOS value is incremented by the string length. The string length may be specified in the <cnt> field, register zero, or as a tagged string (if <cnt> = zero and R0 = >FFFF). String length is described further in Section 2.

If the TOS is specified as a workspace register (T<sub>s</sub> is equal to zero), no limit checking is done.

## POPS

Before the string is moved, limit checking is done. This involves verifying that the following relationship is true:

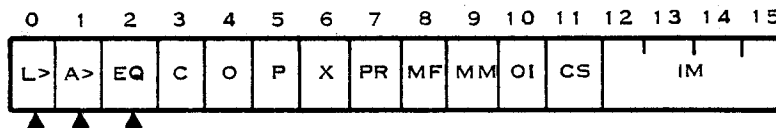
$$\text{stack limit} \leq \text{old TOS} \leq \text{new TOS} \leq \text{bottom of stack}$$

If this relationship does not hold, the stack overflow bit in the error interrupt status register (see paragraph 4.5.1.2) is set and a level two interrupt is taken. Stacks are described further in Section 2.

If the length of the string is 16 bytes or more, the checkpoint register is used for interrupts. If an interrupt occurs during execution, checkpoint data is stored in the checkpoint register. After the interrupt is serviced, execution continues from where it stopped. Upon completion of the instruction, the checkpoint register is set to -1.

The checkpoint register value plus one is used as an initial index into the string. To access the first byte (lowest address) in the string, the checkpoint register must be set to -1 before the instruction is executed.

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



*Execution results:* Refer to Figure 3-3.  $((ga_s)) \rightarrow (ga_d)$

*Application notes:* If  $T_d$  is equal to three, the indicated register is incremented by the string length. If  $T_s$  is equal to three, its indicated register is incremented by six (the length of a stack descriptor).

The following paragraphs describe an example of the pop string from stack instruction.

If the stack boundaries, upper and lower, are in the range from  $235A_{16}$  to  $2373_{16}$ , the TOS points to address  $2362_{16}$ , and OUTPUT points to the address  $4478_{16}$ , then the instructions

```
SETO    R 6
```

```
LABEL  POPS @STACK, @OUTPUT, 10, R6
```

move 10 bytes, beginning at the TOS, to the byte string at location OUTPUT. The TOS descriptor value changes to  $236C_{16}$ . This example is shown figuratively:



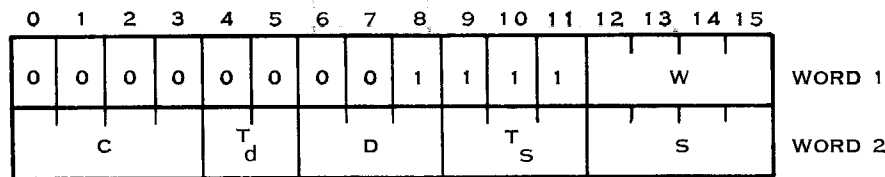


# PSHS

## 3.103 PUSH STRING TO STACK — PSHS

Opcode: 00F0

Instruction Format: 12



Applicability: Processor set C

Syntax definition:

[<label>]b..PSHSb..<ga, >,<ga<sub>a</sub> >,<cnt>][,<ckpt>]b..<comment>

Trailing commas in the operand list may be omitted. The checkpoint register may be omitted from the instruction if a default has been specified with the CKPT directive.

Example:

**LABEL PSHS @STRING,@STACK,14,R6**      Push 14 bytes starting at location STRING onto the stack. The stack parameters are a three-word block at location STACK. Workspace register six is the checkpoint register.

**Definition:** The byte string starting at the source address is moved to the stack specified by the stack descriptor at the destination address and the TOS is decremented by the string length. The last byte of the string is moved first, as in the MVSR instruction. The stack is managed as a byte stack (that is, the top of stack address (TOS) may be even or odd).

The stack descriptor is specified by the word(s) at the destination address. If TOS is specified in a workspace register (T<sub>a</sub> equals zero), limit checking is not done. Otherwise, limit checking is done before the string is moved. This involves verifying that the following relationship is true.

$$\text{stack limit} \leq \text{new TOS} \leq \text{old TOS} \leq \text{bottom of stack}$$

If this relationship does not hold, the stack overflow bit in the error interrupt status register (see paragraph 4.5.1.2) is set and a level two interrupt is taken. Stacks are described further in Section 2.

If the length of the string is 16 bytes or more, the checkpoint register is used for interrupts. If an interrupt occurs during execution, checkpoint data is stored in the checkpoint register. After the interrupt is serviced, execution continues from where it stopped. Upon completion of the instruction, the checkpoint register is set to -1.

## PSHS

The checkpoint register value plus one is used as an initial index into the string. To access the last byte (highest address) of the string, the checkpoint register must be set to -1 before the instruction is executed.

If a string has a length of zero, no data is moved, and status bits 0 through 2 are set to zero. If a tagged string has a length of one, only the tag byte is moved, and status bits 0 through 2 are set to zero.

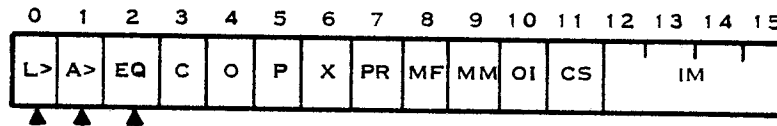
If the checkpoint register is set to a value greater than -1, the bytes moved are taken from inside the string, starting with the byte at <ckpt> + 1.

### NOTE

PSHS sets the <ckpt> register to -1 at completion.

Figure 3-3 illustrates the PSHS and POPS instructions.

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



*Execution results:* (ga<sub>s</sub>)→((ga<sub>d</sub>))

*Application notes:* If T<sub>d</sub> is equal to three, the indicated register is incremented by six. If T<sub>s</sub> is equal to three, the indicated register is incremented by the string length.

The following paragraphs describe an example of the push string to stack instruction. The stack boundaries must be set prior to execution of this instruction (refer to the paragraph on stacks in Section 2).

If the stack boundaries, upper and lower, are in the range from 235A<sub>16</sub> to 2373<sub>16</sub>, and the TOS points to address 236F<sub>16</sub>, then the instruction

```
SETO R6
```

```
LABEL PSHS @STRING,@STACK,14,R6
```

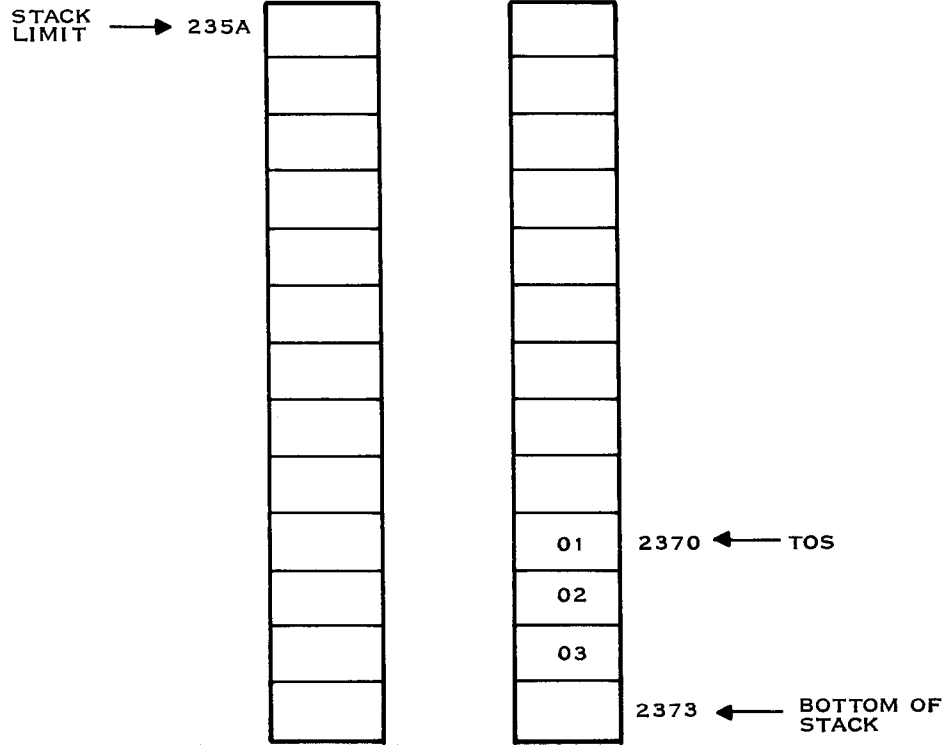
moves 14 bytes, starting at location STRING, onto the stack specified by the stack descriptor block at location STACK. The TOS descriptor value changes to 2361<sub>16</sub>. This example is shown figuratively:

# PSHS

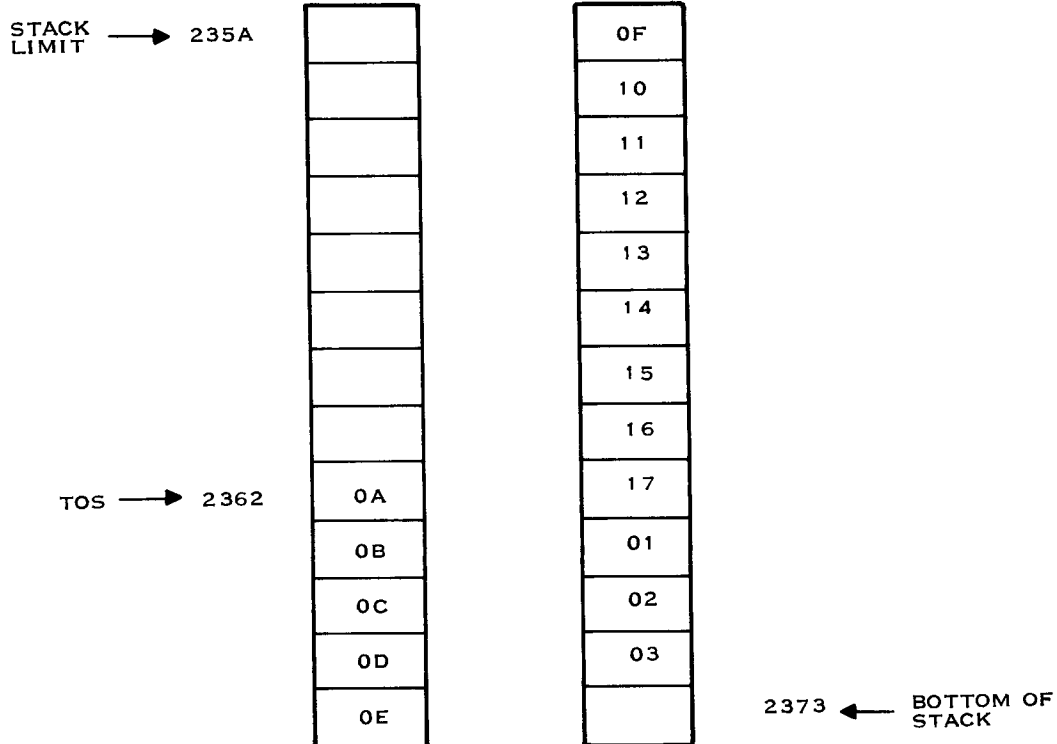
STRING VALUE

0A
0B
0C
0D
0E
0F
10
11
12
13
14
15
16
17

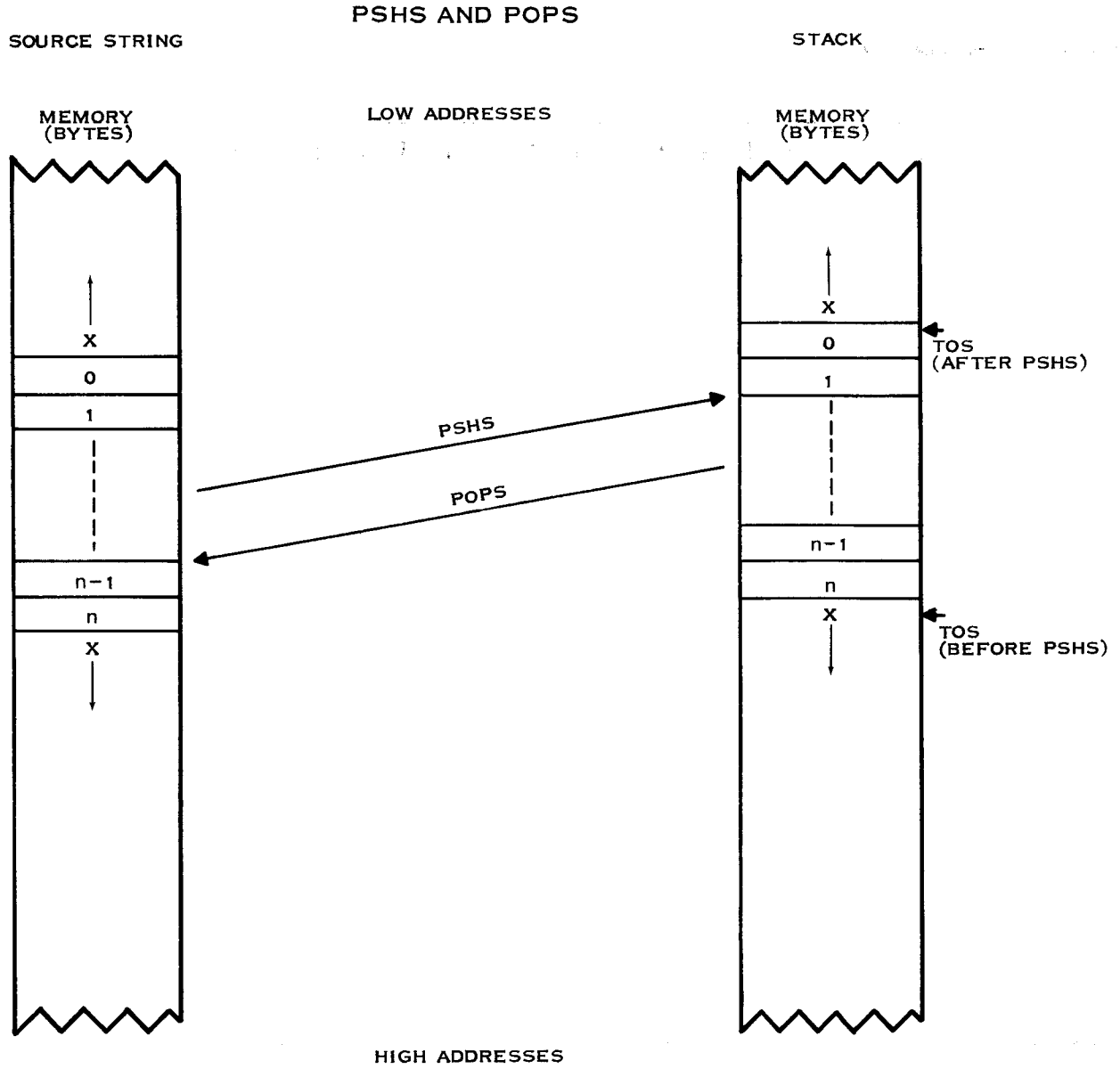
STACK BEFORE EXECUTION



STACK AFTER EXECUTION



# PSHS



X = MEMORY DATA THAT IS NOT AFFECTED BY PSHS OR POPS.

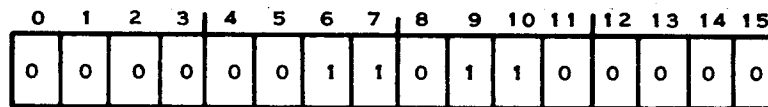
**Figure 3-3. PSHS or POPS Representation**

## RSET, RTO

### 3.104 RESET — RSET

Opcode: 0360

Instruction Format: 7



**Applicability:** Processor set A, privileged instruction

**Syntax definition:**

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

**Example:**

LABEL RSET            Start over.

**Definition:** The RSET instruction clears the interrupt mask, which disables all except level zero interrupts and clears the “disabled interrupt condition” set by the DINT instruction. It also resets all directly connected input/output devices and those CRU devices that provide for reset in the interface with the CRU. RSET also resets all pending interrupts and turns the clock off. RSET is a privileged instruction.

When privileged mode bit (bit 7 of ST register) is set to zero, instruction executes normally. When the privileged mode bit is set to one and execution of RSET instruction is attempted, an error interrupt occurs.

**Status bits affected:** None

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

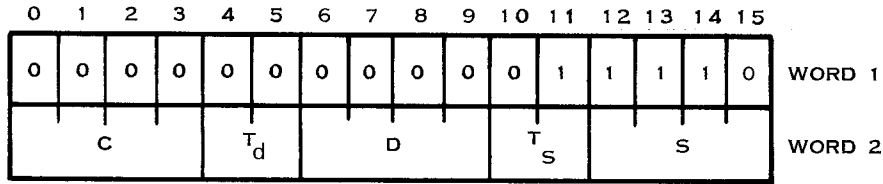
**Application notes:** Use the RSET instruction to reset the interrupt mask to zero, turn off the clock, and (depending on the device and interface) clear any pending interrupt and reset interface electronics.

### 3.105 RIGHT TEST FOR ONES — RTO

Opcode: 001E

**RTO**

**Instruction Format:** 11



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b.. RTOb..<ga<sub>s</sub>>,<ga<sub>d</sub>>,<cnt>]b..[<comment>]

**Example:**

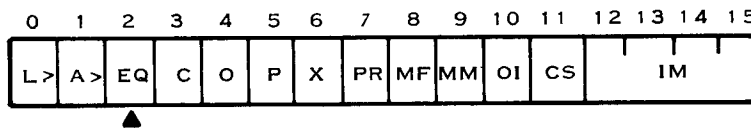
LABEL RTO @TST,@CNT,4

Locate the rightmost one in the byte string starting at location TST and add the one's bit position to the word at location CNT.

**Definition:** The multibyte value at the source address is examined for the rightmost one. The bit position value is added to the word at the destination address. If the value at the source address is zero, nothing is added to the destination and status register bit 2 is set to a one; otherwise, status bit 2 is set to zero. The number of bytes of precision of the source value is determined by the <cnt> field.

If <cnt> equals zero, the count is taken from the four LSBs of workspace register zero. If the four LSBs of workspace register zero are zero, the count is 16.

**Status bits affected:** Equal



**Execution results:** (ga<sub>d</sub>) + index to the rightmost one bit in (ga<sub>s</sub>)→(ga<sub>d</sub>)

**Application notes:** If T<sub>s</sub> is equal to three, the indicated register is incremented by the byte count.

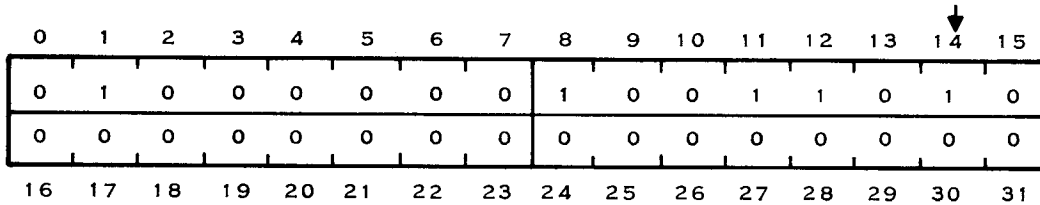
The following is an example of the right test for one instruction. TEST is a pointer to a four-byte string at memory address 449C<sub>16</sub> (the values contained in these bytes are 40<sub>16</sub>, 9A<sub>16</sub>, 0<sub>16</sub>, 0<sub>16</sub> respectively), and CNT contains the value BFC7<sub>16</sub>. The instruction

LABEL RTO @TEST,@CNT,4

## RTO, RTWP

checks for the rightmost one in the four bytes beginning at location TEST, and adds the bit position to the value in CNT. The result, in this example, is the value BFD5<sub>16</sub> being placed in CNT.

This example is shown figuratively:



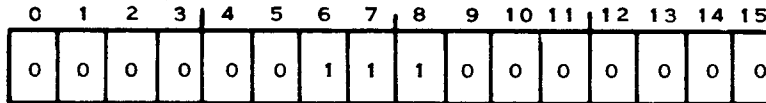
CNT BEFORE EXECUTION  
BFC7<sub>16</sub>  
CNT AFTER EXECUTION  
BFD5<sub>16</sub>

The equal bit of the status register is reset; the other bits of the status register are unaffected.

### 3.106 RETURN WITH WORKSPACE POINTER — RTWP

Opcode: 0380

Instruction Format: 7



Applicability: Processor set A

Syntax definition:

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

Example:

LABEL RTWP

Return from the subroutine called by the BLWP and restore the WP register, PC, and ST register with their contents previous to the BLWP.

**Definition:** Replace the contents of the WP register with the contents of the current workspace register 13. Replace the contents of the PC with the contents of the current workspace register 14.

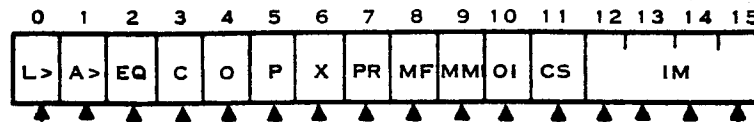


## RTWP, S

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

With the privileged mode bit (bit 7) of the ST register set to one, only bits 0 through 6 and bit 10 of workspace register 15 are placed in the corresponding bits of the ST register. When bit 7 of the ST register is set to zero, the instruction places bits 0 through 15 of workspace register 15 into bits 0 through 15 of the ST register.

*Status bits affected:* Restores status bits 0 through 6 and 10 or 0 through 15 to the value contained in workspace register 15.



*Execution results:* (Workspace register 13) → (WP)

(Workspace register 14) → (PC)

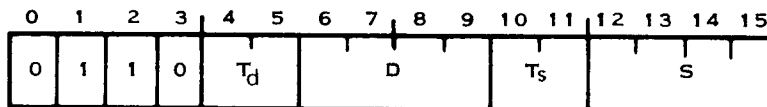
(Workspace register 15) → (ST)

*Application notes:* Use the RTWP instruction to restore the execution environment after the completion of execution of an interrupt, a BLWP instruction, or an XOP instruction. Refer to Section 4 for additional information.

### 3.107 SUBTRACT WORDS — S

*Opcode:* 6000

*Instruction Format:* 1



*Applicability:* Processor set A

*Syntax definition:*

[<label>]b..Sb..<ga>,<ga>b..[<comment>]

*Example:*

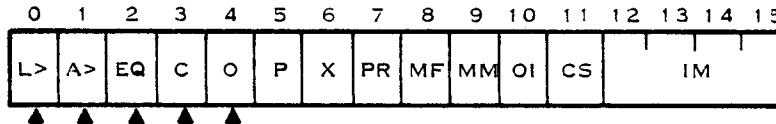
LABEL S R2,R3

Subtract the contents of workspace register two from the contents of workspace register three.

## S, SB

**Definition:** Subtract a copy of the source operand from the destination operand and place the difference in the destination operand. The AU compares the difference to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry out of bit 0, the carry status bit sets. When there is an overflow (the difference cannot be represented within a word as a two's complement value), the overflow status bit sets. The source operand remains unchanged.

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



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

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

S @OLDVAL,@NEWVAL

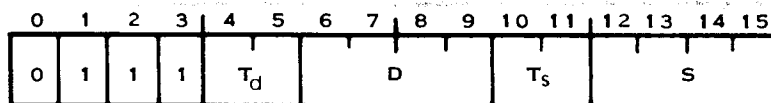
results in the contents of NEWVAL changing to  $6FFE_{16}$ . The logical greater than, arithmetic greater than, carry, and overflow status bit set, while the equal status bit resets.

Refer to Section 2 for a detailed description of normalization and single precision floating point instructions.

### 3.108 SUBTRACT BYTES — SB

**Opcode:** 7000

**Instruction Format:** 1



**Applicability:** Processor set A

**Syntax definition:**

[<label>]b..SBb..<ga\_s>,<ga\_d>b..[<comment>]

## SB, SBO

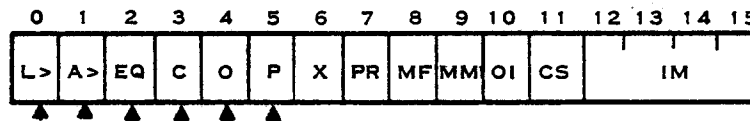
**Example:**

LABEL SB R2,R3

Subtract the leftmost byte of workspace register two from the leftmost byte of workspace register three and place the result in the leftmost byte of workspace register three.

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

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



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

**Application notes:** Use the SB instruction to subtract signed integer bytes. For example, if workspace register six contains the value  $121C_{16}$ , memory location  $121C_{16}$  contains the value  $2331_{16}$ , and workspace register one contains the value  $1344_6$ , then the instruction

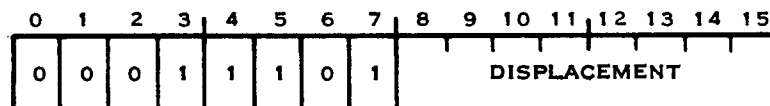
SB \*R6+,R1

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

### 3.109 SET CRU BIT TO LOGIC ONE — SBO

**Opcode:** 1D00

**Instruction Format:** 2





## SBZ, SD

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

When the privileged mode bit (bit 7) of the ST register is set to zero, the SBZ instruction executes normally. When bit 7 is set to one and the effective CRU address is equal to or greater than E00<sub>16</sub>, an error interrupt occurs and the instruction is not executed.

**Status bits affected:** None

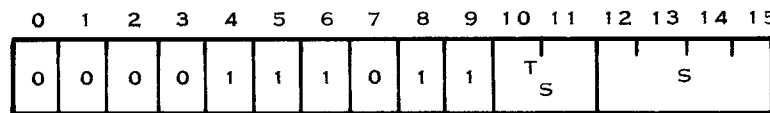
**Execution results:** The CRU bit addressed by the sum of the contents of workspace register 12 (bits 3 through 14) + displacement is set to zero.

**Application notes:** Use the SBZ instruction to set a CRU bit to a logic zero. Refer to Section 4 for additional application notes.

### 3.111 SUBTRACT DOUBLE PRECISION REAL — SD

**Opcode:** 0EC0

**Instruction Format:** 6



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..SDb..<ga,>b..[<comment>]

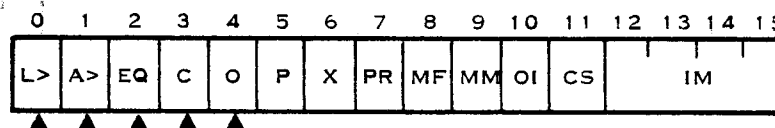
**Example:**

LABEL SD R5

Subtract the contents of workspace registers 5 through 8 from the FPA and place the result in the FPA.

**Definition:** The four-word value at the source address is subtracted from the FPA (R0-R3). The result is placed in the FPA.

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



# SD

Execution results: (FPA) - (ga.) → FPA

Application notes: The result of the SD instruction is compared to zero and status register bits 0, 1, and 2 reflect the comparison. If status register bits 3 and 4 are set to zero and one, respectively, underflow has occurred. If they are both set to one, overflow has occurred.

If T, is equal to three, the indicated register is incremented by eight.

The following is an example of a subtract double precision real instruction. R5-R8, after normalization, contain the value .040077AB<sub>16</sub>, as shown figuratively:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R5	0	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0
R6	0	0	0	0	0	1	1	1	0	1	1	1	1	0	1	0
R7	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
R8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

} NORMALIZED HEXADECIMAL FRACTION

The double precision FPA (R0-R3) contains, after normalization, the value .200000A<sub>16</sub>, as shown figuratively:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
R1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R2	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
R3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

} NORMALIZED HEXADECIMAL FRACTION

The instruction

LABEL SD 5

subtracts the contents of R5-R8 from the FPA and places the result, .1BFF88F5<sub>16</sub>, in the FPA, shown figuratively:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	1	1
R1	1	1	1	1	1	1	1	1	1	0	0	0	1	0	0	0
R2	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0
R3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

} NORMALIZED HEXADECIMAL FRACTION

## SD, SEQB

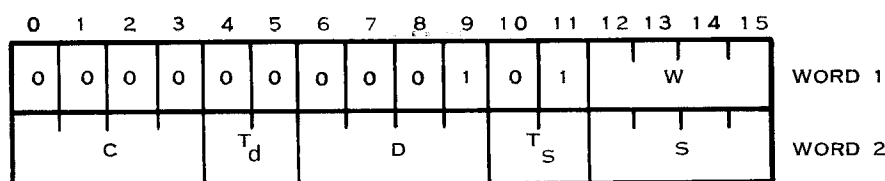
The logical greater than and arithmetic greater than bits of the status register are set; the equal, carry, and overflow bits of the status register are reset.

Refer to Section 2 for a detailed description of normalization and single precision floating point instructions.

### 3.112 SEARCH STRING FOR EQUAL BYTE — SEQB

Opcode: 0050

Instruction Format: 12



Applicability: Processor set C

Syntax definition:

[<label>]b...SEQBb...<ga, >,<ga<sub>a</sub> >,<cnt>[,<ckpt>]b...[<comment>]

The checkpoint register may be omitted from the instruction if a default has been specified with the CKPT assembler directive.

Example:

```

SETO R6
LABEL SEQB @INSEQ,@OUSEQ,,R6
    
```

Search the string starting at location OUSEQ for a byte equal to the one specified in location INSEQ, using the length in workspace register zero, and using workspace register six as the checkpoint register.

**Definition:** The word at the source address is used to specify a mask byte (MSB) and a data byte (LSB) for the search. The destination is a byte string. The search byte is compared to the bytes in the string starting at the location specified by D and T<sub>d</sub>. Only the unmasked bits of each byte are examined. The search byte is equal to the data byte ANDed with the mask byte. Each byte of the string is ANDed with the mask byte before being compared to the search byte.

The string length may be specified in the <cnt> field, in register zero, or as a tagged string (if <cnt> = 0 and R0 = >FFFF).

## SEQB

If a byte in the string is found which is equal to the search byte, an index pointer to the byte is stored in the checkpoint register <ckpt> and status bit 2 is set. To continue the search the instruction must be reexecuted. As bytes are found that are equal to the search byte, the checkpoint register is updated. When the last byte in the string is tested and found not equal to the search byte, status bit 2 is cleared and the checkpoint register is set to -1. If the last byte is equal to the search byte, the checkpoint register is set to point to the last byte with status bit 2 equal to one. Reexecution from this state sets status bit 2 equal to zero and the checkpoint register to -1.

If the length of the string is 16 bytes or more, the checkpoint register is used for interrupts. When reexecuted after the interrupt is serviced, the instruction continues the search where it left off. For more information on reexecutable instructions, see paragraph 3.39.1.

### NOTE

The checkpoint register value plus one acts as an initial index into string. To access the beginning of the string, the checkpoint register must be set to -1 (>FFFF) before SEQB is first executed. The first byte of a tagged string is not tested. If the string length is zero (or one for tagged strings), no search is performed, status bits 0 through 2 are all equal to zero, and the checkpoint register equals -1.

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

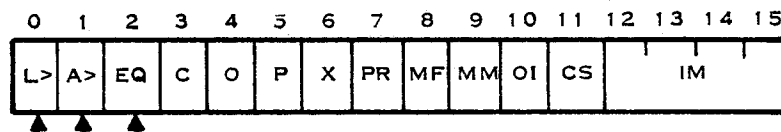


Table 3-5 lists conditions and status bit results for the SEQB.

**Table 3-5. SEQB/SNEB Status Bit Conditions**

Condition	Status Bit	Result	
		SEQB	SNEB
String length is zero	0-2	000	001
Last byte in the string is equal to the search byte	0-2	001	001
The last byte in the string is not equal to the search byte	0-2	(XX0)	(XX0)



## SEQB

Status bits 0 through 2 always reflect the results of the last comparison.

*Execution results:* An index pointer to the byte in (ga<sub>a</sub>) matching the specified byte in (ga<sub>a</sub>) is returned in the checkpoint register.

*Application notes:* When the SEQB instruction is executed, the contents of the checkpoint register are incremented and then used as a starting index into the string at which to begin the search. The instruction could be used in a loop that terminates when the equal bit (ST2) in the status register is reset upon completion of the SEQB instruction.

If T<sub>a</sub> is equal to three, two is added to the indicated register. If T<sub>a</sub> is equal to three, the indicated register is incremented by the string length.

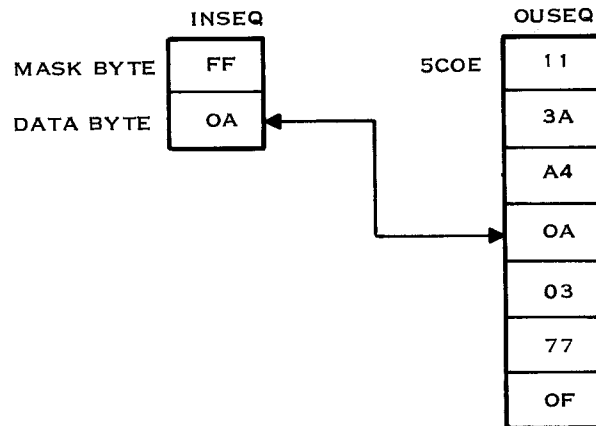
The following is an example of a search string for equal byte instruction. Location INSEQ contains the value FF0A<sub>16</sub>, OUSEQ addresses a byte string at memory address 5C0E<sub>16</sub>, and workspace register zero contains the value seven. The instructions

```

        SETO    R6
LABEL   SEQB   @INSEQ,@OUSEQ,,R6
    
```

initialize the checkpoint register (R6) and compare the data byte of INSEQ to the bytes starting at location OUSEQ for an equal byte, using the string length specified in R0. The first half of the word of INSEQ is the mask byte, and the second byte is the data byte.

In this example a match is found when the fourth byte of OUSEQ is compared to the data byte of INSEQ, as shown figuratively:



When the byte is found in OUSEQ matching the data byte of INSEQ, an index pointer to the byte is stored in the checkpoint register, in this case 3<sub>16</sub>, and the equal bit of the status register is set.

In general, after completion of the SEQB instruction, if the equal bit of the status register is set, the checkpoint register contains the index information to the equal byte in the byte string.

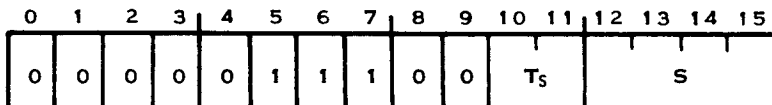
If the equal bit of the status register is reset and the checkpoint register is equal to a -1, then the entire byte string was searched and no equal byte was found.

## SETO, SLA

### 3.113 SET TO ONE — SETO

Opcode: 0700

Instruction Format: 6



Applicability: Processor set A

Syntax definition:

[<label>]b...SETOb...<ga,>b...[<comment>]

Example:

LABEL SETO R3

Set workspace register three to -1.

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

Status bits affected: None

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

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

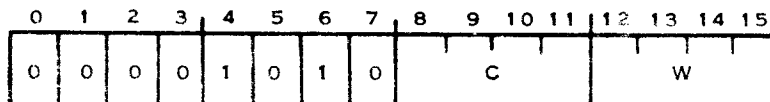
SETO R3

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

### 3.114 SHIFT LEFT ARITHMETIC — SLA

Opcode: 0A00

Instruction Format: 5



Applicability: Processor set A

## SLA, SLAM

**Syntax definition:**

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

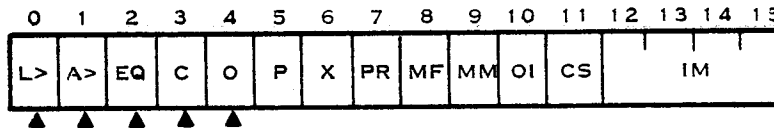
**Example:**

LABEL SLA R2,1

Shift the contents of workspace register two left one bit location.

**Definition:** Shift the contents of the specified workspace register to the left for the specified number of bit positions while filling the vacated bit positions with logic zero values. Note that the overflow status bit sets when the sign of the word changes during the shifting operation. The carry bit contains the last value shifted out of bit position 0.

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



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

**Application notes:** The following is an example of an arithmetic left shift. Workspace register 10 contains the value 1357<sub>16</sub>. The instruction

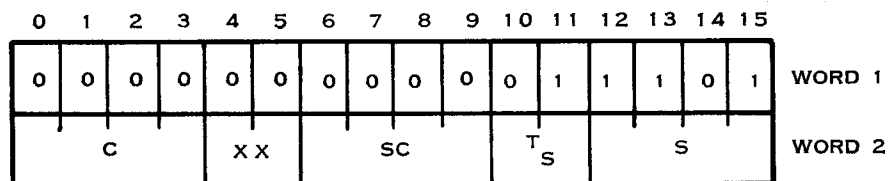
SLA R10,5

changes the contents of workspace register 10 to 6AE0<sub>16</sub>. The logical greater than, arithmetic greater than, and overflow status bits set, while the equal and carry status bits reset.

### 3.115 SHIFT LEFT ARITHMETIC MULTIPLE PRECISION — SLAM

**Opcode:** 001D

**Instruction Format:** 13



# SLAM

**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..SLAMB..<ga, >,<cnt>][,<scnt>]b..[<comment>]

**Example:**

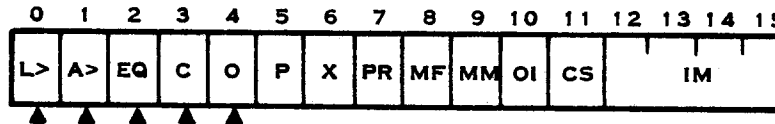
LABEL SLAM @BIT,6,8

Shift the 6-byte field, BIT, eight bits to the left.

**Definition:** The multibyte value at the source address is shifted left by the number of bits specified by <scnt>. The <cnt> is the number of bytes of precision. If <cnt> is zero or is not present, bits 12 through 15 of workspace register zero are used. If bits 12 through 15 are zero, the precision is 16 bytes.

If <scnt> is zero or is not present, bits 4 through 7 of workspace register zero are used. If bits 4 through 7 are zero, the shift count is zero. Bits shifted out of the most significant end are shifted into status register bit 3; the bit positions at the least significant end are filled with zeros as they are vacated.

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



Bit 3 is a copy of the last bit shifted out of the most significant end of the multibyte value. Bit 3 is cleared to zero if the shift count is zero. Bit 4 indicates that the value of the MSB changed during the shift.

**Execution results:** The source address is shifted left by the number of bits specified in <scnt>.

**Application notes:** If T, is equal to three, the indicated register is incremented by the byte count. The result of the SLAM instruction is compared to zero and status register bits 0, 1, and 2 reflect the comparison. Status register bit 3 is a copy of the last bit shifted out of the most significant end of the multibyte value. If the shift count is zero, status register bit 3 is cleared to zero. Status register bit 4 indicates the arithmetic overflow.

The following is an example of a shift left arithmetic multiple precision. Location BIT contains the value 1357<sub>16</sub>. The instruction

LABEL SLAM @BIT,2,5

shifts the contents of BIT to the left five bits, resulting in the value of BIT changing to 6AE0<sub>16</sub>.

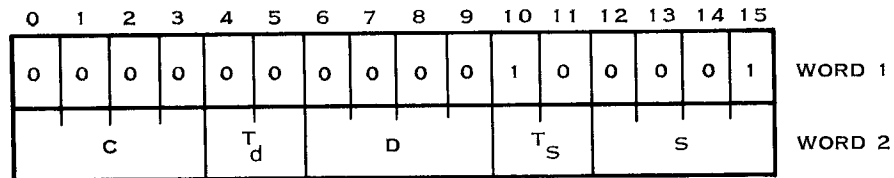
## SLAM, SLSL

The logical greater than, arithmetic greater than, and overflow bits of the status register are set; the equal and carry bits are reset.

### 3.116 SEARCH LIST LOGICAL ADDRESS — SLSL

*Opcode:* 0021

*Instruction Format:* 20



*Applicability:* Processor set C

*Syntax definition:*

[<label>]b..SLSLb..<cond>,<ga,>,<ga,>b..[<comment>]

*Example:*

LABEL SLSL EQ,@TAB,@NTAB

Search, using data in TAB, until an equal condition is met. The contents of NTAB specify the initial address of the list to search.

*Definition:* The source operand is a five-word control block (LSCB). The destination operand is a two-word block specifying the current, or starting, node pointer and the previous node pointer in the first and second words, respectively.

The list search control block (LSCB), beginning with the word specified by the source address, is applied in searching a list. The first word specified by the destination address specifies the initial block address for execution of the instruction. This same word is also a checkpoint in case of interrupts. When an interrupt occurs, the checkpoint data is stored and the program counter is adjusted so that upon return from the interrupt the instruction is reexecuted. On completion of the instruction, the first destination word contains a pointer to the block that matched and the following word contains the pointer to the previous block. When comparing elements of a list, the result of a logical AND operation between the test value and the test mask is compared with the result of a logical AND operation between the list element and the test mask. When no match occurs, the first word of the destination address contains the terminal link value. The second word contains the final block address.

The instruction's <cond> field defines the condition for exit from the search as listed in Table 3-6.

# SLSL

**Table 3-6. Search Termination Conditions**

<cond>Field Entry	Value	Condition
EQ	0	Equal (=)
NE	1	Not Equal (≠)
HE	2	Logical (≥)
L	3	Logical (<)
GE	4	Arithmetic (≥)
LT	5	Arithmetic (<)
LE	6	Logical (≤)
H	7	Logical (>)
LTE	8	Arithmetic (≤)
GT	9	Arithmetic (>)
	10	RESERVED*
	11	RESERVED*
	12	RESERVED*
	13	RESERVED*
	14	RESERVED*
	15	RESERVED*

**Note:**

\* If one of these undefined conditions is specified, an illegal operation error results.

The list search control block, addressed by S and T, is five words long and contains the following information:

WORD 0 — Signed byte displacement to link word (the LSB is ignored)

WORD 1 — Signed byte displacement to compare word (the LSB is ignored)

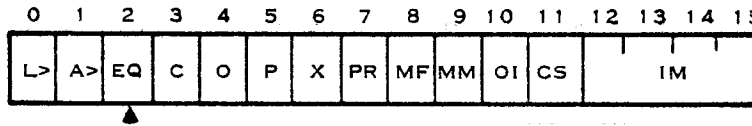
WORD 2 — Test value to be used

WORD 3 — Test mask to be used

WORD 4 — Terminal link value

**SLSL**

Status bit affected: Equal



Execution results: (ga<sub>d</sub>) is searched using table <ga, > until <cond> is met.

Application notes: If T<sub>d</sub> is equal to three, the indicated register is incremented by 10. If T<sub>d</sub> is equal to three, the indicated register is incremented by four. If the search was exited due to a true test condition, status register bit 2 is set to one. If the search was terminated because the end of the list was reached, status register bit 2 is equal to zero.

If the list is empty (indicated by the destination operand containing the terminal link value), the destination operands are unchanged and the equal status bit is reset. If a match is found on the first block, the destination operands are unchanged and the equal bit is set.

The following example of an SLSL instruction searches for a single bit to be set. The first time the search is attempted, a match is found in the second list block.

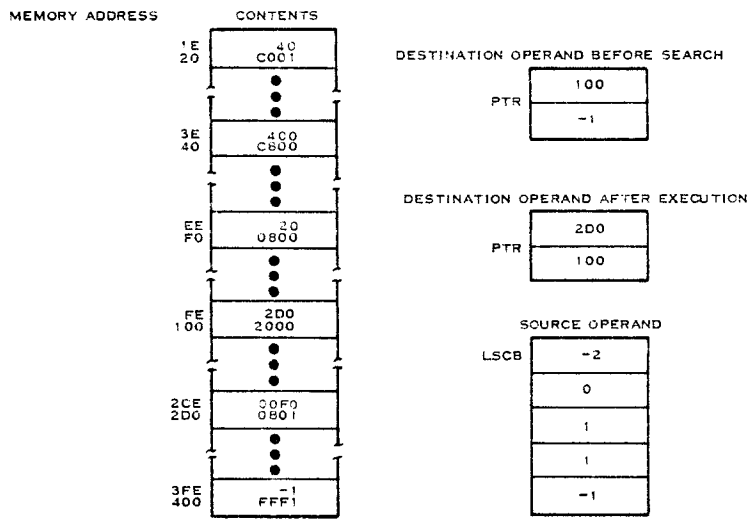
The instruction executed in this example is as follows:

```
SLSL    EQ,@LSCB,@PTR
```

Initially, PTR contains the address of the first word to be searched, >100; and the next word after PTR (PTR + 2) contains the value -1, the value contained in the last link in the list. The first word of LSCB contains -2, which specifies that the address of the next word in the list is found two bytes before the current word. The second word of LSCB contains the displacement to the word to be compared, in this case zero. The third word of LSCB contains the value tested for, in this case one, and the fourth word contains the test mask, also one in this case. The condition of this search is satisfied when a word is found with the rightmost (MSB) bit set to one. The fifth word of LSCB contains the value of the last link in the linked list, -1 in this case.

When this instruction is executed, the value at location >100, which is >2000, is ANDed with the test mask, which produces a value of zero, and the result is compared to the test value. The search condition is not satisfied, so the instruction is executed again. The link is taken from location >100-2, or >FE, which contains the value >200. The value at location >200, which is >0801, is ANDed with the test mask, which produces a one. The result matches the test value and the search is terminated. On termination, location PTR contains the address of the word where the match was found, >200, and PTR + 2 contains the address of the previous word compared, >100.

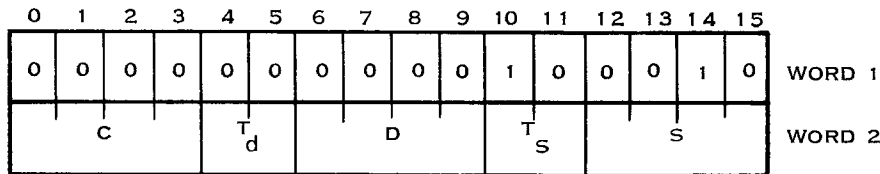
# SLSL, SLSP



### 3.117 SEARCH LIST PHYSICAL ADDRESS — SLSP

Opcode: 0022

Instruction Format: 20



Applicability: Processor set C, privileged instruction

Syntax definition:

[<label>]b..SLSPb..<cond>,<ga>,<ga<sub>d</sub>>b...[<comment>]

Example:

LABEL SLSP NE,@TAB,@NTAB

Search, using the data in TAB, until an NE condition is found. The contents of NTAB specify the initial address of the list to search.

Definition: The SLSP instruction extends the addressability of a list search. SLSP allows for addressing physical memory since all list block addresses are 16-bit bias values.

The list search control block (LSCB) beginning with the word specified by the source address is applied in searching a list. The first word specified by the destination address contains the initial



## SLSP

block bias value for execution of the instruction. This same word is also a checkpoint in case of interrupts. When an interrupt occurs, the checkpoint data is stored and the program counter is adjusted so that upon return from the interrupt the instruction is reexecuted. On completion of the instruction, the first destination word contains the bias value of the block that matched and the following word contains the bias value of the previous block.

The instruction's <cond> field defines the condition for exit from the search. Table 3-6 contains the conditions that allow termination of a search.

SLSP is a privileged instruction.

The list search control block, addressed by the source address, is five words long and contains the following information:

WORD 0 — Signed byte displacement to link word (the LSB is ignored in word zero)

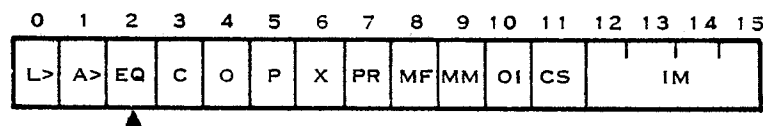
WORD 1 — Signed byte displacement to compare word (the LSB is ignored in word one)

WORD 2 — Test value to be used

WORD 3 — Test mask to be used

WORD 4 — Terminal link value

Status bits affected: Equal



**Execution results:** ( $ga_d$ ) is searched using table ( $ga_s$ ) until <cond> is met.

**Application notes:** When the 990 mapping logic is enabled, the SLSP instruction extends the addressability of a list search. SLSP allows for addressing physical memory since all list block addresses are 16-bit bias values. Status register bit 2 (equal bit) is set/reset to indicate if the condition was met during the search; this condition represents a pass/fail indication rather than an equal/nonequal indication. If the search was exited due to a true test condition, status register bit 2 is set to one. If the search was terminated because the end of the list was reached, status register bit 2 is reset. If the list is empty (indicated by the destination operand containing the terminal link value), the destination operand is unchanged and status register bit 2 is reset. If a match is found on the first block, the destination operand is unchanged and status register bit 2 is set.

If  $T_s$  is equal to three, add 10 to the indicated register. If  $T_d$  is equal to three, add four to the indicated register.

To obtain the physical address, the 16-bit bias values in the link word are multiplied by  $20_{16}$ , giving a 21-bit byte address. The sign of the 16-bit displacement in words zero and one of the list search control block is extended to form a 21-bit bias address to form the memory address of the link word and the bias word, respectively.

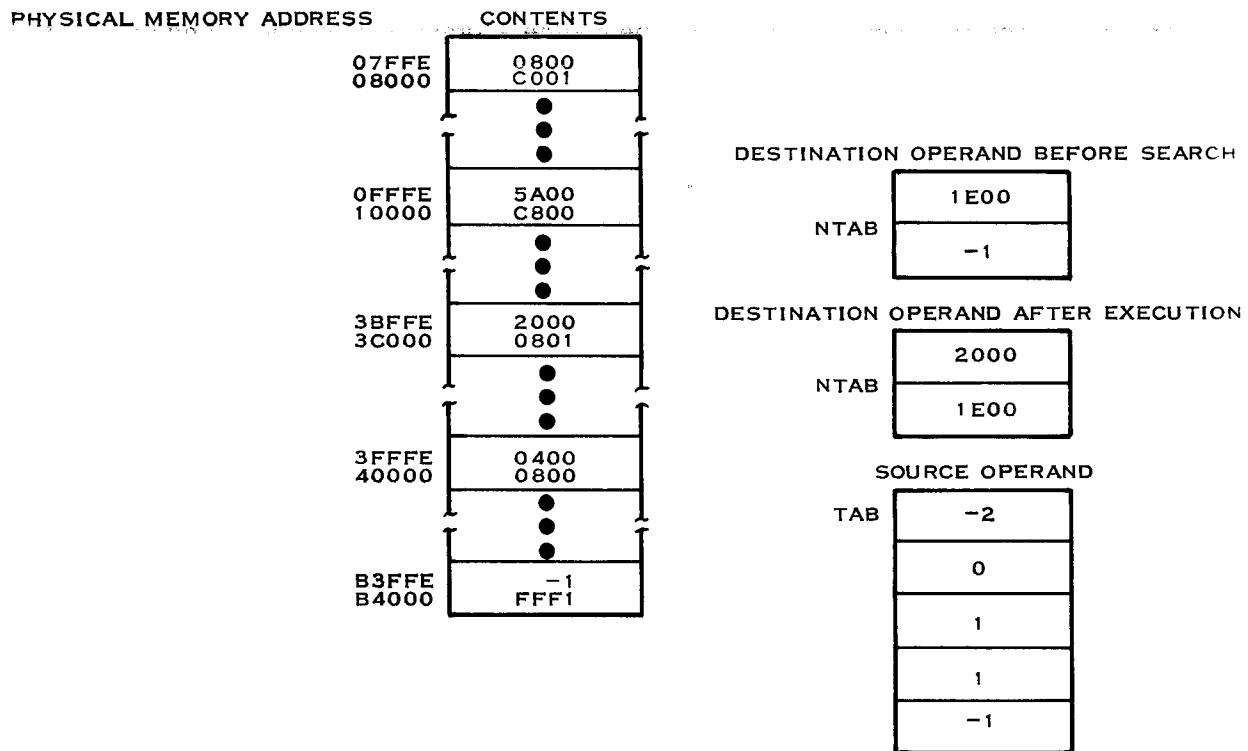
## SLSP, SM

The following example of an SLSP instruction searches for a not equal condition in bit position 15. The not equal condition is met at the second block of the searched addresses.

The instruction executed in this example is as follows:

**SLSP NE,@TAB,@NTAB**

This instruction works exactly like the SLSL instruction except for the manipulation of the logical addresses to obtain physical addresses, described above.

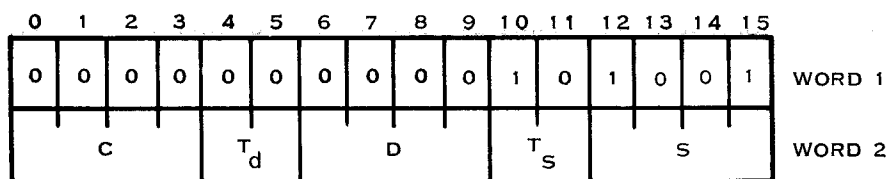


The equal bit of the status register is set; the other bits of the status register are unaffected.

### 3.118 SUBTRACT MULTIPLE PRECISION INTEGER — SM

**Opcode: 0029**

**Instruction Format: 11**



# SM

**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..SMb..<ga,>,<ga<sub>d</sub>>[,<cnt>]b..[<comment>]

**Example:**

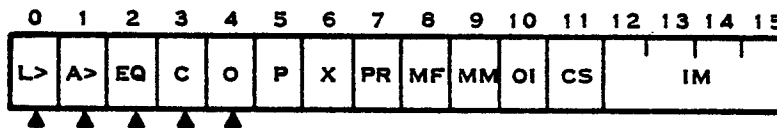
LABEL SM R4,R6,4

Subtract the two's complement four-byte value starting at workspace register four from the two's complement four-byte value starting at workspace register six, and put the result in the four bytes beginning with workspace register six.

**Definition:** The multibyte two's complement integer at the source address is subtracted from the multibyte two's complement integer at the destination address. The result is placed in the destination address. The <cnt> field determines the number of bytes of precision for the integer.

If <cnt> equals zero or is not present, the count is taken from the four LSBs of workspace register zero. If the four LSBs of workspace register zero are zero, the count is 16.

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



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

**Application notes:** If T<sub>s</sub> and/or T<sub>d</sub> is equal to three, the indicated register is incremented by the byte count.

The result of the SM instruction is compared to zero and the status register bits 0, 1, and 2 indicate the results of the comparison. The status register bits 3 and 4 indicate the carry and overflow.

The following is an example of a subtract multiple precision instruction. Workspace registers four and five contain the values 849C<sub>16</sub> and BA72<sub>16</sub>, respectively. The instruction

LABEL SM R4,R5,2

subtracts the two's complement two-byte value in R4 from the two's complement two-byte value in R5 and places the result, in this case 35D6<sub>16</sub>, in R5.

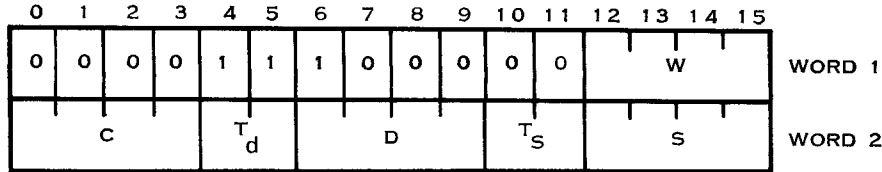
The logical greater than, arithmetic greater than, and carry bits of the status register are set; the equal and overflow bits of the status register are reset.

## SNEB

### 3.119 SEARCH STRING FOR NOT EQUAL BYTE — SNEB

**Opcode:** 0E10

**Instruction Format:** 12



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..SNEBb..<ga>,<ga>,<cnt>[,<ckpt>]b..[<comment>]

Trailing commas from the operand list may be omitted. The checkpoint register may be omitted from the instruction if a default has been specified with the CKPT assembler directive.

**Example:**

```

SETO   R6
LABEL  SNEB @INSEQ,@OUSEQ,,R6
    
```

Search the string starting at location OUSEQ for a byte that is not equal to the one specified in location INSEQ. The string length is specified by the length in workspace register zero or as a tagged string. Workspace register six is used as the checkpoint register.

**Definitions:** The word at the source operand is used to specify a mask byte (MSB) and a data byte (LSB) for the search. The destination operand is a byte string. Only the unmasked bits of each byte in the string are examined. The search byte is equal to the data byte ANDed with the mask byte. Each byte of the string is ANDed with the mask byte before being compared to the search byte.

The search byte is compared to the bytes in the string starting at the location specified by <ga>. The length of the string is specified by the <cnt> field, R0, or as a tagged string (if <cnt> = 0 and R0 = >FFFF).

If a byte in the string is found that is not equal to the search byte, an index pointer to the byte is stored in the checkpoint register and status bit 2 is set to zero. Status bits 0 and 1 are set to reflect the results of the comparison of the search byte with the unequal byte (after being masked).

**SNEB**

To continue the search, the instruction must be reexecuted. As bytes are found that are not equal to the search byte, the checkpoint register is updated. When the last byte in the string is tested and found equal to the search byte, status bit 2 is set. If the last byte is not equal to the search byte, the checkpoint register is set to point to the last byte with status bit 2 equal to zero. Reexecution from this state sets status bit 2 to one and the checkpoint register to -1. If the length of the string is 16 bytes or more, the checkpoint register is used for interrupts.

When reexecuted after an interrupt is serviced, the instruction continues the search where it left off.

**NOTE**

The checkpoint register value plus one acts as an initial index into the string. To access the beginning of the string, the checkpoint register must be set to -1 before SNEB is first executed. The first byte of a tagged string is not searched. If the string length is zero (or one for a tagged string), no search is performed, status bits 0 through 2 are set to zero, zero, and one, respectively, and the checkpoint register is set to -1.

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

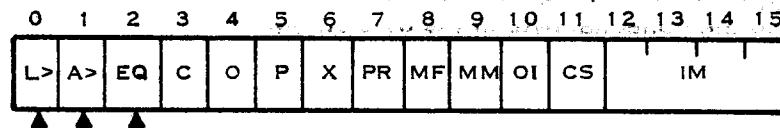


Table 3-5 lists conditions and status bit results for the SNEB.

**Execution results:** (ga<sub>a</sub>):(ga<sub>a</sub>) for a not equal byte

**Application notes:** If T<sub>a</sub> is equal to three, the indicated register is incremented by two. If T<sub>a</sub> is equal to three, the indicated register is incremented by the byte string length.

The following is an example of a search string for not equal byte instruction. Location INSEQ contains the value FF01<sub>16</sub>, location OUSEQ addresses a byte string at memory address 5C0E<sub>16</sub>, and workspace register zero contains the value seven. The instructions

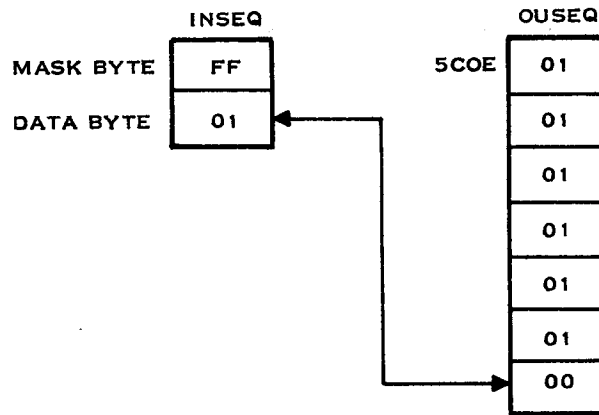
```

SETO      R6
LABEL    SNEB    @INSEQ,@OUSEQ,,R6
    
```

initialize the checkpoint register (R6) and compare the data byte of INSEQ to the bytes addressed by OUSEQ for a nonequal byte using the string length specified in R0. The first half of the word of INSEQ is the mask byte, and the second byte is the data byte.

## SNEB, SOC

In this example a nonequal value is found when the seventh byte of OUSEQ is compared to the data byte of INSEQ, as shown figuratively:



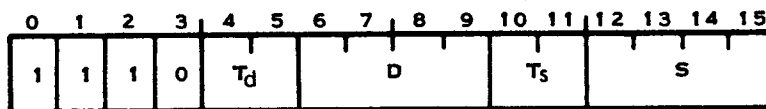
When the nonequal byte is found in the string at OUSEQ matching the data byte of INSEQ, an index pointer to the byte is stored in the checkpoint register, in this case 6<sub>16</sub>. The equal bit of the status register is reset, and the logical greater than and arithmetic greater than status bits are set.

If the equal bit of the status register is set, all bytes searched equal the search byte. If the equal bit of the status register is reset, a nonequal byte was found and the checkpoint register contains the index information to the nonequal byte in the byte string.

### 3.120 SET ONES CORRESPONDING — SOC

Opcode: E000

Instruction Format: 1



Applicability: Processor set A

Syntax definition:

[<label>]b..SOCb..<ga>,<ga>b..[<comment>]

Example:

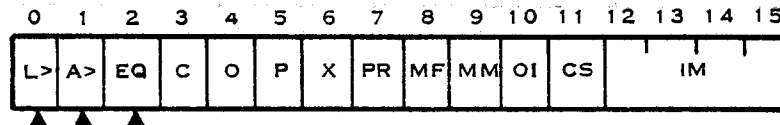
LABEL SOC R3,R2

Set the bits in workspace register 2 that correspond to logic one bits in workspace register 3 to one.

## SOC, SOCB

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

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



**Execution results:** Bits of (ga<sub>d</sub>) corresponding to bits of (ga<sub>s</sub>) equal to one are set to one.

**Application notes:** Use the SOC instruction to OR the 16-bit contents of two operands. For example, if workspace register three contains FF00<sub>16</sub> and location NEW contains AAAA<sub>16</sub>, then the instruction

SOC R3,@NEW

changes the contents of location NEW to FFAA<sub>16</sub> while the contents of workspace register three are unchanged. This is shown as follows:

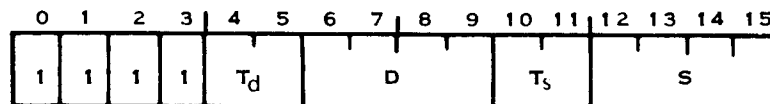
1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0	(Source operand)
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0	(Destination operand)
1 1 1 1 1 1 1 1 0 1 0 1 0 1 0	(Destination operand)

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

### 3.121 SET ONES CORRESPONDING, BYTE — SOCB

**Opcode:** F000

**Instruction Format:** 1



**Applicability:** Processor set A

**Syntax definition:**

[<label>]b...SOCBb...<ga<sub>s</sub>>,<ga<sub>d</sub>>b...[<comment>]

## SOCB, SR

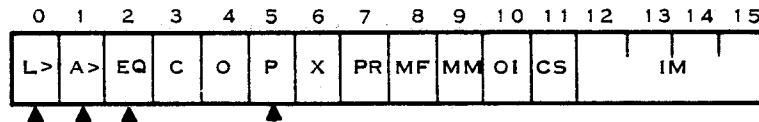
**Example:**

LABEL SOCB R3,@DET

Set the bits in the byte at location DET to one that correspond to the one bits in the first byte of workspace register three.

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

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



**Execution results:** Bits of (ga<sub>s</sub>) corresponding to bits of (ga<sub>s</sub>) equal to one are set to one, (that is, (ga<sub>d</sub>) OR (ga<sub>s</sub>) → (ga<sub>d</sub>)).

**Application notes:** Use the SOCB instruction to OR two byte operands. For example, if workspace register 5 contains the value F013<sub>16</sub> and workspace register eight contains the value AA24<sub>16</sub>, then the instruction

SOCB R5,R8

changes the contents of workspace register eight to FA24<sub>16</sub>, while the contents of workspace register five are unchanged. This is shown as follows:

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

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

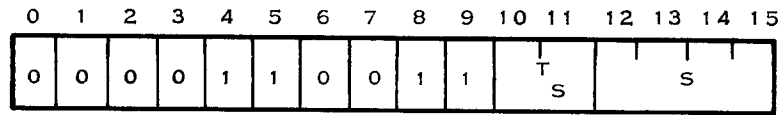
### 3.122 SUBTRACT REAL — SR

**Opcode:** 0CC0



# SR

**Instruction Format:** 6



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..SRb..<ga,>b..[<comment>]

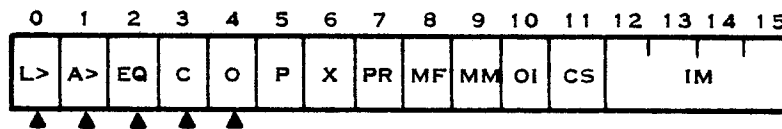
**Example:**

LABEL SR @WORD

Subtract the contents of the two-word real number, beginning at location WORD, from the FPA and replace the FPA with the difference.

**Definition:** The two-word normalized value at the source address is subtracted from the FPA and the result is stored in the FPA (R0-R1).

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

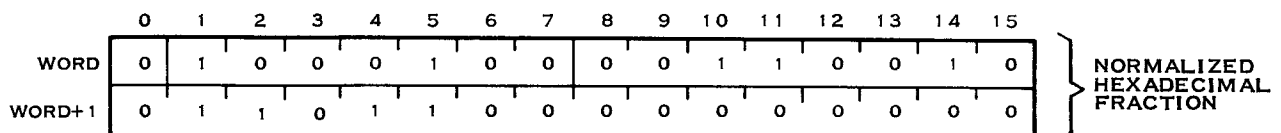


**Execution results:** FPA - (ga<sub>s</sub>)→FPA

**Application notes:** The result of the SR instruction is compared to zero and status register bits 0, 1, and 2 reflect the comparison. If status register bits 3 and 4 are set to zero and one, respectively, underflow has occurred. If they are set to ones, overflow has occurred.

If T<sub>s</sub> is equal to three, the indicated register is incremented by four.

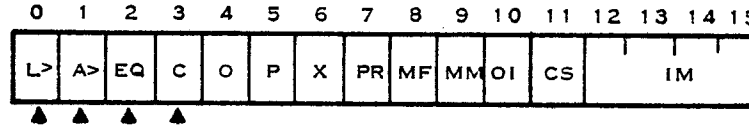
The following is an example of the subtract real instruction. Location WORD, after normalization, contains the value 326C<sub>16</sub>, as shown figuratively:





## SRA, SRAM

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



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

**Application notes:** The following is an example of an arithmetic right shift. Workspace register five contains the value 8824<sub>16</sub>, and workspace register zero contains the value F326<sub>16</sub>. The instruction

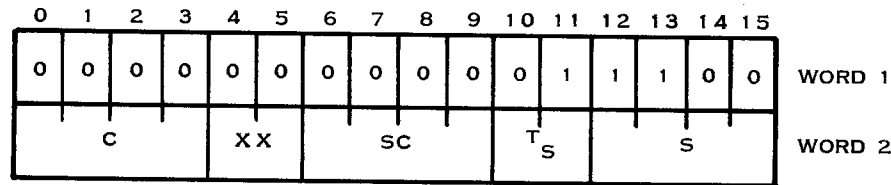
SRA      R5,0

changes the contents of workspace register five to FE20<sub>16</sub>. The logical greater than and carry status bits set, while the arithmetic greater than and equal status bits reset.

### 3.124 SHIFT RIGHT ARITHMETIC MULTIPLE PRECISION — SRAM

**Opcode:** 001C

**Instruction Format:** 13



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..SRAMb..<ga.>,<cnt>[,<scnt>]b..<comment>

**Example:**

LABEL SRAM @BIT,6,8

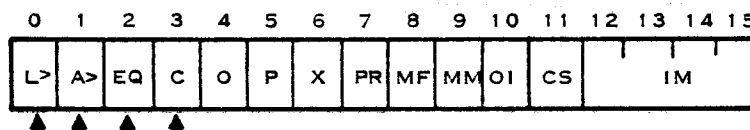
Shift the six-byte field, starting at location BIT, eight bits to the right.

## SRAM, SRC

**Definition:** The multibyte value at the source address is shifted right by the number of bits specified by <scnt>. The count <cnt> is the number of bytes of precision. If <cnt> is zero, bits 12 through 15 of workspace register zero are used. If bits 12 through 15 equal zero, the precision is 16 bytes.

If <scnt> is zero, bits 4 through 7 of workspace register zero are examined to determine this shift count. If bits 4 through 7 are zero, the shift count is zero. Bits shifted out of the least significant end are shifted into status register bit 3. The bit positions at the most significant end are filled with the sign bit as they are vacated.

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



**Execution results:** The source operand is shifted to the right the number of bits specified in <scnt> and the sign is extended.

**Application notes:** The MSB (sign bit) is not changed during execution. The result of the SRAM instruction is compared to zero and the status register bits 0, 1, and 2 reflect the comparison. Status register bit 3 is a copy of the last bit shifted out of the least significant end of the multibyte value. If the shift count is zero, the status register bit 3 is set to zero. If T<sub>s</sub> is equal to three, the indicated register is incremented by the byte count.

The following is an example of a shift right arithmetic multiple precision. Location BIT contains the value 8224<sub>16</sub>, and workspace register zero contains the value F326<sub>16</sub>. The instruction

LABEL SRAM @BIT,2,0

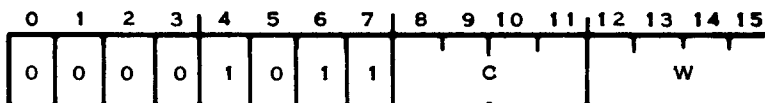
shifts the contents of BIT to the right three bits, causing the value of BIT to change to F044<sub>16</sub>.

The logical greater than and carry bits of the status register are set; the arithmetic greater than and equal bits are reset.

### 3.125 SHIFT RIGHT CIRCULAR — SRC

**Opcode:** 0B00

**Instruction Format:** 5



**Applicability:** Processor set A

## SRC, SRJ

**Syntax definition:**

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

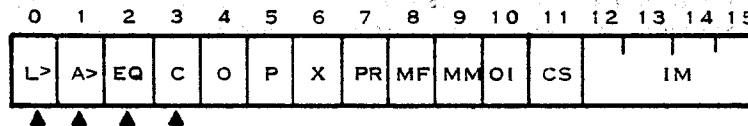
**Example:**

LABEL SRC R7,3

Shift workspace register seven three bit locations to the right filling in the vacated bit positions with the bit shifted out of position 15.

**Definition:** Shift the specified workspace register to the right for the specified number of bit positions while filling vacated bit positions with the bits shifted out of position 15. The carry bit contains the last value shifted out of bit position 15.

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



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

**Application notes:** The following is an example of a circular right shift. Workspace register two contains the value FFEF<sub>16</sub>. The instruction

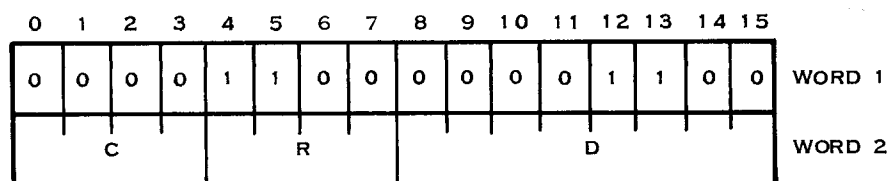
SRC R2, 7

changes the contents of workspace register two to DFFF<sub>16</sub>. The logical greater than and carry status bits set, while the arithmetic greater than and equal status bits reset. Shift left circular is not implemented since SRC can perform the same function: SLC x,n = SRC x,16-n.

### 3.126 SUBTRACT FROM REGISTER AND JUMP — SRJ

**Opcode:** 0C0C

**Instruction Format:** 17



## SRJ

*Applicability:* Processor set C

*Syntax definition:*

[<label>]b..SRJb..<exp>,<cnt>,<wad>b..<comment>]

*Example:*

LABEL SRJ BEGIN,12,R3

Subtract 12 from workspace register three and jump to BEGIN if the result of the subtraction is not equal to zero or does not pass through zero.

*Definition:* The unsigned position integer in the <cnt> field is subtracted from the register specified by <wa>. If the <cnt> field is zero, the value to be subtracted is obtained from workspace register zero as a 16-bit unsigned value. If the <cnt> operand is not present, it defaults to one. If <wa> minus the effective <cnt> does not equal zero or pass through zero (wrap-around), the signed word displacement, <exp>, is added to the program counter.

*Status bits affected:* None

*Execution results:* (wa) - <cnt> → (wa)

If the value of the register has not passed through zero, then the value of the program counter plus the displacement is placed in the program counter.

*Application notes:* The SRJ instruction is useful for controlling a counter for a loop. A loop to transfer a character string in 990/12 instructions could be written as follows:

SOURCE	TEXT	'ABCDEF'
DEST		,
	LI	R6,6
TOP	MOVB	SOURCE-1(R6),DEST-1(R6)
	DEC	R6
	JGT	TOP

This loop could be coded using the SRJ instruction as follows:

TOP	LI	R6,6
	MOVB	SOURCE-1(R6),DEST-1(R6)
	SRJ	TOP,1,R6

## SRJ, SRL

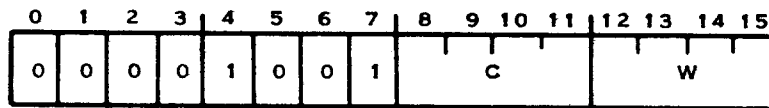
These two loops require the same amount of memory but the SRJ loop has only two instructions in it and runs faster. When the decrement value is greater than two, the SRJ instruction begins to make the loop shorter in size as well as in the number of instructions. The MVSR instruction can also be used to perform this function.

The status register is not affected.

### 3.127 SHIFT RIGHT LOGICAL — SRL

Opcode: 0900

Instruction Format: 5



Applicability: Processor set A

Syntax definition:

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

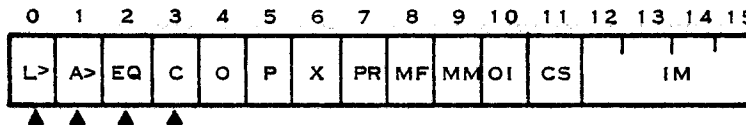
Example:

LABEL SRL R3,7

Shift the contents of workspace register three seven bit locations to the right.

**Definition:** Shift the contents of the specified workspace register to the right for the specified number of bits while filling the vacated bit positions with logic zero values. The carry bit contains the last value shifted out of bit position 15.

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



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

## SRL, STCR

*Application notes:* The following is an example of a logical right shift. Workspace register zero contains the value FFEF<sub>16</sub>. The instruction

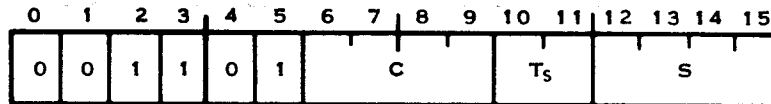
SRL R0,3

changes the contents of workspace register zero to 1FFD<sub>16</sub>. The logical greater than, arithmetic greater than, and carry status bits set, while the equal status bit resets.

### 3.128 STORE CRU — STCR

*Opcode:* 3400

*Instruction Format:* 4



*Applicability:* Processor set A

*Syntax definition:*

[<label>]b..*STCR*b..<ga>,<cnt>b..[<comment>]

*Example:*

READ STCR @BUF,9

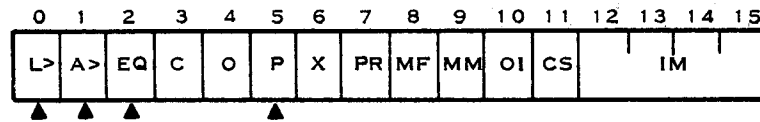
Read nine bits from the CRU and store at location BUF.

*Definition:* Transfer the number of bits specified in the <cnt> field from the CRU to the source operand. The transfer begins from the CRU address specified in bits 3 through 14 of workspace register 12 to the least significant bit of the source operand and fills the source operand toward the most significant bit. When the <cnt> field contains a zero, the number of bits to transfer is 16. If the number of bits to transfer is from 1 to 8, the source operand address is a byte address. Any bit in the memory byte not filled by the transfer is reset to a zero. When the number of bits to transfer is from 9 to 16, the source operand address is a word address. In word mode, if the source operand address is odd, the address is truncated to an even address prior to data transfer. If the transfer does not fill the entire memory word, unfilled bits are reset to zero. When the number of bits to transfer is a byte or less, the bits transferred are compared to zero and the status bits set/reset to indicate the results of the comparison. Also, when the bits to be transferred are a byte or less, the odd parity bit sets when the bits establish odd parity.



## STCR, STD

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



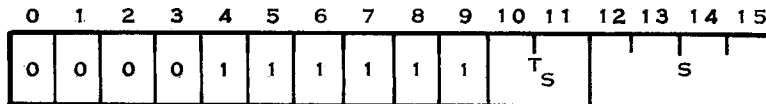
**Execution results:** The number of bits specified by <cnt> are transferred from consecutive CRU lines beginning at the address in workspace register 12 to memory at address <ga,>.

**Application notes:** Use the STCR instruction to transfer a specified number of CRU bits from the CRU to the memory location supplied by the user as the source operand. The CRU base address must be in workspace register 12 prior to the execution of this instruction. Refer to Section 4 for a detailed explanation and examples of the use of the STCR instruction.

### 3.129 STORE DOUBLE PRECISION REAL — STD

**Opcode:** 0FC0

**Instruction Format:** 6



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b...STDb...<ga,>b...[<comment>]

**Example:**

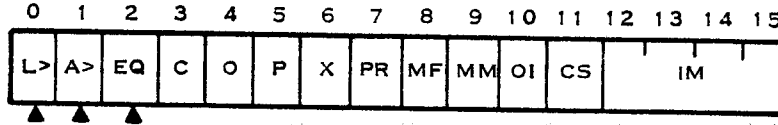
LABEL STD @WORD

Store the value in the floating point accumulator (R0-R3) in memory beginning at location WORD.

**Definition:** The value specified by FPA (R0-R3) is stored in the four words beginning at the address specified by the operand.

## STD, STPC

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



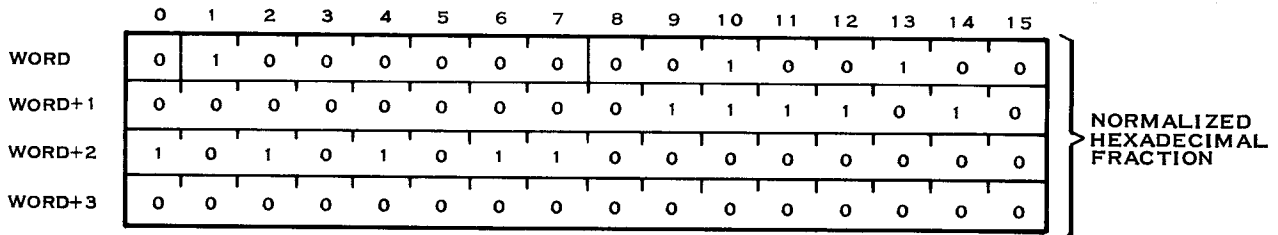
**Execution results:** FPA → (ga)

**Application notes:** The results of the STD instruction are compared to zero, and status register bits 0, 1, and 2 reflect the comparison. If T, is equal to three, the indicated register is incremented by eight.

The following is an example of the store double precision real instruction. The value contained in the four words of the double precision FPA (R0-R3), is .24007AAB<sub>16</sub>. The instruction

LABEL STD @WORD

stores the normalized fraction in the four words specified by WORD, as shown figuratively:

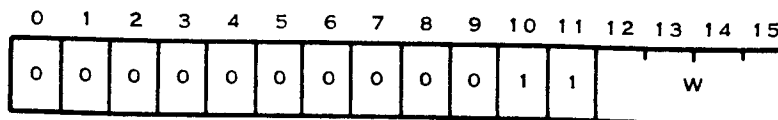


The logical greater than and arithmetic greater than bits of the status register are set; the equal bit is reset.

### 3.130 STORE PROGRAM COUNTER — STPC

**Opcode:** 0030

**Instruction Format:** 18



**Applicability:** Processor set C

**Syntax definition:**

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

## STPC, STR

**Example:**

LABEL STPC R3

The address of the STPC instruction (the program counter before it is incremented) is stored in workspace register three.

**Definition:** The contents of the program counter are placed in <wa>.

**Status bits affected:** None

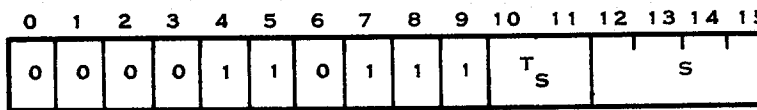
**Execution results:** PC → (wa)

**Application notes:** The STPC instruction stores the contents of the program counter before it is incremented.

### 3.131 STORE REAL — STR

**Opcode:** 0DC0

**Instruction Format:** 6



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..STRb..<ga>b..[<comment>]

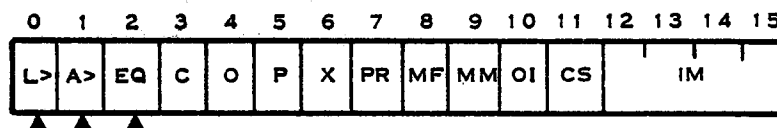
**Example:**

LABEL STR R6

Store the real numbers in the FPA in workspace registers six and seven.

**Definition:** The real number in the FPA (R0-R1) is stored at the source address.

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



## STR, STST

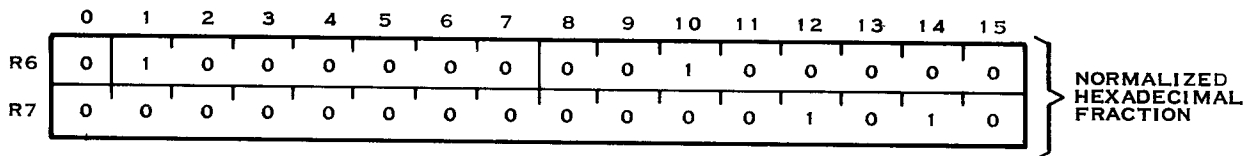
Execution results: FPA → (ga.)

Application notes: The result of the STR instruction is compared to zero, and status register bits 0, 1, and 2 reflect the comparison. If T<sub>r</sub> is equal to three, the indicated register is incremented by four.

The following is an example of the store real instruction. The value contained in the single precision FPA (R0-R1), after normalization, is .2000A<sub>16</sub>. The instruction

LABEL STR R6

stores the normalized fraction in workspace register six and workspace register seven, as shown figuratively:

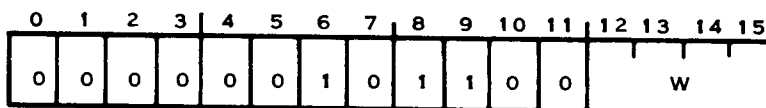


The logical greater than and arithmetic greater than bits of the status register are set; the equal bit is reset.

### 3.132 STORE STATUS — STST

Opcode: 02C0

Instruction Format: 18



Applicability: Processor set A

Syntax definition:

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

Example:

LABEL STST R7

Store the contents of the status register in workspace register seven.

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

## STST, STWP, SWPB

**Status bits affected:** None

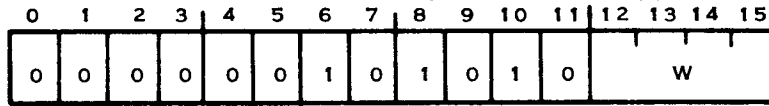
**Execution results:** (ST) → (wa)

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

### 3.133 STORE WORKSPACE POINTER — STWP

**Opcode:** 02A0

**Instruction Format:** 18



**Applicability:** Processor set A

**Syntax definition:**

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

**Example:**

LABEL STWP R6

Store the contents of the workspace pointer in workspace register six.

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

**Status bits affected:** None

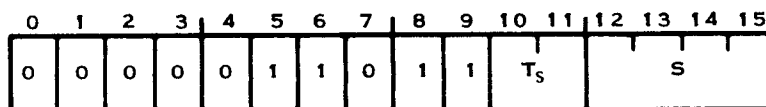
**Execution results:** (WP) → (wa)

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

### 3.134 SWAP BYTES — SWPB

**Opcode:** 06C0

**Instruction Format:** 6



**Applicability:** Processor set A

## SWPB, SWPM

*Syntax definition:*

[<label>]b..SWPBb..<ga,>b..[<comment>]

*Example:*

SWITCH SWPB R3

Move the least significant byte to the most significant byte, and the most significant byte to the least significant byte, in workspace register three.

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

*Status bits affected:* None

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

*Application notes:* Use the SWPB instruction to interchange bytes of an operand prior to executing various byte instructions. For example, if workspace register zero contains 2144<sub>16</sub> and memory location 2144<sub>16</sub> contains the value F312<sub>16</sub>, then the instruction

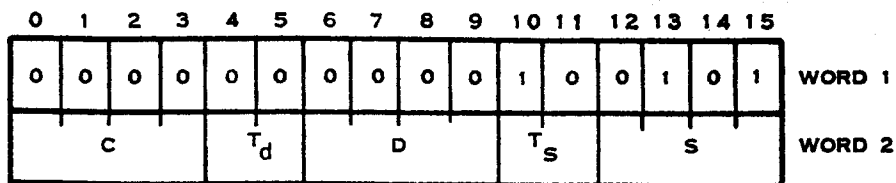
SWPB \*R0 +

changes the contents of workspace register zero to 2146<sub>16</sub> and the contents of memory location 2144<sub>16</sub> to 12F3<sub>16</sub>. The status register remains unchanged.

### 3.135 SWAP MULTIPLE PRECISION — SWPM

*Opcode:* 0025

*Instruction Format:* 11



*Applicability:* Processor set C

*Syntax definition:*

[<label>]b..SWPMb..<ga,><ga<sub>d</sub>>[,<cnt>]b..[<comment>]

## SWPM

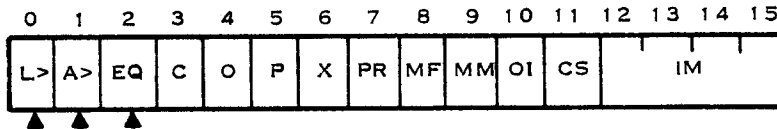
**Example:**

```
LABEL SWPM @TAB,@NTAB,8
```

Place the eight bytes beginning at location TAB into the eight bytes beginning at location NTAB, and place the eight bytes at NTAB into the eight bytes at TAB.

**Definition:** The multibyte value at the source address is swapped with the value at the destination address. The count of the number of bytes of precision of the values is determined by the <cnt> field. If <cnt> is zero or is not present, the four LSBs of R0 are used. If the four LSBs of R0 are zero, the count is 16.

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



**Execution results:** (ga<sub>s</sub>)→(ga<sub>d</sub>) ; (ga<sub>d</sub>)→(ga<sub>s</sub>)

**Application notes:** If T<sub>s</sub> and/or T<sub>d</sub> is equal to three, the indicated register is incremented by the byte count.

The resulting value at the source address of the SWPM instruction is compared to the resulting destination value. Status register bits 0, 1, and 2 reflect the comparison.

The following is an example of a swap multiple precision instruction. TAB is the starting address of an eight-byte string containing the values F312<sub>16</sub>, 2144<sub>16</sub>, 1276<sub>16</sub>, D430<sub>16</sub>, and location NTAB is the starting address of an eight-byte string containing the values 1134<sub>16</sub>, 8417<sub>16</sub>, 4480<sub>16</sub>, 5326<sub>16</sub>. The instruction

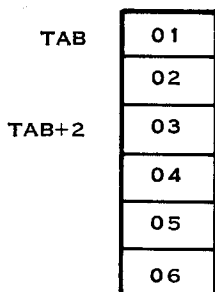
```
LABEL SWPM @TAB,@NTAB,8
```

changes the values of the eight-byte string starting at TAB to 1134<sub>16</sub>, 8417<sub>16</sub>, 4480<sub>16</sub>, and 5326<sub>16</sub>, and changes the values of the eight-byte string starting at NTAB to F312<sub>16</sub>, 2144<sub>16</sub>, 1276<sub>16</sub>, and D430<sub>16</sub>.

The arithmetic greater than bit of the status register is set, and the logical greater than and equal bits are reset.

## SWPM, SZC

An example of where the operands overlap is as follows. A six-byte string starting at location TAB is shown figuratively:



The instruction

```
SWPM @TAB,@TAB + 2,4
```

results in the data at TAB being 03, 04, 01, 02, 03, 04. The instruction

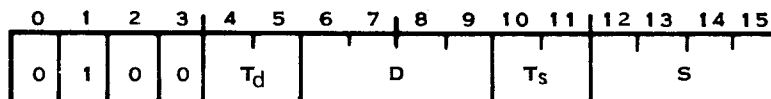
```
SWPM @TAB + 2, @TAB,4
```

results in the data at TAB being 03, 04, 05, 06, 01, 02.

### 3.136 SET ZEROS CORRESPONDING — SZC

Opcode: 4000

Instruction Format: 1



Applicability: Processor set A

Syntax definition:

```
[<label>]b..SZCb..<ga>,<ga>b..[<comment>]
```

Example:

```
LABEL SZC @MASK,R2
```

Reset the bits in workspace register 2 that correspond to the one bits of MASK to zero.

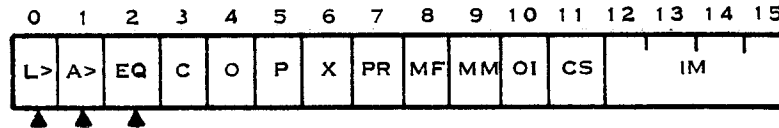
**Definition:** Set to a logic zero the bits in the destination operand that correspond to the bit positions equal to a logic one in the source operand. This operation is effectively an AND operation of



## SZC, SZCB

the one's complement of the source operand and the destination operand. The AU compares the resulting destination operand to zero and sets/resets the status bits to indicate the results of the comparison.

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



*Execution results:* Bits of (ga<sub>d</sub>) corresponding to bits of (ga<sub>s</sub>) equal to one are set to zero, (that is, [NOT (ga<sub>s</sub>) AND (ga<sub>d</sub>)]→(ga<sub>d</sub>))

*Application notes:* Use the SZC instruction to turn off flag bits or AND the contents of the one's complement of the source operand and the destination operand. For example, if workspace register five contains 6D03<sub>16</sub> and workspace register three contains D2AA<sub>16</sub>, then the instruction

SZC R5,R3

changes the contents of workspace register three to 92A8<sub>16</sub>, while the contents of workspace register five remain unchanged. This is shown as follows:

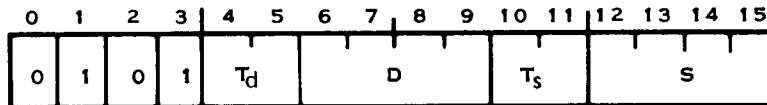
0 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1	(Source operand)
1 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0	(Destination operand)
1 0 0 1 0 0 1 0 1 0 1 0 1 0 0 0	(Destination operand result)

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

### 3.137 SET ZEROS CORRESPONDING, BYTE — SZCB

*Opcode:* 5000

*Instruction Format:* 1



*Applicability:* Processor set A

*Syntax definition:*

[<label>]b..SZCBb..<ga<sub>s</sub>>,<ga<sub>d</sub>>b..[<comment>]

## SZCB, TB

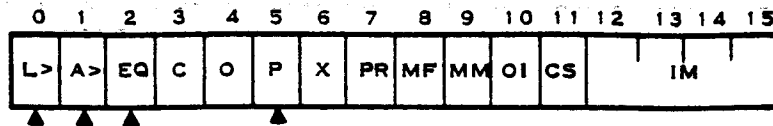
**Example:**

```
LABEL SZCB @MASK,@CHAR
```

Reset the bits in a byte at location CHAR which correspond to the one bits in a byte at location MASK to zero.

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

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



**Execution results:** Bits of  $(ga_d)$  corresponding to bits of  $(ga_s)$  equal to one are set to zero, that is,  $[NOT (ga_s) AND (ga_d) \rightarrow (ga_d)]$ .

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

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

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

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

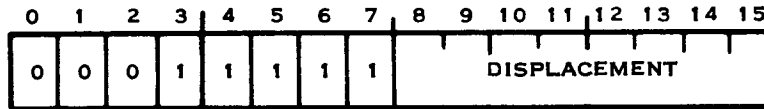
For this example, the logical greater than and arithmetic greater than status bits set, while the equal and odd parity status bits reset.

### 3.138 TEST BIT — TB

**Opcode:** 1F00

## TB, TCMB

**Instruction Format: 2**



**Applicability:** Processor set A, privileged instruction

**Syntax definition:**

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

**Example:**

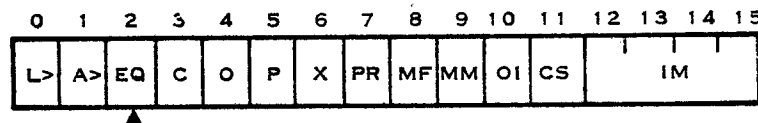
CHECK TB 7

Read bit 7 on CRU and set the equal status bit with the value read.

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

When the privileged mode bit (bit 7) of the ST register is set to zero, the TB instruction executes normally. When bit 7 is set to one and the effective CRU address is equal to or greater than E00<sub>16</sub>, an error interrupt occur and the instruction is not executed.

**Status bits affected:** Equal



**Execution results:** The equal bit is set to the value of the CRU bit addressed by the sum of the contents of workspace register 12 (bits 3 through 14) + displacement.

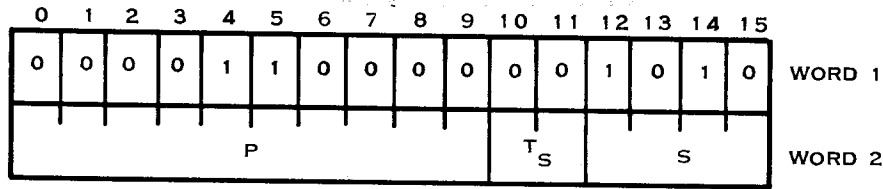
**Application notes:** The TB CRU line logic level test transfers the logic level from the indicated CRU line to the equal status bit without modification. If the CRU line tested is set to a logic one, the equal status bit sets to a logic one and if the line is zero, resets to a zero. JEQ will then transfer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

### 3.139 TEST AND CLEAR MEMORY BIT — TCMB

**Opcode:** 0C0A

## TCMB

Instruction Format: 14



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..TCMBb..<ga,>[,<pos>]b..[<comment>]

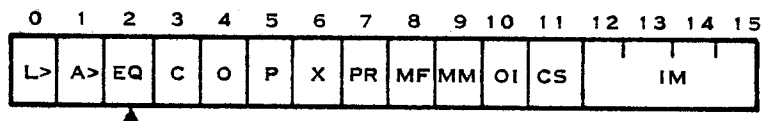
**Example:**

LABEL TCMB @BITMAP(R3),6

Reset the value of bit 6 of the contents of location BITMAP plus workspace register three.

**Definition:** The value of the bit at position <pos> in the bit string starting at the source address is copied into status register bit 2 and then set to zero. If the position operand is not present, the default is 3FF<sub>16</sub>, the maximum value of the position. When the position is 3FF<sub>16</sub>, all 16 bits of workspace register zero are used to determine the bit position, so that the maximum possible position is FFFF<sub>16</sub>. TCMB can work as an interlock instruction in multiprocessor systems; that is, only the CPU executing the TCMB instruction is allowed to access the source memory address.

**Status bits affected:** Equal



**Execution results:** (ga, + pos)→ST2  
0→(ga, + pos)

**Application notes:** If T<sub>s</sub> is equal to three, the indicated register is incremented by one. The previous value of the affected bit is copied into status register bit 2.

The following is an example of the test and clear memory bit instruction. BITMAP contains the value 4E28<sub>16</sub>. The instruction

LABEL TCMB @BITMAP,6

copies the previous value of bit 6 (a one) into the equal bit of the status register (ST2) and clears bit 6 of the value in BITMAP. The value of BITMAP, after execution of this instruction, is 4C28<sub>16</sub>.

## TCMB, TMB

**Multiple CPU Systems:** Several 990/12 CPUs can be connected together to create a multiple CPU system. In these systems, the CPUs must share a common memory. Simultaneous access attempts to memory by more than one CPU can result in a loss of data. To prevent this conflict, software "memory busy" flags in memory can be used. When a processor desires access to memory, it must first check the flag to determine if any other processor is actively using memory. If memory is not busy, the processor sets the busy flag to lock out other processors and begins its memory transfers. When the processor is finished with memory, it clears the busy flag to allow access from other processors.

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

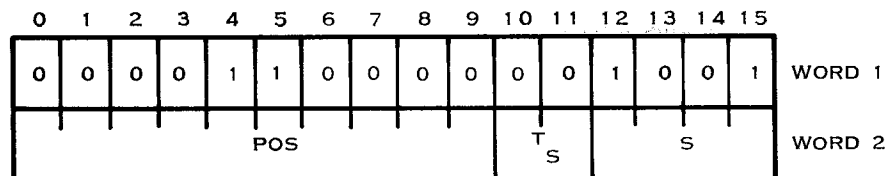
### NOTE

When workspace registers are cached, TCMB in direct register addressing cannot detect a flag changed in the corresponding memory location by another processor. Therefore, TCMB can only be used with indirect, indirect autoincrement, indexed, and symbolic addressing modes when used to control memory access in a multiple CPU system.

### 3.140 TEST MEMORY BIT — TMB

**Opcode:** 0C09

**Instruction Format:** 14



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..TMBb..<ga>[,<pos>]b..[<comment>]

## TMB, TS

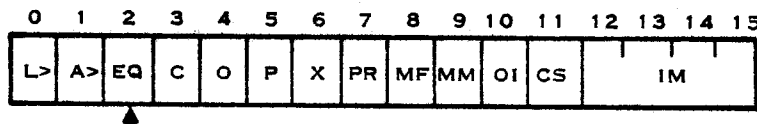
**Example:**

LABEL TMB @BITMAP(R3),6

Test bit 6 at the location specified by BITMAP plus workspace register three.

**Definition:** The bit at position <pos> in the bit string starting at the source address is copied into status register bit 2. If <pos> is all ones, the position is taken from all 16 bits of workspace register zero. When the position is not present, it defaults to all ones (>3FF).

**Status bits affected:** Equal



**Execution results:** ((ga,) + pos)→ST2 in status register

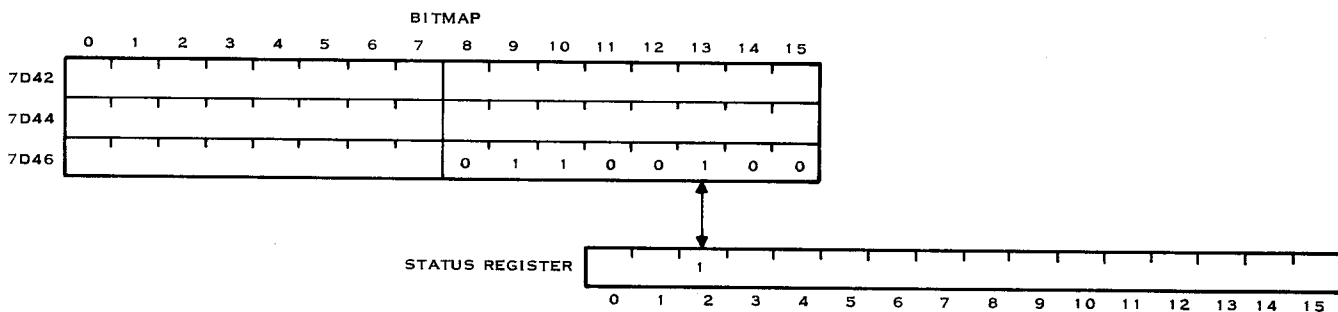
**Application notes:** If T, is equal to three, the indicated register is incremented by one.

Status register bit 2 is set or cleared.

The following is an example of the test memory bit instruction. BITMAP points to memory address 7D42<sub>16</sub>, and workspace register three contains the value of five. The instruction

LABEL TMB @BITMAP(R3),5

copies the value of bit 5 of the source address, in this example location 7D47<sub>16</sub>, into bit 2 of the status register; shown figuratively:



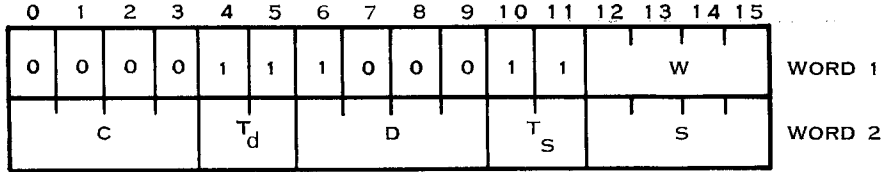
Status register bit 2 is the only bit of the status register affected.

### 3.141 TRANSLATE STRING — TS

**Opcode:** 0E30

**TS**

**Instruction Format:** 12



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..TSb..<ga>,<ga>,<cnt>[,<ckpt>]b..<comment>

Trailing commas on the operand list may be omitted. The checkpoint register may be omitted from the instruction if a default has been declared using the CKPT assembler directive

**Example:**

LABEL TS @NTAB,@TAB,10,R1

Replace the TAB values with the corresponding NTAB values.

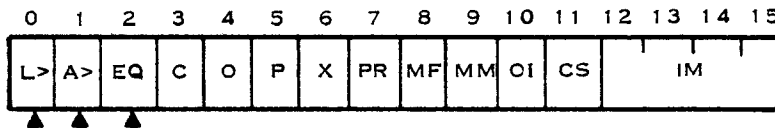
**Definition:** The bytes in the string starting at the destination address are used as indexes into a 256-byte translation table at the source address. Each byte in the destination string is replaced by its respective value from the table pointed to by the source address. The length of the string may be specified by the <cnt> field, by workspace register zero, or as a tagged string (if <cnt> = 0 and R0 = >FFFF).

If the length of the string is 16 bytes or more, the checkpoint register is used for interrupts. If an interrupt occurs during execution, checkpoint data is stored in the checkpoint register. After the interrupt is serviced, execution continues from where it stopped. Upon completion of the instruction, the checkpoint register is set to -1.

The checkpoint register value plus one is used as an initial index into the string. To access the first byte (lowest address) in the string, the checkpoint register must be set to -1 before the instruction is executed.

If the string length is zero (or one for a tagged string), no data is translated (and status bits 0 through 2 are set to zero). If the string is tagged, the tag byte is not translated.

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



## TS

The translated string is compared to zero (as a signed, two's complement value), and the status bits 0, 1, and 2 are set accordingly. The tag byte in a tagged string is not used in the comparison.

**Execution results:** For each byte in  $ga_d$ ,

$$(ga_s + (ga_d)) \rightarrow ga_d$$

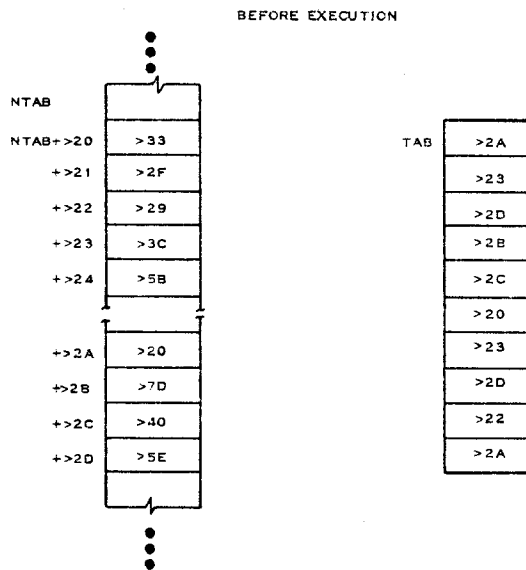
**Application notes:** If  $T_s$  is equal to three, the indicated register is incremented by the string length. If  $T_s$  is equal to three, the indicated register is incremented by 256.

The following is an example of the translate string instruction. NTAB points to a 256-byte table and the TAB points to a string of bytes, that are indexes into the table pointed to by NTAB. The instructions

```

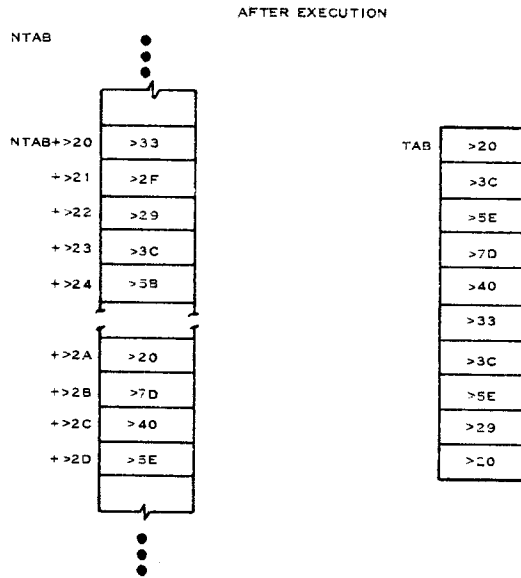
        SETO      R1
        LABEL TS @NTAB,@TAB,10,R1
    
```

replace the value in the first ten bytes of TAB, with the value of the respective bytes in NTAB. For instance, if the first byte of TAB contains the hexadecimal representation for an asterisk (\*), which is 2A, then the index into the table would be  $NTAB + 2A$ . If the value in the byte at  $NTAB + 2A$  is the hexadecimal representation of a blank (b), which is 20<sub>16</sub>, this value, after execution of the instruction, replaces the index value of the first byte of TAB. This example is shown figuratively:





# TS, TSMB

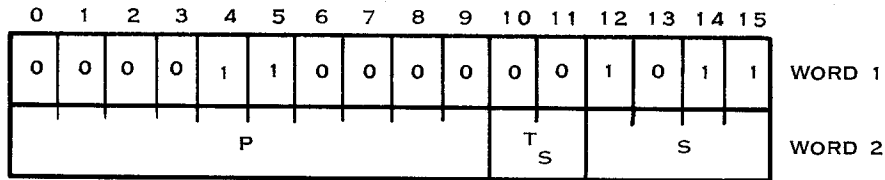


The logical greater than and arithmetic greater than bits of the status register are set, and the equal bit is reset.

### 3.142 TEST AND SET MEMORY BIT — TSMB

Opcode: 0C0B

Instruction Format: 14



Applicability: Processor set C

Syntax definition:

[<label>]b..TSMBb..<ga,>[,<pos>]b..[<comment>]

Example:

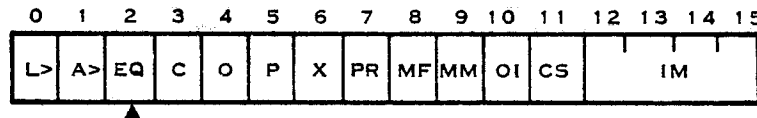
LABEL TSMB @ BITMAP(R3),6

Set bit 6 of the contents of location BITMAP plus workspace register three to one.

## TSMB

**Definition:** The bit at position <pos> in the bit string starting at the source address is placed in status register bit 2 and then set to one. If the position operand is not present, it defaults to  $3FF_{16}$ , the maximum value of the position. When the position is  $3FF_{16}$ , all 16 bits of workspace register zero are used to determine the bit position. TSMB can work as an interlock instruction in multiprocessor systems; that is, only the CPU executing the TSMB instruction is allowed to access the source memory address.

**Status bits affected:** Equal



**Execution results:**  $(ga_r + pos) \rightarrow ST2$   
 $1 \rightarrow (ga_r + pos)$

**Application notes:** If T, is equal to three, the indicated register is incremented by one.

The following is an example of the test and set memory bit instruction. BITMAP contains the value  $4C28_{16}$ . The instruction

```
LABEL TSMB @BITMAP,6
```

resets the equal bit of the status register (ST2), and sets bit 6, of the value of BITMAP, to one. The value of BITMAP, after execution of this instruction, is  $4E28_{16}$ .

**Multiple CPU Systems:** Several 990/12 CPUs can be connected together to create a multiple CPU system. In these systems, the CPUs must share a common memory. Simultaneous access attempts to memory by more than one CPU can result in a loss of data. To prevent this conflict, software "memory busy" flags in memory can be used. When a processor desires access to memory, it must first check the flag to determine if any other processor is actively using memory. If memory is not busy, the processor sets the busy flag to lock out other processors and begins its memory transfers. When the processor is finished with memory, it clears the busy flag to allow access from other processors.

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

## TSMB, X

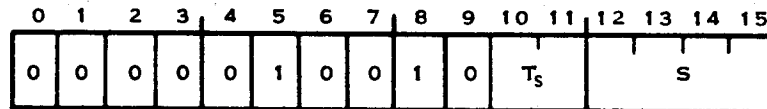
### NOTE

When workspace registers are cached, TSMB in direct register addressing cannot detect a flag changed in the corresponding memory location by another processor. Therefore, TSMB can only be used with indirect, indirect autoincrement, indexed, and symbolic addressing modes when used to control memory access in a multiple CPU system.

### 3.143 EXECUTE — X

*Opcode:* 0480

*Instruction Format:* 6



*Applicability:* Processor set A

*Syntax definition:*

[<label>]b..Xb..<ga,>b..[<comment>]

*Example:*

LABEL X @TABLE(R5)

Execute the instruction at the address specified by adding the contents of register 5 to the value of TABLE.

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

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

*Execution results:* An instruction at <ga, >is executed instead of the X instruction.

*Application notes:* Use the X instruction to execute the source operand as an instruction. This is primarily useful when the instruction to be executed is dependent upon a variable factor.

## X, XF

### NOTE

Execution of instructions from the workspace cache on a 990/12 computer causes the workspace cache to be disabled with corresponding degradation of performance.

#### 3.143.1 X Applications

The execute instruction may be used to execute an instruction that is not in sequence without transferring control to the desired instruction. One useful application is to execute one of a table of instructions, selecting the desired instruction by using an index into the table of instructions. The computed value of the index determines which instruction is executed.

A table of shift instructions is an example of the use of the X instruction. Place the following instructions at location TBLE:

TBLE	SLA	R6,3	SHIFT WORKSPACE REGISTER 6
	SLA	R7,3	SHIFT WORKSPACE REGISTER 7
	SLA	R8,3	SHIFT WORKSPACE REGISTER 8
TABEND	EQU	\$-2	

A character is placed in the most significant byte of workspace register five to select the workspace register to be shifted to the left three bit positions. ASCII characters A, B, and C specify shifting workspace registers six, seven, and eight, respectively. Other characters are ignored. The following code performs the selection of the shift desired:

SRL	R5,8	MOVE TO LOWER BYTE
AI	R5,-'A'	SUBTRACT TABLE BIAS
JLT	NOSHFT	ILLEGAL
SLA	R5,1	MAKE IT A WORD INDEX
CI	R5,TABEND	-TBLE
JGT	NOSHFT	ILLEGAL
X	@TBLE(R5)	
NOSHFT	EQU	\$

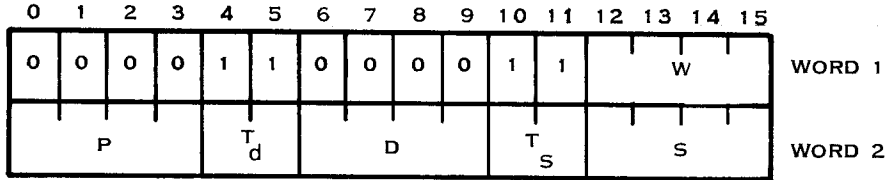
When using the X instruction, if the substituted instruction contains a T<sub>s</sub> field or a T<sub>a</sub> field that results in a two word instruction, the computer accesses the word following the X instruction as the second word, not the word following the substituted instruction. When the substituted instruction is a jump instruction with a displacement, the displacement must be computed from the X instruction, not from the substituted instruction.

#### 3.144 EXTRACT FIELD — XF

**Opcode:** 0C30

**XF**

**Instruction Format:** 16



**Applicability:** Processor set C

**Syntax definition:**

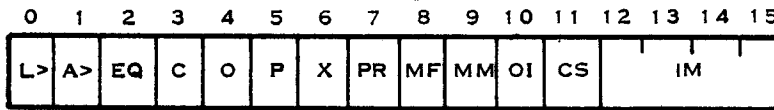
[<label>]b..XFb..<ga<sub>s</sub>>,<ga<sub>d</sub>>,<pos>,<wid>b..[<comment>]

**Example:**

LABEL XF @CORE,@OPUT,(3,6)

**Definition:** The bit field of width <wid> beginning at position <pos> in the word at the source address is stored, right-justified, in the word at the destination address. Leading zeros fill the most significant bit positions at the destination. If either <pos> or <wid> is zero, the position or width is taken from workspace register zero. In this case, bits 4 through 7 of workspace register zero indicate the position and bits 12 through 15 determine the width. If bits 4 through 7 are zero, the position is zero. If bits 12 through 15 are zero, the width is 16. If <pos> plus <wid> is greater than 16, the remainder of the extracted value is taken from the next word in memory, starting at the most significant bit. The source and destination operands must start on a word boundary.

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



**Execution results:** (ga<sub>s</sub>) → (ga<sub>d</sub>)

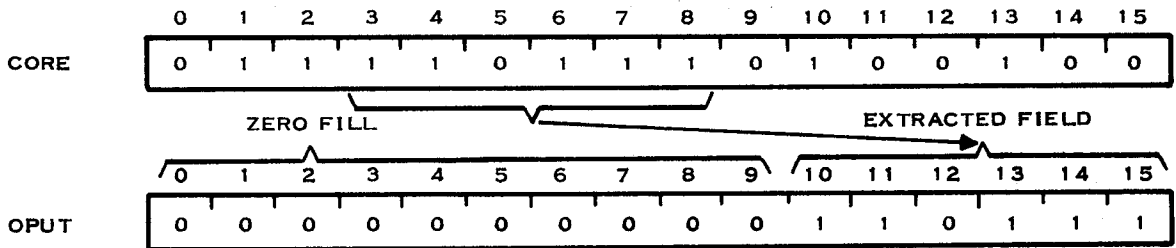
**Application notes:** The result of the XF instruction stored in the memory location specified by <ga<sub>d</sub>> is compared to zero and status register bits 0, 1, and 2 reflect the results of the comparison. If T<sub>s</sub> or T<sub>d</sub> is equal to three, the indicated register is incremented by two.

The following is an example of the extract field instruction. CORE contains the value 7BA4<sub>16</sub>. The instruction

LABEL XF @CORE,@OPUT,(3,6)

## XF, XIT

places the six-bit value of CORE, starting at bit position 3, into OPUT, right-justified. The most significant bit positions of OPUT are filled with zeros. The new value of OPUT is 0037<sub>16</sub>. The example is shown figuratively:

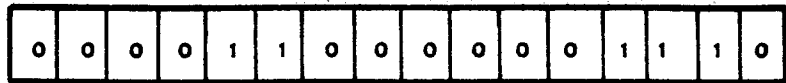


The logical greater than and arithmetic greater than bits of the status register are set, and the equal bit of the status register is reset.

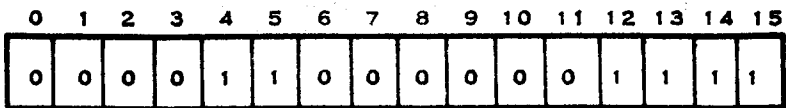
### 3.145 EXIT FROM FLOATING POINT INTERPRETER — XIT

Opcode: 0C0E or 0C0F

Instruction Format: 7



or



Applicability: Processor set C

Syntax definition:

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

Example:

LABEL XIT

Exit from the floating point interpreter.

Definition: The XIT instruction is used to exit from the floating point interpreter. It is effectively a NOP.

## XIT, XOP

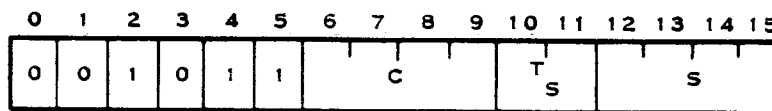
Status bits affected: None

Application notes: XIT is generated by the S option when compiling a FORTRAN program. This instruction is treated as a NOP.

### 3.146 EXTENDED OPERATION — XOP

Opcode: 2C00

Instruction Format: 9



Applicability: Processor set A

Syntax definition :

[<label>]b..XOPb..<ga>,<cnt>b..[<comment>]

Example:

LABEL XOP @DATA,12

Execute extended operation 12 using data stored beginning at location DATA

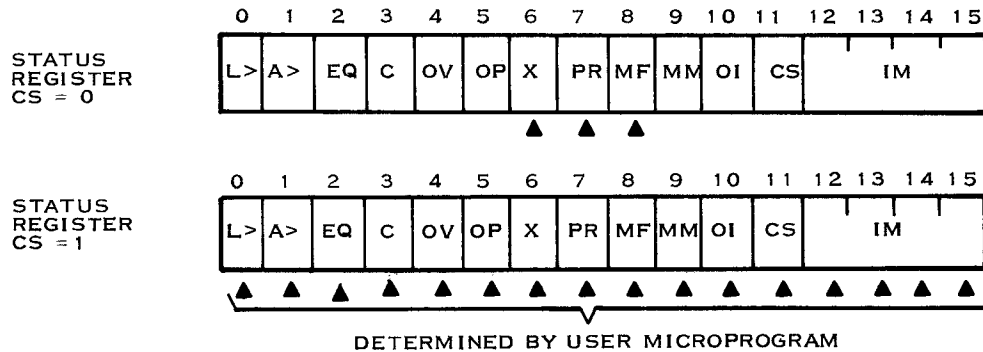
**Definition:** If the status register bit 11 (WCS enable) is zero, the <cnt> field specifies the extended operation transfer vector in memory. The two memory words at that location contain the WP contents and PC contents for the XOP instruction subroutine. The memory location for these two words is derived by multiplying the <cnt> field contents by four and adding the product to 0040<sub>16</sub>. The two memory words at this location must contain the necessary WP and PC values prior to the XOP instruction execution.

The effective address of the source operand is placed in workspace register 11 of the XOP workspace. The WP contents are placed in workspace register 13 of the XOP workspace. The PC contents are placed in workspace register 14 of the XOP workspace. The status register contents are placed in workspace register 15 of the XOP workspace. Control is transferred to the new PC address and the XOP is executed. (XOP execution is similar to an interrupt trap execution.) When the extended operation is executed, the privileged mode and map file bits in the status register are set to zero.

If the XOP instruction is executed with the WCS (writable control store) enable bit (status register bit 11) set to one, the effective (source) address is calculated and deposited in a hardware register internal to the CPU. Control is then transferred to the microcode instruction in the WCS word specified by the <cnt> field. Refer to the *Model 990 Computer MDS-990 Microcode Development System Programmer's Guide*.

## XOP

**Status bits affected:** If status register bit 11 is zero, the extended operation, privileged mode, and map file bits are affected. If status register bit 11 is one, the status register may be affected by the microcode being executed in WCS.



**Execution results:** If status register bit 11 is zero

(ga<sub>i</sub>) → (workspace register 11)  
 (0040<sub>16</sub> + (<cnt>)\*4) → (WP)  
 (0042<sub>16</sub> + (<cnt>)\*4) → (PC)  
 (WP) → (workspace register 13)  
 (PC) → (workspace register 14)  
 (ST) → (workspace register 15)  
 0 → ST8    0 → ST9  
 0 → ST7    0 → ST10  
 1 → ST6    0 → ST11

If status register bit 11 is one

(ga<sub>i</sub>) → CPU register

**Application notes:** When WCS is enabled (ST 11 = 1), the table of transfer vector subroutine addresses, and the subroutines within WCS, must be set up by the LCS instruction or the operating system prior to execution of the XOP instruction. Entry to these subroutines might be through microcode jumps in the first 16 words of WCS. Return to the next machine instruction must be handled by microcode.

### 3.146.1 Extended Operations

Extended operation instructions permit the extension of the existing instruction set to include additional instructions. In the Model 990 Computer, the instructions are implemented by software routines. Interface between a user program and the standard TI executives is implemented as XOP 15.

The extended operation instruction also implements microcode routines in the writable control store. For a writable control store routine to be executed, bit 11 of the status register must be set to one. The writable control store should be loaded by use of the load control store (LCS) instruction, or by other means detailed in the *Model 990/12 Computer MDS-990 Microcode Development System Programmer's Guide*. Execution of the XOP instruction with status bit 11 set to zero implements the standard software extended operations.



**XOP**

Memory locations 0040<sub>16</sub> through 007E<sub>16</sub> are used for XOP vectors for software-implemented XOPs. Vector contents are user-supplied WP and PC addresses for the XOP routine workspace and starting address. Table 3-7 contains the addresses and contents of the 16 XOP vectors. These vectors must be supplied and loaded prior to the XOP instruction execution.

**Table 3-7. XOP Vectors**

Memory Address	XOP Number	Vector Contents
0040	0	WP address for XOP workspace
0042	0	PC address for XOP routine
0044	1	WP address for XOP workspace
0046	1	PC address for XOP routine
0048	2	WP address for XOP workspace
004A	2	PC address for XOP routine
004C	3	WP address for XOP workspace
004E	3	PC address for XOP routine
0050	4	WP address for XOP workspace
0052	4	PC address for XOP routine
0054	5	WP address for XOP workspace
0056	5	PC address for XOP routine
0058	6	WP address for XOP workspace
005A	6	PC address for XOP routine
005C	7	WP address for XOP workspace
005E	7	PC address for XOP routine
0060	8	WP address for XOP workspace
0062	8	PC address for XOP routine
0064	9	WP address for XOP workspace
0066	9	PC address for XOP routine
0068	10	WP address for XOP workspace
006A	10	PC address for XOP routine
006C	11	WP address for XOP workspace
006E	11	PC address for XOP routine
0070	12	WP address for XOP workspace
0072	12	PC address for XOP routine
0074	13	WP address for XOP workspace
0076	13	PC address for XOP routine
0078	14	WP address for XOP workspace
007A	14	PC address for XOP routine
007C	15	WP address for XOP workspace
007E	15	PC address for XOP routine

When the program module contains a software XOP instruction, the AU locates the XOP WP and PC words in the XOP reserved memory location and loads the WP and PC. When the WP and PC are loaded, the AU transfers control to the XOP instruction set through a context switch. When the context switch is complete, the XOP workspace contains the calling routine return data in WRs 13, 14, and 15.

# XOP

The XOP instruction passes one operand to the XOP (input to the XOP routine in workspace register 11 of the XOP workspace). At the completion of the software XOP, the XOP routine should return to the calling routine with an RTWP instruction that restores the execution environment of the calling routine to that in existence at the call to the XOP.

An example of a software XOP, shown in Figure 3-4, causes XOP number two to be executed on the data stored at the address contained in workspace register one of the calling program module. Prior to the execution of the XOP, the PC contains the address of the XOP \*R1,2 instruction and the WP contains the address of the calling program workspace. At this point, the PC increments by two, to 922, and the XOP is executed. This execution is a context switch in which the XOP routine gains control of the execution sequence. Workspace register one of the calling program module contains the data address for the operand that is passed to the XOP routine.

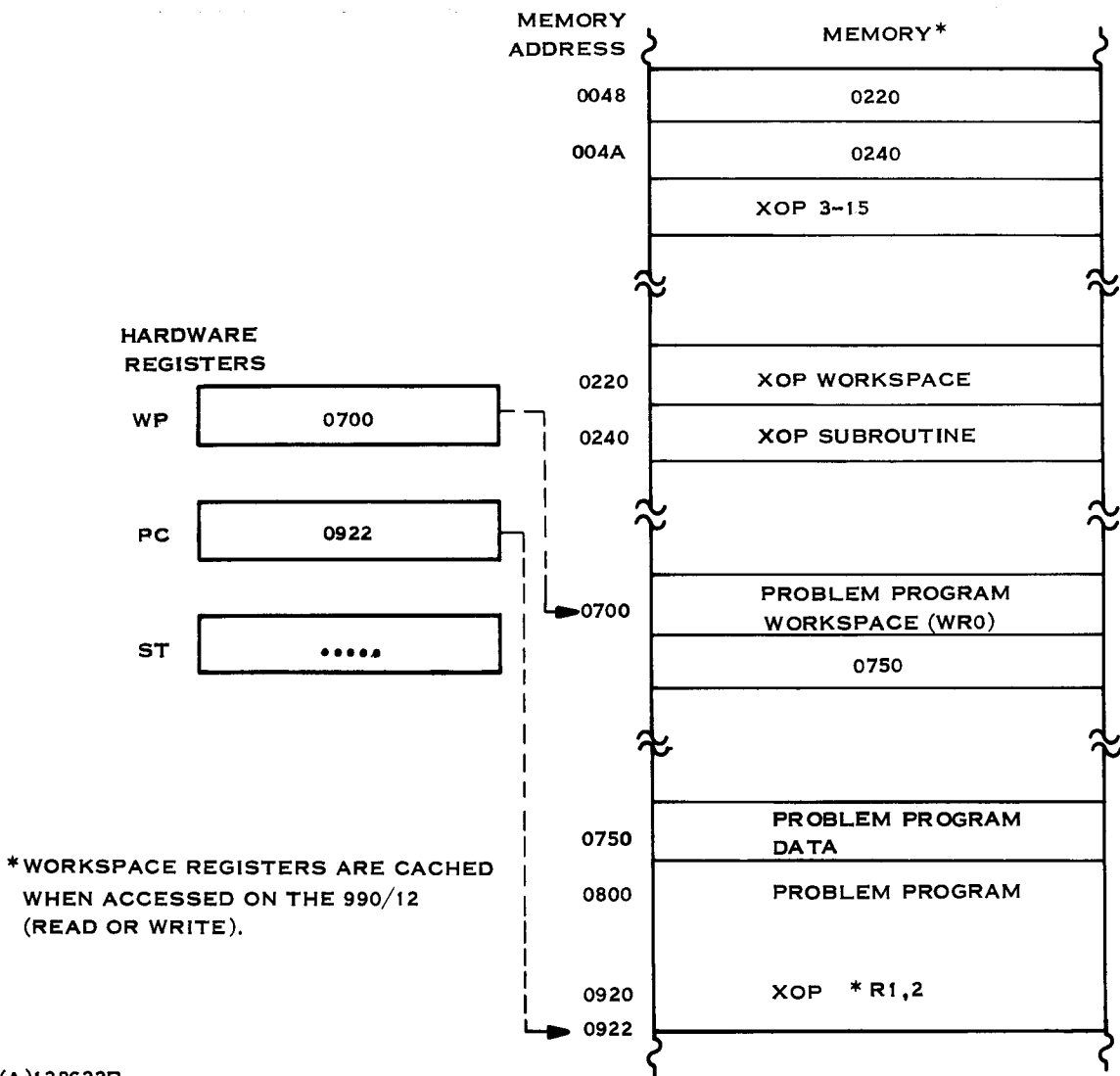


Figure 3-4. Extended Operation Example

# XOP

After the context switch is complete and the XOP subroutine is in control (Figure 3-5), the PC contains the starting address of the XOP subroutine and the WP contains the address of the XOP subroutine workspace. Workspace register 11 of the XOP subroutine contains the effective address of the data to be used as an operand. Workspace registers 13, 14, and 15 contain the return control information, which is used to return control to the main program module when the XOP subroutine completes execution.

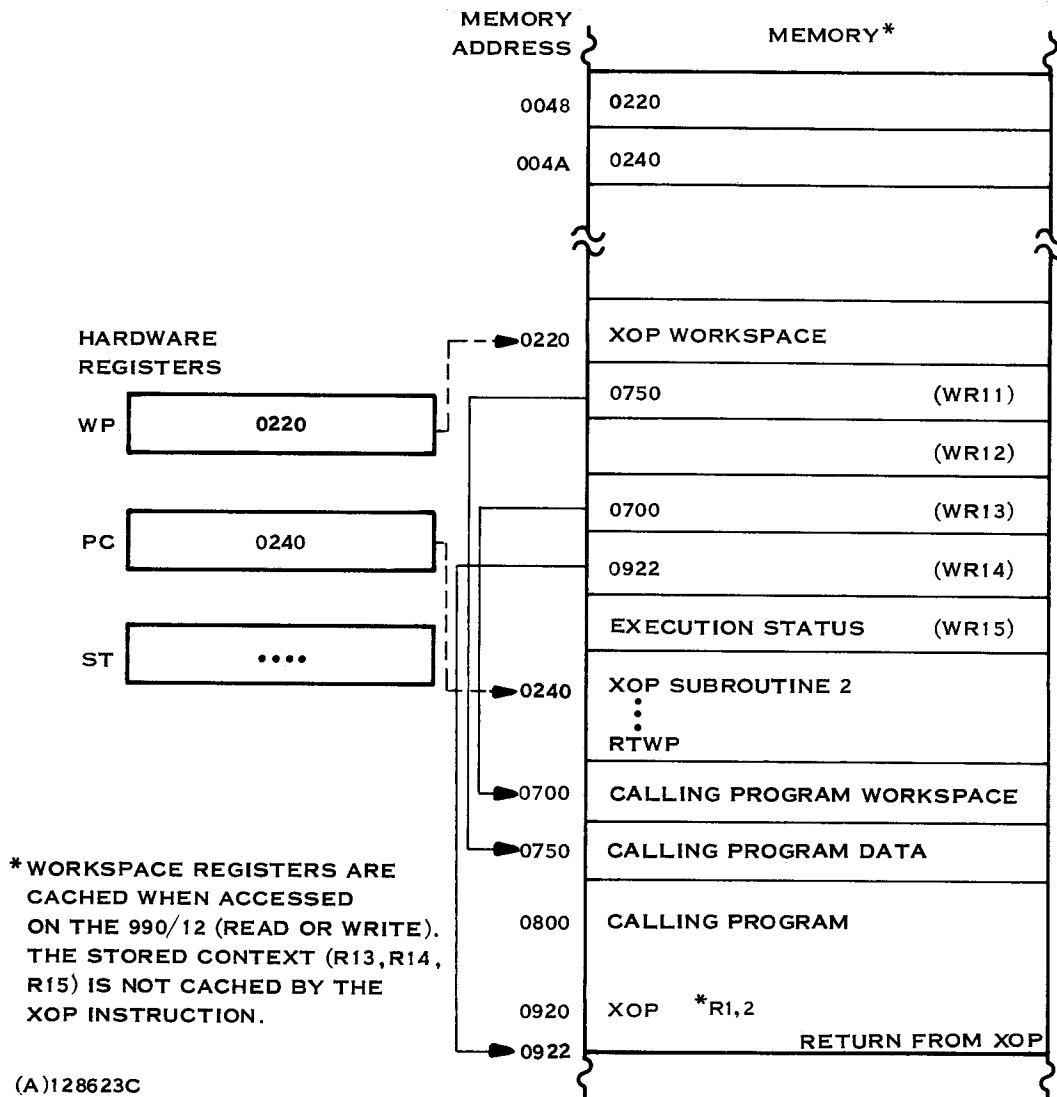


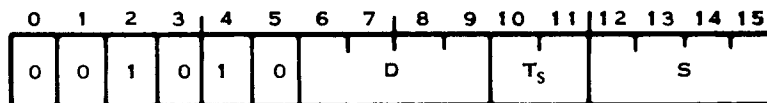
Figure 3-5. Extended Operation Example after Context Switch

# XOR

## 3.147 EXCLUSIVE OR — XOR

Opcode: 2800

Instruction Format: 3



Applicability: Processor set A

Syntax definition:

[<label>]b...XORb...<ga<sub>s</sub>>,<wa<sub>d</sub>>b...[<comment>]

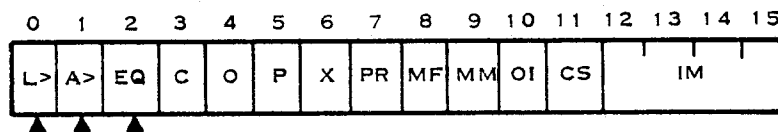
Example:

LABEL XOR @WORD,R3

Perform the logical exclusive OR of the memory word at location WORD and the contents of workspace register three. Place the result in workspace register three.

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

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



Execution results: (ga<sub>s</sub>) XOR (wa<sub>d</sub>) → (wa<sub>d</sub>)

(that is, [(ga<sub>d</sub>) AND NOT (wa<sub>d</sub>)] OR [(wa<sub>d</sub>) AND NOT (ga<sub>d</sub>)] → (wa<sub>d</sub>)

**Application notes:** Use the XOR instruction to perform an exclusive OR on two-word operands. For example, if workspace register two contains D2AA<sub>16</sub> and location CHANGE contains the value 6D03<sub>16</sub>, then the instruction

XOR @CHANGE,R2

## XOR, XORM

results in the contents of workspace register two changing to BFA9<sub>16</sub>. Location CHANGE remains 6D03<sub>16</sub>. This is shown as follows:

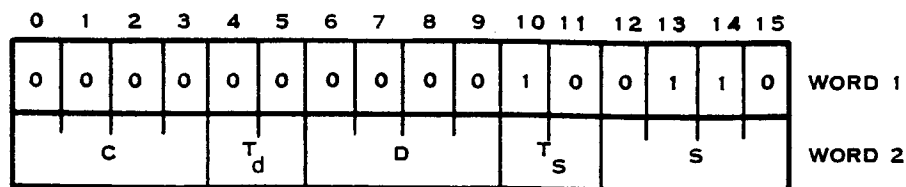
0 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1	(Source operand)
1 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0	(Destination operand)
1 0 1 1 1 1 1 1 0 1 0 1 0 0 1	(Destination operand result)

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

### 3.148 EXCLUSIVE OR MULTIPLE PRECISION — XORM

Opcode: 0026

Instruction Format: 11



Applicability: Processor set C

Syntax definition:

[<label>]b..XORMb..<ga>,<ga>[,<cnt>]b..[<comment>]

Example:

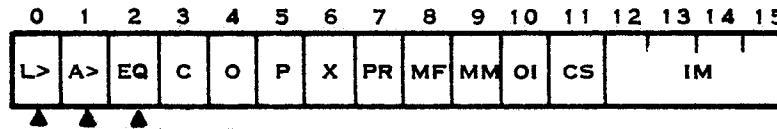
LABEL XORM @TAB,@NTAB,14

Perform the logical exclusive OR between the 14 bytes starting at location TAB and the 14 bytes starting at location NTAB. The result is placed in the 14 bytes starting at location NTAB.

**Definition:** A bit-by-bit exclusive OR operation is performed between the multibyte two's complement integer at the source address and the corresponding bits of the multibyte two's complement integer at the destination address. The result is placed in the destination address. The <cnt> is the number of bytes of precision of the integer. If <cnt> equals zero, the count is taken from the four LSBs of workspace register zero. If the four LSBs of workspace register zero are zero, the count is 16.

## XORM

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



Execution results:  $(ga_n) \text{ XOR } (ga_d) \rightarrow (ga_d)$

Application notes: The result of the XORM instruction is compared to zero and the status register bits 0, 1, and 2 indicate the results of the comparison. If  $T_n$  and/or  $T_d$  is equal to three, the indicated register is incremented by the byte count.

The following is an example of the exclusive OR multiple precision instruction. TAB addresses a 16-byte string and NTAB addresses a 16-byte string, as shown figuratively:

TAB	1F		NTAB	14
	33			0A
	B7			88
	5C			4C
	25			17
	77			9B
	13			BB
	39			65
	A5			D3
	E0			F4
	AA			77
	99			00
	4C			C2
	EF			33
	DE			B7
	A2			C9

**XORM, XV**

The instructions

```

CLR    R0
XORM  @TAB,@NTAB,0

```

perform a bit-by-bit exclusive OR operation between the multibyte two's complement integer at TAB and the corresponding bits of the multibyte two's complement integer at NTAB, placing the results in NTAB. The results of this instruction are shown figuratively:

NTAB	0B
	39
	3F
	10
	32
	EC
	A8
	5C
	76
	14
	DD
	99
	8E
	DC
	69
6B	

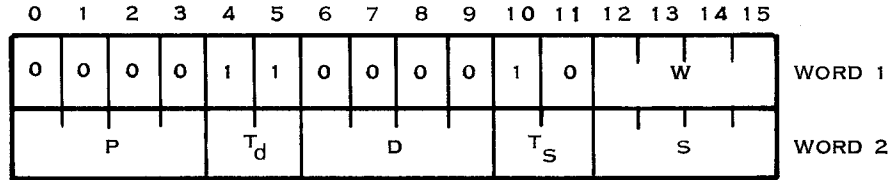
The logical greater than and arithmetic greater than bits of the status register are set, and the equal bit is reset.

**3.149 EXTRACT VALUE — XV**

*Opcode:* 0C20

# XV

Instruction Format: 16



**Applicability:** Processor set C

**Syntax definition:**

[<label>]b..XVb..<ga<sub>s</sub>>,<ga<sub>d</sub>>,<pos>,<wid>b..[<comment>]

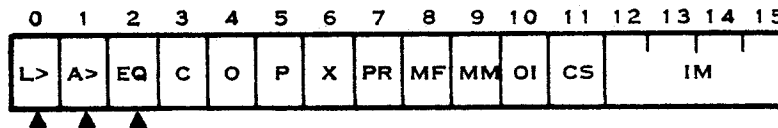
**Example:**

LABEL XV @WORD,@NWORD,(7,7)

Extract a seven-bit value, beginning with bit 7 from the word at location WORD and place the value in the word at location NWORD.

**Definition:** The bit field of width <wid>, beginning at position <pos>, in the word at the source address is stored right-justified in the word at the destination address. The MSB of the extracted field fills the vacant bit positions at the destination. If either <pos> or <wid> are zero, the position or width is taken from workspace register zero. In this case, bits 4 through 7 of workspace register zero indicate the position and bits 12 through 15 determine the width. If bits 4 through 7 are zero, the position is zero. If bits 12 through 15 are zero, the width is 16. If <pos> plus <wid> is greater than 16, the remainder of the extracted value is taken from the next word in memory, starting at the most significant bit. The source and destination operands must start on a word boundary.

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



**Execution results:** (ga<sub>s</sub>) → (ga<sub>d</sub>)

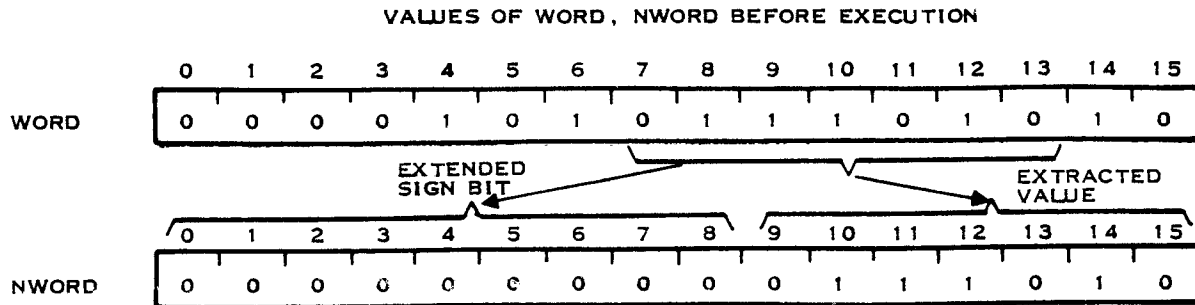
**Application notes:** The result of the XV instruction, stored in memory at the address specified by <ga<sub>d</sub>>, is compared to zero, and the status register bits 0, 1 and 2 reflect the results of the comparison. If T<sub>s</sub> or T<sub>a</sub> is equal to three, the indicated register is incremented by two.

The following is an example of the extract value instruction. WORD contains the value 0AEA<sub>16</sub>. The instruction

LABEL XV @WORD,@NWORD,(7,7)



extracts the seven-bit value starting at bit position 7 of WORD, and places the seven-bit value in NWORD, right justified (bits 9 through 15). The MSB of the extracted value (bit seven) fills the vacant bit positions (bits 0 through 8). The new value of NWORD is 003A<sub>16</sub>, shown figuratively:



The logical greater than and arithmetic greater than bits of the status register are set; the equal bit of the status register is reset.

# Application Notes

---

## 4.1 GENERAL

This section provides information about and examples of specific applications of the 990 assembly language. This section is divided into four parts. The first part describes ways of implementing, passing data to, and controlling subroutines. The second part contains information for upgrading a program written on a 990/10 to run on a 990/12 to take maximum advantage of the 990/12 features. The third part describes reentrant programming techniques. The fourth part details considerations for writing a program that does not run under an operating system and for writing device service routines for devices not supported by the standard operating systems. More detailed information on writing device service routines for nonstandard devices may be found in the reference manual for the operating system under which the device service routine is to run.

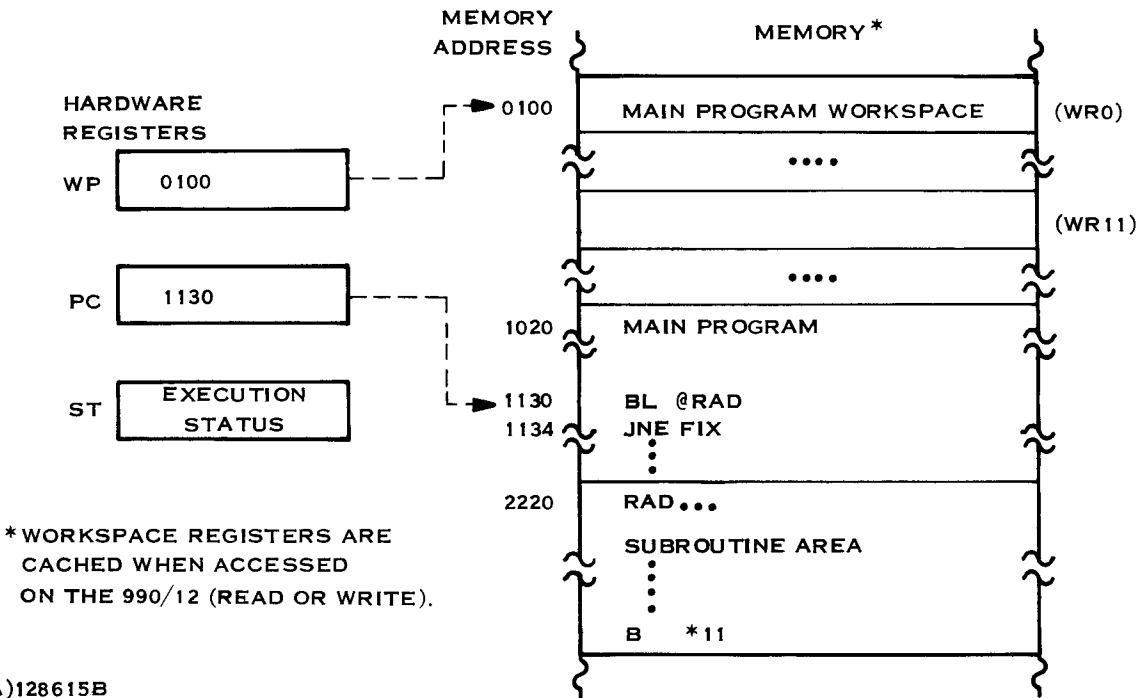
## 4.2 SUBROUTINES

There are two types of subroutine linkage available with the Model 990 Computer. One type uses the same set of workspace registers that the calling routine uses, and is called a common workspace subroutine. The BL instruction and BLSK instruction store the contents of the program counter and transfer control to the subroutine. Another type is called a context switch subroutine. The BLWP instruction stores the contents of the WP register, the program counter, and the status register. The instruction makes the subroutine workspace active and transfers control to the subroutine. The BLSK instruction executes on the 990/12 only.

### 4.2.1 BL Instruction Common Workspace Subroutine Example

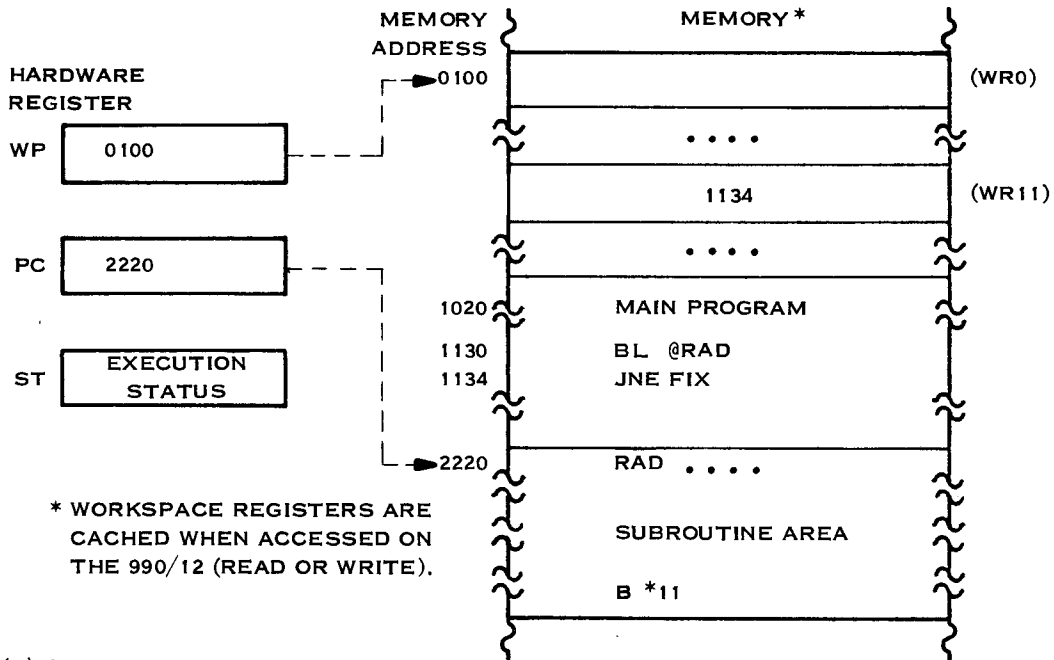
Figure 4-1 shows an example of memory contents prior to a BL call to a subroutine. The contents of workspace register 11 are not important to the main routine. When the BL instruction is executed, the CPU stores the contents of the PC in workspace register 11 of the main routine and transfers control to the instruction located at the address indicated by the operand of the BL instruction. This type of subroutine uses the main program workspace. Figure 4-2 shows the memory contents after the call to the subroutine with the BL instruction.

When the instruction at location  $1130_{16}$  is executed (BL @RAD), the present contents of the PC, which point to the next instruction, are saved in workspace register 11. WR11 would then contain an address of  $1134_{16}$ . The PC is then loaded with the address of label RAD, which is address  $2220_{16}$ . This example subroutine returns to the main program with a branch to the address in WR11 (B \*11).



(A)128615B

Figure 4-1. Common Workspace Subroutine Example



(A)128616B

Figure 4-2. PC Contents After BL Instruction Execution

### 4.2.2 BLSK Common Workspace Subroutine Example

Figure 4-3 shows an example of memory contents prior to a BLSK call to a subroutine. The stack is located in the data area of the main routine. When the BLSK instruction is executed, the CPU stores the contents of the PC in two consecutive bytes on top of the stack and transfers control to the instruction located at the address indicated by the operand of the BLSK instruction. This type of subroutine uses the main program workspace. Figure 4-4 shows the memory contents after the call to the subroutine with the BLSK instruction. The BLSK instruction executes on the 990/12 only.

When the instruction at location  $2000_{16}$  is executed (BLSK R5, @ SUBR), the present contents of the PC, which point to the next instruction, are pushed on to the stack, as two consecutive bytes. The next instruction address is  $2004_{16}$ , so the top byte of the stack contains  $20_{16}$ , and the second byte contains  $04_{16}$ . The PC is then loaded with the address of label SUBR, which is address  $2500_{16}$ . This example subroutine returns to the main program with an indirect branch through workspace register five. The top of stack is updated by using autoincrement to the return.

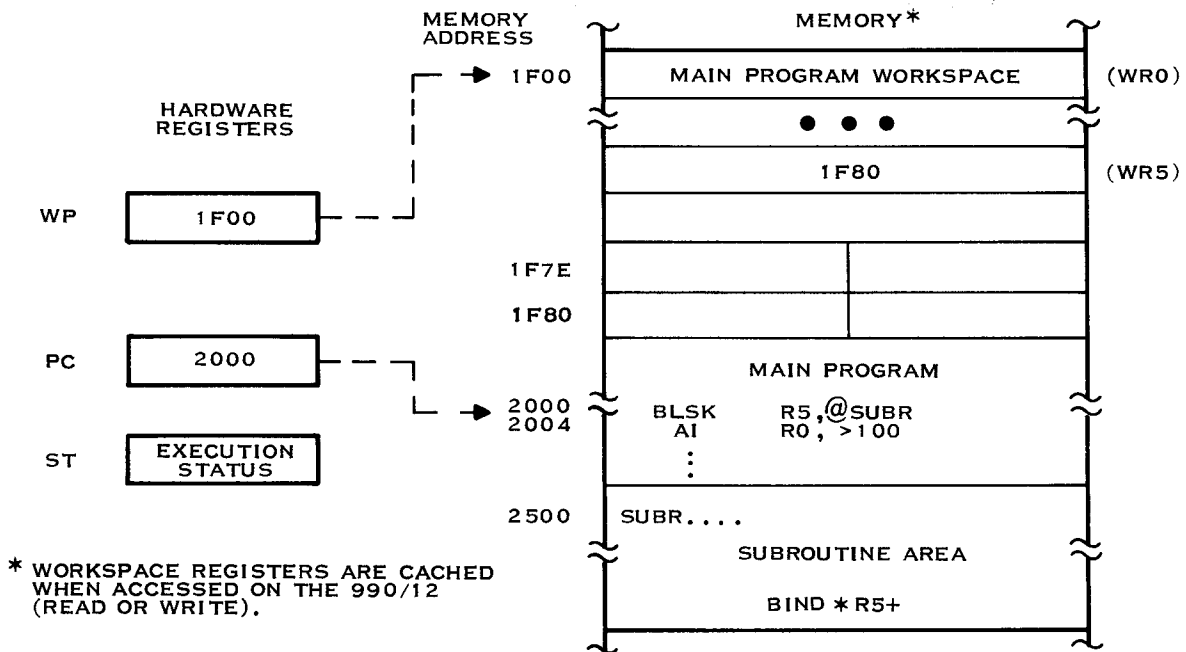


Figure 4-3. Before Execution of BLSK Instruction

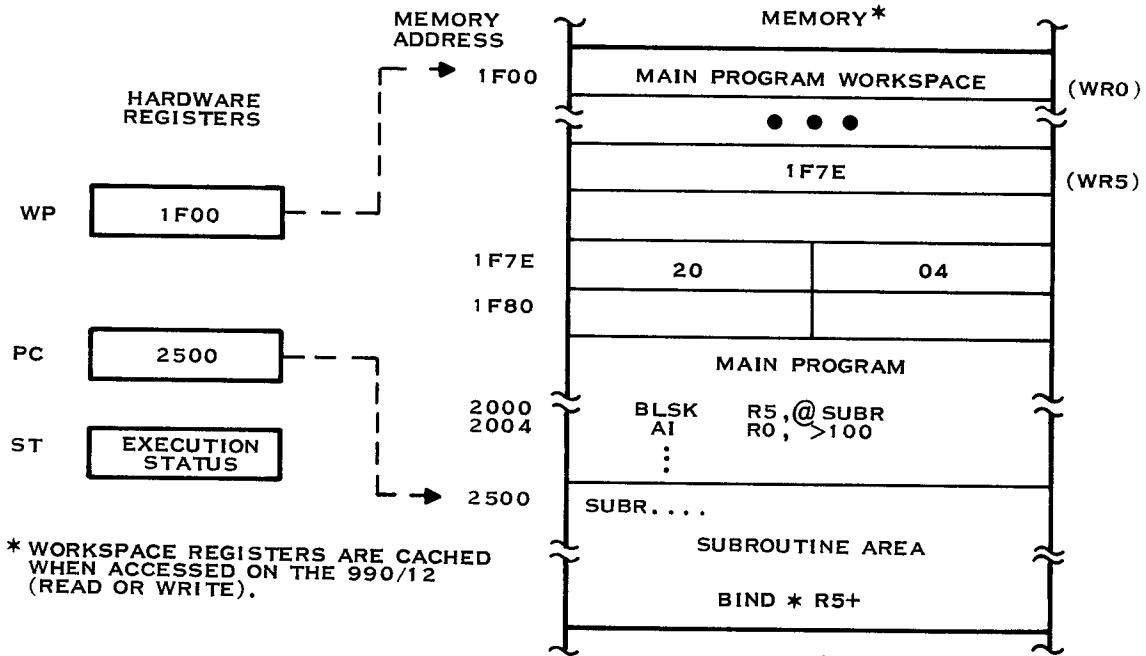


Figure 4-4. After Execution of BLSK Instruction

### 4.2.3 Context Switch Subroutine Example

Figure 4-5 shows the example memory contents prior to the call to the subroutine. The contents of workspace register 13, 14, and 15 are not significant. When the BLWP instruction is executed at location 0300, there is a context switch from the main program to the subroutine. The context switch then places the main program WP, PC, and ST register contents in workspace registers 13, 14, and 15 of the subroutine. This saves the environment of the main program for return. The operand of the BLWP instruction specifies that the address vector for the context switch is in workspace registers five and six. The address in workspace register five is placed in the WP register and the address in workspace register six is placed in the PC.

After the instruction at location 0300 is executed, the memory contents are shown in Figure 4-6. This illustration shows the subroutine in control, with the WP pointing to the subroutine workspace and the PC pointing to the first instruction of the subroutine. The contents of the status register are not reset prior to the execution of the first instruction of the subroutine, so the status indicated will actually be the status of the main program execution. A subroutine may then execute in accordance with the status of the main program.

This example subroutine contains a RTWP return from the subroutine. The results of executing the RTWP instruction are shown in Figure 4-7. Control is transferred to the main program at the instruction following the BLWP to the subroutine. The status register is restored from workspace register 15, and the workspace pointer points to the workspace of the main program.



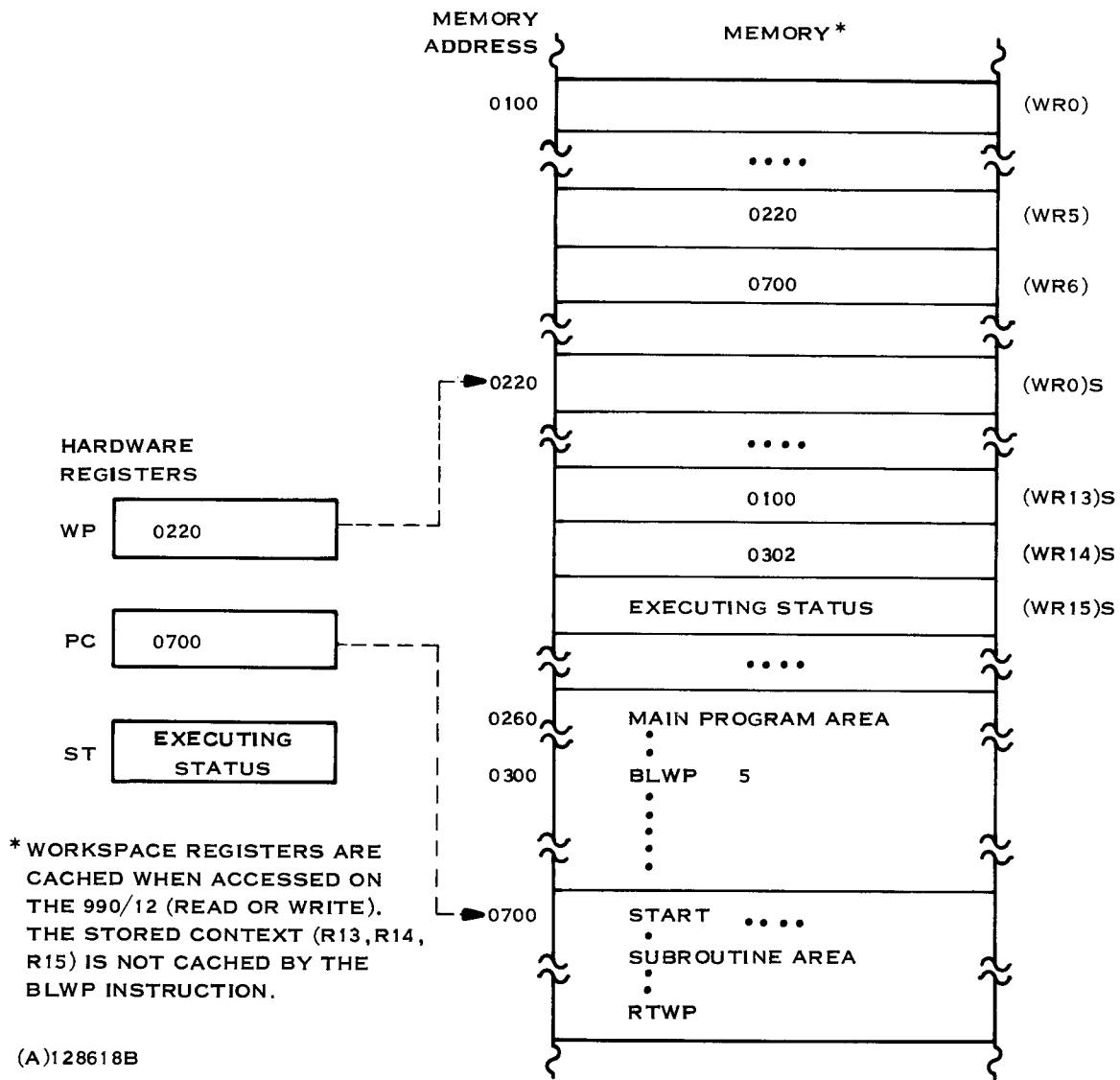


Figure 4-6. After Execution of BLWP Instruction

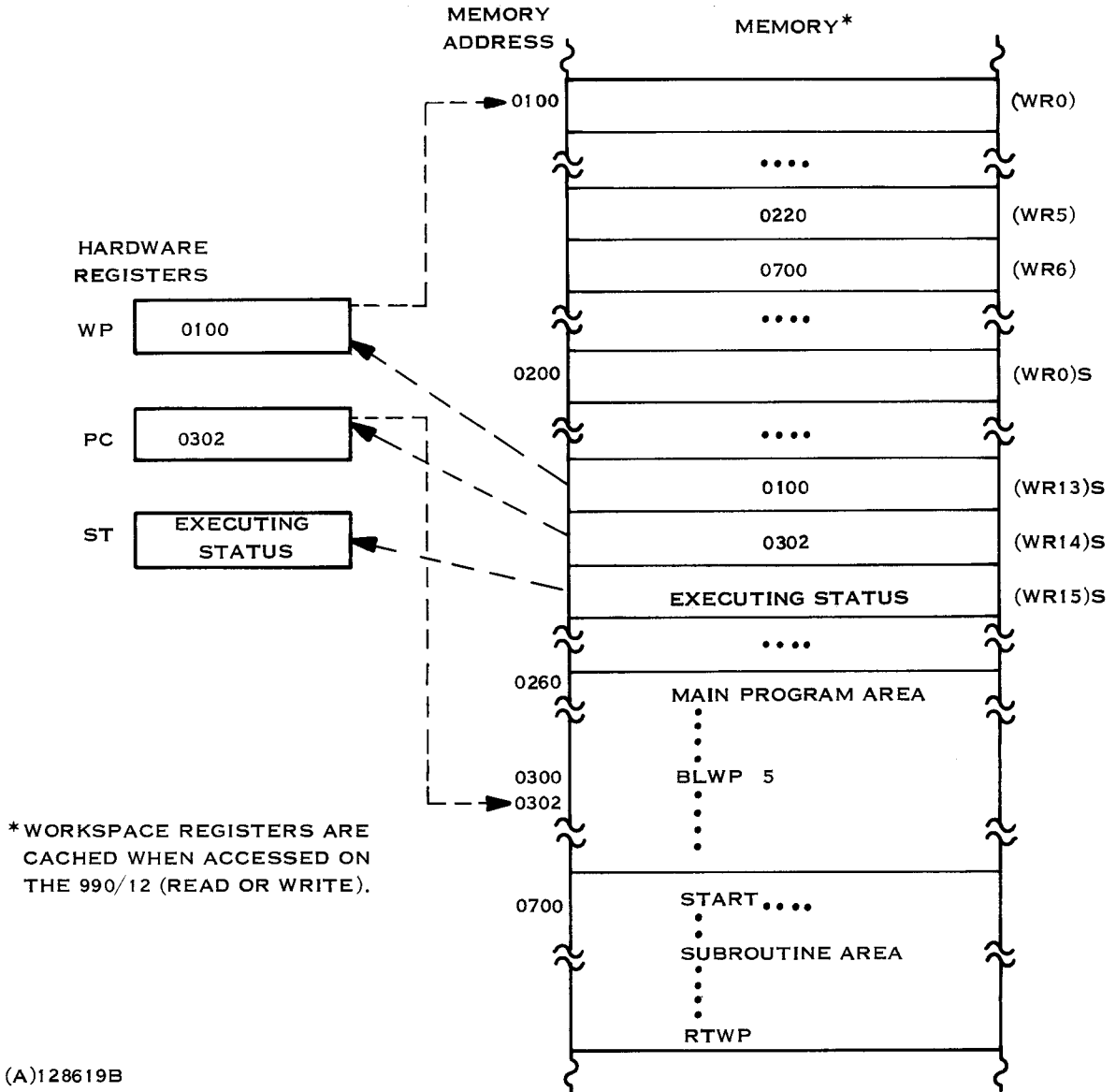


Figure 4-7. After Execution of RTWP Instruction



#### 4.2.4 Passing Data to Subroutines

When a subroutine is entered with a context switch (BLWP), data may be passed using either the contents of workspace register 13 or 14 of the subroutine workspace. Workspace register 13 contains the memory address of the calling program's workspace. The calling program's workspace may contain data to be passed to the subroutine. Workspace register 14 contains the memory address of the next memory location following the BLWP instruction. This location and following locations may contain data to be passed to the subroutine.

The following example shows passing of data to a subroutine by placing the data following the BLWP instruction:

```

        BLWP      @SUB      SUBROUTINE CALL
        DATA    V1        DATA
        DATA    V2        DATA
        DATA    V3        DATA
        JEQ      ERROR     RETURN FROM SUBROUTINE, TEST
        .          .        FOR ERROR (Subroutine sets
        .          .        the equal status bit to one
        .          .        for error.)

SUB     DATA    SUBWS,SUBPRG ENTRY POINT FOR SUB
        .          .        AND SUB WRKSPCE
        .
        .
SUBWS   BSS      32
SUBPRG  MOV      *14 + ,1   FETCH V1 PLACED IN WR1
        MOV      *14 + ,2   FETCH V2 PLACED IN WR2
        MOV      *14 + ,3   FETCH V3 PLACED IN WR3
        .
        .
        .
        RTWP          RETURN FROM SUBROUTINE
    
```

The three MOV instructions retrieve the variables from the main program module and place them in workspace registers one, two, and three of the subroutine. The autoincrement (+) feature increments register 14 (PC) past the three DATA words to the next executable instruction in the main program.

When the BLWP instruction is executed, the main program module status is stored in workspace register 15 of the subroutine. If the subroutine returns with a RTWP instruction, this status is placed in the status register after the RTWP instruction is executed. The subroutine may alter the status register contents (stored in the subroutine workspace register 15) prior to executing the RTWP instruction. The calling program can then test the appropriate bit of the status word, the equal bit in this example, with jump instructions.

A BL instruction can be used to pass parameters to a subroutine. This instruction places the originating PC value in workspace register 11. Therefore, the subroutine must fetch the parameters relative to the contents of workspace register 11 rather than the contents of workspace register 14 as in the BLWP example. The following example demonstrates parameter passing with a BL instruction.

	BL	@SUBR	BRANCH TO SUBROUTINE
	DATA	PARM1,PARM2	PASSED PARAMETERS STORED IN NEXT TWO MEMORY WORDS
	JEQ	ERROR	TEST FOR ERROR (Subroutine sets the equal status bit to one for error)
	.		
	.		
SUBR	EQU	\$	
	MOV	*R11+,R0	GET VALUE OF FIRST PARAMETER AND PUT IN WR0
	MOV	*R11+,R1	GET VALUE OF SECOND PARAMETER AND PUT IN WR1 (R11 is incremented past the locations of the two data words and now indicates the address of the next instruction in main program)
	.		
	.		
	B	*11	

A BLSK instruction can also be used to pass parameters to a subroutine. This instruction places the originating PC value on the stack. Therefore, the subroutine must pop the PC value from the stack, and retrieve the parameters relative to the location of the popped PC value. The following example demonstrates parameter passing with a BLSK instruction. This example is applicable to the 990/12 only.

STACK	BSS	20	20 BYTE STACK
STEND	EQU	\$	
	.		
	.		
	LI	R5,STEND	LOAD REGISTER WITH CURRENT TOP OF STACK
	BLSK	R5,@SUBR	BRANCH TO SUBROUTINE
	DATA	PARM1,PARM2	PASSED PARAMETERS STORED IN NEXT TWO WORDS OF MEMORY
	JEQ	ERROR	TEST FOR ERROR (Subroutine sets the equal status bit if an error occurs)
	.		
	.		
SUBR	EQU	\$	
	POPS	R5,R6,2	POP TWO BYTES FROM STACK (PC value)
	MOV	*R6+,R0	GET FIRST PARAMETER
	MOV	*R6+,R1	GET SECOND PARAMETER (R6 is incremented past the locations of the two data words and now indicates

		the address of the next instruction in the main program)
PSHS	R6,R5,2	RESTORE CORRECT RETURN ADDRESS TO STACK

#### 4.2.5 Non-Reentrant Subroutine Control

The following paragraphs discuss ways of controlling the execution of non-reentrant subroutines through the use of the ABS, TCMB, and TSMB instructions. TCMB and TSMB execute only on the 990/12. A non-reentrant subroutine contains a code that modifies the code or data contained within itself so that the subroutine cannot be successfully executed by more than one user concurrently. Reentrant programming techniques are described in paragraph 4.4. The use of reentrant code is to be encouraged, since it increases the efficiency of the machine. In particular, a device service routine that services multiple devices at different interrupt priority levels must be reentrant.

As noted in Section 3, the techniques described below for using ABS, TCMB, and TSMB to control subroutine execution may also be used to control access to common memory in multiple-CPU systems.

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

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

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

	SETO	@SWITCH	INITIALIZES SWITCH NEGATIVE <sup>1</sup>
	.	.	.
	.	.	.
TEST	ABS	@SWITCH	TEST SWITCH <sup>2</sup>
	JLT	CALL	IF NEGATIVE, TRANSFER <sup>3</sup>
	XOP	@TMDLY,15	IF NOT, WAIT <sup>4</sup>
	JMP	TEST	TEST AGAIN
	.	.	.

CALL	BL SETO	@SUBR @SWITCH	USE SUBROUTINE RESET SWITCH <sup>5</sup>
	.		
	.		
	.		
SUBR	.....		SUBROUTINE ENTRY
	.		
	.		
	.		
	.		
SWITCH	B DATA	*11 0	SUBROUTINE RETURN STORAGE AREA FOR SWITCH
TIMDLY	DATA	>200,10	TIME DELAY SUPERVISOR CALL BLOCK

<sup>1</sup>Set SWITCH to all ones, making it negative.

<sup>2</sup>If SWITCH negative, set to positive value to prevent subsequent entry.

<sup>3</sup>If value in SWITCH was negative, the JLT instruction transfers control.

<sup>4</sup>Supervisor call pointing to data block defining time delay request. Used to wait for a time period before retesting SWITCH. While in a time delay, other programs can be executed, thus leaving the SUBR available for use. Time delay supervisor calls are supported by the operating systems. Reference the applicable operating system manual.

<sup>5</sup>Upon return, reset SWITCH to negative value to permit future use.

**4.2.5.2 TSMB and TCMB Instructions.** The test and set memory bit (TSMB) and test and clear memory bit (TCMB) instructions can be used to test flags in memory like the ABS instruction in the paragraph above. The following example is similar to the one above, except it uses the TSMB instruction for switch control. The TSMB instruction is useful when there are several common subroutines, each of which can be used by only one program at a time. These subroutines use a bit map, with each bit indicating the availability of the subroutine (0 = available, 1 = in use). The example program uses the TSMB instruction to determine if the subroutine is available, and if available, to set the flag and use the subroutine. TSMB and TCMB execute on the 990/12 only.

Location MAP is a 16-bit word; each bit can be used to control a common subroutine. Only bit zero is used in this example. The assembly language program does a TSMB to bit zero of location MAP. This bit is tested before it is set. If the bit equals zero, the subroutine SUBR is called. If the bit equals one, the program enters delay mode, and tests the bit again after the delay.

The test and clear memory bit (TCMB) can be used in the same way, except that a one would indicate the subroutine is available, and a zero would indicate the subroutine is active.

	TCMB	@MAP,0	INITIALIZE CONTROL BIT TO ZERO
	.		
	.		
TEST	TSMB	@MAP,0	TEST CONTROL, BIT <sup>1</sup>
	JNE	CALL	IF ZERO, TRANSFER <sup>2</sup>
	XOP	@TMDLY,15	IF NOT ZERO, WAIT <sup>3</sup>
	JMP	TEST	TEST AGAIN AFTER DELAY
	.		
	.		
CALL	BL	@SUBR	BRANCH TO SUBROUTINE
	TCMB	@MAP,0	RESET CONTROL BIT <sup>4</sup>
	.		
	.		
SUBR	....	....	SUBROUTINE ENTRY
	.		
	.		
	RT		SUBROUTINE RETURNS <sup>5</sup>
MAP	DATA	0	
TMDLY	DATA	200,10	TIME DELAY SUPERVISOR CALL BLOCK

<sup>1</sup> If bit zero of MAP equals zero, set to one to prevent subsequent entry. If bit zero of MAP equals one, the bit is unchanged.

<sup>2</sup> The instruction reads the bit in MAP into status register bit two (equal). If status bit two equals zero, the jump not equal (JNE) instruction transfers control.

<sup>3</sup> The XOP instruction performs a call to the operating system requesting a time delay. The task goes into a wait state before retesting bit zero of MAP. While this task is in a time delay, other programs can be executed, leaving the subroutine available for use. Reference the applicable operating system manual.

<sup>4</sup> Upon return, set bit zero of MAP to zero to permit further use of the subroutine.

<sup>5</sup> RT is a pseudo-instruction discussed in Section 6.

### 4.3 990/10 TO 990/12 UPGRADE CONSIDERATIONS

These paragraphs are written specifically for the user who is upgrading his computing system from a Model 990/10 Computer to a Model 990/12 Computer. The advanced architecture of the 990/12 may require modification of existing 990/10 code due to the differences in instruction execution or speed. None of the current release 990 software has upgrade problems, but users writing their own programs or transporting their own programs may encounter the problems discussed below.

### 4.3.1 Cache Usage

The Model 990/12 Computer features a high-speed workspace register cache memory. The cache is loaded with workspace data on a demand-fill basis, and subsequent accesses to the data are made to the cache instead of memory. The 990/12 also features an optional TILINE memory cache, with 1K 16-bit words of high speed general memory. Both of these features can be used to significantly enhance program operation.

The workspace is designed to hold high usage data and addresses. By keeping as much of this data as possible in the workspace, the programmer gains the efficiency of the workspace register cache. Once the data is loaded into the cache, accesses are made to the cache instead of memory. Therefore, only the first read of a workspace register requires a full main memory cycle. All other accesses to that register are made from the high-speed cache. This offers a significant increase in processor speed. When the workspace pointer is changed, the registers that have had a write cycle made to them are written to memory.

#### NOTE

Many of the two-word instructions on the 990/12 have the general source and destination address fields in the second word of the instruction. If the register direct addressing mode is selected for these instructions ( $T_s$  or  $T_d = 0$ ), this addressing mode usually operates slower than the other addressing modes (that is,  $T_s$  or  $T_d = 1, 2, \text{ or } 3$ ). It will not operate faster, as might be expected due to the workspace register cache, because the hardware will employ the workspace cache overlap circuitry for the direct addressing case. This case is similar to the case described in paragraph 4.3.3.2.

The memory cache option provides a similar increase in processor speed. The memory cache is loaded on a two-word demand fill basis. When an even numbered memory word (word zero, two, four,...) is addressed in memory, that word and the next word are loaded into the cache. When an odd numbered memory word (one, three, five,...) is addressed in memory, that word and the previous word are loaded into the cache, provided that it has not yet been loaded into the cache. Once the initial odd numbered word has been loaded into memory, the even-odd pairs are loaded. Therefore, the usual minimum hit ratio is 50%, and typically the hit ratio is much higher.

#### NOTE

The workspace cache is intended to hold frequently used program data. Execution of instructions from the cache, using the X instruction for example, causes the cache to be disabled with accompanying degradation of performance.

### 4.3.2 Execution Differences

The 990/12 CPU contains several features that enhance instruction throughput or expand the capability of the machine. Some of these features cause instructions to execute somewhat differently when transferred from the 990/10 to the 990/12. These cases are rare, and most programs written for the 990/10 will transfer to the 990/12 without modification. Each case, and methods to correct the operation, is described below.

**4.3.2.1 ABS Instruction.** Section 3 and the application notes in this section describe the use of the ABS instruction for multiprocessor control.

The 990/12 normally locks out all other processors from accessing memory while the ABS instruction is accessing the memory control flag. However, when the flag is located in a workspace register, the memory fetch traps to the workspace cache. Control of memory is released when the cache is accessed. Therefore, the flag could be read incorrectly by another processor. When the flag is incorrectly read as reset, two or more CPUs may access memory at the same time. When the flag is incorrectly read as set, the memory access may be delayed.

This problem can be solved by keeping all memory interlock control flags in memory locations that are not used as workspace registers.

**4.3.2.2 Second Word Modification.** The 990/12 central processor contains a prefetch register, which usually contains the next memory word following the currently executing instruction. If an instruction modifies the next word in memory, the modification occurs in memory but not in the prefetch register. Therefore, an instruction cannot modify the next word in memory and have the central processor use the correct value.

For example, in the code segment shown below, the 990/12 will execute the unmodified JMP instruction. The MOVB instruction modifies the signed displacement in the JMP instruction in memory, but the JMP instruction executes from the prefetch register.

	MOVB	R0,@LABEL + 1
LABEL	JMP	DUMMY

In general, code modification is an unreliable programming practice that produces non-reentrant code. It is not recommended on the 990/12.

**4.3.2.3 Illegal Opcodes.** Some instructions on the 990/10 have “don’t care” bits in the opcode field (the bits can be either zero or one). On the 990/12, these instructions have specific values for the “don’t care” bits, and any values besides the specified values will cause an illegal opcode error. For example, the opcode of the IDLE instructions on the 990/10 is 000000110100XXXX, where the Xs represent either zero or one. The opcode of the IDLE instruction on the 990/12 is 0000001101000000, in which the value of the last four bits must be 0000. Executing the IDLE instruction on the 990/12 with any of the last four bits set to one will cause an illegal opcode error on the 990/12, but it will execute normally on the 990/10. The instructions that have specific bit values on the 990/12 and not on the 990/10 are as follows:

IDLE	Idle
LWPI	Load Workspace Pointer Immediate
LIMI	Load Interrupt Mask Immediate
RSET	Reset

RTWP	Return Workspace Pointer
CKON	Clock On
CKOF	Clock Off
LREX	Load or Restart Execution

**4.3.2.4 Workspace Crossing Map Segment Boundaries.** On the 990/12, the workspace register file should never be located so that it extends over a map segment boundary. Access to registers that cross the end of the map boundary could cause the wrong physical address to be generated, thus filling the cache with bad data. This problem can be solved by either allocating space for the entire workspace or by not locating the workspace at the end of the task.

**4.3.2.5 Deferred Mapping Error.** In some cases, workspace register data is not written to memory until a context switch occurs. If a write to a workspace register causes a mapping error, the error will be deferred until the context switch occurs.

#### NOTE

If an unallocated workspace register is referenced, and it does not cross a map segment boundary, program data or instructions may be overwritten.

**4.3.2.6 Error Status Register.** The error status register on the 990/10 computer contains five significant bits, CRU bits 11-15 at CRU base >1FC0. On the 990/12, the error status register contains 16 significant bits, CRU bits 0-15 at CRU base >1FC0. A program transferred from the 990/10 to the 990/12 may require that the additional bits be masked when the error status register is read on the 990/12. The error status register is described in paragraph 4.5.1.2.

**4.3.2.7 990/12 CPU Status Register.** Bits 9-11 of the 990/10 status register are unused. On the 990/12, these three bits store relevant information. If a 990/10 program writes to any of bits 9-11, unexpected execution results may occur on the 990/12.

**4.3.2.8 Map Diagnostic Hardware.** Bits 9-11 of the 990/10 and the 990/12 have memory mapping diagnostic hardware to isolate faults in the memory mapping logic. Both computers have diagnostic control bits at CRU base address >1FA0. But, the map diagnostic hardware on the 990/10 is different from the diagnostic hardware on the 990/12, and the control bits serve different functions. Programs written for the 990/10, that use the map diagnostic hardware will not execute correctly on the 990/12. The CRU diagnostic control bits are described in paragraph 4.5.1.3.

**4.3.2.9 TILINE Access to Workspace Cache.** The workspace cache is accessible only to the 990/12 central processor. TILINE master devices other than the central processor address the workspace data that is in main memory. If parameters are passed between the CPU and another TILINE device through the workspace, errors in operation can result. Therefore, all such flags and parameters should be kept in main memory, and not in workspace registers.

#### 4.3.3 Performance Differences

When a program is transferred from a 990/10 to a 990/12, differences in instruction execution speed may sometimes require modification to the transferred code, even though the instructions execute identically on the 990/12. These cases are described below.





## PROGRAM TWO

WP	DATA 0	WORKSPACE REGISTER 0
WR1	DATA 0	WORKSPACE REGISTER 1
.		
.		
.		
LWPI WP	LOAD WORKSPACE POINTER	
INC @ WR1	INCREMENT WORKSPACE REGISTER ONE	

**4.3.3.4 Instruction Execution from Workspace Registers.** Instructions can be executed from any location in memory, including cached workspace registers. However, when instructions are executed from the cache, there is a significant increase in instruction execution time. This problem is a special case of the workspace register addressing discussed in paragraph 4.3.3.3. The instructions that are cached are not addressed in direct mode, so the time for fetching the instruction is longer than if the instruction was in memory.

**4.3.3.5 LDS Instruction.** On the 990/12 computer, an LDS instruction addressing workspace register eight or above directly causes the workspace cache to be dumped into the program starting at the instruction after the LDS. This problem may be avoided by using workspace register seven or less or by addressing the registers greater than seven indirectly.

**4.3.3.6 User Device Service Routines.** Device service routines written by the user can be entered on the operating system to support nonstandard devices. User DSRs transported from the 990/10 to the 990/12 may not operate properly because of timing differences and the workspace cache. This situation pertains to user DSRs written for other Model 990 Computers or other 990 operating systems. The typical problems encountered are described in paragraphs 4.3.2.9 (workspace cache) and 4.3.3.1 (timing loops), respectively.

More information on adding a user device service routine to a particular operating system may be found in the reference manual for that operating system.

#### 4.4 REENTRANT PROGRAMMING

Reentrant programming is a technique that allows the same program code to be used for several different applications while maintaining the integrity of the data used with each application. The common program code and its associated constants are stored in one area in memory. Each function that uses that code is then assigned a unique workspace and data area so that as it executes the common code, its variable data is developed without affecting the variable data associated with any of the other functions that use the program. With this arrangement one function can execute the common code routine and be interrupted in the middle of the routine by another function that also uses the same routine. The second function then uses the routine for its purpose and returns control to the first function so that it can proceed from the point of interruption without returning to the start of the routine. Reentrant programming of this type lends itself well to servicing similar peripheral devices that interface with the computer at different priority levels. The following characteristics apply to a reentrant procedure:

- The procedure does not contain data except data common to all tasks.

- The procedure does not alter the contents of any word in the procedure, whether that word contains data or an instruction.
- Data that is unique to one or more tasks is in the data division for the task and is either in a workspace or is indirectly addressed.

A very important application of a reentrant procedure is one that controls a process using several sets of identical control devices through identical sets of CRU lines. Each task using the reentrant procedure addresses a unique set of control devices that controls a set of equipment to perform the same process concurrently. The workspace for each task contains the CRU base address in workspace register 12 for the set of control devices for the task. The procedure addresses a control device by a displacement from the base address. For each task, the base address in workspace register 12 of its workspace controls the proper device. Figure 4-8 shows a procedure common to sixteen tasks, each of which uses an identical set of CRU lines at different CRU base addresses.

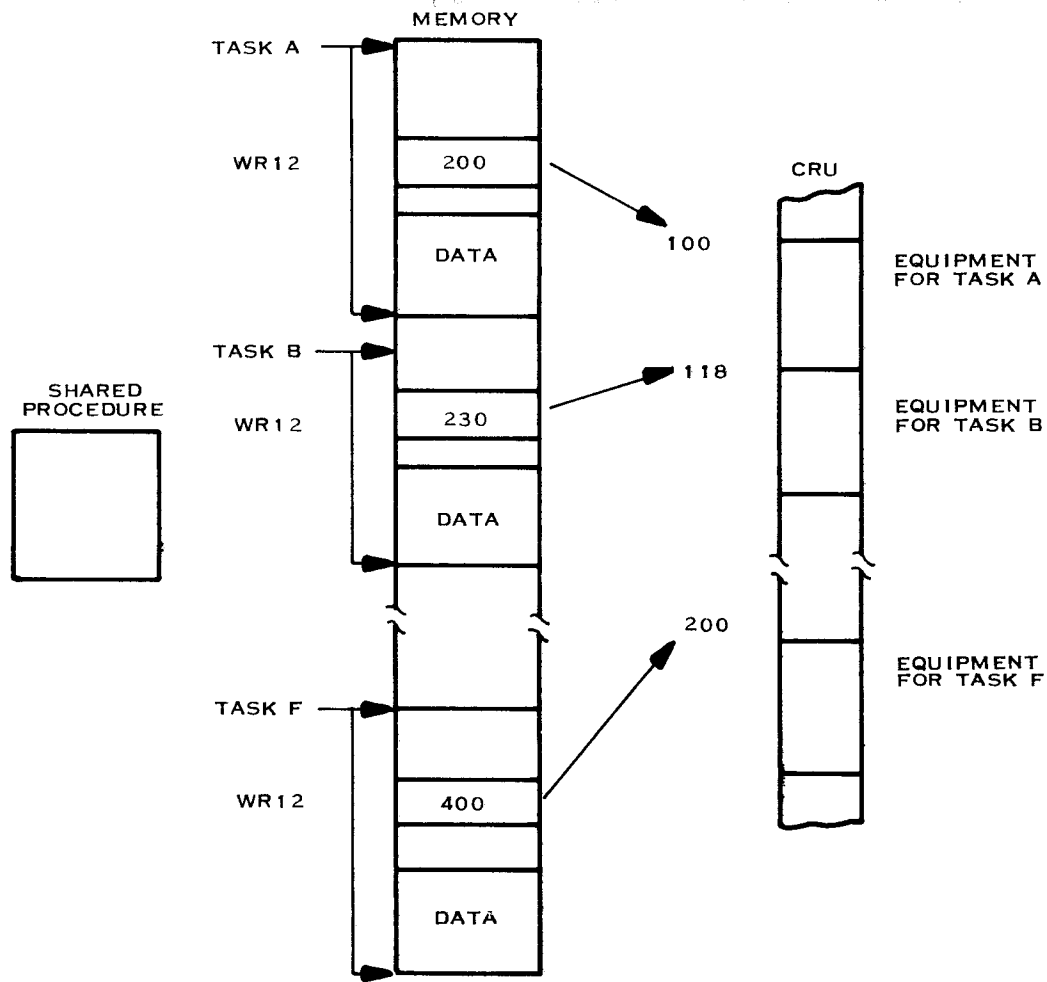


Figure 4-8. Reentrant Procedure for Process Control

The following is an example of reentrant code. The following assumptions apply:

- Workspace register 14 contains the address of a word that contains the size of a buffer, in bytes.
- Workspace register nine contains the start address of that buffer.
- Label NOTFND is the location that contains the first instruction of a routine that is to be executed if the buffer does not contain a carriage return character.
- Label FOUND is the location of the first instruction of a routine that is to be executed when the buffer contains a carriage return.

The reentrant code is as follows:

```

ENTER      MOV      *R14,R3      GET BUFFER SIZE
           MOV      R9,R8        GET START ADDRESS
           A        R3,R8        POINT TO END OF BUFFER
LOOK       C        R9,R8        CHECK FOR END
           JH       NOTFND       BRANCH AT END
           CB       *R9+,@CARRET CHECK CHARACTER
           JNE      LOOK         BRANCH WHEN NOT FOUND
FOUND      CHARACTER FOUND
           .
           .
           .
CARRET     BYTE     >D

```

The code is reentrant because it is not altered during execution of the code. Also, when execution resumes following an interruption, the workspace for the code again becomes active, and contains the correct values for resuming the execution as if execution had not been interrupted.

Another possible version of the same code is as follows:

```

ENTER      MOV      *R14,@ADDLOC
           MOV      R9,R8
           AI       R8,$$
ADDLOC     EQU      $-2
LOOK       C        R9,R8
           JH       NOTFND
           CB       *R9+,@CARRET
           JNE      LOOK
FOUND      .
           .
           .
CARRET     BYTE     >D

```

The code performs the same function by storing the buffer length in the word that contains the immediate operand of an AI instruction. As long as only one task using this code is active, there is no problem. However, if one task is interrupted after storing a value in ADDLOC and before executing the AI instruction, and another task executes the code, the size of the buffer for the first task is lost. The code is not reentrant because it alters data within itself.

## 4.5 WRITING STAND-ALONE PROGRAMS

A program that does not execute under the control of an operating system must handle for itself those services that it requires that are normally handled by the operating system. Before writing a stand-alone program, the programmer should thoroughly understand operating system theory and implementation, as well as the features of the hardware on which the program is to run. The system design document for one of the operating systems, available from TI upon request, and the hardware manuals referenced in the preface to this manual would be valuable resources in this instance. The rest of this section contains general information about the implementation of various facets of a stand-alone program. Much of the information is also applicable to writing a device service routine for a nonsupported device; however, the operating system reference manuals contain detailed information about writing device service routines and adding them to the particular operating system.

The two main functions that a stand-alone program must provide are interrupt handling and device service routines to interface to the peripherals. The 990 computer provides a flexible interrupt management structure, which is described in paragraph 4.5.1. Interface to peripheral devices is accomplished through the CRU and TILINE, which are described in paragraphs 4.5.2 and 4.5.3, respectively.

### 4.5.1 Interrupts

The 990 computer general interrupt structure was described in detail in paragraph 1.2.5. The three highest interrupt levels are predefined by 990 computer hardware.

**4.5.1.1 Predefined Interrupts.** Level zero is the power-on interrupt for the Model 990 Computer, used for power-on initialization of the processor. Level one is the power-failing interrupt, used when ac power begins to fail. At this point, the computer has seven milliseconds of program time before operation is halted. Interrupt level two is the system error interrupt, detailed below. Interrupt levels five or 15 can be defined as the line frequency clock interrupt. The remaining interrupt levels are available for assignment to devices on the CRU and TILINE. Several interrupt lines may be combined at one level. Any interrupt request must remain active until the interrupt is taken and must be reset before the interrupt subroutine is complete.

**4.5.1.2 System Error Interrupt.** Interrupt level two is defined as the system error interrupt. Eleven conditions may cause a system error interrupt (although the breakpoint and the 12-ms test clock are forced error interrupts). These conditions are listed in Table 4-1.

**Table 4-1. Error Interrupt Status Register (CRU Base 1FC0<sub>16</sub>)**

Error	Input Bit	Output Bit <sup>1</sup>
TIMEOUT (Unimplemented Memory was Addressed)	15	15
PRIVOP (Privileged Violation)	14	14
ILLOP (Illegal Instruction Code)	13	13
MER (Memory Data Error)	12	12
MAPERR (Address Beyond Map)	11	11
EV (Execution Attempt in Execute-protected Memory) <sup>2</sup>	9	9
WV (Write Attempt in Write-protected Memory) <sup>2</sup>	8	8
SO (Stack Overflow/Underflow) <sup>2</sup>	7	7
BP (Breakpoint Address Encountered)	6	6
CK (12-ms Test Clock) <sup>2</sup>	5	5
AO (Arithmetic Overflow)	4	4
ID (990/10-990/12 Indicator)	0	

**Notes:**

<sup>1</sup>Individually cleared by SBZ. Any SBO to bits 0 through 7 of this register sets all bits to one for diagnostic purposes.

<sup>2</sup>These bits have meaning on a 990/12 only.

To isolate the cause of the error, read the CRU error interrupt status register using the STCR instruction. The interrupt is cleared by executing a set CRU bit to zero (SBZ) instruction. A sample routine for doing this is shown in Figure 4-9. The error interrupt status register is cleared by any of the following operations.

- Reset (RSET instruction)
- Power-up
- CRU output operations to CRU base address 1FC0<sub>16</sub> (clear the individual bits).

TABLE	EQU	>1000	Initialize the table entry address. TABLE starts at address HEX 1000.
*			
READ	EQU	\$	Entry point.
	LI	R3, TABLE	Initialize recovery routine index.
	LI	R12, >1FC0	Initialize CRU base address.
	STCR	R 1, 12	Read error interrupt register.
	JEQ	OUT	If all bits equal zero, exit routine.
TESBIT	TB	0	Test the bit in the error register.
	JNE	INCRMN	If the bit is not set, increment the registers and test the next bit.
	BLWP	*R3	Branch to the recovery routine.
	SBZ	0	Reset the bit in the error register.
	JMP	READ	Retest the error register.
INCRMN	INCT	R12	Increment the CRU address.
	INCT	R3	Increment recovery routine.
	INCT	R3	Index by four.
	JMP	TESBIT	Test the next bit.
OUT	RTWP		Return.

Figure 4-9. Error Interrupt Handling Routine

This example routine shows how the error interrupt status register is read, the error recovery routines are entered, and the error is cleared. Error recovery routines are entered through TABLE, a table of workspace pointers and program counters similar to the interrupt vectors. This routine reads the error register and enters the recovery routine if an error bit is set. If an error bit is not set, the next bit is tested. When all the bits are reset, the routine is exited.

**4.5.1.3 Error Interrupt Trace Memory.** The 990/12 has a trace memory of sixteen 32-bit words containing trace information useful in error recovery and for diagnostic purposes. When a system error interrupt occurs, the trace memory contains a trace of the 15 memory cycles or workspace accesses prior to and including setting of the system error interrupt. The trace memory does not stop immediately upon the setting of a system error interrupt, but stores one more workspace access or memory cycle to make the 16th word. The functions of the bits of the 32-bit trace words are listed in Table 4-2. Each 32-bit word is composed of two 16-bit words that are read consecutively by the CRU.

The error interrupt trace memory is read by software through a 16-bit CRU register at CRU base address 1FA0<sub>16</sub>. The first 16-bit word read from this register after an error interrupt is the first 16 bits of the last 32-bit trace memory word. The second 16-bit word read from this register is the second 16 bits of the last trace memory word. The next 16-bit word read is the first 16 bits of the next-to-the-last trace memory word. The entire error interrupt trace memory is transferred in this manner. Each read (STCR) should be followed by an SBO or SBZ to bit zero to decrement the trace pointer.

The output bits at CRU address 1FA0<sub>16</sub> are listed in Table 4-3. Output bit zero is used to decrement the trace memory pointer. The remaining bits are used for control and breakpoint operations. Power reset clears all bits to zero. I/O reset does not affect this register. The breakpoint system is discussed in the next paragraph.

**Table 4-2. Error Interrupt Trace Memory Data Word Bit Functions**

First Word CRU Bit Number	Function
0-15	Least significant 16 bits of saved TILINE address.
<b>Second Word CRU Bit number</b>	
0-3	Most significant four bits of saved TILINE address.
4	End of Instruction Flag. "1" = E.O.I.
5	Workspace Access Flag. "1" = W.A.
6	TILINE Read/Write Flag. "1" = Write.
7	TILINE Access Flag. "1" = T.A.
8	Workspace Read/Write Flag. "1" = Write.
9	Privileged Violation. "1" = P.V.
10	Illegal Opcode. "1" = I.O.
11	Mapping Error. "1" = Error. Indicates an attempt to address memory beyond the limits set in the active map file limit registers.
12	Memory Data Error. "1" = Error.
13	TILINE Time Out. "1" = Timeout. Indicates an attempt to address unimplemented TILINE addresses (memory).
14	Execution Violation. "1" = E.V. Indicates an attempt to execute from a mapped memory segment that has been flagged as nonexecutable.
15	Write Violation. "1" = W.V. Indicates an attempt to write to a mapped memory segment that has been flagged as nonwritable.

**Table 4-3. CRU Output Bit Assignments for Error Interrupt Trace Control and Map Control (CRU Base Address 1FA0<sub>16</sub>)**

Output Bit	Function
0	SBO or SBZ instruction to bit zero decrements error interrupt trace memory display pointer.
1	Test clock enable. SBO enables the test clock; SBZ disables the test clock.
2	TILINE cache enable. SBO = disable SBZ = enable
3	Mapping enable. SBO enables memory mapping, SBZ disables mapping except for addresses to the TILINE peripheral control space and to the loader and self-test PROM.
4 <sup>1</sup>	SBO or SBZ to bit four clears the mapping violation bit in the error interrupt status register.



**Table 4-3. CRU Output Bit Assignments for Error Interrupt Trace Control and Map Control (CRU Base Address 1FA0<sub>16</sub>) (Continued)**

5	Breakpoint on map zero or map one. SBZ = map 0, SBO = map one.
6, 7 = 0,0 <sup>2</sup>	Breakpoint occurs on any read.
6, 7 = 0,1 <sup>2</sup>	Breakpoint occurs on any instruction stream fetch.
6, 7 = 1,0 <sup>2</sup>	Breakpoint occurs on any write.
6, 7 = 1,1 <sup>2</sup>	Breakpoint on any address occurrence.
8	Diagnostic interrupt enable. SBO = enable. See paragraph 4.5.1.6
9	Diagnostic memory error. SBO = enable. See paragraph 4.5.1.7
10 11 12	CRU output bits 10, 11, and 12 develop PROM address lines to address one of eight 512-word sections of the loader and self-test PROM. Bit 12 is the most significant bit.
13	Breakpoint qualifier in privileged mode. SBO disables breakpoint system when processor is in privileged mode (ST7 = 0).
14, 15	Reserved.

**Notes:**

<sup>1</sup> Also cleared by SBZ to CRU output bit 11 at CRU base address 1FC0<sub>16</sub>, power-up, or RESET instruction.

<sup>2</sup> Breakpoint bit in the error interrupt status register is set after memory request that satisfied the breakpoint.

**4.5.1.4 Breakpoint System.** The Model 990/12 Computer features a programmable breakpoint system for diagnostic use. The breakpoint system is implemented using the following:

- 16-bit register at CRU base 1F80<sub>16</sub>
- Error interrupt status register
- Error trace memory
- 16-bit register at CRU base 1FA0<sub>16</sub>.

Bits 5 through 7 and 13 of the error interrupt trace control and map control register (CRU base 1FA0<sub>16</sub>) are used to indicate the breakpoint condition(s). These bits are listed in Table 4-2. CRU bits 1 (LSB) through 15 (MSB) of the breakpoint register (CRU Base 1F80<sub>16</sub>) are used for the breakpoint word address. Bit 0 is the breakpoint enable (1 = enable). When the breakpoint system is enabled and the proper memory reference (Table 4-2) is made to the address that corresponds to the value in the breakpoint register, the error interrupt trap (level two) is taken and the breakpoint flag is set in the error interrupt status register (bit 6 of CRU base 1FC0<sub>16</sub>). The breakpoint address and breakpoint enable are cleared by the hardware when the breakpoint is encountered or when a reset (RSET) or power-up occurs.

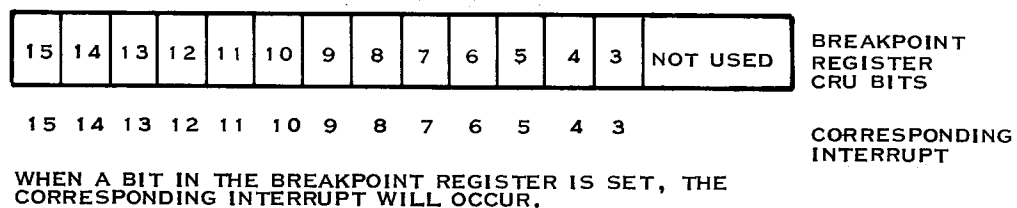
**NOTE**

The breakpoint enable should be set last using an SBO instruction.  
The breakpoint register can be loaded using the LDCR instruction.

**4.5.1.5 Twelve-Millisecond Test Clock.** When CRU bit 1 at CRU base address 1FA0<sub>16</sub> is set to a one, the 12-ms (± 15%) test clock interrupt is enabled. When the clock interrupt occurs, the error interrupt level two is taken and the 12-ms test clock flag (CK) is set in the error interrupt status register (bit 5 of CRU base 1FC0<sub>16</sub>). The interrupt is cleared by clearing the CK flag.

This test clock interrupt is used by a service routine that gathers statistical data for analysis of the operating system software and is used to determine the percentage of time available to the users. This function is cleared and disabled by power-up or RSET.

**4.5.1.6 Forced Interrupts.** The error interrupt trace control and map control register outputs and the breakpoint register can be programmed to initiate interrupt levels 3 through 15. When bit 8 of the error interrupt trace control and map control register is set, CRU bits 3 through 15 of the breakpoint register correspond to interrupt levels 3 through 15, respectively. Setting one of the breakpoint bits causes the corresponding interrupt to occur. This is illustrated in Figure 4-10.



**Figure 4-10. Breakpoint Register Interrupts**

**4.5.1.7 Forced Memory Errors.** Bit 9 of the error interrupt trace control and map control register can be used to cause a memory error. By setting CRU bit 9 in the register, any memory cycle addressing the loader and self-test PROMs will generate a memory error. This causes a level two interrupt (error interrupt), with bit 12 of the error interrupt status register set.

## 4.5.2 CRU Input/Output

The communications register unit (CRU) performs single and multiple bit programmed input/output in the Model 990 Computer. All input consists of reading CRU line logic levels into memory, and output consists of setting CRU output lines to bit values from a word or byte of memory. The CRU provides a maximum of 4096 input and output lines that may be individually selected by a 12-bit address. The 12-bit address, located in bits 3 through 14 of workspace register 12, is the base address of all CRU communications.

I/O to supported devices is provided through the use of I/O supervisor calls to the operating system. For these CRU devices, it is not necessary to use the instructions described in the following paragraphs. The information provided here is for writing routines for nonstandard device handling.

**4.5.2.1 CRU I/O Instructions.** Five instructions provide for communications with CRU lines. Three of them are single-bit transfer instructions, used for control functions such as turning on a motor, clearing interrupts, testing status bits, or testing an external sensor. The three single-bit instructions are SBO, SBZ, and TB. The remaining two instructions are multiple-bit transfer instructions, used for transferring data between memory and the CRU input/output devices. The multiple-bit instructions are LDCR and STCR.

- SBO — Set CRU bit to one. This instruction sets a CRU output line to a logic one. If the device on the CRU line is a data module, SBO results in zero volts at the data module terminal corresponding to the addressed bit.
- SBZ — Set CRU bit to zero. This instruction sets a CRU output line to a logic zero. If the device on the CRU line is a data module, SBZ results in a float (no signal applied) at the data module terminal corresponding to the addressed bit.
- TB — Test CRU bit. This instruction reads the digital input bit and sets the equal status bit (bit 2) to the value of the digital input bit.
- LDCR — Load Communications Register. This instruction transfers the number of bits (1 through 16) specified by the C field of the instruction onto the CRU from the source operand. When less than nine bits are specified, the source operand address is a byte address. When more than eight bits are specified, the source operand is a word address. The CRU address is the address of the first CRU bit affected.
- STCR — Store Communications Register. This instruction transfers the number of bits specified by the C field of the instruction from the CRU to the source operand. When less than nine bits are specified, the source operand is a byte address. When nine or more bits are specified, the source operand is a word address.

**4.5.2.2 CRU Base Address Development.** The processor develops a CRU bit address for the single-bit operations by adding the CRU base address contained in workspace register 12 (R12) to the signed displacement contained in bits 8 through 15 of the instruction. The displacement allows two's complement addressing from base minus 128 bits through base plus 127 bits. Bits 3 through 14 of the base address in R12 are added to the signed displacement specified in the instruction, with the sign bit (bit 8) extended to the left four bit positions to make a 12-bit value, and

the result is loaded onto the CRU address bus. For example, with register 12 set to >200, the effective base address in bits 3 through 14 is equal to >100, and the instruction SBO 15 sets CRU line >10F to one. If register 12 contains the value >21E, the effective base address in bits 3 through 14 is >10F, and the instruction SBO 0 also sets CRU line >10F to one.

For multiple-bit instructions, the address of the first bit on the CRU to be read or set is specified in bits 3 through 14 of workspace register 12. The number of bits to be read or set is specified in the C field of the instruction. Because the CRU address development ignores the LSB of R12, and effectively divides the address by two, the address value loaded into R12 must be double the intended CRU bit number to compensate.

The LDCR instruction fetches a word from memory and right-shifts it to transfer it serially to the CRU output bits. If the LDCR instruction specifies eight or fewer bits, the bits are taken right-justified (the least significant bits) from the addressed byte of the memory word. If the LDCR instruction specifies nine or more bits, the bits are taken right-justified (the least significant bits) from the entire memory word. As the bits are transferred to the CRU interface, the CRU address is incremented for each successive bit. The addressing mechanism results in an order reversal of the bits; that is, bit 15 (or bit 7) of the memory word becomes the lowest addressed bit in the CRU and bit 0 becomes the highest addressed bit in the CRU field.

The STCR instruction transfers data from the CRU to a specified memory word. If the operation involves the transfer of eight bits or less, the transferred data is stored right-justified in the specified memory byte with leading bits set to zero. The remaining byte of the word is not affected. If the transfer involves 9 to 16 bits, the transferred data is stored right-justified in the entire word, with leading bits set to zero. When the input from the CRU device is complete, the first bit from the CRU is in the least significant bit position of the memory word or byte. Figure 4-11 illustrates the multiple-bit transfer operations.

**4.5.2.3 SBO Example.** Assume that a control device turns on a motor when the computer sets a one on CRU line >10F and that workspace register 12 contains >200, making the base address in bits 3 through 14 equal to >100. The following instruction sets CRU line >10F to one:

**SBO 15**

If register 12 contains >21E, the effective base address in bits 3 through 14 is >10F, and the following instruction also sets CRU line >10F to one:

**SBO 0**

If a data module is connected as the CRU device, either instruction places zero volts on output line 15 of the module without affecting other lines.

**4.5.2.4 SBZ Example.** Assume that a control device that shuts off a valve when the computer sets a zero on a CRU line is connected to CRU line two, and that workspace register 12 contains zero. The following instruction sets CRU line two to zero:

**SBO 2**

If workspace register 12 contains four, the following instruction also sets CRU line two to zero:

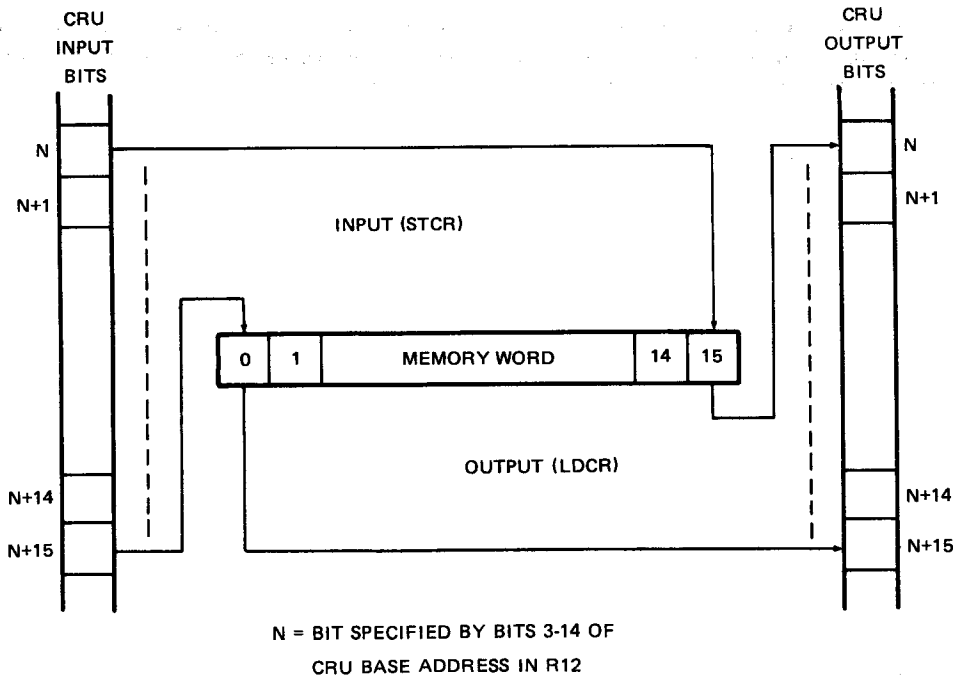
**SBO 0**

If a data module is connected as the CRU device, output line two would float at a voltage determined by the characteristics of the control device. No other CRU line would be affected.

**4.5.2.5 TB Example.** Assume that workspace register 12 contains >0140, making the effective base address in bits 3 through 14 equal to >A0. The following instructions would test the input on CRU line >A4 and execute the instructions beginning at location RUN when the CRU line is set to one. When the CRU line is set to zero, the instructions beginning at location WAIT are executed.

	TB	4	TEST CRU LINE 4
	JEQ	RUN	IF ON, GO TO RUN
WAIT	.		IF OFF, CONTINUE
	.		
	.		
RUN	.		

If register 12 is set to >148, the instruction TB 0 has the same effect. The TB instruction sets the logic level of the equal bit of the status register to the level on line four of the CRU device.



(A)133121A

Figure 4-11. LDCR/STCR Data Transfer

**4.5.2.6 LDCR Example.** Assume that a device is connected to the CRU and that the base address in workspace register 12 is set to CRU line >40. The following instruction outputs a character from memory location TOM by setting CRU output lines >40 through >47 to the bit configuration of the character.

```
LDCR @TOM,8
```

Assuming memory location TOM begins on an even address, the bits would be transferred as shown in Figure 4-12.

If the LDCR instruction is changed to LDCR @TOM,9, nine bits beginning with the least significant bit at address TOM are transferred to nine CRU lines, >40 through >48. Figure 4-13 shows the data transfer.

**4.5.2.7 STCR Example.** As in the LDCR example, if a device is connected to the CRU and the base address in workspace register 12 is set to CRU line >40, the following instruction places the seven bits of the character read from the device into the seven least significant bits of the byte at the address in workspace register two:

```
STCR *R2,7 READ CHARACTER
```

This STCR instruction stores the bits as shown in Figure 4-14.

If the STCR instruction is changed to STCR \*R2,0, sixteen bits would be transferred from the CRU lines specified by workspace register 12 to the address specified by the contents of register 2. The transfer of data is shown in Figure 4-15.

**4.5.3 CRU Interface Example**

The following paragraphs contain an example of programming for a CRU device to aid the user in programming any CRU device that the executive does not support. A medium-speed line printer having the characteristics listed in Table 4-4 is used as an example device, although this device is supported by the executives.

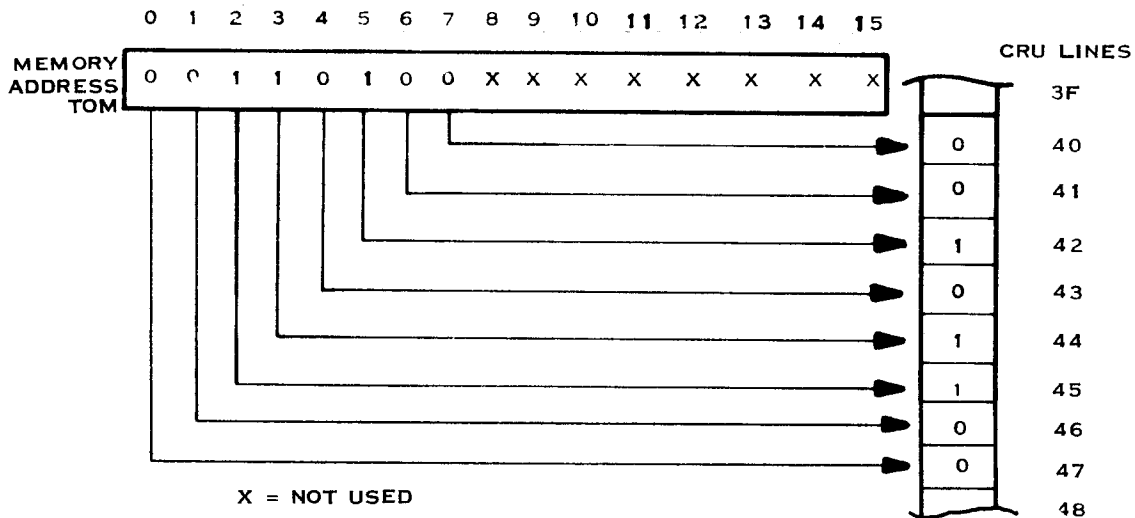


Figure 4-12. LDCR Byte Transfer

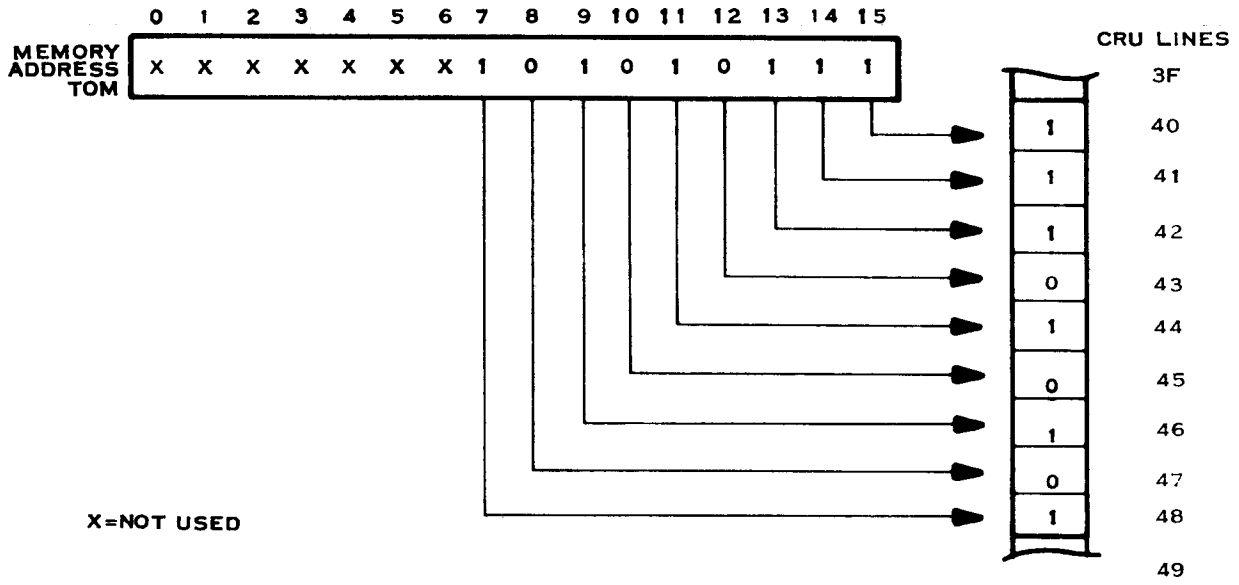


Figure 4-13. LDCR Word Transfer

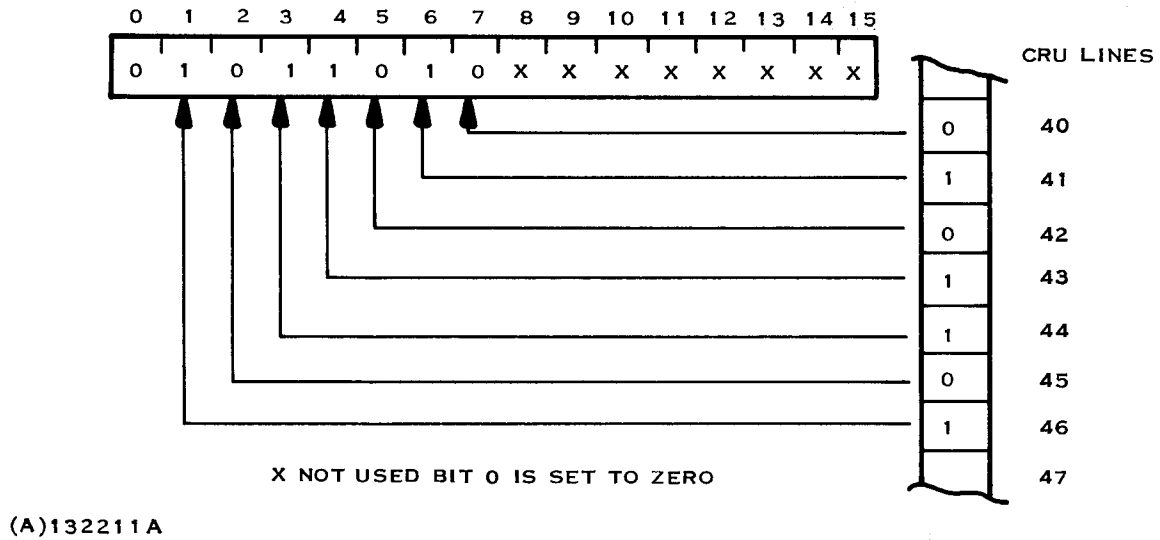


Figure 4-14. STCR Byte Transfer

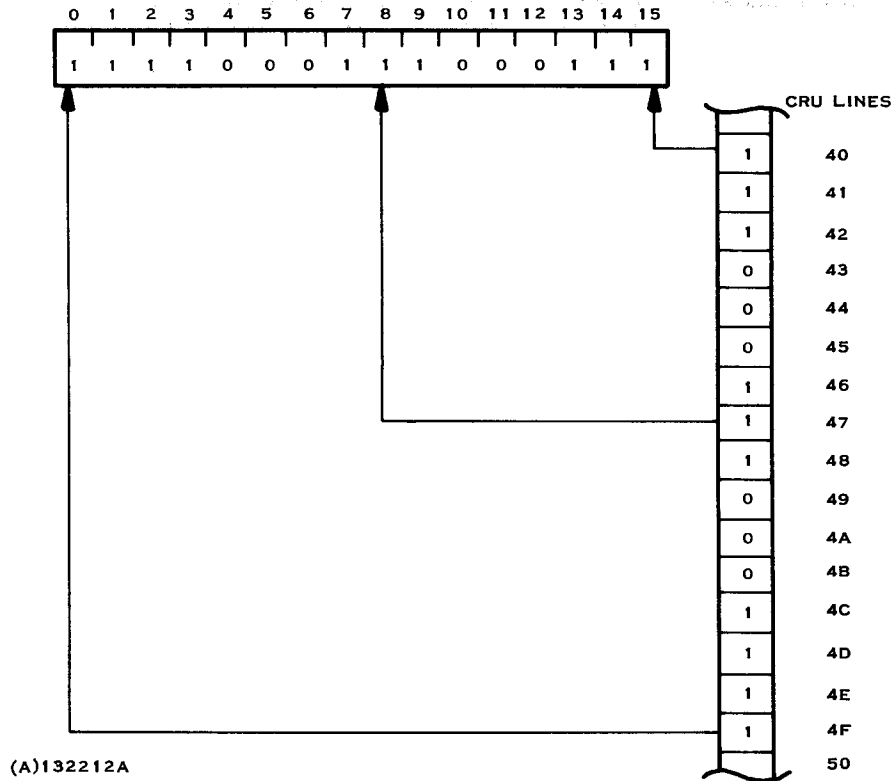


Figure 4-15. STCR Word Transfer

Table 4-4. Medium-Speed Line Printer Characteristics

Function	Description
Print line length	80 characters maximum
Paper width	Variable, up to 9 1/2 inches, sprocket-fed
Character format	5X7 dot matrix, 10 characters per inch (horizontal) 6 lines per inch (vertical)
Printer speed	60 lines per minute for 80 character lines or 150 lines per minute for 20 character lines
Printer input buffer	80 characters
Buffer data rate	75,000 characters per second (8-bit characters supplied in parallel) maximum



**4.5.3.1 Software Interface Requirements.** The control characters recognized by the line printer and the control and response signals for the printer are listed in Table 4-5. An arbitrary CRU signal arrangement shown in Figure 4-16 has been selected for this example.

**Table 4-5. Printer Control and Response Signals**

Signal	Definition	Hexadecimal Value
<b>Control Characters</b>		
LF	Line Feed	0A <sub>16</sub>
CR	Carriage Return	0D <sub>16</sub>
TOF	Top of Form	0C <sub>16</sub>
PD	Printer Deselect	13 <sub>16</sub>
<b>Discrete Signals</b>		
PP	Printer Prime →	
PSD	Printer Selected ←	
PF	Printer Fault ←	
BSY	Printer Busy ←	
IM	Interrupt Mask →	
IR	Interrupt Reset →	
ACK	Acknowledge ←	
PS	Strobe →	

**Notes:**

- Signal from printer
- Signal to printer

**4.5.3.2 Software Routines Required.** To properly operate the medium-speed line printer, software routines must provide initialization, character transfer, and end-of-data reporting. The following paragraphs define these operations and provide specific programming examples.

CRU BITS	CRU OUTPUTS	PRINTER OUTPUTS (CRU INPUTS)
0	DATA BIT 0 (LSB)	BUSY
1	DATA BIT 1	FAULT
2	DATA BIT 2	SELECTED
3	DATA BIT 3	NOT USED
4	DATA BIT 4	NOT USED
5	DATA BIT 5	NOT USED
6	DATA BIT 6	NOT USED
7	DATA BIT 7 (MSB)	NOT USED
8	STROBE	NOT USED
9	PRIME	NOT USED
10	NOT USED	NOT USED
11	NOT USED	NOT USED
12	NOT USED	NOT USED
13	NOT USED	NOT USED
14	INTERRUPT MASK	NOT USED
15	INTERRUPT RESET	ACKNOWLEDGE -

Figure 4-16. CRU Bit Assignments

4.5.3.3 Initialization. Initialization should occur when power is applied to the system. A generalized approach to initialization with specific printer initialization follows:

	AORG	0	
	DATA	PWRONW	INITIALIZE POWER ON
	DATA	PWRONP	INTERRUPT VECTOR
	.		OTHER VECTORS
	.		
	RORG		REMAINDER OF PROGRAM RELOCATABLE
PWRONP	EQU	\$	POWER ON INITIALIZATION
	.		OTHER INITIALIZATIONS
	.		
PRBASE	EQU	>120	CONNECTED TO MODULE SELECT 9
PRIME	EQU	9	RESET PRINTER
STROBE	EQU	8	TAKE DATA
MASK	EQU	14	INTERRUPT MASK
INT	EQU	15	INTERRUPT RESET
BUSY	EQU	0	PRINTER BUSY
*			

\*PRINTER INITIALIZATION

\*

L1	12,PRBASE	LOAD CRU BASE ADDRESS
SBZ	PRIME	SET PRIME TO ZERO
SBZ	STROBE	SET STROBE TO ZERO
SBZ	MASK	MASK INTERRUPTS

**4.5.3.4 Character Transfer.** Character transfer can be accomplished as follows by the use of a subroutine call. The assumptions for the subroutine are as follows:

- Workspace register 8 (WR8) contains the address of the data to be printed.
- Workspace register 9 (WR9) is used for temporary storage.
- Workspace register 10 (WR10) contains the number of characters to transfer.
- Workspace register 12 (WR12) contains the CRU base address.

The following subroutine is one method of transferring characters to the printer:

```

PRINTR    EQU    $          PRINT SUBROUTINE
*
*SET UP INTERRUPTS
*
        SBZ    INT          RESET INTERRUPT
        LIM1   15          ENABLE LEVEL 15
        SBO    MASK        ENABLE INTERRUPTS
*
*TEST FOR PRINTER BUSY, PRINT IF NOT
*BUSY, WAIT FOR ANY INTERRUPT IF BUSY
*AND RETRY TEST
*
TSTBSY    TB           BUSY    TEST BUSY BIT
          JEQ          PRINT    JUMP IF NOT BUSY
          IDLE         WAIT IF BUSY
          JMP          TSTBSY   RETRY TEST
*CHARACTER PRINT SUBROUTINE
*
PRINT     EQU    $          START
          MOVB    *8+,9     WR9 CONTAINS PRINT CHAR
          INV     9          INVERT BITS (LOW TRUE DATA)
          LDCR   9,8        OUTPUT TO PRINTER
          SBO    STROBE     PULSE STROBE LINE
          SBZ    STROBE     ABOUT 1.5 MICROSECONDS
          DEC    10         DECREMENT CHARACTER COUNT
    
```

	JEQ	EXIT	EXIT IF COMPLETE
	JMP	TSTBSY	GO FOR NEXT CHARACTER
*EXIT CODE			
*			
EXIT	SBZ	MASK	MASK INTERRUPT
	RTWP		RETURN TO CALLER

**4.5.3.5 End-of-Data Reporting.** End-of-data reporting in the example subroutine is the exit code, which is executed when the character count in WR10 reaches zero. The code masks the interrupt and returns control to the calling routine.

**4.5.3.6 Interrupt Routine.** In the character transfer subroutine, the CPU enters an idle state when the printer is busy. The occurrence of any enabled interrupt signal causes the CPU to resume processing. The printer is assumed to be connected at interrupt level 15, and all levels are enabled following execution of the LIM1 15 instruction. The following interrupt routine resets the printer interrupt and returns control to the instruction following the interrupted instruction, the JMP TSTBSY instruction, in this case:

	AORG	>3C	INTERRUPT LEVEL 15 VECTOR
	DATA	PRIWP	WORKSPACE ADDRESS
	DATA	PRIPC	PROGRAM ADDRESS
.			
PRIWP	RORG	\$\$-24	SET WORKSPACE ADDRESS RELATIVE TO
	DATA	PRBASE	CRU BASE ADDRESS
	RORG	\$\$+6	RESERVE REMAINDER OF WORKSPACE
PRIPC	EQU	\$	INTERRUPT ROUTINE
	SBZ	INT	RESET INTERRUPT
	RTWP		

**4.5.3.7 Programming Notes.** A specific device to be programmed might require more sophisticated routines. These examples are intended to show possible ways of programming input and output for a CRU device. Error tests may be included to transfer control to error recovery routines when errors are detected. Error recovery routines may simply indicate the occurrence of an error or may correct or overcome the error.

#### 4.5.4 TILINE Input/Output

The set of machine instructions that communicate with CPU memory may be used to communicate with devices connected to the TILINE by coding the addresses in the instructions with the TILINE addresses for the device. The hardware supplies the five most significant bits, each having a value of one, to convert the 16-bit memory addresses to 21-bit TILINE addresses. The actual TILINE addresses for a device and the significance of data transferred to these addresses are device dependent. The operating systems support I/O to the standard disk units. The user programs executing under an operating system use the I/O supervisor call to perform I/O to the disk.

**4.5.4.1 TILINE Interface Example.** The following paragraphs contain an example of programming for a TILINE device to aid the user in programming any TILINE device that the executive does not support. Figure 4-17 shows a typical TILINE device, a disk controller, which is used for this example. Actual input/output for the disk is provided by the executive. This example is only intended to illustrate the principles involved in TILINE input/output programming.

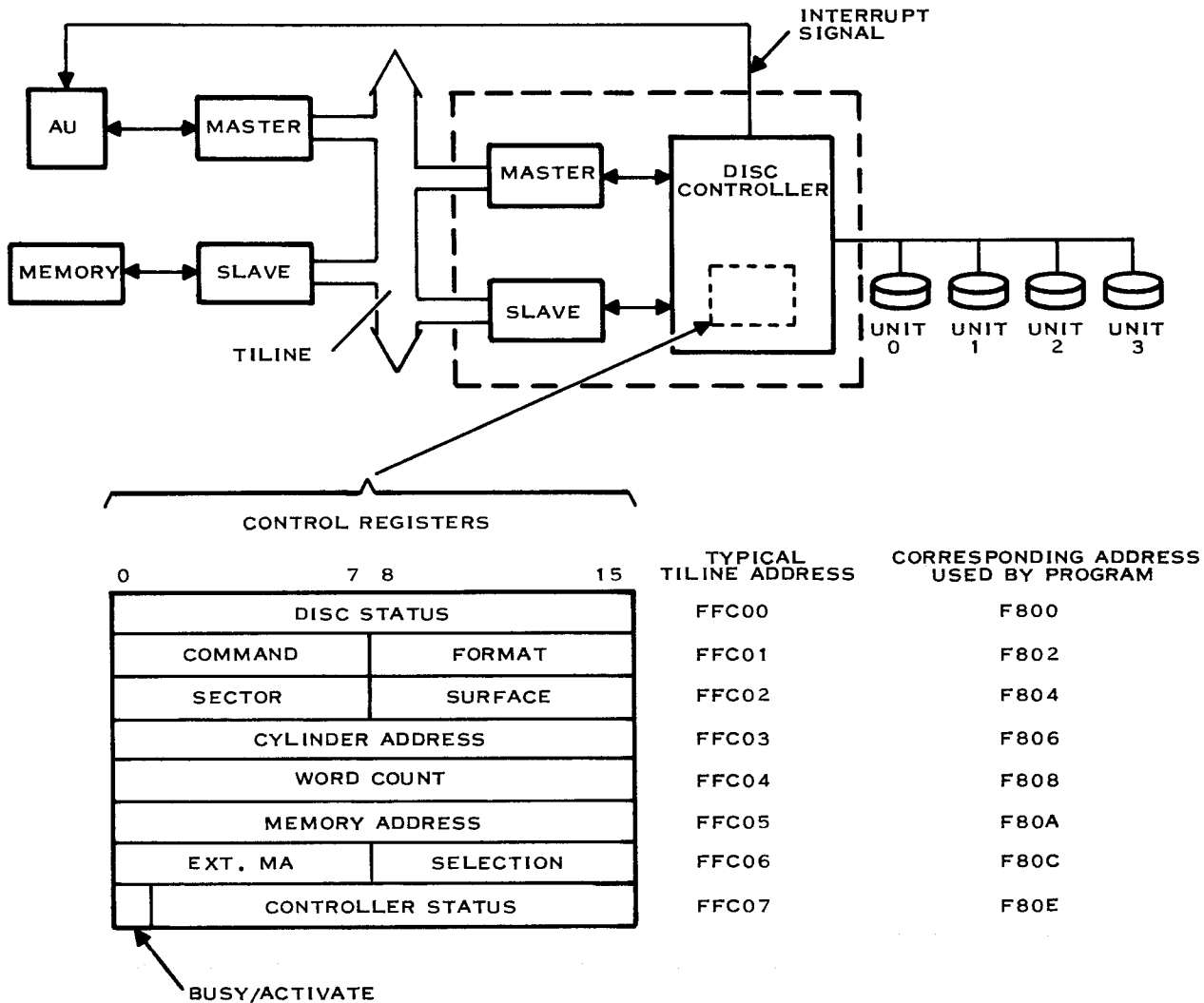


Figure 4-17. TILINE Device Controller Example

**4.5.4.2 Peripheral Controller Application.** Controllers for peripheral devices connected to the TILINE have a master interface for transferring data to and from memory. They also have a slave interface for receiving command information from the AU and for sending status information to the AU. A simplified block diagram for a disk controller is shown in Figure 4-17. Typical use of control registers accessed by the slave interface is also shown in Figure 4-17. It is assumed that address  $F800_{16}$  is assigned to the controller. This corresponds to TILINE address  $FFC00_{16}$ . A program would operate the disk controller by moving the appropriate parameters into the control registers and setting the activate bit as follows:

PARAMS	DATA	>017F	COMMAND,FORMAT
	DATA	>0000	SECTOR,SURFACE
	DATA	0	CYLINDER ADDRESS
	DATA	>1000	WORD COUNT
	DATA	BUFF	MEMORY ADDRESS
	DATA	>0201	EXT. MA,SELECTION
*			
*TEST FOR DISK CONTROLLER BUSY			
*			
	LI	7,>7FFF	WR7 = BUSY TEST MASK
	COC	@>F80E,7	TEST FOR BUSY
	JNE	BUSY	IF NOT
*			
*TRANSFER DISK PARAMETER LIST FROM			
*MEMORY TO DISK CONTROLLER			
*			
	LI	8,PARAMS	WR8 = PARAMETER LIST ADDRESS
	LI	9,>F802	WR9 = DISK CONTROLLER ADDRESS
FILL	MOV	*8 + , *9 +	MOVE LIST WORD TO CONTROLLER
	CI	9,>F80E	STOP WHEN WR9 = CONT. STATUS ADDR.
	JNE	FILL	
	INV	7	WR7 = >8000
	SOC	7,*9	SET ACTIVATE BIT

The disk controller performs the action requested in the command register. During the time the controller is active, the busy flag is on. When the operation is complete the busy flag is turned off and an interrupt signal is generated. The interrupt can be connected to an external interrupt input line, as determined by the system designer.

# Assembler Directives

---

## 5.1 GENERAL

The Model 990 Computer assembly language is processed by the software development system macro assembler, SDSMAC. The section describes the SDSMAC assembler and the directives processed by SDSMAC.

## 5.2 SDSMAC ASSEMBLER

The program development system assembler, SDSMAC, is a two-pass assembler that assembles object code for the 990 computer. SDSMAC executes under the control of an operating system.

A two-pass assembler reads the source statements of a program two times. The first time (first pass), the assembler maintains the location counter, builds a symbol table similar to those in a one-pass assembler, and processes any macro definitions for expansion. The two-pass assembler also copies the source statements for reading during the second pass, but does not assemble any object code. During the second pass, the assembler reads the copy of the source statements, and assembles the object code using the operation codes and the symbol table completed during the first pass.

The only restrictions on forward references are instances in which the value of the symbol affects the location counter.

SDSMAC supplies the additional capability of macro-instructions or macros. A macro is a user-defined set of assembler language source statements. Macro definitions assign a name to the macro and define the source statements of the macro. The macro name may then be used in the operation field of a source statement of the program to cause the assembler to insert the predefined source statements and assemble them along with the other source statements of the program. The macro capability of SDSMAC allows the user to perform the following functions:

- Define macros to specify frequently used sequences of source code
- Define macros for problem-oriented sequences of instructions to provide a means of programming that may be more meaningful to users who are not computer-oriented.

Macros are defined in a macro language consisting of the verbs described in Section 7. In addition to the macro language, SDSMAC supports a number of assembler directives, which are described in subsequent paragraphs of this section.

### 5.3 ASSEMBLER DIRECTIVES

Assembler directives and machine instructions in source programs supply data to be included in the program and control the assembly process. The Model 990 Computer assembler supports a number of directives in the following categories:

- Directives that affect the location counter
- Directives that affect the assembler output
- Directives that initialize constants
- Directives that provide linkage between programs
- Miscellaneous directives.

#### 5.3.1 Directives That Affect the Location Counter

As an assembler reads the source statements of a program, a component of the assembler called the location counter advances to correspond to the memory locations assigned to the resulting object code. The first nine of the assembler directives listed below initialize the location counter and define the value as relocatable, absolute, or dummy. The last three directives advance the location counter to provide a block or an area of memory for the object code to follow. The word boundary directive also ensures a word boundary (even address). The directives are as follows:

- Absolute origin
- Relocatable origin
- Dummy origin
- Data segment
- Data segment end
- Common segment
- Common segment end
- Program segment
- Program segment end
- Block starting with symbol
- Block ending with symbol
- Word boundary



**NOTE**

Any use of RORG, AORG, CSEG, or PSEG that results in the program counter having an odd value causes the byte at the preceding even address to be set to zero. It is good programming practice to include an EVEN directive immediately following any of these directives.

**5.3.1.1 Absolute Origin — AORG.**

*Syntax definition:*

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

AORG places a value in the location counter and defines the succeeding locations as absolute. Use of the label field is optional. When a label is used, it is assigned the value that the directive places in the location counter. The operation field contains AORG. The operand field is optional, but when used, contains a well-defined expression (wd-exp). The assembler places the value of the well-defined expression in the location counter. The comment field is optional and may be used only when the operand field is used. When no AORG directive is entered, no absolute addresses are included in the object program. When the operand field is not used, the length of all preceding absolute code replaces the value of the location counter.

The following example shows an AORG directive:

```
AORG >1000 + X
```

Symbol X must be absolute and must have been previously defined. If X has a value of 6, the location counter is set to 1006<sub>16</sub> by this directive. Had a label been included, the label would have been assigned the value 1006<sub>16</sub>.

**5.3.1.2 Relocatable Origin — RORG.**

*Syntax definition:*

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

RORG places a value in the location counter; if encountered in absolute code, it also defines succeeding locations as program-relocatable. When a label is used, it is assigned the value that the directive places into the location counter. The operation field contains RORG. The operand field is optional, but when it is used, the operand must be an absolute or relocatable expression (exp) that contains only previously defined symbols. The comment field may be used only when the operand field is used.

When the operand field is not used, the length of the program segment, data segment, or specific common segment of a program replaces the value of the location counter. For a given relocation type X (data-, common-, or program-relocatable), the length of the X-relocatable segment at any time during an assembly is either of the following values:

- The maximum value the location counter has ever attained as a result of the assembly of any preceding block of X-relocatable code
- Zero, if no X-relocatable code has been previously assembled

Since the location counter begins at zero, the length of a segment and the next available address within that segment are identical.

If the RORG directive appears in absolute or program-relocatable code and the operand field is not used, the location counter value is replaced by the current length of the program segment of that program. If the directive appears in data-relocatable code without an operand, the location counter value is replaced by the length of the data segment. Likewise, in common-relocatable code, the RORG directive without an operand causes the length of the appropriate common segment to be loaded into the location counter.

When the operand field is used, the operand must be an absolute or relocatable expression (exp) that contains only previously defined symbols. If the directive is encountered in absolute code, a relocatable operand must be program-relocatable; in relocatable code, the relocation type of the operand must match that of the current location counter. When it appears in absolute code, the RORG directive changes the location counter to program-relocatable and replaces its value with the operand value. In relocatable code, the operand value replaces the current location counter value, and the relocation type of the location counter remains unchanged.

The following example shows a RORG directive:

```
RORG $-20 OVERLAY TEN WORDS
```

The \$ symbol refers to the location following the preceding relocatable location of the program. This has the effect of backing up the location counter ten words. The instructions and directives following the RORG directive replace the ten previously assembled words of relocatable code, permitting correction of the program without removing source records. If a label had been included, the label would have been assigned the value placed in the location counter.

```
SEG2 RORG
```

The location counter contents depend upon preceding source statements. Assume that after defining data for a program that occupied  $44_{16}$  bytes, an AORG directive initiated an absolute block of code. The absolute block is followed by the RORG directive in the preceding example. This places  $0044_{16}$  in the location counter and defines the location counter as relocatable. Symbol SEG2 is a relocatable value,  $0044_{16}$ . The RORG directive in the above example would have no effect except at the end of an absolute block or a dummy block.

### 5.3.1.3 Dummy Origin — DORG.

*Syntax definition:*

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

DORG places a value in the location counter and defines the succeeding locations as a dummy block or section. When assembling a dummy section, the assembler does not generate object code but operates normally in all other respects. The result is that the symbols that describe the layout of the dummy section are available to the assembler during assembly of the remainder of the program. The label is assigned the value that the directive places in the location counter. The operation field contains DORG. The operand field contains an expression (exp) which may be either absolute or relocatable. Any symbol in the expression must have been previously defined.

When the operand field is absolute, the location counter is assigned the absolute value. When the operand is relocatable, the location counter is assigned the relocatable value and the same relocation type as the operand. When this occurs, space is reserved in the section that has that relocation type.

The following example shows a DORG directive:

```
DORG 0
```

The effect of this directive is to cause the assembler to assign values relative to the start of the dummy section to the labels within the dummy section. The example directive would be appropriate to define a data structure. The executable portion of the module (following a RORG directive) should use the labels of the dummy section as indexed addresses. In this manner, the data is available to the procedure regardless of the memory area into which the data is loaded.

The following example shows another use of the DORG directive:

```
RORG 0
.
.
. (code as desired)
.
DORG $
.
. (data segment)
.
END
```

The example of the DORG directive would be appropriate for the executable portion (procedure division) of a disk-resident procedure that is common to more than one task, and that executes under the disk executive. The code corresponding to the dummy section must be assembled in another program module. In this manner, the data in the task portion (dummy section) is available to the procedure portion.

The DORG directive may also be used with data-relocatable or common-relocatable operands to specify dummy data or common segments. The following example illustrates this usage:

```

CSEG 'COM1'

DORG $ "$" HAS A COMMON-RELOCATABLE VALUE
.
.
.
LAB1 DATA $

MASK DATA >F000
.
.
.
CEND

SZC @MASK,@LAB1(R3)
    
```

In the example, no object code is generated to initialize the common segment COM1, but space is reserved and all common-relocatable labels describing the structure of the common block (including LAB1 and MASK) are available for use throughout the program.

#### 5.3.1.4 Block Starting with Symbol — BSS.

*Syntax definition:*

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

BSS advances the location counter by the value of the well-defined expression (wd-exp) in the operand field. Use of the label field is optional. When a label is used, it is assigned the value of the location of the first byte in the block. The operation field contains BSS. The operand field contains a well-defined expression that represents the number of *bytes* to be added to the location counter. The comment field is optional.

The following example shows a BSS directive:

```
BUFF1 BSS 80 CARD INPUT BUFFER
```

This directive reserves an 80-byte buffer at location BUFF1.

#### 5.3.1.5 Block Ending with Symbol — BES.

*Syntax definition:*

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

BES advances the location counter according to the value in the operand field. Use of the label field is optional. When the label is used, it is assigned the value of the location following the block. The operation field contains BES. The operand field contains a well-defined expression that represents the number of *bytes* to be added to the location counter. The comment field is **optional**.

The following example shows a BES directive:

```
BUFF2 BES >10
```

The directive reserves a 16-byte buffer. Had the location counter contained  $100_{16}$  when the assembler processed this directive, BUFF2 would have been assigned the value  $110_{16}$ .

### 5.3.1.6 Word Boundary — EVEN.

*Syntax Definition:*

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

EVEN places the location counter on the next word boundary (even byte address). When the location counter is already on a word boundary, the location counter is not altered. Use of the label field is optional. When a label is used, the value in the location counter after processing the directive is assigned to the label. The operation field contains EVEN. The operand field is not used, and the comment field is optional.

The following example shows an EVEN directive:

```
WRF1 EVEN WORKSPACE REGISTER FILE ONE
```

The directive assures that the location counter contains a word boundary address and assigns the location counter address to label WRF1. Use of an EVEN directive preceding or following a machine instruction or a DATA directive is redundant. The assembler advances the location counter to an even address when it processes a machine instruction or a DATA directive.

### 5.3.1.7 Data Segment — DSEG.

*Syntax definition:*

```
[<label>]b..DSEGb..[<comment>]
```

DSEG places a value in the location counter and defines succeeding locations as data-relocatable. Use of the label field is optional. When a label is used, it is assigned the data-relocatable value that the directive places in the location counter. The operation field contains DSEG. The operand field is not used, and the comment field is optional. One of the following values is placed in location counter:

- The maximum value the location counter has ever attained as a result of the assembly of any preceding block of data-relocatable code
- Zero, if no data-relocatable code has been previously assembled

The DSEG directive defines the beginning of a block of data-relocatable code. The block is normally terminated with a DEND directive (see paragraph 5.3.1.8). If several such blocks appear throughout the program, they together comprise the data segment of the program. The entire data segment may be relocated independently of the program segment at link-edit time and, therefore, provides a convenient means of separating modifiable data from executable code.

In addition to the DEND directive, the following directives will properly terminate the definition of a block of data-relocatable code: PSEG, CSEG, AORG, and END. The PSEG directive, like DEND, indicates that succeeding locations are program-relocatable. The CSEG and AORG directives effectively terminate the data segment by beginning a common segment or an absolute segment, respectively. The END directive terminates the data segment as well as the program.

The following example illustrates the use of both the DSEG and the DEND directives.

```

RAM DSEG START OF DATA AREA
.
.
.
.
<Data-relocatable code>
.
.
.
.
ERAM DEND

LRAM EQU ERAM-RAM

```

The block of code between the DSEG and DEND directives is data-relocatable. RAM is the symbolic address of the first word of this block; ERAM is the data-relocatable byte address of the location following the code block. The value of the symbol LRAM is the length in bytes of the block.

### 5.3.1.8 Data Segment End — DEND.

*Syntax definition:*

```
[<data>]b..DENDb..[<comment>]
```

DEND terminates the definition of a block of data-relocatable code by placing a value in the location counter and defining succeeding locations as program-relocatable. Use of the label field is optional. When a label is used, it is assigned the value of the location counter *prior* to modification. The operation field contains DEND. The operand field is not used, and the comment field is optional. One of the following values is placed in the location counter as a result of this directive:

- The maximum value the location counter has ever attained as a result of the assembly of any preceding block of program-relocatable code
- Zero, if no program-relocatable code has been previously assembled

If encountered in common-relocatable or program-relocatable code, this directive functions as a CEND or PEND (and a warning message is issued); like CEND and PEND, it is invalid when used in absolute code. See paragraph 5.3.1.7 for an example of the DEND directive.

### 5.3.1.9 Common Segment — CSEG.

*Syntax definition:*

```
[<label>]b..CSEGb..'<string>'b..[<comment>]
```

CSEG places a value in the location counter and defines succeeding locations as common-relocatable (that is, relocatable with respect to a common segment). Use of the label field is optional. When a label is used, it is assigned the value that the directive places in the location counter. The operation field contains CSEG, and the operand field is optional. The comment field may be used only when the operand field is used.

If the operand field is not used, the CSEG directive defines the beginning of (or continuation of) the blank common segment of the program. When the operand field is used, it must contain a character string of up to six characters enclosed in quotes. (If the string is longer than six characters, the assembler prints a truncation error message and retains the first six characters of the string.) If this string has not previously appeared as the operand of a CSEG directive, the assembler associates a new relocation type with the operand, sets the location counter to zero, and defines succeeding locations as relocatable with respect to the new relocatable type. When the operand string has been previously used in a CSEG, the succeeding code represents a continuation of that particular common segment associated with the operand. The location counter is restored to the maximum value it previously attained during the assembly of any portion of the particular common segment.

The following directives will properly terminate the definition of a block of common-relocatable code: CEND, PSEG, DSEG, AORG, and END. The block is normally terminated with a CEND directive (see paragraph 5.3.1.10). The PSEG directive, like CEND, indicates that succeeding locations are program-relocatable. The DSEG and AORG directives effectively terminate the common segment by beginning a data segment or an absolute segment. The END directive terminates the common segment as well as the program.

The CSEG directive permits the construction and definition of independently relocatable segments of data that several programs may access or reference at execution time. The segments are the assembly language counterparts of FORTRAN blank COMMON and labeled COMMON, and in fact permit assembly language programs to communicate with FORTRAN programs which use COMMON. Information placed in the object code by the assembler permits the link editor to relocate all common segments independently and to make appropriate adjustments to all addresses that reference locations within common segments. Locations within a particular common segment may be referenced by several different programs if each program contains a CSEG directive with the same operand or no operand.

The following example illustrates the use of both the CSEG and the CEND directives:

```

COM1A CSEG 'ONE'
    <Common-relocatable code, type 'ONE'>
    .
    .
    .
    CEND
COM2A CSEG 'TWO'
    .
    .
    .
    <Common-relocatable code, type 'TWO'>
    .
    .
    .
COM2B CEND
COM1C CSEG 'ONE'
    .
    .
    .
    <Common-relocatable code, type 'ONE'>
    .
    .
    .
COM1B CEND

    COM1L DATA COM1B-COM1A LENGTH OF SEGMENT 'ONE'
    COM2L DATA COM2B-COM2A LENGTH OF SEGMENT 'TWO'
    
```

The three blocks of code between the CSEG and the CEND directives are common-relocatable. The first and third blocks are relocatable with respect to one common relocation type; the second is relocatable with respect to another. The first and third blocks comprise the common segment 'ONE', and the value of the symbol COM1L is the length in bytes of this segment. The symbol COM2A is the symbolic address of the first word of common segment 'TWO'; COM2B is the common-relocatable (type 'TWO') byte address of the location following the segment. (Note that the symbols COM2B and COM1C are of different relocation types and possibly different values.) The value of the symbol COM2L is the length in bytes of common segment 'TWO'.

### 5.3.1.10 Common Segment End — CEND.

*Syntax definition:*

```
[<label>]b..CENDb..[<comment>]
```

CEND terminates the definition of a block of common-relocatable code by placing a value in the location counter and defining succeeding locations as program-relocatable. Use of the label field



is optional. When a label is used, it is assigned the value of the location counter prior to modification. The operation field contains CEND. The operand field is not used, and the comment field is optional. One of the following values is placed in the location counter as a result of this directive:

- The maximum value the location counter has ever attained as a result of the assembly of any preceding block of program-relocatable code
- Zero, if no program-relocatable code has been previously assembled

If encountered in data- or program-relocatable code, this directive functions as a DEND or PEND (and a warning message is issued); like DEND and PEND, it is invalid when used in absolute code. See paragraph 5.3.1.9 for an example of the use of the CEND directive.

### 5.3.1.11 Program Segment — PSEG.

*Syntax Definition:*

```
[<label>] PSEG <comment>
```

PSEG places a value in the location counter and defines succeeding locations as program-relocatable. When a label is used, it is assigned the value that the directive places in the location counter. The operation field contains PSEG. The operand field is not used and the comment field is optional. One of the following values is placed in the location counter:

- The maximum value the location counter has ever attained as a result of the assembly of any preceding block of program-relocatable code
- Zero, if no program-relocatable code has been previously assembled

The PSEG directive is provided as the program-segment counterpart to the DSEG and CSEG directives. Together, the three directives provide a consistent method of defining the various types of relocatable segments. The following sequences of directives are functionally identical:

DSEG	DSEG
.	.
.	.
.	.
<Data-relocatable code>	<Data-relocatable code>
.	.
.	.
.	.
DEND	.
CSEG	CSEG
.	.
.	.
.	.
<Common-relocatable code>	<Common-relocatable code>
.	.
.	.
.	.
CEND	.

PSEG	PSEG
.	.
.	.
.	.
<Program-relocatable code>	<Program-relocatable code>
.	.
PEND	.
.	.
END	END

**5.3.1.12 Program Segment End — PEND.**

*Syntax definition:*

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

The PEND directive is provided as the program-segment counterpart to the PEND and CEND directives. Like those directives, it places a value in the location counter and defines succeeding locations as program-relocatable (however, since PEND properly appears only in program-relocatable code, the relocation type of succeeding locations remains unchanged). Use of the label field is optional. When a label is used, it is assigned the value of the location counter *prior* to modification. The operation field contains PEND. The operand field is not used, and the comment field is optional. The value placed in the location counter by this directive is simply the maximum value ever attained by the location counter as a result of the assembly of all preceding program-relocatable code. If encountered in data- or common-relocatable code, this directive functions as a DEND or CEND (and a warning message is issued). Like DEND and CEND, it is invalid when used in absolute code. See paragraph 5.3.1.11 for an example of the use of the PEND directive.

**5.3.2 Directives That Affect the Assembler Output**

This category includes the directive that supplies a program identifier in the object code and five directives that affect the source listing. The directives in this category are as follows:

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

**5.3.2.1 Output Options — OPTION.**

*Syntax definition:*

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

OPTION specifies output and list options to the assembler. No label is entered with the OPTION directive. The operation field contains OPTION. The operand field contains one or more keywords to specify the desired options. The comment field is optional. The keywords are as follows:

- XREF — Print a cross-reference listing at the end of the source and object listing.
- RXREF — Print a reduced cross-reference list at the end of the source listing. (The reduction deletes all symbols defined in statements COPY'd into the source that are not referenced in the original source.)
- SYMT — Output a symbol table in the object code that contains all symbols in the program.
- NOLIST — Suppress printing of any listing and override other directives and keywords.
- TUNLST — Limit the listing for text directives to a single line.
- DUNLST — Limit the listing for data directives to a single line.
- BUNLST — Limit the listing for byte directives to a single line.
- MUNLST — Limit the listing for macro expansion to a single line.
- FUNL — Override all unlist directives.

The following example shows an OPTION directive:

```
OPTION XREF,SYMT
```

The directive in the example specifies the printing of a cross-reference listing and the output of a symbol table with the object code.

### 5.3.2.2 Program Identifier — IDT.

*Syntax definition:*

```
[<label>]IDT'<string>'[<comment>]
```

IDT assigns a name to the program. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains IDT. The operand field contains the program name (string), a character string of up to eight characters within single quotes. When a character string of more than eight characters is entered, the assembler prints a truncation error message and retains the first eight characters as the program name. The comment field is optional.

The following example shows an IDT directive:

```
IDT 'CONVERT'
```

The directive assigns the name CONVERT to the program to be assembled. The program name is printed in the source listing as the operand of the IDT directive and appears in the page heading of the source listing. The program name is placed in the object code, but serves no purpose during the assembly.

#### NOTE

Although SDSMAC will accept lowercase letters and special characters within the quotes; ROM loaders, for example, will not. Therefore, only uppercase letters and numerals are recommended.

#### 5.3.2.3 Page Title — TITL.

*Syntax definition:*

```
[<label>]TITL'<string>'[<comment>]
```

TITL supplies a title to be printed in the heading of each page of the source listing. When a title is desired in the heading of the first page of the source listing, a TITL directive must be the first source statement submitted to the assembler. This directive is not printed in the source listing. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains TITL. The operand field contains the title (string), a character string of up to 50 characters enclosed in single quotes. When more than 50 characters are entered, the assembler retains the first 50 characters as the title and prints a truncation error message. The comment field is optional; the assembler does not print the comment but does increment the line counter.

The following example shows a TITL directive:

```
TITLE'***REPORT GENERATOR***'
```

This directive causes the title \*\*\*REPORT GENERATOR\*\*\* to be printed in the page headings of the source listing. When a TITL directive is the first source statement in a program, the title is printed on all pages until another TITL directive is processed. Otherwise, the title is printed on the next page after the directive is processed, and on subsequent pages until another TITL directive is processed.

#### 5.3.2.4 List Source — LIST.

*Syntax definition:*

```
[<label>]LIST[<comment>]
```

LIST restores printing of the source listing. This directive is required only when a no source list (UNL) directive is in effect to cause the assembler to resume listing. This directive is not printed in the source listing, but the line counter increments. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains LIST. The operand field is not used. Use of the comment field is optional, but the assembler does not print the comment.

The following example shows a LIST directive:

```
LIST
```

The directive causes the source listing to be resumed with the next source statement.

### 5.3.2.5 No Source List — UNL.

*Syntax Definition:*

```
[<label>]b..UNLb..[<comment>]
```

UNL inhibits printing of the source listing. The UNL directive is not printed in the source listing, but the line counter increments. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains UNL. The operand field is not used. Use of the comment field is optional, but the assembler does not print the comment.

The following example shows a UNL directive:

```
UNL
```

The UNL directive inhibits printing of the source listing. The UNL directive can be used to reduce assembly time and the size of the source listing.

### 5.3.2.6 Page Eject — PAGE.

*Syntax definition:*

```
[<page>]b..PAGEb..[<comment>]
```

PAGE causes the assembler to continue the source program listing on a new page. The PAGE directive is not printed in the source listing, but the line counter increments. Use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains PAGE. The operand field is not used. Use of the comment field is optional, but the assembler does not print the comment.

The following example shows a PAGE directive:

```
PAGE
```

The directive causes the assembler to begin a new page of the source listing. The next source statement is the first statement listed on the new page. Use of the page directive to begin new pages of the source listing at the logical divisions of the program improves documentation of the program.

### 5.3.3 Directives That Initialize Constants

This category includes directives that place values in successive bytes or words of the object code, and a directive that places characters of text in the object code to be displayed or printed. It also includes a directive that initializes a constant for use during the assembly process. The directives are as follows:

- Initialize byte
- Initialize word
- Initialize text
- Define assembly-time constant
- Define checkpoint register
- Define workspace pointer

#### 5.3.3.1 Initialize Byte — BYTE.

*Syntax definition:*

```
[<label>]b..BYTEb..<exp>[,<exp>]...b..[<comment>]
```

BYTE places one or more values in one or more successive bytes of memory. Use of the label field is optional. When a label is used, the location at which the assembler places the first byte is assigned to the label. The operation field contains BYTE. The operand field contains one or more expressions separated by commas. The expressions must contain no external references. The assembler evaluates each expression and places the value in a byte as an eight-bit two's complement number. When truncation is required, the assembler prints a truncation warning message and places the rightmost portion of the value in the byte. The comment field is optional.

The following example shows a BYTE directive:

```
KONS BYTE >F + 1,-1,'D'-'=' ,0,'AB'-'AA'
```

The directive initializes five bytes, starting with a byte at location KONS. The contents of the resulting bytes are 00010000, 11111111, 00000111, 00000000, and 00000001.

#### 5.3.3.2 Initialize Word — DATA.

*Syntax definition:*

```
[<label>]b..DATAb..<exp>[,<exp>]...b..[<comment>]
```

Data places one or more values in one or more successive words of memory. The assembler advances the location counter to a word boundary (even) address. Use of the label field is optional. When a label is used, the location at which the assembler places the first word is assigned to the label. The operation field contains DATA. The operand field contains one or more expressions separated by commas. The assembler evaluates each expression and places the value in a word as a 16-bit two's complement number. The comment field is optional.

The following example shows a DATA directive:

```
KONS1 DATA 3200,1 + 'AB',-'AF',>F4A0,'A'
```

The directive initializes five words, starting with a word at location KONS1. The contents of the resulting words are 0C80<sub>16</sub>, 4143<sub>16</sub>, BEBA<sub>16</sub>, F4A0<sub>16</sub>, and 0041<sub>16</sub>. Had the location counter contents been 010F<sub>16</sub> prior to processing this directive, the value assigned to KONS1 would be 0110<sub>16</sub>.

### 5.3.3.3 Initialize Text — TEXT.

*Syntax definition:*

```
[<label>]b..TEXTb.[-]'<string>'b..[<comment>]
```

TEXT places one or more characters in successive bytes of memory. The assembler negates the last character of the string when the string is preceded by a minus (–) sign (unary minus). Use of the label field is optional. When a label is used, the location at which the assembler places the first character is assigned to the label. The operation field contains TEXT. The operand field contains a character string of up to 52 characters enclosed in single quotes, which may be preceded by a unary minus sign. The comment field is optional.

The following example shows a TEXT directive:

```
MSG1 TEXT 'EXAMPLE' MESSAGE HEADING
```

The directive places the eight-bit ASCII representations of the characters in successive bytes. When the location counter is on an even address, the result, in hexadecimal representation, is 4558, 414D, 504C, and 45XX. XX represents the contents of the rightmost byte of the fourth word, which are determined by the next source statement. The label MSG1 is assigned the value of the first byte address in which 45 is placed. Another example, showing the use of a unary minus, is as follows:

```
MSG2 TEXT -'NUMBER'
```

When the location counter is on an even address, the result, in hexadecimal representation, is 4E55, 4D42, and 45AE. The label MSG2 is assigned the value of the byte address in which 4E is placed.

### 5.3.3.4 Define Assembly-Time Constant — EQU.

*Syntax definition:*

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

#### NOTE

<exp> may not be a REF'd symbol.

EQU assigns a value to a symbol. The label field contains the symbol to be given a value. The operation field contains EQU. The operand field contains an expression. Use of the comment field is optional.

The following example shows an EQU directive:

```
SUM EQU R5 WORKSPACE REGISTER 5
```

The directive assigns an absolute value to the symbol SUM, making SUM available to use as a workspace register address. Another example of an EQU directive is as follows:

```
TIME EQU HOURS
```

The directive assigns the value of previously defined symbol HOURS to symbol TIME. When HOURS appears in the label field of a machine instruction in a relocatable block of the program, the value is a relocatable value. The two symbols may be used interchangeably. Symbols in the operand field must be previously defined.

### 5.3.3.5 Checkpoint Register — CKPT.

Syntax definition:

```
[<label>]b..CKPTb..<wa>b..[<comment>]
```

CKPT reserves a register <wa> for use by instructions using byte strings. Use of the comment field and label field is optional. The CKPT directive is available on the 990/12 only.

The following example shows a CKPT:

```
CKPT R6
```

The CKPT directive is used by the Format 12 instructions to imply a checkpoint register. The checkpoint register is used to decrease the number of operands necessary in the instruction, thereby increasing the readability.

Multiple CKPT directives may appear in a module. The most recent CKPT directive operand is used in the assembly of an instruction requiring an implied checkpoint register.

CKPT example:

	CKPT	R6
	.	.
	.	.
1.	MOVS	@A,@B,,R6
	.	.
	.	.



```

2.  CKPT      R7
    .
    .
3.  MOVS     @A,@B
    .
    .
4.  MOVS     @A,@B,,R6
    .
    .
5.  CKPT      R4
    .
    .
6.  MOVS     @A,@B
    .
    .

```

In this program segment for line one, the workspace register R6 is used as a checkpoint register. For line three, an implied checkpoint register is used, in this case, R7, which was defined in line two. In line four the explicitly specified CKPT register (R6) overrides the implied CKPT register (R7). In line five, a new implied CKPT register is specified (R4), so the CKPT register for line six is R4.

### 5.3.3.6 Workspace Pointer — WPNT.

*Syntax definition:*

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

WPNT defines the current workspace to the assembler. WPNT generates no object code. The user must provide a machine instruction to actually place the value in the workspace register. The symbol in the label field, when used, must represent a word (even) address and must have been previously defined. The operation field contains WPNT. The operand field contains the label assigned to the workspace. The comment field is optional.

The following example shows a WPNT directive:

```
WPNT WORK
```

The directive in the example is appropriate when the workspace at location WORK is the active workspace. The assembler stores the value of label WORK as the current workspace address, and from this information identifies symbolic addresses as workspace registers when the symbolic addresses have values less than WORK plus 15. The assembler also recognizes WORK or a label equal to WORK as workspace register zero. Symbolic addresses having values outside this range are considered to be symbolic memory addresses.

### 5.3.4 Directives That Provide Linkage Between Programs

The category consists of two directives that enable program modules to be assembled separately and integrated into an executable program. One directive places one or more symbols defined in the module into the object code, making them available for linking. The other directive places symbols used in the module but defined in another module into the object code, allowing them to be linked. The directives are:

- External Definition
- External Reference
- Secondary External Reference
- Force Load

#### 5.3.4.1 External Definition — DEF.

*Syntax definition:*

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

DEF makes one or more symbols available to other programs for reference. The use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains DEF. The operand field contains one or more symbols, separated by commas, to be defined in the program being assembled. The comment field is optional.

The following example shows a DEF directive:

```
DEF ENTER,ANS
```

The directive causes the assembler to include symbols ENTER and ANS in the object code so that these symbols are available to other programs.

#### 5.3.4.2 External Reference — REF.

*Syntax definition:*

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

REF provides access to one or more symbols defined in other programs. The use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains REF. The operand field contains one or more symbols, separated by commas, to be used in the operand field of a subsequent source statement. The comment field is optional.

The following example shows a REF directive:

```
REF ARG1,ARG2
```

The directive causes the assembler to include symbols ARG1 and ARG2 in the object code so that the corresponding addresses may be obtained from other programs.

If a symbol is listed in the REF statement, then a corresponding symbol must also be present in a DEF statement in another source module. If a one-to-one matching of symbols does not occur, then an error occurs at link edit time. The system will generate a summary list of all “unresolved references”.

#### NOTE

If a symbol in the operand field of a REF directive is the first operand of a DATA directive which results in the assembler placing the symbol at absolute location zero, the symbol will not be linked correctly. Use of the symbol at other locations will be correctly linked.

#### 5.3.4.3 Secondary External Reference — SREF.

*Syntax definition:*

```
[<label>]b..SREFb..<symbol>[,<symbol>]...b..[<comment>]
```

SREF provides access to one or more symbols defined in other programs. The use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The operation field contains SREF. The operand field contains one or more symbols, separated by commas, to be used in the operand field of a subsequent source statement. The comment field is optional.

The following example shows a SREF directive:

```
SREF ARG1,ARG2
```

The directive causes the link editor to include symbols ARG1 and ARG2 in the object code so that the corresponding addresses may be obtained from other programs.

SREF unlike REF does not require a symbol to have a corresponding symbol listed in a DEF statement of another source module. The symbol will be an unresolved reference, but no error message will be given.

#### NOTE

If a secondary referenced (SREFed) symbol is used as the operand of a DATA directive, and the value of the symbol is loaded at program-relocatable location zero, and only at location zero, the symbol will not be correctly linked.

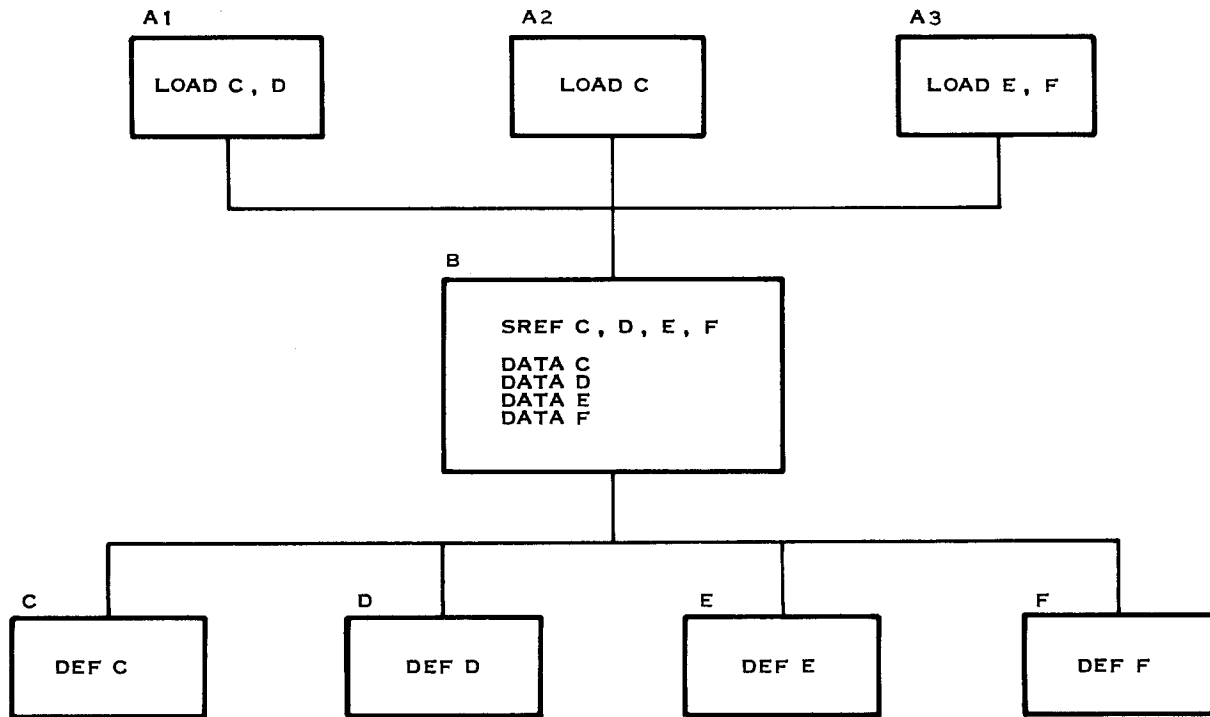
### 5.3.4.4 Force Load — LOAD.

*Syntax definition:*

```
[<label>]b..LOADb..<symbol>[,<symbol>]...b..[<comment>]
```

The load directive is like a REF, but the symbol does not need to be used in the module containing the LOAD. The symbol used in the LOAD must be defined in some other module. LOADs are used with SREFs. If one-to-one matching of LOAD-SREF pairs and DEF symbols does not occur, then unresolved references will occur during link editing.

The following example shows the use of the SREF and the LOAD directives:



Module A1 uses a branch table in module B to get one of the modules C, D, E, or F. Module A1 knows which of the modules C, D, E, and F it will need. Module B has an SREF for C, D, E, and F. Module C has a DEF for C. Module D has a DEF for D. Module E has a DEF for E. Module F has a DEF for F. Module A1 has a LOAD for the modules C and D it needs. Module A2 has a LOAD for the module C it needs. Module A3 has a LOAD for the modules E and F it needs.

The LOAD and SREF directives permit module B to be written to handle the most involved case and still be linked together without unneeded modules since A1 only has LOAD directives for the modules it needs.

When a link edit is performed, automatic symbol resolutions will pull in the modules appearing in the LOAD directives.

If the link control file included A1 and A2, modules C and D would be pulled in while modules E and F would not be pulled in. If the link control file included A3, modules E and F would be pulled in while modules C and D would not be pulled in. If the link control file included A2, module C would be pulled in while modules D, E, and F would not be pulled in.

### 5.3.5 Miscellaneous Directives

This category includes the remainder of the assembler directives which are not applicable to the other categories. The directives are as follows:

- Define extended operation
- Program end
- Copy source file
- Conditional assembly directives
- Define operation
- Define source line length
- Set maximum macro nesting level

#### 5.3.5.1 Define Extended Operation — DXOP.

*Syntax definition:*

```
[<label>]b..DXOPb..<symbol>,<wo-exp>b..[<comment>]
```

DXOP assigns a symbol to be used in the operation field in subsequent lines to specify an extended operation. The use of the label field is optional. When a label is used, the current value in the location counter is assigned to the label. The operation field contains DXOP. The operand field contains a symbol followed by a comma and a well-defined expression. The symbol assigned to an extended operation must not be used in the label or operand field of any other statement. The assembler assigns the symbol to the extended operation specified by the well-defined expression, which must have a value in the range of 0 to 15. The comment field is optional.

The following example shows a DXOP directive:

```
DXOP DADD,13
```

The directive defines DADD as extended operation 13. When the assembler recognizes the symbol DADD in the operation field, it assembles an XOP instruction that specifies extended operation 13. The following example shows the use of the symbol DADD in a source statement:

```
DADD @LABEL1(4)
```

The assembler places the operand field contents in the T, and S fields of an XOP instruction and places 13 in the D field.

### 5.3.5.2 Program End — END.

*Syntax definition:*

```
[<label>]▯..END▯..[<symbol>]▯..[<comment>]
```

END terminates the assembly. The last source statement of a program is the END directive. When any source statements follow the END directive, they are considered part of the next assembly. Use of the label field is optional. When a label is used, the current value in the location counter is assigned to the symbol. The operation field contains END. Use of the operand field is optional. When the operand field is used, it contains a program-relocatable or absolute symbol that specifies the entry point of the program. When the operand field is not used, no entry point is placed in the object code. The comment field may be used only when the operand field is used.

The following example shows an END directive:

```
END START
```

The directive causes the assembler to terminate the assembly of this program. The assembler also places the value of START in the object code as an entry point.

When a program executes in a stand-alone mode and is loaded by the ROM loader, it must supply an entry point to the loader. When no operand is included in the END directive and that program is loaded by the ROM loader, the loader transfers control to the entry point of the loader and attempts to load another object program.

### 5.3.5.3 Copy Source File — COPY.

*Syntax definition:*

```
[<label>]▯..COPY▯..<file name>▯..[<comment>]
```

COPY changes the source input for the assembler. Use of the label field is optional. The operation field contains COPY. The operand field contains a file name from which the source statements are to be read. The file name may be the following:

- An access name recognized by the operating system
- A synonym form of an access name

The comment field is optional.

The following example shows a COPY directive:

```
COPY .SFILE
```

The directive in the example causes the assembler to take its source statements from a file SFILE. At the end-of-file of SFILE, the assembler resumes taking source statements from the file or device from which it was taking source statements when the COPY directive was processed. A COPY directive may be placed in a file being copied, which results in nested copying of files.

### 5.3.5.4 Conditional Assembly Directives — ASMIF, ASMELS, ASMEND.

*Syntax definition:*

```
[<label>]b.ASMIFb.<wd-exp>b..[<comment>]
```

```
.
```

```
.
```

```
Assembly language statements
```

```
.
```

```
.
```

```
b..ASMELSB[<comment>]
```

```
.
```

```
.
```

```
Assembly language statements
```

```
.
```

```
.
```

```
b..ASMENDB.[<comment>]
```

Three directives, ASMIF, ASMELS, and ASMEND, furnish conditional assembly capability in SDSMAC. The three function as IF-THEN-ELSE brackets for blocks of assembly language statements. When the expression in the operand field of an ASMIF evaluates to a non-zero (or true) value, the block of statements enclosed by either ASMIF-ASMEND or ASMIF-ASMELS is assembled. If the block is terminated by ASMELS, the block enclosed by ASMELS-ASMEND is not assembled. When the expression on an ASMIF evaluates to zero (or false), the block of statements immediately following ASMIF is not assembled. If an alternate ASMELS-ASMEND block occurs, it is assembled.

Statements, including macro calls, which occur in portions not assembled are treated as comments. Macros that are treated as comments will not be expanded. Statements that are treated as comments are not assembled. The ASMIF expression must be well defined when it is encountered.

#### NOTE

ASMIF, ASMELS, and ASMEND may not appear as macro model statements. ASMIF-ASMELS-ASMEND constructs may be nested.

The following example shows the use of conditional assembly.

```

SDSMAC 947075 *E    00:01:36
                                                    PAGE 0001

0001      *
0002      *      THIS IS AN EXAMPLE OF A USE OF CONDITIONAL ASSEMBLY
0003      *      TO INCLUDE VARIOUS LEVELS OF DEBUG INFORMATION.
0004      *
0005      *      A SYMBOL IS DEFINED WHICH INDICATES THIS LEVEL
0006      *      0 - NO DEBUG
0007      *      5 - ENTRY /EXIT SHORT DUMPS
0008      *      10- THE ABOVE, OUTER LOOP SHORT DUMPS, ENTRY/EXIT
0009      *      LONG DUMPS
0010      *      15- ALL THE ABOVE & INNER LOOP SHORT DUMPS
0011      *
0012      0000  DEBUG  EQU  12
0013      *      A VALUE OF 12 FOR DEBUG WILL GIVE THE FIRST 3 LEVELS
0014      *      OF DEBUG INFORMATION
0015      *
0016      REF  SRTDMP,LNGDMP
0017      * PROGRAM ENTRY POINT
0018      DEF  ENTRY
0019 0000  ENTRY
0020      ASMIF  DEBUG          IF DEBUG=0, SKIP THIS BLOCK*
0021      ASMIF  DEBUG>5
0022 0000 06A0          BL  LNGDMP          ENTRY POINT SHORT DUMP *
0002 0000
0023      ASMELS
0024      BL  SRTDMP          ENTRY POINT LONG DUMP *
0025      ASMEND
0026      ASMEND *****
0027 0004 018B          MOV  R11,R6          (SAVE RETURN ADDRESS)
0028      *
0029      * <CODE>
0030      *
0031      * OUTER LOOP
0032 0006  LABEL1
0033 0006 0205          LI   R5,>100
0008 0100
0034      * INNER LOOP
0035 000A  LABEL2
0036 000A 0204          LI   R4,6
000C 0006

SDSMAC 947075 *E    00:01:36
                                                    PAGE 0002

0037      *
0038      * <CODE>
0039      *
0040      ASMIF  DEBUG>=15
0041      BL  SRTDMP          INNER LOOP SHORT DUMP
0042      ASMEND
0043 000E 0604          DEC  R4
0044 0010 15FC          JGT  LABEL2
0045      *
0046      * <CODE>
0047      *

```



```

0048          ASMIF DEBUG>=10
0049 0012 06A0          BL  SRTDMP          OUTER LOOP SHORT DUMP
          0014 0000

0050          ASMEND
0051 0016 0605          DEC  R5
0052 0018 15F6          JGT  LABEL1
0053          *
0054          * <CODE>
0055          *
0056          * EXIT POINT
0057          ASMIF DEBUG          IF DEBUG=0, SKIP THIS BLOCK*
0058          ASMIF DEBUG>5          *
0059 001A 06A0          BL  LNGDMP          EXIT LONG DUMP          *
          001C 0002

0060          ASMELS          *
0061          BL  SRTDMP          EXIT SHORT DUMP          *
0062          ASMEND          *
0063          ASMEND          *****
0064 001E 0456          B   *R6          <RETURN THROUGH SAVED REGISTER
0065          END

NO ERRORS

```

### 5.3.5.5 Define Operation — DFOP.

#### Syntax definition:

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

DFOP defines a synonym for an operation. Use of the label field is optional. The operation field contains DFOP. The operand field contains a symbol that is the synonym for an operation, and the operation, which may be the mnemonic operation code of a machine instruction, a macro name, or the symbol of a previous DFOP or DXOP directive. The comment field is optional.

The following example shows a DFOP directive:

```
DFOP LOAD,MOV
```

The directive in the example defines LOAD for a synonym for the MOV machine instruction. The LOAD symbol might be more meaningful where the source is a symbolic memory location and the destination is a workspace register. The machine code for the MOV instruction is assembled whenever either symbol appears in the operation field of a source statement. A single symbol may appear in more than one DFOP directive in the same assembly, and an operation symbol may appear in the label field of a DFOP directive. When an operation symbol appears as the defined symbol of a DFOP directive, the corresponding operation is not available unless it had appeared in the operand field of a previous DFOP directive. The effect of a group of DFOP directives is shown in the following example:

```

DFOP      HOLD,LWPI          HOLD DEFINED TO BE LWPI
DFOP      LWPI,SOMMAC        LWPI REDEFINED AS MACRO SOMMAC
.
.
DFOP      SAVE,HOLD          SAVE DEFINED AS HOLD (LWPI)
DFOP      HOLD,BLWP          HOLD REDEFINED AS BLWP

```

DFOP            LWPI,SAVE            LWPI RESTORED

The first pair of DFOP directives substitutes macro SOMMAC for the LWPI machine instruction that may be specified by the symbol HOLD. The second pair of DFOP directives changes the symbol by which the LWPI machine instruction is specified to SAVE and the symbol by which the BLWP instruction is specified to HOLD. The last DFOP directive restores the symbol LWPI to the LWPI machine instruction.

### 5.3.5.6 Set Right Margin — SETRM.

*Syntax definition:*

```
[<label>]b..SETRMb..<wd-exp>b..[<comment>]
```

The SETRM directive allows the programmer to specify the number of characters in the source line to be considered by the assembler. Use of the label field is optional. The operation field contains SETRM. The operand field contains a well-defined expression in the range from 10 to 80. Use of the comment field is optional.

The assembler uses the value of the expression to determine which is the last character of the source line to be considered for the assembly. Initially, the assembler scans 60 characters. The new value may range from 10 to 80. This value may be changed at any point in the assembly and any number of times.

#### NOTE

The assembler ordinarily prints only 60 source line characters. The programmer may have more lines printed by specifying a value higher than 80 for the PRINT WIDTH parameter of the SCI procedure to execute the assembler.

### 5.3.5.7 Set Maximum Macro Nesting Level — SETMNL.

*Syntax definition:*

```
[<label>]b..SETMNLb..<exp>b..[<comment>]
```

The SETMNL directive allows the programmer to change the maximum macro nesting stack level as required. The assembler maintains a count of the number of levels of macro nesting and declares an error if this count exceeds the maximum number allowed. The default maximum is 16. The SETMNL directive may be used to set the allowed maximum to greater or less than 16.

## 5.4 SYMBOLIC ADDRESSING TECHNIQUES

The assembler processes symbolic memory addresses so that the user may perform the following functions:

- Use the symbolic memory address of a workspace register to address the workspace register
- Omit the @ character to identify a symbolic memory address

When the assembler processes a symbol as an operand of a machine instruction, it compares the value of the current workspace. When the value is equal to the workspace address, or is greater by 15 or less, the symbol represents a workspace and the assembler assembles a workspace register address. Otherwise, the assembler assembles a symbolic memory address. A WPNT directive or an LWPI instruction supplies the address of the current workspace to the assembler. Without this capability, two symbols are frequently assigned to the same address. The following example illustrates this type of coding:

```

SUM      EQU      0          ASSIGN SUM FOR WORKSPACE
                                REGISTER 0
QUAN     EQU      1          ASSIGN QUAN FOR WORK-
                                SPACE REGISTER 1
.
.
WS1      DATA    0          WORKSPACE REGISTER 0
QUANT    DATA    0          WORKSPACE REGISTER 1
FIVE     DATA    5          WORKSPACE REGISTER 2
.
.
        MOV      @FIVE,@QUANT  INITIALIZE QUANTITY
        BLWP    @SUB1          BRANCH TO SUBROUTINE
.
.
SUB1     DATA    WS1        TRANSFER VECTOR
        DATA    ENT1        FOR SUBROUTINE
ENT1     A        QUAN,SUM   ADD QUAN TO SUM

```

The two initial EQU directives assign meaningful labels to be used as workspace register addresses in the subroutine. The labels of the DATA statements are required to access the same memory locations in the main program when another workspace is active. The following code would produce the same object code when assembled:

```

SUM      DATA    0          WORKSPACE REGISTER 0
QUAN     DATA    0          WORKSPACE REGISTER 1
FIVE     DATA    5          WORKSPACE REGISTER 2

```

```

      .
      MOV      FIVE,QUAN      INITIALIZE QUANTITY
      BLWP    SUB1           BRANCH TO SUBROUTINE
      .
SUB1  XVEC    SUM            TRANSFER VECTOR FOR SUBROUTINE
ENT1  A      QUAN,SUM      ADD QUAN TO SUM

```

The MOV instruction in the main program results in symbolic memory addresses for both operands. The BLWP instruction uses transfer vector SUB1, provided by the XVEC directive labeled SUB 1. The XVEC directive also provides a WPNT directive that identifies SUM as the address of the current workspace. The A instruction uses the symbol QUAN (as used in the MOV instruction) but results in a workspace register address, because QUAN is now workspace register one.

The @ character is considered redundant if the following conditions are true:

- All symbols in the expression have been previously defined and the resulting value of the expression is greater than 15, or
- Another @ character prefaces the expression.

The following notations for the MOV instruction in the previous example would generate the same object and result in an error-free assembly:

```

MOV @FIVE,@QUAN
MOV FIVE,QUAN
MOV @@FIVE,@@QUAN

```

#### NOTE

When the @ is omitted from a symbolic expression, the symbol must be defined before its use. If the symbol is not first defined, a register reference is assumed. If later the symbol is defined as a memory reference, an OPERAND CONFLICT PASS1/PASS2 error is generated.

# Pseudo-Instructions

---

## 6.1 GENERAL

A pseudo-instruction is a convenient way to code an operation that is actually performed by a machine instruction with a specific operand. The Model 990 Computer assembly language includes three pseudo-instructions. The pseudo-instructions are as follows:

- No operation
- Return
- Transfer vector

## 6.2 NO OPERATION — NOP

*Syntax definition:*

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

NOP places a machine instruction in the object code. NOP has no effect on execution of the program other than adding to the execution time. Use of the label field is optional. When the label field is used, the label is assigned the location of the instruction. The operation field contains NOP. The operand field is not used. Use of the comment field is optional.

Enter the NOP pseudo-instruction as shown in the following example:

```
MOD NOP
```

Location MOD contains a NOP pseudo-instruction when the program is loaded. Another instruction may be placed in location MOD during execution to implement a program option. The assembler supplies the same object code as if the source statement had contained the following instruction:

```
MOD JUMP $ + 2
```

## 6.3 RETURN — RT

*Syntax definition:*

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

RT places a machine instruction in the object code to return control to a calling routine from a subroutine. Use of the label field is optional. When the label field is used, the label is assigned the location of the instruction. The operation field contains RT. The operand field is not used. Use of the comment field is optional.

Enter the RT pseudo-instruction as shown in the following example:

```
RT
```

The assembler supplies the same object code as if the source statement had contained the following instruction:

```
B *11
```

When control is transferred to a subroutine by execution of a BL instruction, the link to the calling routine is stored in workspace register 11. An RT pseudo-instruction returns control to the instruction following the BL instruction in the calling routine.

#### 6.4 TRANSFER VECTOR — XVEC

*Syntax definition:*

```
<label>XVEC<wp address>[,<subr address>][<comment>]
```

The XVEC pseudo-instruction is a means of coding the transfer vector for a subroutine. XVEC places a set of assembler directives in the source code to provide a transfer vector for a BLWP instruction. XVEC also provides a WPNT directive to define the newly active workspace to the assembler. The label field contains the label of the resulting transfer vector. The operation field contains XVEC. The operand field contains the label (wp address) of the workspace that becomes active when the BLWP instruction is executed. Optionally, the wp address may be followed by a comma and the label (subr address) or the first instruction to be executed in the subroutine. When the second operand is omitted, the assembler assumes that the first instruction to be executed follows the transfer vector. The use of the comment field is optional.

Enter the XVEC pseudo-instruction as shown in the following example:

```
SUBRA XVEC WKSPA,ENTRYA
```

Transfer of control to a subroutine at location ENTRYA with a workspace at location WKSPA becoming the active workspace is coded as follows:

```
BLWP SUBRA
```

The resulting object code and assembler processing is the same as would result from the following directives:

```
SUBRA DATA WKSPA
      DATA ENTRYA
      WPNT WKSPA
```

Alternatively, the XVEC pseudo-instruction may be entered as follows:

```
SUBRA XVEC WKSPA
```

In this case, the executable code of the subroutine must immediately follow the XVEC pseudo-instruction. The resulting object code and assembler processing is the same as would result from the following directives:

```
SUBRA DATA WKSPA  
DATA $+2  
WPNT WKSPA
```

#### NOTE

No executable code that requires a different active workspace than that of the subroutine may be entered between the XVEC pseudo-instruction and the subroutine entry address.

# Macro Language

---

## 7.1 GENERAL

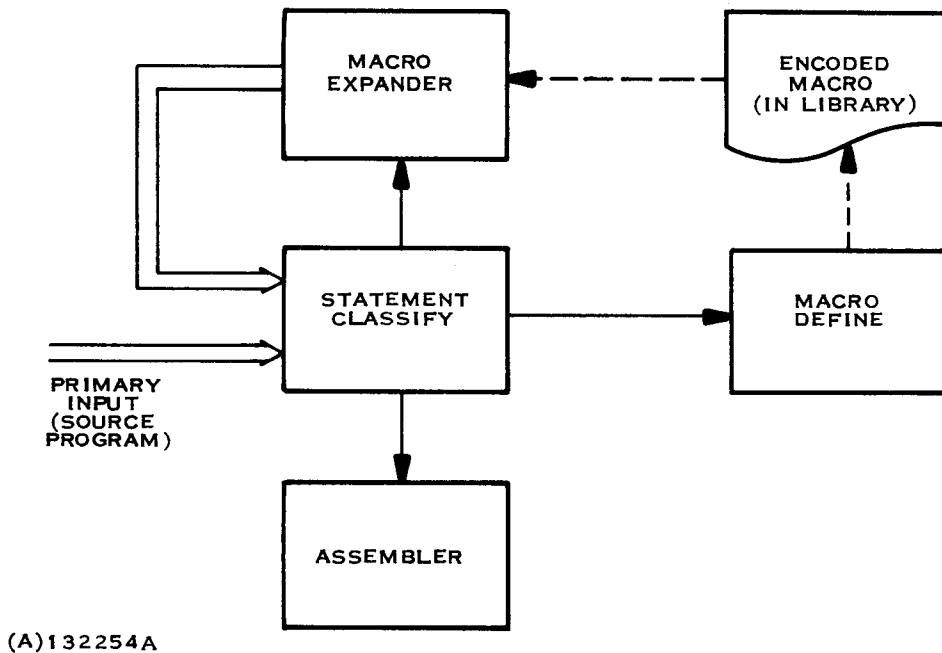
The SDSMAC assembler supports a language used to define macros. A macro definition is a set of statements which, when expanded, produce assembly language source statements (machine instructions and assembler directives). Expansion occurs when a macro call is encountered in an assembly source program. When the assembler processes a macro call, it substitutes the predefined source statements generated from the macro definition for the macro call source statement and assembles the substituted statements as if they had been included in the source program. Macro definitions may be placed in a macro library for use in subsequent programs. This section describes the macro language, the verbs used to define macros, and the macro library directives.

## 7.2 PROCESSING OF MACROS

Figure 7-1 illustrates the data paths between the basic assembler, the macro translator (consisting of the statement classify, macro define, and macro expander modules), and the macro library. The statement classify module processes all source statements to detect macro definitions and macro calls, ignoring nonmacro statements. A macro language statement, \$MACRO, identifies the beginning of a macro definition, and \$END identifies the end of a macro definition. Statements that occur between these two statements constitute a macro definition and are passed to the macro define module. The macro define module processes the macro definition. If a LIBOUT directive has preceded the macro definition, the macro define module writes the macro definition to the specified macro library. If the macro definition is not written to a macro library, the macro may be called only from the program in which it is defined.

The statement classify module scans each operation field of the source code for a match to a defined macro name. This means that a macro definition may be substituted for an assembler mnemonic. If a macro is defined to have the same name as an assembler mnemonic, LI for example, when LI appears in the source program the macro definition will be executed rather than the LI assembler operation. When the statement classify module recognizes a macro call by finding a macro name in an operation field, it passes the name to the macro expander module, which accesses the specified macro definition. The macro expander module first searches memory to see if the macro has been previously defined in the source code. If the macro definition is not found in the source code, the macro expander module next searches the macro libraries available to it. Macro libraries are made available for the reading of macro definitions by specifying the macro library name either in a LIBIN directive, described in paragraph 7.6, or in the system macro library input parameter to the assembler.





**Figure 7-1. Macro Assembler Block Diagram**

Macro definitions may appear anywhere in a program prior to macro calls that activate the definitions and they may be unique to a program or shared by many programs. The macro call is expanded as specified in the macro definition. The source statements that result from this expansion are used as input by the statement classify module. Macro calls are not expanded if they occur in unassembled ASMIF or ASMELS code blocks.

During expansion of a macro call, a macro language statement may call another macro using the \$CALL macro language verb, or a resulting source statement may call a macro by having the macro name in the operation field. A nesting of macro calls can occur in the expansion of one macro call. The macro processor suspends processing of the current macro, processes the new macro, and then resumes processing the original macro at the point of interruption. The macro translator allows a macro to be recursive. To prevent the assembler from getting into an infinite loop, the maximum nesting level for macros is 16. However, the SETMNL directive may be used to change the established maximum as required.

### 7.3 MACRO LIBRARY

A macro library is a directory defined to the operating system. It must be created before any macros may be written to it. Each member file on the directory contains a macro definition; the file name is the name of the macro. Two assembler directives, LIBIN and LIBOUT, identify macro libraries from which macro definitions may be read and to which macro definitions may be written, respectively. Macro library directives are described in paragraph 7.6. In addition, a system macro library, containing macros defined by the user, may be made available to the user program for reading macro definitions by the assembler input parameters.

A macro library may also be used to store extended operation definitions, as defined by the DXOP assembler directive, and synonyms for operations, as defined by the DFOP assembler directive. A LIBOUT directive must precede the DXOPs and DFOPs in order for the definitions or synonyms to be written to the specified macro library. When the definitions or synonyms are written to a macro library, they are placed in a subdirectory named D\$DFX\$, which is automatically created in the macro library. If the results of DXOPs and DFOPs are not written to a macro library, such results are only available to the program that defines them. Specifying a macro library as an assembler input option is equivalent to specifying the library in a LIBIN directive. Definitions may be read from the library, but nothing is written to the library unless a LIBOUT directive has been issued.

The purpose of the macro library is to reduce execution time and memory overhead associated with using macros. Execution time is reduced by encoding the macro definitions only once and by making the definitions available for subsequent assembler runs. Memory requirements are reduced since macro definitions not being defined or expanded reside only in the directory on disk. The macro definitions become part of the source program only when defined in that source program or when a macro is called in the source program.

## 7.4 MACRO LANGUAGE ELEMENTS

The elements of the macro language are labels, strings, constants, operators, variables, parameters, symbols, keywords, and verbs. A macro definition consists of statements containing macro language verbs and model statements. The macro language verbs are discussed in paragraph 7.5. A model statement specifies the structure of a single assembly language statement. The macro definition inserts items, such as symbols, mnemonics, and addresses, into the model statement to produce an assembly language statement which is then inserted into the assembly language source program. The elements of the macro language are explained in detail in the following paragraphs. Model statements are described in paragraph 7.5.13.

### 7.4.1 Labels

A macro language label consists of one or two characters. The first must be an alphabetic character (A...Z, \$, and ;) optionally followed by an alphanumeric character (A...Z, \$, ;, 0...9). Macro language labels are used to determine the processing sequence of statements in a macro definition when the statements are not processed in sequential order. Labels have no significance in the actual assembly language. The following are examples of valid macro language labels:

```
L1 MA C
```

### 7.4.2 Strings

The literal strings of the macro language consist of one or more characters enclosed in single quotes. They are identical to the character strings used in the assembly language. An example of a string is 'ONE'. Another example is ' ' (a blank).

### 7.4.3 Constants and Operators

Constants for the macro language are defined in the same way as constants for the assembly language. The arithmetic, logical, and relational operators of the assembly language, described in Section 2, also apply to the macro language. In addition, the macro language permits the concatenation of macro variable components with literal strings, characters of model statements, and other macro variables. Concatenation is described in paragraph 7.4.5.4.

### 7.4.4 Variables

A macro definition may include variables, which must follow the same rules as labels. That is, a variable may be no more than two characters in length and the first character must be alphabetic. Variables are declared using the \$VAR macro language verb and are assigned values by the \$ASG verb. Macro variables are local and may be used only in the macro that defines them.

### 7.4.5 Parameters

A parameter is a variable that is declared in the \$MACRO statement at the beginning of the macro definition. Parameters follow the same rules as macro labels. A parameter may be no more than two characters long and the first character must be alphabetic. Unlike variables, parameters do not have to be declared using the \$VAR verb. Parameters are declared when they appear in the \$MACRO statement. The initial values for parameters are determined by the corresponding arguments in the macro call statement. Argument refers to the value passed from the calling program to the macro. Parameter refers to the value that the macro receives. For example, assume that a macro definition begins with the following statement:

```
ADD $MACRO AU,AD
```

This statement defines a macro, called ADD, having parameters AU and AD. The macro might be called from the assembly source program with the following statement:

```
ADD NUM, *3
```

This statement calls macro ADD with arguments NUM and \*3. When the macro is called, the assembler assigns the values and attributes of argument NUM to parameter AU and assigns the values and attributes of argument \*3 to parameter AD. The number of arguments passed to a macro does not have to be the same as the number of parameters accepted by the macro. The assignment of values and attributes to parameters is described in detail in paragraph 7.5.1.

Parameters are a special kind of variable. However, unlike variables, parameters do not have to be declared using the \$VAR verb or have initial values assigned to them using the \$ASG verb. A parameter in a macro is assigned the same initial value as the value of the corresponding argument in the macro call statement. Like variables, the initial value of a parameter may be altered using the \$ASG verb.

**7.4.5.1 Macro Symbol Table.** The macro translator maintains a macro symbol table (MST) for each macro definition, which has an entry for each variable or parameter defined within the macro. Entries are made for each parameter appearing in a \$MACRO statement and for each variable declared using the \$VAR verb. Each entry consists of the variable or parameter itself and four pieces of information about it called the string, value, length, and attribute components. When a parameter is placed in the MST, all of the components are set to values determined by the argument in the macro call corresponding to that parameter. When a variable is declared in a \$VAR statement, an entry for it is made in the MST but none of the component values are set. Variable component values are set later in the macro definition by the \$ASG macro language verb.

The string component is a character string, up to 255 characters in length, assigned to the macro variable or parameter by the macro expander. The string component of a parameter is the string representing the corresponding argument in the macro call statement. In the example macro ADD, discussed in paragraph 7.4.5, after the macro is called, the string component of parameter AU is set to the string NUM. The string component may be used to make the linkage between a variable within a macro and a symbol in the assembly source program. This linkage is explained in detail in paragraph 7.4.6.1.

The value component is the value associated with a variable or parameter. The value component of a variable is null until a value is assigned to the variable using the \$ASG verb. The value component of a parameter is determined by the corresponding argument in the macro call statement. If the argument is a symbol, the value component of the corresponding parameter is set to the value associated with the symbol in the calling program. If the argument is an integer, the value component is set to the binary equivalent of that integer. If the argument is an operand list (multiple arguments, see paragraph 7.5.1), the value component is set to the number of operands in the list. If the argument is an indexed memory address, the value component of the corresponding parameter is set to the index register number.

The length component is the number of characters in the string component. The length component cannot be modified directly by the user. The length component is set when the string component is set.

The attribute component is a group of bits used as flags. Each bit specifies an attribute of a variable or parameter. For parameters, the attributes describe the way that the parameter was passed to the macro and the addressing mode used if the argument is an address. The attribute component is set automatically only for parameters. Although attributes may be set for variables within a macro definition, they are meaningful primarily for parameters.

As an example of the way the MST is set for parameters, assume that a macro definition in a program begins with the following statement:

```
ADD $MACRO AU, AD
```

This statement identifies a macro, ADD, having parameters AU and AD. Subsequently a macro call appears in the program as follows:

```
ADD NUM, *3
```

This statement is a macro call by name (the only way that macros can be called from an assembly source program) and identifies the arguments NUM and \*3, which are matched to the parameters in the \$MACRO statement. When the macro call statement is processed, entries are built in the MST for parameters AU and AD. The parameter entries would be as follows:

Symbol	String	Value	Length	Attributes
AU	NUM	(as defined in source)	3	\$PCALL
AD	*3	binary 3	2	\$PCALL,\$PIND

NUM is a symbol that may have been previously defined in the assembly source program. The value component of AU is the value that NUM has in the source program. The attributes \$PCALL and \$PIND are defined in Table 7-4.

Assume that another call to the same macro appears later in the program as follows:

```
ADD VAL(5),SUM
```

The components of parameters AU and AD are changed to correspond to the arguments of the new call as follows:

Symbol	String	Value	Length	Attributes
AU	VAL(5)	5 (index register)	6	\$PCALL,\$PNDX
AD	SUM	(as defined in source)	3	\$PCALL

When an argument of a macro call is a symbol that has previously been defined in the assembly source program, the value component for the corresponding parameter is set to the value that the symbol has in the assembly source program. When an argument of a macro call is not a previously defined symbol (a character string passed to the macro, for example), it is treated by the assembler as a symbol and an entry for it is made in the assembler symbol table. The value component of the parameter is set to zero, since the symbol is undefined, and the attribute component shows that the variable is a parameter appearing in a macro call. If no value is subsequently defined for the assumed symbol, the assembler generates a warning message about the undefined symbol, which may be ignored. The warning message may be avoided by assigning a value to the symbol, using the variable name and the SV variable qualifier, defined in Table 7-2.

**7.4.5.2 Referencing Variables.** Each component of a macro variable may be accessed individually. Reference to a variable is made in either binary mode or string mode. In the binary mode, the referenced macro variable component is treated as a signed 16-bit value. Binary mode access is made by writing the variable name and a variable qualifier, separated by a period. Variable qualifiers are described in paragraph 7.4.5.3. A reference to the binary mode value of the length component of variable AU in the example above would be AU.L, and the corresponding value is the 16-bit value, 06. A reference to the string component of a macro variable in binary mode is, by definition, the 16-bit value of the ASCII representation of the first two characters in the string. The binary mode value of the string component of variable AD is >5355, which is the ASCII representation of SU.

String mode access of macro variable components is signified by enclosing the qualified variable in colons (:), for example :AD.S:. Colons are always used in pairs to enclose a variable name. The string mode value of a component, other than the string component, is the decimal character string whose value is the binary value of the component. In the previous example, the string mode value of the length component of AD would be the character string 3. If the value of SUM in the source assembly program is >28, then the string mode value of the value component of AD would be the character string 40, which is the decimal equivalent of >28. Since the string component of a macro variable is a string, the string mode value of a string component is the entire string.

**7.4.5.3 Variable Qualifiers.** The components of a variable may be referenced using the qualifiers shown in Table 7-1. The variable name is followed by a period (.) and the single letter qualifier. Access or modification of a component using the qualifiers in Table 7-1 accesses or modifies the component values in the MST.

**Table 7-1. Variable Qualifiers**

Qualifier	Meaning
S	The string component of the variable.
A	The attribute component of the variable.
V	The value component of the variable.
L	The length component of the variable.

The following examples show qualified variables:

AU.S	String component of variable AU. In the first example of the macro call for the macro ADD, AU.S equals the binary equivalent for NU or >4E55. If the colon (:) has indicated string mode, the string component is the characters NUM (:AU.S: = NUM).
AU.A	Attribute component of variable AU. The various attributes may be accessed by use of logical operators and keywords.
AU.V	Value component of variable AU. In the first example of the macro call for the macro ADD, this would be the value of the symbol NUM in the assembly source program.
AU.L	Length component of variable AU. In the first example of the macro call for the macro ADD, AU.L = 3.

Except in an \$ASG statement (described in a subsequent paragraph), an unqualified variable means the string component of the variable. In the two following examples, the concatenated strings are equivalent:

:CT.S:WAY	Variable CT qualified
:CT:WAY	Variable CT unqualified

**7.4.5.4 Concatenation.** Concatenation is indicated by writing character strings next to string mode references to macro variables. Concatenation is especially useful when a previously defined string is augmented with additional characters. For example, if the string component of variable CT is the string ONE, concatenation could be expressed as follows:

:CT.S:WAY

or

:CT:WAY

The resulting string would be the same as if ONEWAY had been written. The small *b* represents a space. If the string component of the variable CT is changed to the string TWO, the result of the concatenation in the example would be TWOWAY. Strings and variables may be concatenated as desired and the variable need not be first. Components of variables that are represented by a binary value (for example, CT.V and CT.L) are converted to their ASCII decimal equivalent before concatenation.

For example:

:CT.S:WAY:CT.L:

is expanded as

ONEWAY3

since the length component of the variable CT is three.

#### 7.4.6 Symbols

Symbols that have been defined in the assembly source program may be used in the macro definition. If the symbol NUM has been defined in the assembly source program to have the value >28, NUM may be used in the macro definition and has the value >28 when used. When a symbol is defined in the assembly source program, an entry is made in the assembler symbol table (AST) which specifies several pieces of information about the symbol, similar to the way that an entry in the MST specifies information about macro variables. When a symbol is linked to a macro variable, the information in the AST becomes available to the macro.

Expansion of a macro may be varied according to the contents of the assembler symbol table (AST) and may change portions of the contents of the AST. The AST contains an entry for each symbol defined in the source program. The five parts of the entry in the AST that the macro language may access are called the value component, attribute component, string component, length component, and user attribute component.

The value component is the value defined to the symbol in binary form.

The attribute component is a group of bits, each of which represents an attribute of the symbol. Symbol attributes are defined in a later paragraph.

The string component is used to contain a string of up to 255 characters associated with the symbol by the macro translator. The string component is null until the macro translator places a character string in it.

The length component is determined by the string component and contains the number of characters in the string component. This component cannot be directly modified by the user. The value of this component is determined when the string component is modified.

The user attribute component is an eight-bit field which allows the user to define special attributes for a symbol. The significance of the user attribute component may be defined as desired in the user program, either as a group of eight flags or as a one-byte value.

As an example of the way that values are set in the AST when a symbol is defined, assume that the following statements appear in an assembly source program:

```
START    EQU    >5F
          RORG   >100
WSP      BSS    32
```

The symbol table components would be as follows:

Symbol	Value	Attribute	String	Length	User Attributes
START	005F		0	0	0
WSP	0100	\$REL	0	0	0

In this section, symbol table entry components are referred to as symbol components. Using keywords, a macro definition may access any component of any symbol in the AST that has been linked to a macro variable. Symbols that are arguments of the macro call are automatically linked to the corresponding macro parameter in the \$MACRO statement at the time of the macro call. A detailed example of parameter value assignments is included in the discussion of the \$MACRO verb. Symbols that are not arguments of the macro call must be linked to a macro variable before the symbol components can be accessed from the macro definition. Symbol values may be used directly by using the symbol name.

**7.4.6.1 Linked Symbols.** Symbols that are not arguments of a macro call may be linked to macro variables by using the \$ASG statement to assign the symbol name to the string component of the variable. For example, if macro variable G has been defined in the macro definition and the symbol TEN has been defined in the assembly source program, the statement \$ASG 'TEN' TO G.S allows the use of symbol component qualifiers to access or modify the components of symbol TEN. Note the use of the quotes around TEN. If the single quotes are not present, the value of the symbol is used rather than the string representing the symbol. The string component becomes the decimal equivalent of the value of the symbol in the assembly source program, and the symbol is not linked. The \$ASG verb is described in detail in a subsequent paragraph.

When a symbol in the AST is the string component of a macro variable, it is referred to as a linked symbol and the symbol qualifiers in Table 7-2 may be used to access the components of that symbol. The symbol value of an AST symbol is accessible directly using the symbol name without a qualifier. To access other components of a symbol that is not an argument of a macro call, the



symbol must be assigned as a string component of a macro variable (linked) and the symbol component qualifiers of Table 7-2 applied to that variable. A linked symbol is a symbol that is the string component of a variable. Access or modification of a component using the qualifiers in Table 7-2 accesses or modifies the component values in the AST.

**Table 7-2. Variable Qualifiers for Symbol Components**

Qualifier	Meaning
SS	String component of a linked symbol
SV	Value component of a linked symbol
SA	Attribute component of a linked symbol
SL	Length component of a linked symbol
SU	User attribute component of a linked symbol

The following are examples of qualified variables that specify components of linked symbols:

- B.SS** String component of the symbol linked to variable B. This value is null unless the \$ASG statement has been used to assign a string to it.
- G2.SV** Value component of the symbol that is linked to variable G2. If G2.S has been defined as MASK and the statement MASK EQU >FF has been encountered in the assembly source, then G2.SV = >FF. In string mode, :G2.SV: equals the characters 255. If MASK is used in the macro definition, it has the value >FF.
- NO.SA** Attribute component of the symbol linked to variable NO. The various attributes may be accessed by use of logical operators and keywords, as described later.
- V2.SL** Length component of the symbol linked to variable V2. If a string has been assigned to the symbol that is V2.S, then V2.SL is the length of that string.
- NV.SU** User attribute component of the symbol linked to variable NV. This component is zero until an \$ASG statement sets the bits in the component. This component is eight bits long. The bits may be defined and used as desired in the macro definition.

### 7.4.7 Keywords

The macro language recognizes keywords to specify the attributes of linked symbols and macro parameters. Each attribute is associated with a specific bit position in the attribute component word. Each keyword is equated to a data word that has the bit set in the position corresponding to the attribute represented by the keyword. Thus, keywords may be used with logical operators and the attribute component to test or set a specific attribute of a symbol or parameter. Logical operators are described in Section 2.

**7.4.7.1 Symbol Attribute Component Keywords.** The keywords listed in Table 7-3 may be used with a logical operator and the symbol attribute component (.SA) to test or set the corresponding attribute component in the AST.

**Table 7-3. Symbol Attribute Keywords**

Keyword	Meaning
\$REL	Symbol is relocatable
\$REF*	Symbol is an operand of an REF directive
\$DEF*	Symbol is an operand of a DEF directive
\$STR	Symbol has been assigned a string component
\$VAL	Symbol has been assigned a value
\$MAC	Symbol is defined as a macro name
\$UNDF	Symbol is not defined
<b>Note:</b>	
	* A symbol which is an operand of an SREF directive has both \$REF and \$DEF set.

The following examples show expressions that use symbol attribute component keywords:

P5.SA&\$STR	This is an AND operation between the attribute component of the symbol linked to variable P5 and the data word corresponding to keyword \$STR. The expression has a nonzero value when the symbol linked to variable P5 has had a string assigned to it (symbol string component is not null); otherwise, the expression has a value of zero. This expression would typically be used with an \$IF statement to determine if the symbol linked to variable P5 has had a string assigned to it.
CT.SA + + \$REL	This is an OR operation between the attribute component of the symbol linked to variable CT and the data word corresponding to keyword \$REL. The value of the

expression is always greater than zero. This expression might be used in an \$ASG statement to set the attribute bit specifying that the symbol linked to variable CT is relocatable.

The assembler uses the attribute components to determine the type of symbol with which it is dealing. All of the mnemonic opcodes representing assembly language instructions have entries in the assembler symbol table. If a macro is defined to have the same name as an assembler mnemonic, the \$MAC attribute is set for that entry. When the assembler encounters that name in an operation field, rather than execute the named assembler operation, it searches for the macro definition first in memory, to see if the macro has been defined in this assembly program, and then in the macro libraries available to it. When a macro is defined, as most of them are, with a name that does not match any assembler mnemonic, an entry is made for that name in the assembler symbol table and the \$UNDF attribute is set. When the assembler encounters the macro name in an operation field, it finds that the symbol is undefined and again searches memory and the available macro libraries for the macro definition.

**7.4.7.2 Parameter Attribute Keywords.** The keywords listed in Table 7-4 may be used with a logical operator and the macro variable attribute qualifier (.A) to test or set the corresponding attribute bit in the macro symbol table (MST) attribute component. These attribute keywords may be used to test or set attributes of both parameters and variables in the MST. Variables do not have values in the attribute component until a value has been assigned in the macro definition.

**Table 7-4. Parameter Attribute Keywords**

Keyword	Meanings
\$PCALL	Parameter appears as a macro call argument.
\$POPL	Parameter is an operand list, described in a subsequent paragraph. The value component contains the number of operands in the list.
\$PNDX	Parameter is an indexed memory address. The value component contains the index register number.
\$PIND	Parameter is an indirect workspace register address.
\$PATO	Parameter is an indirect auto-increment address.
\$PSYM	Parameter is a symbolic memory address.

The following examples show expressions that use parameter attribute keywords:

P6.A&\$PCALL	This is an AND operation between the attribute component of variable RA and the data word corresponding to keyword \$PCALL. The expression has a nonzero value when variable P6 is a parameter passed in a macro call; otherwise, the expression evaluates to zero.
--------------	---

RA.A + + \$PSYM

This is an OR operation between the attribute component of the variable RA and the data word corresponding to keyword \$PSYM. The expression always has a value greater than zero. This expression might be used in an \$ASG statement to set the attribute bit specifying that variable RA is a symbolic memory address.

## 7.5 VERBS

The macro language supports 12 verbs that are used in macro language statements. Any statement in a macro definition that does not contain a macro language verb in the operation field is processed as a model statement. The verbs are described in the following paragraphs.

### 7.5.1 \$MACRO

*Syntax definition:*

```
[<macro name>]b . . $MACROb . . [<parm>],[<parm>]. . b . . [<comment>]
```

The \$MACRO statement must be the first statement of a macro definition. It assigns a name to the macro and declares the parameters for the macro. The macro name consists of from 1 to 6 alphanumeric characters, the first of which must be alphabetic. Each <parm> is a parameter for the definition. Parameters are described in paragraph 7.4.5. The operand field may contain as many parameters as the right margin allows and must contain all parameters used in the macro definition. The comment field may not be used if there are no parameters.

The macro definition is used in the expansion of macro calls. The syntax for a call from the assembly source program is as follows:

```
[<label>]b..<macro name>b..  $\left[ \begin{array}{l} \langle \text{operand} \rangle \\ \langle \text{operand list} \rangle \end{array} \right] \left[ \begin{array}{l} \langle \text{operand} \rangle \\ \langle \text{operand list} \rangle \end{array} \right] \dots b..[\langle \text{comment} \rangle]$ 
```

When the label field contains a label, the label is assigned to the location of the first object code or dummy object code of the expanded macro instruction. The macro name specifies the macro definition to be used. Each operand may be any expression or address type recognized by the assembler, or a character string enclosed in quotes. Alternatively, an operand list may be used. An operand list is a group of operands enclosed in parentheses and separated by commas (when two or more operands are in the list). An operand list is processed as a set after removal of the outer parentheses during macro expansion.

Operands (or operand lists) may be nested in parentheses in the macro call for use within macro definitions.

For example,

```
ONE $MACRO P1,P2
```

specifies 2 parameters.

A call such as

```
ONE PAR1,PAR2
```

results in

PAR1 being associated with P1 and PAR2 being associated with P2.

However, a call such as

```
ONE PAR1,(PAR21,PAR22)
```

results in

PAR1 being associated with P1, while the list PAR21,PAR22 is associated with P2.

Now if :P2: or :P2.S: is used as an operand in a model statement, it has the effect of being a two operand list.

Processing of each macro call in a source program causes the macro expander to associate the first parameter in the \$MACRO statement with the first operand or operand list on the macro call line and the second parameter with the second operand or operand list, etc. Each parameter receiving a value has the \$PCALL attribute set. When the macro definition has more parameters specified than the number of operands in the macro call, the \$PCALL attribute is not set for the excess parameters. The \$PCALL attribute is also not set if an operand is null, that is, the call line has two commas adjacent or an operand list of zero operands. Expansion of the macro can be conditioned on the number of operands by testing this attribute, \$PCALL.

For example, a macro definition containing

```
AMAC $MACRO P1,P2,P3
```

when called by

```
AMAC AB1,AB2
```

sets \$PCALL in parameters P1 and P2 but not for P3.

Similarly,

```
AMAC XY,,XY3
```

causes \$PCALL to be set for P1 and P3 but not for P2.

When the macro call has more operands than the number of parameters in the \$MACRO statement, the excess operands are combined with the operand or operand list corresponding to the

last parameter to form an operand list (or a longer operand list). In the example , the operands of the two macro calls in the following code would be assigned to the parameters in the same way:

ONE	EQU	9	
TWO	EQU	43	
THREE	EQU	86	
FIX	\$MACRO	P1,P2	MACRO FIX
	.		
	.		
	FIX	ONE,TWO,THREE	MACRO INSTRUCTION
	FIX	ONE,(TWO,THREE)	MACRO INSTRUCTION

Both calls result in the following parameter assignments:

P1.S = >4F4E	P2.S = >5457
:P1.S: = ONE	:P2.S: = TWO,THREE
P1.A = \$PCALL	P2.A = \$PCALL,\$POPL
P1.L = 3	P2.L = 9
P1.V = 9	P2.V = 2

The following example shows the symbol value assignments prior to, and the resulting parameter assignments after a call to example macro PARM.

*Example:*

A	EQU	7	
B	EQU	15	
C	DATA	17	
D	DATA	63	
E	EQU	95	
F	EQU	11	
G	EQU	58	
H	EQU	101	
I	EQU	119	
PARM	\$MACRO	P1,P2,P3,P4,P5,P6,P7,P8,P9	
	.		
	.		
	PARM	A,,B,(,),C,(D),(E)(F),(G,(H,I)),*R7 +	

## Parameter assignments:

P1.S = >0041	P2.S = >0000	
:P1.S: = A	:P2.S: =	(no string)
P1.A = \$PCALL	P2.A =	(zeros)
P1.L = 1	P2.L = 0	
P1.V = 7	P2.V = 0	
P3.S = >0042	P4.S = >0000	
:P3.S: = B	:P4.S: =	(no string)
P3.A = \$PCALL	P4.A = \$POPL	
P3.L = 1	P4.L = 0	
P3.V = 15	P4.V = 0	
P5.S = >0043	P6.S = >0044	
:P5.S: = C	:P6.S: = D	
P5.A = \$PCALL	P6.A = \$PCALL,\$POPL	
P5.L = 1	P6.L = 1	
P5.V = 17	P6.V = 1 (number of items in operand list)	
P7.S = >2845	P8.S = >472C	
:P7.S: = (E)(F)	:P8.S: = G,(H,I)	
P7.A = \$PCALL,\$PNDX	P8.A = \$PCALL,\$POPL	
P7.L = 6	P8.L = 7	
P7.V = 11 (index register)	P8.V = 2 (number of items in operand list)	
P9.S = >2A52		
:P9.S: = *R7 +		
P9.A = \$PCALL,\$PATO		
P9.L = 4		
P9.V = 7 (register number)		

**7.5.2 \$VAR***Syntax definition:*

```
h..$VARh..<var>[,<var>]...h..[<comment>]
```

The \$VAR statement declares the variables for a macro definition. The \$VAR statement is required only if the macro definition contains one or more variables other than parameters. More than one \$VAR statement may be included and each \$VAR statement may declare more than one variable. Each <var> in the operand is a variable as previously described.

The following is an example of a \$VAR statement:

```
$VAR A,CT, V3 THREE VARIABLES FOR A MACRO
```

The example declares variables A, CT, and V3. A, CT, and V3 must not have been declared as parameters. The \$VAR statement does not assign values to any components of the variables. \$VAR statements may appear anywhere in the macro definition to which they apply; however, each variable must be declared before the first statement that uses the variable. It is logical to place \$VAR statements immediately following the \$MACRO statement.

### 7.5.3 \$ASG

*Syntax definition:*

$$\text{b..\$ASGb..} \left\{ \begin{array}{l} \langle \text{expression} \rangle \\ \langle \text{string} \rangle \end{array} \right\} \text{bTOb} \langle \text{var} \rangle \text{b..} [\langle \text{comment} \rangle]$$

The \$ASG statement assigns values to the components of a variable. Variables that are not parameters have no values for components until values are assigned using \$ASG statements. Components previously assigned to parameters or to variables by \$ASG statements may be assigned new values with \$ASG statements.

The expression operand may be any expression valid to the assembler and may contain binary mode variable references and the keywords in Tables 7-3 and 7-4.

#### NOTE

The binary mode value of a string component or symbol string component used in an expression is the binary value of the first two characters of the string. Thus, if GP.S has the string LAST, the value used for GP.S in an expression in the <string> is the hexadecimal number >4C41, which is the ASCII representation for LA.

A string may be one or more characters enclosed in single quotes or the concatenation of a literal with the string mode value of a qualified variable. The <var> may be either an unqualified variable or a qualified variable.

When the operands are both unqualified variables, all components of the source (first) variable are transferred to the destination (second) variable.

When the source variable is qualified or is a quoted string and the destination variable is unqualified, an error results. When the destination variable is qualified, only the specified component receives the corresponding component of the expression or string. An exception to this occurs when a string is assigned to the string component of a variable or symbol; the length component of that variable or symbol is set to the number of characters in the assigned string. If the attribute component of the destination variable is to be changed, only those attributes that can be tested using keywords are changed. Other attributes maintained by the assembler may or may not be changed as appropriate.

#### NOTE

A qualified variable that specifies the length component is illegal as a target in a \$ASG statement. Also, a qualified variable that specifies the attribute component or the value component of a macro variable that was declared to be a macro language label (for the purpose of a \$GOTO) is illegal as the target in a \$ASG statement.



The following examples show the use of the \$ASG statement:

\$ASG P3 TO V3	Assign all the components of variable P3 to variable V3.
\$ASG :P3.S.'ES' TO P3.S	Concatenate string ES to the string component of variable P3 and set the string component to the result. Also, add 2 to the length component of the new value.
\$ASG CT.A + +\$PSYM TO CT.A	Set the bit in the attribute component of variable CT to indicate the symbolic address attribute.

Variables P3,V3, and CT must have been previously declared either as parameters in a \$MACRO statement or as variables in a \$VAR statement.

The \$ASG statement may be used to modify symbol components as shown in the following examples. Assume that P3.V = 6 and P3.S = SUB.

\$ASG 'TEN' TO G.S	Assigns TEN as the string component of variable G. When TEN is a label in the AST, this statement allows the use of symbol component qualifiers to modify the components of symbol TEN.
\$ASG P3.V TO G.SV	Sets the value component of the symbol linked to variable G to the value component of variable P3. In this case, the value component of TEN is set to six.
\$ASG 'A':P3.S:'S' TO G.SS	Concatenates string A with the string component of variable P3 and with string S and places the result in the string component of the symbol linked to variable G. Also sets the length component of the same symbol. Thus, the string component of TEN is ASUBS and the length component is five.

#### 7.5.4 \$SBSTG

*Syntax definition:*

```
b..$SBSTGb..<var>,<expr>,b..TOb..<var>[<comment>]
```

The \$SBSTG statement is used to access substrings of the string component of a variable or symbol. The substrings so accessed are stored in the string component of a variable or symbol. The \$SBSTG statement is a special case of the \$ASG statement.

The two variables (<var>) must have been previously declared in the \$MACRO statement or in a \$VAR statement and must be qualified by either .S or .SS. The two expressions may be any valid integer macro expressions. An integer macro expression may contain any qualified macro variable, constant, or assembler symbol, but it must not contain any quoted strings or any colons.

The first <var> specifies the source string. The variable or symbol so specified must have been assigned a string component, and that string must be at least one character long.

The first expression is the first-character expression, which indicates where in the source string the desired substring begins, the first character of the source string counting as one. The expression must evaluate to a positive integer less than or equal to the length of the source string.

The second expression is the length expression, which indicates how many characters to take from the source string to make up the substring. The length expression must evaluate to a non-negative integer less than or equal to the length of the source string plus one, minus the first-character expression, that is, substring length  $\leq$  source string length + 1 – first character expression.

If any of the conditions specified for the source string, the first-character expression, or the length expression is not met, an error is reported and the destination string component is not changed.

The following examples show the results of using \$SBSTG:

\$ASG 'ABCDEFGH' TO V.S.	V.S = ABCDEFGH
\$SBSTG V.S,3,3 TO W.S	W.S = CDE
\$SBSTG V.S,V.L/2 + 2,3 TO W.S	W.S = FGH
\$SBSTG V.S,1,V.L – 8 TO W.S	W.S = (null string)

### 7.5.5 \$NAME

*Syntax definition:*

```
<label> . . $NAME . . [<comment>]
```

The \$NAME statement associates a macro language label with a macro language statement. When a label is required for branching within a macro definition, it must be provided by a \$NAME statement. The \$NAME statement performs no processing in the expansion of a macro instruction.

The following example shows a \$NAME statement:

AB \$NAME BRANCH TO THIS POINT	A \$GOTO statement with AB as an operand branches to this point.
\$ASG P3 TO V3	Expansion of the macro instruction continues with the \$ASG statement.

### 7.5.6 \$GOTO

*Syntax definition:*

```
h . $GOTO h . <label> h . [<comment>]
```

The \$GOTO statement branches within a macro definition either to a \$NAME statement or to an \$END statement. The label is a macro language label of either type of statement.

The following example shows a \$GOTO statement:

```
$GOTO AB
```

Branch to a \$NAME statement having the label AB and execute the following statement, or to an \$END statement having the label AB.

### 7.5.7 \$EXIT

*Syntax definition:*

```
h . $EXIT h . [<comment>]
```

The \$EXIT statement terminates processing of the macro expansion. The \$EXIT statement has the same effect as a \$GOTO statement with the label of the \$END statement as the operand.

The \$EXIT is used in a macro that has several possible termination points to mark each termination point except the last. The \$END statement is used only at the end of the macro definition.

### 7.5.8 \$CALL

*Syntax definition:*

```
h . $CALL h . <macro name> h . [<comment>]
```

The \$CALL statement initiates processing of the macro definition named in the operand field. The operands passed to the macro being expanded are mapped to the parameters of the macro specified in the \$CALL statement. When the macro expander executes an \$END statement or an \$EXIT statement in the called macro, processing returns to the statement following the \$CALL statement in the calling macro.

The following is an example of a \$CALL statement:

```
$CALL CONV
```

Activates the macro definition CONV. The parameters of the calling macro are passed as the operands of the macro CONV.

**7.5.9 \$IF**

*Syntax definition:*

```

h..$IFh..<expression>h..[<comment>]

```

The \$IF statement provides conditional processing in a macro definition. An \$IF statement is followed by a block of macro language statements terminated by an \$ELSE statement or an \$ENDIF statement. When the \$ELSE statement is used, the \$ELSE statement is followed by another block of macro language statements terminated by an \$ENDIF statement. The expression in the \$IF statement may use arithmetic or logical operators to produce a zero or nonzero value. When the expression has a nonzero value, the block of statements following the \$IF statement is executed. The expression in the \$IF statement may also use relational operators to produce a true or false value. When the expression is true, the block of statements following the \$IF statement is executed. When the \$ELSE statement is used and the expression in the \$IF statement has a zero value or is false, the block of statements following the \$ELSE statement is executed. Thus, the condition of the \$IF statement may determine whether or not a block of statements is processed, or which of two blocks of statements is processed. A block may consist of zero or more statements. While logical operators may be used in the expression of an \$IF statement, the NOT operation is a bit-by-bit operation and the NOT of a nonzero number is, except in the case of >FFFF, still a nonzero number.

The <expression> may be any expression as defined for the \$ASG statement and may include qualified variables and keywords. The expression defines the condition for the \$IF statement.

**NOTE**

The expression is always evaluated in binary mode. Specifically, the relational operations (<, >, =, #, =, etc.) operate only on the binary mode value of the macro variable. As a result, comparisons of two character strings may be done only on the initial two character positions.

The following examples show conditional processing in macro definition:

```

.
.
.
$IF      KY.SV      Process the statement of BLOCK A when
.          BLOCK A  the value component of the symbol
.              linked to variable KY contains a
$ELSE    .          nonzero value. Process the statements
.          BLOCK B  of BLOCK B when the component contains
.              zero. After processing either block
$ENDIF   .          of statements, continue processing at
.              the statement following the $ENDIF
.              statement.

```

<pre> \$IF      T.A&amp;\$PCALL = 0 . .      BLOCK A . \$ENDIF </pre>	<p>Process the statements of BLOCK A when the attribute component of parameter T indicates that parameter T was not supplied in the macro instruction. If parameter T was supplied, do not process the statements of BLOCK A. Continue processing at the statement following the \$ENDIF statements in either case.</p>
<pre> \$IF      T.L = 5 . .      BLOCK A . \$ENDIF </pre>	<p>Process the statements of BLOCK A when the length component of variable T is equal to 5. If the length component of the variable is not equal to 5, do not process the statements of BLOCK A. Continue processing at the statement following the \$ENDIF statement.</p>

### 7.5.10 \$ELSE

*Syntax definition:*

```

h . . $ELSE h . . [comment]

```

The \$ELSE statement begins an alternate block to be processed if the preceding \$IF expression was false or zero.

### 7.5.11 \$ENDIF

*Syntax definition:*

```

h .. $ENDIF h .. [comment]

```

The \$ENDIF statement terminates the conditional processing initiated by an \$IF statement in a macro definition. Examples of \$ENDIF statements and their use are shown in a preceding paragraph.

### 7.5.12 \$END

*Syntax definition:*

```

[label] h . . $END h . . [comment]

```

The \$END statement marks the end of the macro definition. When executed, the \$END statement terminates the processing of the macro definition. The label may be used in a \$GOTO statement to terminate processing of the macro definition.

The following is an example of an \$END statement:

```

$END FIX                                Terminates the definition of macro FIX.

```

### 7.5.13 Model Statements

As mentioned earlier, a macro definition consists of statements that contain macro language verbs and model statements. A model statement always results in an assembly language source statement and may consist only of an assembly language statement or portions of an assembly language statement combined with string mode qualified variable components using the colon operator (:). In any case, the resulting source statement must be a legal assembler language statement or an error results. The following examples show model statements:

MOVB R6,R7

This model statement is itself an assembly language source statement that contains a machine instruction.

:P7.S: ~~SO~~SO~~C~~C P2.S:,R8V4.S:

This model statement begins with the string component of variable P7. Three blanks, SOC, and three more blanks are concatenated to the string. The string component of variable P2 is concatenated to the result, to which R8 and three blanks are concatenated. A final concatenation places the string component of variable V4 in the model statement. The result is an assembly language machine instruction having the label and comment fields and part of the operand field supplied as string components

:MS.S:

This model statement is the string component of variable MS. Preceding statements in the macro definition must place a valid assembly language source statement in the string component to prevent assembly errors.

#### NOTE

Conditional assembly directives may not appear as operations in a model statement. Comments supplied in model statements may not contain periods (.) since SDSMAC scans comments in the same way as model statements and improper use of punctuation may cause syntax errors.

The following example shows a complete macro, MSG, consisting of two macro statements and two model statements. This macro expects a quoted character string as a parameter and returns a tagged string, with the length of the string in a BYTE statement followed by the string itself, enclosed in single quotes, in a TEXT statement. The expression :MS.L: - 2 is used so that the single quotes are not counted as part of the string.

MSG

\$MACRO MS

BYTE :MS.L: - 2

TEXT :MS.S:

**SEND**

When the statement MSG 'THIS IS A MESSAGE' appears in the assembly source program, the **statements:**

BYTE 19-2

TEXT 'THIS IS A MESSAGE'

are inserted into the source program in place of it.

## 7.6 ASSEMBLER DIRECTIVES TO SUPPORT MACRO LIBRARIES

Two directives support the use of libraries of macros in SDSMAC. These two directives are LIBOUT, which is used to build or add to a library of macro definitions, and LIBIN, which is used to recall a previously built macro library.

### 7.6.1 LIBOUT Directive

*Syntax definition:*

**LIBOUT** <library-access-name>

The LIBOUT directive declares a macro library where macro definitions are written during an assembly. The library must have been previously created by a CFDIR (create file directory) utility command. Macro definitions appearing in the assembler input stream following a LIBOUT directive are written to the specified library upon successful translation. Macro definitions appearing prior to the first LIBOUT directive remain in memory and are not written to any library. Multiple LIBOUT directives may appear in a single assembly. Each successive output library supercedes its predecessor so that only one output library is in effect at a time. The same library may be specified on multiple LIBOUT directives. Furthermore, a library may be used for both input and output simultaneously. Macro definitions are written to the library, using the replace option that will redefine any macro with the same library name. Hence, a macro library may be maintained (updated) without difficulty.

In addition to macro definitions, a subdirectory of the macro library with the name D\$DFX\$ contains the results of DXOP and DFOP directives and the results of macro names that redefine an assembly language instruction, directive, or pseudo-instruction appearing within the span of the current LIBOUT directive.

The macro definitions, DXOPs, and DFOPs are written to the library completely replacing any prior definitions of the symbols on that macro library. For example, if a macro library contained a macro definition for the symbol LOCK and a subsequent assembly encounters a DFOP LOCK, ABS statement while a LIBOUT directive to that library is in effect, the macro library will contain information that LOCK is another name for the instruction ABS. The macro definition which existed on the library previously will have been deleted.

## 7.6.2 LIBIN Directive

*Syntax definition:*

```
↳ .LIBIN↳ .<library-access-name>
```

The LIBIN directive declares a macro library to be used in the current assembly. The library must have been previously created and must contain only macro definitions and DFOP and DXOP directives previously encoded during assembly (by use of the LIBOUT directive). Multiple LIBIN directives may appear in a single assembly. When the LIBIN directive is encountered, the library directory is examined for any redefinition of assembler instructions, and their existence is flagged. No further use is made of the macro library until an undefined operation is encountered. At that time, the macro library is searched for a possible macro definition of the operation. In the case of multiple macro libraries, the search order is inverse to the order of presentation, that is, the last macro library is searched first. The system macro library, specified as an assembler input option, is always searched last.

The system macro library contains only those macros that have been defined by the user and written to that library using the LIBOUT directive. When the system macro library is specified as an assembler input option, it may only be read from, not written to. To write to the system macro library, the library name must be specified in a LIBOUT directive.

## 7.6.3 Macro Library Management

A macro library may be listed, added to, deleted from, and replicated using a combination of utility commands provided by the operating system and the macro assembler LIBIN and LIBOUT directives.

To list or replicate a macro library, use the utility commands provided by the operating system.

To add to an existing macro library or to change an existing macro definition, DFOP, or DXOP, use only the LIBOUT directive provided by the macro assembler. Do not use utility commands for copying files to copy a macro definition to another macro library.

To delete macro definitions, DFOPs, and DXOPs, use the utility commands provided by the operating system to delete files. In the following examples, assume that a macro library with the name

```
SYSTEM.MACROS
```

is present.

- a. If the result of the DFOP

```
DFOP T,TEXT
```

is to be deleted, use the delete file utility command to delete the file:

```
$.SYSTEM.MACROS.D$DFX$.T
```



- b. If the result of the DXOP

DXOP SVC,15

is to be deleted, use the delete file utility command to delete the following file in the same manner as above:

.SYSTEM.MACROS.D\$DFX\$.SVC

- c. If a macro definition for CALL is to be deleted, use the delete file utility command to delete the following file:

.SYSTEM.MACROS.CALL

- d. If a macro definition that redefines an assembly language instruction, directive, or pseudo-instruction is to be deleted, then two files must be deleted. If the macro name was TEXT then delete:

.SYSTEM.MACROS.TEXT  
.SYSTEM.MACROS.D\$DFX\$.TEXT

If only one of these is deleted, either an invalid opcode assembly error will result or the intended macro will not be used.

## 7.7 MACRO EXAMPLES

Macros may simply substitute a machine instruction for a macro instruction, or they may include conditional processing, access the assembler symbol table, and employ recursion. Several examples of macro definitions are described in the following paragraphs.

### 7.7.1 Macro GOSUB

Macro GOSUB is an example of a macro that substitutes a machine instruction for the macro instruction. The macro definition consists of three macro language statements, one of which is a model statement, as follows:

GOSUB	\$MACRO	AD	Defines macro GOSUB and declares a parameter, AD.
	BL	:AD.S:	A model statement that results in a BL instruction with the string component of the parameter as operand.
	\$END	GOSUB	Terminates macro GOSUB.

The syntax of the macro instruction for the GOSUB macro is defined as follows:

[<label>]b . .GOSUBb . .<address>b . .[<comment>]

When a label is used, it is effectively the label of the resulting BL machine instruction. The address may be any address form that is valid for a BL instruction. When a comment is used, it applies to the macro instruction. For example, the following macro instruction is valid for the GOSUB macro:

```
GOSUB @SUBR
```

The statement in the example results in a machine instruction to branch and link to a subroutine at location SUBR, as follows:

```
BL @SUBR
```

Another example shows the macro instruction that could be used if the subroutine address were in workspace register 8.

```
NEXIT GOSUB *R8
```

The resulting instruction would be as follows:

```
BL *R8
```

### 7.7.2 Macro EXIT

Macro EXIT is an example of a macro that supplies a machine instruction and an assembler directive the first time the macro is executed and a machine instruction each successive time. The macro requires an EQU directive to be placed in the source program prior to calling the macro, and the definition consists of nine macro language statements, including two model statements. The definition is as follows:

EXIT	\$MACRO		Defines macro EXIT with no parameters.
	\$VAR	L	Defines variable L.
	\$ASG	'F1' TO L.S	Assign F1 to the string component of variable L to allow access to symbol F1 in the assembler symbol table.
	XOP	@TERM,15	Model statement — places an XOP machine instruction in the source program.
	\$IF	L.SV	If the value component of symbol F1 is a nonzero value, perform the next two statements and terminate the macro. Otherwise, terminate the macro.
TERM	BYTE	16	Model statement — places a byte directive referenced by the XOP instruction following the XOP instruction.

	\$ASG	O TO L.SV	Set the value component of symbol F1 to zero. Any further calls to macro EXIT will omit the preceding model statement and this statement.
	\$ENDIF		Defines the end of conditional processing.
	\$END		End of macro definition.
	.		
	.		
F1	EQU	1	Defines F1 with a value of 1. This is not part of the macro definition, but is a source statement. It must precede the first macro call for macro EXIT and may precede the definition.

The syntax of the macro instruction for the EXIT macro is defined as follows:

[<label>]b..EXIT

When a label is used, it is effectively the label of the XOP machine instruction resulting from the macro. The first time the macro is called, the following source statements are placed in the program:

```

XOP          @TERM,15
TERM        BYTE          16
    
```

Subsequent calls for the macro result in the following:

```

XOP          @TERM,15
    
```

### 7.7.3 Macro ID

Macro ID is an example of a macro having a default value. The macro supplies two DATA directives to the source program. The macro consists of nine macro language statements, four of which are model statements. The definition is as follows:

ID	\$MACRO	WS,PC	Defines ID with parameters WS and PC.
	DATA	:WS.S:	Model statement — places a DATA directive with the string of the first parameter as the operand in the source program.
	\$IF	PC.A&\$PCALL	Tests for presence of parameter PC.

DATA	:PC.S;,15	Model statement — places a DATA directive in the source program. The first operand is the string of the second parameter, and the second operand is 15. This statement is processed if the second parameter is present.
\$ELSE		State of alternate portion of definition.
DATA	START,15	Model statement — places a DATA directive in the source program. The first operand is label START, and the second operand is 15. This statement is processed if the second parameter is omitted.
START		Model statement — places a label START in the source program. This statement is processed if the second parameter is omitted.
\$ENDIF		End of conditional processing.
\$END		End of macro.

This macro could be used to place a three-word vector at the beginning of a program. The first word could be the workspace address, the second the entry point, and the third the value 15 to be placed in the ST register. The first operand of the macro instruction would be the workspace address, and the second operand would be the entry point. When the executable code immediately follows the vector and the entry point is the first word of executable code, the second parameter may be omitted. The syntax definition of the macro instruction for macro ID is as follows:

```
<label> ID <address> [, <address>] . [<comment>]
```

The label becomes the label of the three-word vector, and the addresses may be expressions or symbols.

The following is an example of a macro call for macro ID:

```
PROG1 ID WORK1,BEGIN
```

The resulting source code would be as follows:

```
PROG1 DATA WORK1
DATA BEGIN,15
```

When the entry point immediately follows the macro instruction, the macro call could be coded as follows:

```
PROG2 ID WORK2
```

This would result in the following source code:

```

PROG      DATA      WORK2
          DATA      START,15

START

```

This form of the macro call imposes two restrictions on the source program. The source program may not use the label START and may not call macro ID more than once. The user may prevent problems with labels supplied in macros by reserving certain characters for use in macro-generated labels. A macro definition may maintain a count of the number of times it is called and use this count in each label generated by the macro.

The assembler marks statements inserted into the source code during macro expansion with an asterisk (\*) preceding the line number. The actual macro definition appears in the source code where it is defined only, not where it is called.

#### 7.7.4 Macro UNIQUE

```

0001          IDT 'UNIQUE'
0003          THIS EXAMPLE DEMONSTRATES A METHOD FOR CREATING UNIQUE
0004          LABELS USING THE MACRO LANGUAGE. EACH CALL OF THE MACRO
0005          * GENERATES A UNIQUE LABEL OF THE FORM 'U;;xxx' WHERE 'xxx'
0006          * IS A NUMBER
0007          LABEL $MACRO
0008          * DECLARE A VARIABLE TO USE IN THE MACRO
0009          $VAR L
0010          * ASSIGN THE CHARACTER STRING OF A SYMBOL THAT WILL HOLD
0011          * A COUNTER VALUE AND THE LAST LABEL GENERATED
0012          $ASG 'U;;;;;' TO L.S
0013          * INCREMENT THE SYMBOL VALUE OF 'U;;;;;' TO OBTAIN THE
0014          * LABEL VALUE
0015          $ASG L.SV + 1 TO L.SV
0016          * CREATE THE LABEL AND SAVE IN THE SYMBOL STRING COMPONENT
0017          * GENERATE THE LABEL IN THE NEXT LABEL FIELD. NOTE THAT
0018          * MODEL STATEMENT STARTS IN COLUMN 1
0019          U;; : L. SV:
0020          $END
0021          *
0022          * NOW GENERATE SOME LABELS
0023          *
0024          LABEL
*0001 0000  U;;1
0025 0000  0000  DATA 0,1
0002 0001
0026          LABEL
*0001 0004  U;;2
0027          LABEL
*0001 0004  U;;3

```

```

0028 0004 0004 DATA 4
0029 END
NO ERRORS

```

### 7.7.5 Macro GENCMT

Macro GENCMT is an example showing how to implement both those comments that appear in the macro definition only, and those comments that appear in the expansion of the macro. When this macro is called, the statement in line six generates a comment.

```

0001 IDT 'GENCMT'
0002 GENCMT $MACRO
0003 $VAR V
0004 *THIS IS A MACRO DEFINITION COMMENT
0005 $ASG '**' TO V.S
0006 :V.S:THIS IS A MACRO EXPANSION COMMENT
0007 $END
0008 GENCMT
*0001 *THIS IS A MACRO EXPANSION COMMENT
0009 0000 0000 DATA 0,1
0010 0002 0001
0010 GENCMT
*0001 *THIS IS A MACRO EXPANSION COMMENT
0011 GENCMT
*0001 *THIS IS A MACRO EXPANSION COMMENT
0012 0004 0004 LABEL DATA 4
0013 END

```

### 7.7.6 Macro LOAD

```

0001 IDT 'LOAD'
0002 *
0003 *GENERALIZED LOAD IMMEDIATE MACRO
0004 *
0005 *THIS MACRO DEMONSTRATES USE OF THE MACRO
0006 *SYMBOL ATTRIBUTES $PSYM,$PNDX,$PATO,$PIND.
0007 *
0008 *OPERANDS: D (DESTINATION)MAY BE REGISTER,
0009 * INDIRECT, SYMBOLIC,
0010 * OR AUTO-INC.
0011 * V (VALUE) SHOULD BE LITERAL VALUE.
0012 *
0013 *
0014 *IF THE FIRST OPERAND IS NOT A REGISTER, IT
0015 *WILL BE MOVED INTO THE SCRATCH REGISTER
0016 *BEFORE PERFORMING THE LOAD. THE SCRATCH
0017 *REGISTER IS ASSUMED TO BE R0.
0018 *
0019 *
0020 *THIS SYMBOL DEFINITION OR'S TOGETHER ALL
0021 *ADDRESSING MODES BUT 'REGISTER'.

```

```

0022          *
0023          001E      COMPLEX EQU $PATO + + $PSYM + + $PNDX + + $PIND
0024          *
0025          *THIS MACRO WILL MASK OUT THE REGULAR 'LI'
0026          *INSTRUCTION, SO THE 'DFOP' FOR 'LI' IS
0027          *USED TO DEFINE A SYNONYM FOR THE 'LI'
0028          *INSTRUCTION.
0029          *
0030          DFOP LI$,LI
0031          LI          $MACRO D,V
0032          $IF D.A&COMPLX
0033          LI$ R0,:V:
0034          MOV R0,:D:
0035          $ELSE
0036          LI$ :D,:V:
0037          $ENDIF
0038          $END
0039 0000 0000  LOC      DATA 0
0040          LI *R5,25
*0001 0002 0200      LI$ R0,25
          0004 0019
*0002 0006 C540      MOV R0,*R5
0041          LI R12,4
*0001 0008 020C      LI$ R12,4
          000A 0004
0042          LI 12(R13),16
*0001 000C 0200      LI$ R0,16
          000E 0010
*0002 0010 CB40      MOV R0,12(R13)
          0012 000C
0043          LI @LOC,111
*0001 0014 0200      LI$ R0,111
          0016 006F
*0002 0018 C800      MOV R0,@LOC
          001A 0000'
0044          *
0045          *NOTE THAT THE FOLLOWING CASE DOES NOT
0046          *GENERATE THE DESIRED CODE. TO CORRECTLY
0047          *DETECT MEMORY LOCATION REFERENCES, LABELS
0048          *SHOULD HAVE '@' SIGNS PRECEDING THEM.
0049          *
0050          LI LOC,111
*0001 001C 0200      LI$ LOC,111
          001E 006F
***** REGISTER REQUIRED
0051          END
0001  ERRORS, LAST ERROR AT 0050

```

## 7.7.7 Macro TABLE

```

0001                                IDT 'TABLE'
0002                                *
0003                                *THIS MACRO DEMONSTRATES RECURSIVE PROCESSING
0004                                *
0005                                *WHEN MORE OPERANDS ARE PASSED TO A MACRO
0006                                *THAN WERE INCLUDED IN THE DEFINITION, ALL THE
0007                                *SURPLUS OPERANDS ARE ASSIGNED (WITH THE
0008                                *COMMAS BETWEEN THEM) TO THE LAST PARAMETER.
0009                                *THIS IS A USEFUL FEATURE WHEN RECURSIVE
0010                                *PROCESSING IS NEEDED.
0011                                *
0012                                *THE EXPECTED OPERAND FOR THE 'OR' MACRO IS A
0013                                *LIST OF BIT PATTERNS 16 BITS IN WIDTH. THIS
0014                                *MACRO USES RECURSION TO 'OR' THE BITS
0015                                *TOGETHER. 'TEMP' IS A SYMBOL USED BY THE
0016                                *MACRO.
0017                                *
0018                                0000 TEMP EQU 0
0019                                OR $MACRO A,B
0020                                $VAR T
0021                                $ASG 'TEMP' TO T.S
0022                                $ASG A.V + + T.SV TO T.SV
0023                                $IF B.A&$PCALL
0024                                    OR :B.S:
0025                                $ELSE
0026                                    DATA :T.SV:
0027                                    $ASG 0 TO T.SV
0028                                $ENDIF
0029                                $END
0030                                OR >100
*0001 0000 0100 DATA 256
0031                                OR 1,2,4,8
*0001                                OR 2, 4, 8
*0001                                OR 4, 8
*0001                                OR 8
*0001 0002 000F DATA 15
0032                                OR 1, 1, 2, 4, 8
*0001                                OR 1, 2, 4, 8
*0001                                OR 2, 4, 8
*0001                                OR 4, 8
*0001                                OR 8
*0001 0004 000F DATA 15
0033                                OR >11,>1100
*0001                                OR>1100
*0001 0006 1111 DATA 4369
0034                                END
NO ERRORS

```



7.7.8 Macro LISTS

```

0001             IDT 'LISTS'
0002             *
0003             *THE PREORD AND ENDORD MACROS DEMONSTRATE
0004             *RECURSION AND LIST PROCESSING.
0005             *
0006             *
0007             *INPUTS: A PARENTHESIZED EXPRESSION OF
0008             *   THE FOLLOWING FORM:
0009             *
0010             *           A,OP,C
0011             *
0012             *           A = PARENTHESIZED EXPRESSION
0013             *           OP = OPERATION
0014             *           (MULTIPLICATION IS REPRESENTED
0015             *           AS A NULL PARAMETER, SIMILAR
0016             *           TO ITS REPRESENTATION IN
0017             *           ALGEBRAIC EXPRESSIONS)
0018             *           B = PARENTHESIZED EXPRESSION
0019             *
0020             *OUTPUTS: UNPARENTHESIZED EXPRESSION IN
0021             *   PREORDER (PREORD), OR ENDORDER
0022             *   (ENDORD).
0023             *****
0024             *PREORDER MACRO DEFINITION
0025             *
0026             PREORD $MACRO A,OP,B
0027             $VAR C VARIABLE TO HOLD '**' FOR COMMENTS
0028             *
0029             *PRINT THE OPERATION
0030             *
0031             $ASG '**' TO C.S
0032             $IF OP.A&$PCALL = 0
0033             $ASG '**' TO OP.S
0034             $ENDIF
0035             :C: :OP:
0036             *
0037             *PRINT THE FIRST OPERAND
0038             *
0039             $IF A.A&$POPL
0040             PREORD :A:
0041             $ELSE
0042             :C: :A:
0043             $ENDIF
0044             *
0045             *PRINT THE SECOND OPERAND
0046             *
0047             $IF B.A&$POPL
0048             PREORD :B:
0049             $ELSE

```

```

0050      :C: :B:
0051          $ENDIF
0052          $END
0053          *****
0054      *ENDORDER MACRO DEFINITION
0055      *
0056      ENDORD $MACRO A,OP,B
0057          $VAR C VARIABLE TO HOLD '*' FOR COMMENTS.
0058          $ASG '*' TO C.S
0059      *
0060      *PRINT THE FIRST OPERAND
0061      *
0062          $IF A.A&$POPL
0063          ENDORD :A:
0064          $ELSE
0065      :C: :A:
0066          $ENDIF
0067      *
0068      *PRINT THE SECOND OPERAND
0069      *
0070          $IF B.A&$POPL
0071          ENDORD :B:
0072          $ELSE
0073      :C: :B:
0074          $ENDIF
0075      *
0076      *PRINT THE OPERATION
0077      *
0078          $IF OP.A&$PCALL = 0 THEN
0079          $ASG '*' TO OP.S
0080          $ENDIF
0081      :C: :OP:
0082          $END
0083      *
0084      *SAMPLE MACRO CALLS
0085      *
0086      PREORD A,/,B
0087      * /
0088      * A
0089      * B
0090      ENDORD A,/,B
0091      *A
0092      *B
0093      */
0094      PREORD (A, + ,B),,(6,/, (2,-,B))
0095      **
0096          PREORD A,-,B
0097      * +
0098      *A
0099      *B
0100          PREORD 6,/, (2,-,B)

```

```

*0001 */
*0002 *6
*0003 PREORD 2,-,B
*0001 *-
*0002 *2
*0003 *B
0089 ENDORD (A,+,B),,(6/, (2,+,B))
*0001 ENDORD A,+,B
*0001 *A
*0002 *B
*0003 *-
*0002 ENDORD 6/, (2,-,B)
*0001 *6
*0002 ENDORD 2,-,B
*0001 *2
*0002 *B
*0003 *-
*0003 */
*0003 **
0090 PREORD ((X,+,Y),/(X,+,Y)),+,(1/,Z)
*0001 *-
*0002 PREORD (X,+,Y),/(X,+,Y)
*0001 */
*0002 PREORD X,+,Y
*0001 *+
*0002 *X
*0003 *Y
*0003 PREORD X,-,Y
*0001 *-
*0002 *X
*0003 *Y
*0003 PREORD 1/,Z
*0001 */
*0002 *1
*0003 *Z
0091 ENDORD ((X,+,Y),/(X,-,Y)),-(1/,Z)
*0001 ENDORD (X,+,Y),/(X,-,Y)
*0001 ENDORD X,+,Y
*0001 *X
*0002 *Y
*0003 *+
*0002 ENDORD X,-,Y
*0001 *X
*0002 *Y
*0003 *-
*0003 */
*0002 ENDORD 1/,Z
*0001 *1
*0002 *Z
*0003 */
*0003 *-

```

THE FOLLOWING SYMBOLS ARE UNDEFINED

A  
B  
X  
Y  
Z

NO ERRORS

# Relocatability and Program Linking

---

## 8.1 INTRODUCTION

The assembler for the Model 990 Computer supplies both absolute and relocatable object code that may be linked as required to form executable programs from separately assembled modules. This section contains guidelines to assist the user in taking full advantage of these capabilities.

## 8.2 RELOCATION CAPABILITY

Relocatable code includes information that allows a bootstrap loader to place the code in any available area of memory. This relocation capability allows the most efficient use of available memory and is required for disk-resident programs executed under an operating system. Absolute code must be loaded into a specified area of memory. Absolute code is appropriate for code that must be placed in dedicated areas of memory and may be used for memory-resident programs executing under operating systems.

Object code generated by an assembly is a representation of machine language instructions, addresses, and data comprising the assembled program. The code may include absolute segments, program-relocatable segments, data-relocatable segments, and numerous common-relocatable segments. In assembly language source programs, symbolic references to locations within a relocatable segment are called relocatable addresses. These addresses are represented in the object code as displacements from the beginning of a specified segment. A program-relocatable address, for example, is a displacement into the program segment. At load time, all program-relocatable addresses are adjusted by a value equal to the load address. Data-relocatable addresses are represented by a displacement into the data segment. There may be several types of common-relocatable addresses in the same program, since distinct common segments may be relocated independently of each other. A subsequent section of this manual describes the representation of these relocatable addresses in the object code.

### 8.2.1 Relocatability of Source Statement Elements

Elements of source statements are expressions, constants, and symbols. The relocatability of an expression is a function of the relocatability of the symbols and constants that make up the expression. An expression is relocatable when the number of relocatable symbols or constants added to the expression is one greater than the number of relocatable symbols or constants subtracted from the expression. (All other valid expressions are absolute.) When the first symbol or constant is unsigned, it is considered to be added to the expression. When a unary minus follows a subtraction operator, the effective operation is addition. For example, when all symbols in the following expressions are relocatable, the expressions are relocatable:

```
LABEL + 1  
LABEL + TABLE + - INC  
- LABEL + TABLE + INC
```

Decimal, hexadecimal, and character constants are absolute. Assembly-time constants defined by absolute expressions are absolute, and assembly-time constants defined by relocatable expressions are relocatable.

Any symbol that appears in the label field of a source statement other than an EQU directive is absolute when the statement is in an absolute block of the program. Any symbol that appears in the label field of a source statement other than an EQU directive is relocatable when the statement is in a relocatable block of the program.

The relocatability of expressions having logical and relational operators follows similar rules to those for expressions containing only arithmetic operators. The result of a logical operation between a relocatable constant or symbol and an absolute constant or symbol is relocatable. A logical operation between two relocatable elements of an expression is invalid. Relational operators result in an absolute value, zero or one. The relation is the assembly-time relation and ignores the effect of relocation on relocatable values.

To summarize, a location is either absolute or relocatable and may contain either absolute or relocatable values. The example program in Appendix G includes absolute locations with relocatable contents and relocatable locations with absolute contents.

### **8.3 PROGRAM LINKING**

Since the assembler includes directives that generate the information required to link program modules, it is not necessary to assemble an entire program in the same assembly. A long program may be divided into separately assembled modules to avoid a long assembly or to reduce the symbol table size. Also, modules common to several programs may be combined as required. Program modules may be linked by the link editor to form a linked object module that may be stored on a library and/or loaded as required. The following paragraphs define the linking information that must be included in a program module.

#### **8.3.1 External Reference Directives**

Each symbol from another program module must be placed in the operand field of an REF or SREF directive in the program module that requires the symbol.

#### **8.3.2 External Definition Directive**

Each symbol defined in a program module and required by one or more other program modules must be placed in the operand field of a DEF directive.

#### **8.3.3 Program Identifier Directive**

Program modules that are to be linked by the link editor should include an IDT directive. The module names in the character strings of the IDT directives should be unique.

#### **8.3.4 Linking Program Modules**

The link editor builds a list of symbols from REF directives as it links the program modules. The link editor matches symbols from DEF directives to the symbols in the reference list. The link editor follows linking commands to determine the modules to be linked. Refer to Section 5 for linking commands generatable from the assembler.

# Assembler Output

## 9.1 INTRODUCTION

The Model 990 Computer prints a source listing of the assembly code and the error or warning messages when these conditions are encountered. This section discusses the source listing and lists the error/warning codes output by the assembler. The object code format is also covered in this section.

## 9.2 SOURCE LISTING

The source listings show the source statements and the resulting object code. A typical listing is shown with the example program in Appendix G.

The assembler produces a cover sheet as the first output in the listing. This cover page contains a table that provides a record of the files and devices used during the assembly process. An example of this output is as follows:

### ACCESS NAMES TABLE

PAGE 0001

```

SOURCE ACCESS NAME =          .SUSAN.SRC.TEST1
OBJECT ACCESS NAME =
LISTING ACCESS NAME =        .SUSAN.LIST.TEST1
ERROR ACCESS NAME =
OPTIONS =                     12,XREF,SYMT,TUNLST,MUNLST
MACRO LIBRARY PATHNAME =     .SDSMAC.MACRODEF

```

```

LINE      KEY      NAME
0001      LI      .SDSMAC.MACRODEF
           =>.SDSMAC.MACRODEF
0001      LO      MACROS
           =>.SDSMAC.MACRODEF
0002      A       DSC.SYSTEM.TABLES.DOR
           =>DS01.SYSTEM.TABLES.DOR
0003      LI      SDSMAC.MACRODEF
           =>.SDSMAC.MACRODEF

```

The output has two sections:

- A listing of the parameters that were passed to the assembler via SCI.
- A list of access names encountered during the first pass of the assembly.

In the first section, any parameters that had no value are left blank. The fields in the second section are labeled as follows:

- LINE — This field contains the record number in which the access name was encountered.
- KEY — This field contains one of the following:
  - LI — indicating a LIBIN usage,
  - LO — indicating a LIBOUT usage,
  - one character — indicating a copy file to be given this character as a key.
- NAME — This field contains two access names. The first name is an image of the name on the source record. The second name, appearing after the =>, is the result of synonym substitution on the first name.

Each page of the source listing has a title line at the top of the page. Any title supplied by a TITL directive is printed on this line, and a page number is printed to the right of the title area. The printer skips a line below the title line and prints a line for each source statement listed. The line for each source statement contains a source statement number, a location counter value, object code assembled, and the source statement as entered. When a source statement results in more than one word of object code, the assembler prints the location counter value and object code on a separate line following the source statement for each additional word of object code. The source listing lines for a machine instruction source statement are shown in the following example:

```

0018          0156          C820          MOV          @INT + 3.@3
              0158          012B'
              015A          0003
    
```

The source statement number, 0018 in the example, is a four-digit decimal number. Source records are numbered in the order in which they are entered whether they are listed or not. The TITL, LIST, UNL, and PAGE directives are not listed, and source records between a UNL directive and a LIST directive are not listed. The difference between source record numbers printed indicates how many source records are not listed.

The next field on a line of the listing contains the location counter value, a hexadecimal value. In the example, 0156 is the location counter value. Not all directives affect the location counter, and those that do not affect the location counter leave this field blank. Specifically, of the directives that the assembler lists, the IDT, REF, DEF, DXOP, EQU, SREF, LOAD, and END directives leave the location counter field blank.

The third field normally contains a single blank. However, the assembler places a dash in this field when warning errors are detected.



The fourth field contains the hexadecimal representation of the object code placed in the location by the assembler, C820 in the example. The apostrophe following the field of the second line in the example indicates that the contents, 012B, are program-relocatable. A quote (") in this location would indicate that the location is data-relocatable, while a plus (+) would indicate that the label INIT is relocatable with respect to a common segment. All machine instructions and the BYTE, DATA, and TEXT directives use this field for object code. The EQU directive places the value corresponding to the label in the object code field.

The fifth field contains the characters of the source statement as they were scanned by the assembler. The length of this field is determined by the SETRM assembler directive and the PRINT WIDTH parameter supplied as an assembler input parameter. Default in both cases is 60 characters.

Spacing in this field is determined by the spacing in the source statement. The four fields of source statements will be aligned in the listing only when they are aligned in the same character positions in the source statements or when tab characters are used.

The machine instruction used in the example specifies the symbolic memory addressing mode for both operands. This causes the instruction to occupy three words of memory and three lines of the listing. The object code corresponds to the operands in the order in which they appear in the source statement.

### 9.3 ASSEMBLER ERROR MESSAGES

The assembler issues two kinds of error messages, depending on whether the assembly completed normally or abnormally. Normal completion errors are printed in the listing of the assembly and are described in paragraph 9.3.2 and Table 9-2. Abnormal completion error messages are described in the paragraph that follows.

#### 9.3.1 Abnormal Completion Error Messages

The ERROR ACCESS NAME parameter for the procedure that executes the assembler allows the programmer to specify the disposition of abnormal completion error messages and warning messages. The error access name may specify a file name, a listing device, device DUMMY, or be left blank. If a file name is specified, the file is automatically created, if it does not already exist, and may be displayed or printed like any other sequential file when the assembly has terminated. If a device name is specified, it is assumed to be a printer and the output is sent to the device upon completion of the assembly. This means that if a terminal name (ST01 for example) is specified, the output is sent to it as if it were a printer, so that the messages scroll rapidly up the screen and may not be recalled. If device DUMMY is specified, the output is discarded. This practice is not recommended. If the error access name is left blank, the messages are written to an error file for the terminal that bid the assembler and are displayed on the screen when the assembler terminates.

Most abnormal completion error messages are issued by the operating system under which the assembler runs and are described in the applicable operating system reference manual. Messages in this category include all messages dealing with file I/O errors. The errors listed in Table 9-1 are issued by the assembler and indicate some sort of problem with the assembler. If one of these messages is displayed, the user should first attempt to run the assembly again. In

most cases, these errors are caused by a transient error condition that does not persist. If the messages reappear on subsequent assemblies, load a fresh copy of the assembler from a backup. If the error still persists, contact a customer representative.

**Table 9-1. Abnormal Completion Messages**

---

ATTEMPT TO POP EMPTY STACK — SDSMAC BUG  
DIRECTIVE EXPECTED — SDSMAC BUG  
UNEXPECTED END OF PARSE — SDSMAC BUG  
ERROR MAPPING PARSE — SDSMAC BUG  
INVALID OPERATION ENCOUNTERED — SDSMAC BUG  
NO OP CODE — SDSMAC BUG  
INVALID LISTING ERROR ENCOUNTERED  
SYMBOL TABLE ERROR  
MACRO EXPANSION ERROR  
BUG — INVALID SDSLIB COMMAND ID  
UNKNOWN ERROR PASSED, CODE = XXXX

---

### 9.3.2 Assembly Listing Errors

SDSMAC prints the following error message on successive lines of the listing when an error is detected:

```
* * * ERROR DESCRIPTION  
  
LAST ERROR AT XXXX
```

The error description is the brief description shown in Table 9-2. The second line identifies the statement in which the previous error was detected.

At the end of the listing is an error summary as follows:

```
NNNN ERRORS, LAST ERROR AT XXXX, YYYY WARNINGS
```

NNNN is the count of the errors in the assembly. XXXX identifies the last error detected in the assembly; YYYY is the count of the warnings in the assembly. The second line of the error messages link the error messages so that the user may begin at the error summary message and readily locate all error messages. In an error-free assembly, the final message is as follows:

```
NO ERRORS, NO WARNINGS or NO ERRORS, XXXX WARNINGS
```

Several errors detected by the assembler (such as arithmetic overflow while evaluating expressions) are considered to be only warning errors. The programmer should examine the code generated when warning messages occur since the results may or may not be the code expected. Warning error messages are written only to the error file and are not included in the listing; however, a dash is placed in column 11 of the listing where the warning error occurred. Warning messages do not include an indication of a previous warning or error.

**WARNING**

The dash in column 11 is the only warning indication printed in the program listing. If the error file is assigned to device DUMY, corresponding warning messages will be lost.

**Table 9-2. Assembly Listing Errors**

Error Message	Possible Causes
ABSOLUTE VALUE REQUIRED	
ATTEMPT TO INDEX BY REGISTER ZERO	A warning.*
BAD ACCESS NAME SYNTAX	A warning.*
'CEND' ASSUMED	A warning.*
CLOSE (') MISSING	
COMMA MISSING	
COMMON TABLE OVERFLOW	Too many common segments used (127 maximum)
CONDITIONAL ASSEMBLY NESTING ERROR	An if-then-else construct is in error. The following conditions could cause this:  A) Missing ASMENDs. B) Surplus ASMELSSs. C) Surplus ASMENDs.
'DEND' ASSUMED	A warning.*
DIRECTORY OPEN ERROR	Check that any synonyms are valid and that no other processor is currently writing to the macro library.
DIRECTORY READ ERROR	An I/O error was encountered while trying to read a macro library directory. Verify that no other processor is currently writing to that macro library.
DIRECTORY REQUIRED	The access name specified is not an existing directory. Verify that all synonyms are correct and that the macro library does indeed exist; it cannot be autocreated.
DIRECTORY WRITE ERROR	Verify that no other processor is currently writing to that macro library.

Table 9-2. Assembly Listing Errors (Continued)

Error Message	Possible Causes
DISPLACEMENT TOO BIG	An instruction requiring an operand with a fixed upper limit was encountered that overflowed this limit. An example is the JMP instruction, whose single operand must evaluate to within >7F words distance from the current program counter.
'DSEG' ASSUMED	This is a warning that the following two statements have the same result:  CSEG 'DATA' DSEG
DUPLICATE DEFINITION	A) The symbol appears more than once in the label field of the source.  B) The symbol appears as an operand of a REF statement as well as in the label field of the source.  C) An attempt was made to define a macro variable or macro language label that was previously defined in the macro.
ERROR EXPANDING CALL	The symbol in the operand field of the \$CALL statement is not a defined macro.
ERROR ON COPY OPEN	The access name specified as the operand of a copy directive cannot be opened. Check that the synonyms are correct and that the file is not currently being written to by another processor.
EXPRESSION SYNTAX ERROR	A) Unbalanced parentheses.  B) Invalid operations on relocatable symbols.
INDIRECT(*)MISSING	
INVALID \$ASG VARIABLE	A) An attempt was made to change the length component of a variable.  B) An attempt was made to change the attribute component or the value component that was declared as a macro language label.

Table 9-2. Assembly Listing Errors (Continued)

Error Message	Possible Causes
INVALID CHARACTER IN SYMBOL - BLANK USED	C) The destination variable is not present or is not a symbol.  A warning.* The legal characters to be used in symbols under SDSMAC are A-Z, 0-9, ',', and '\$'.
INVALID CONDITION	The List Search instructions require conditions to be specified as one of the operands. The following are legal conditions: EQ, NE, HE, L, GE, LT, LE, H, LTE, GT.
INVALID CRU OR SHIFT VALUE	A warning.*
INVALID DIRECTIVE IN ABSOLUTE CODE	The directives PEND, DEND, and CEND have no meaning in absolute code.
INVALID EXPRESSION	This may indicate invalid use of a relocatable symbol in arithmetic.
INVALID MACRO EXPRESSION	Invalid construct in \$ASG statement.
INVALID MACRO VARIABLE	The target variable specified on an \$ASG or a \$GOTO verb is not a valid target variable.
INVALID MODEL STATEMENT	A macro symbol in a model statement must be followed with either a colon operator (:) or an end-of-record.
INVALID OPCODE	The second field of the source record contained an entry that is not a defined instruction, directive, pseudo-op, DXOP, DFOP, or macro name.
INVALID OPTION	A warning. The only legal options are the following:  XREF SYMT NOLIST MUNLST TUNLST BUNLST DUNLST FUNL 10 or 12 (or suitable abbreviation).

Table 9-2 Assembly Listing Errors (Continued)

Error Message	Possible Causes
INVALID RELOCATION TYPE	Only PSEG relocatable or absolute symbols are allowed as the operand of an END statement.
INVALID USE OF CONDITIONAL ASSEMBLY	A conditional assembly directive may not appear as a model statement.
INVALID USE OF REF'D SYMBOL	REF'D symbols may appear in expressions only under certain circumstances (see paragraph 2.5.1).
INVALID \$ASG EXPRESSION	The expression is not present.
INVALID \$IF EXPRESSION	The expression either is not present or does not evaluate to an integer value.
LABEL REQUIRED	\$NAME statements must begin with a label of maximum length two. \$MACRO statements must begin with a label of maximum length six.
MACRO DEFINITION DISCARDED DUE TO ERRORS	An error was detected during the assembly of the macro definition. Use of the macro name in succeeding lines will cause error messages.
MACRO LIBRARY READ ERROR	A LIBIN was in effect and the statement was a macro in a specified macro library, but an I/O error was encountered when reading it.
MACRO LIBRARY WRITE ERROR	The current LIBOUT library could not be used at completion of a macro definition. Check that the macro is not currently being written by another processor.
MACRO SYMBOL TRUNCATED.	<p>A warning.*</p> <p>The maximum length for a macro symbol is two characters. The following are legal macro symbols: A, A.S, B2. SV.</p> <p>The following are illegal macro symbols: CNT, CNT.A, PM2.SL.</p>

Table 9-2. Assembly Listing Errors (Continued)

Error Message	Possible Causes
MAX MACRO NESTING STACK DEPTH OVERFLOW	<p>A) A macro calls itself recursively more than the allowed maximum number of times.</p> <p>B) More levels of macro calling have been used than the allowed maximum.</p>
MEMORY EXCEEDED	The program counter overflowed the value >FFFF.
MISSING \$END	
MODEL STATEMENT TRUNCATED.	<p>A warning.* When expanded, the model statement exceeded 80 characters in length.</p>
OPEN '(' MISSING	<p>A parenthesized operand is required with the Extract Field, Extract Value, Insert Field, and Invert Order of Field Insert instructions.</p>
OPERAND CONFLICT PASS1/PASS2	<p>The assembler defaults currently undefined symbols to register uses in the first pass if that symbol is used in an ambiguous way. If during the second pass it is discovered that the symbol was not a register use, this error will result. An example is as follows:</p> <pre data-bbox="883 1360 1187 1503">           BL      SUB           .           . SUB      EQU     \$ </pre> <p>If this had been coded as follows, no ambiguity would have existed due to the explicit "@" sign:</p> <pre data-bbox="883 1650 1214 1793">           BL      @SUB           .           . SUB      EQU     \$ </pre>

Table 9-2. Assembly Listing Errors (Continued)

Error Message	Possible Causes
OPERAND MISSING	On instructions having a fixed number of operands, too few appeared before encountering a blank. On instructions having a variable number of operands, such as DATA, a comma may have been encountered with no operand following it. An expression extending beyond the right margin could cause this problem.
'PEND' ASSUMED	A warning.*
REGISTER REQUIRED	
STRING REQUIRED	
STRING TRUNCATED	A warning.* Check the syntax for the directive in question to determine the maximum length for the string.
SYMBOL TRUNCATED	A warning.* The maximum length for a symbol is six characters.
SYMBOL REQUIRED	
SYMBOL USED IN BOTH REF AND DEF	This is a conflicting, duplicate definition.
SYNTAX ERROR	
'TO' MISSING	'TO' is a required part of the syntax for the \$ASG macro verb.
UNDEFINED MACRO VARIABLE	The target variable specified in an \$ASG or a \$GOTO verb is undefined.
UNDEFINED SYMBOL	A) A symbol is used that did not appear in the label field of a source record.  B) The use requires definition in the first pass and is undefined when the assembler first encounters it.
VALID OPCODE REQUIRED	The defining symbol (that is, the second operand) is not a valid instruction or directive.



Table 9-2. Assembly Listing Errors (Continued)

Error Message	Possible Causes
VALUE TRUNCATED	A warning.* Overflow is checked after every operation in an arithmetic expression. This may result in several truncations in one expression.
WORKSPACE ADDRESS NOT PREVIOUSLY DEFINED	The operand field must have been previously defined. Note that the WPNT directive (or implied WPNT) is ignored. Any previous WPNT is also ignored from this point on.
\$IF - \$ELSE - \$ENDIF CONSTRUCT IN ERROR	Possible errors are as follows: A) Surplus \$ELSEs. B) Surplus \$ENDIFs. C) Missing \$ENDIFs.
\$MACRO INVALID WITHIN MACRO DEFINITION	A) The \$END verb belonging to the previous macro was missing. B) A \$MACRO verb was unintentionally included.
MACRO STRING OVERFLOW	While building a concatenated string, the string length exceeded 255 characters.
<b>Note:</b>	
	*Warnings are defined by a dash (-) in column 11 of the assembled program listing.

## 9.4 CROSS-REFERENCE LISTING

The assembler prints an optional cross-reference listing following the source listing. The format of the listing is shown in Figure 9-1. In the left column, the assembler prints each symbol defined or referenced in the assembly. In the second column, the attribute of the symbol is indicated as a single character. The characters that appear in the second column, and their meanings, are listed in Table 9-3. The third column contains a four-digit hexadecimal number, the value assigned to the symbol. The number of the statement that defines the symbol appears in the fourth column, unless the symbol is undefined. For undefined symbols, the fourth column is left blank. The right column contains a list of the numbers of statements that reference the symbol, or is left blank if the symbol is never used.

CROSS REFERENCE

LABEL	VALUE	DEFN	REFERENCES
ADDT	01A8'	0325	0314
ADSR D	01A0'	0316	0342 0343 0348 0349
GT	0006	0997	
OBTCHN R		0088	
SQUIB U			0127 0233

Figure 9-1. Cross-Reference Listing Format

Table 9-3. Symbol Attributes

Character	Meaning
R	External reference (REF)
D	External definition (DEF)
U	Undefined
M	Macro name
S	Secondary reference (SREF)
L	Force load (LOAD)

9.5 OBJECT CODE

The assembler produces object code that may be linked to other object code modules or programs and loaded into the Model 990 Computer, or which may be loaded into the computer directly. Object code consists of records containing up to 71 ASCII characters each. The format, described in the next paragraph, permits correction using a keyboard device. Reassembly to correct errors is unnecessary. An example of output code is shown in Figure 9-2.

```
00000SAMPRG 90040C0000A0020BC06DB000290042C0020A0024BC81BC002A7F219F
A0028B0241B0000BCB41B0002B0380A00CAC0052C00A2B02E0C0032B0200B0F0F7F1DEF
A00D6BC0A0C00CAB04C3BC160C00CCBC1A0C00D0BC072B0281B3A00A00ECB02217F151F
A00EEB0900B06C1A00EAB1102A00F2B0543B11F8B2C20C0032BC101B0B44BE0447F18EF
A0100BDD66B0003B0282C00A2B11EDB03407F832F
200CE0010C      7FCABF
:
```

Figure 9-2. Object Code Example

### 9.5.1 Object Code Format

The object record consists of a number of tag characters, each followed by one to three fields as defined in Table 9-4. The first character of a record is the first tag character, which tells the loader which field or fields follow the tag. The next tag character follows the end of the field or fields associated with the preceding tag character. When the assembler has no more data for the record, the assembler writes the tag character seven followed by the checksum field and the tag character F, which requires no fields. The assembler then fills the rest of the record with blanks and a sequence number and begins a new record with the appropriate tag character.

Tag character zero is followed by two fields. Field one contains the number of bytes of program-relocatable code, and field two contains the program identifier assigned to the program by an IDT directive. When no IDT directive is entered, the field contains blanks. The linker uses the program identifier to identify the program, and the number of bytes of program-relocatable code to determine the load bias for the next module or program. The assembler places a single tag character zero at the beginning of each program.

The tag character M, used only when data or common segments are defined in the program, is followed by three fields. Field one contains the length, in bytes, of data- or common-relocatable code, field two contains the data or common segment identifier, and field three contains a "common number." The identifier is a six-character field containing the name \$DATA*b* for data segments and \$BLANK for blank common segments. If a named common segment appears in the program, an M tag will appear in the object code with an identifier field corresponding to the operand in the defining CSEG directive(s). Field three of the M tag consists of a four-character hexadecimal number defining a unique common number to be used by other tags that reference or initialize data of that particular segment. For data segments, this common number is always zero. For common segments (including blank common), the common numbers are assigned in increasing order, beginning at one and ending with the number of different common segments. The maximum number of common segments that a program may contain is 127.

Tag characters 1 and 2 are used with entry addresses. Tag character 1 is used when the entry address is absolute. Tag character 2 is used when the entry address is relocatable. Field one contains the entry address in hexadecimal. One of these tags may appear at the end of the object code file. The associated field is used by the linker to determine the entry point at which execution starts when the linking is complete.

Tag characters 3, 4, and X are used for external references. Tag character 3 is used when the last appearance of the symbol in field two of the tag is in program-relocatable code. Tag character 4 is used when the last appearance of the symbol is in absolute code. The X tag is used when the last appearance of the symbol in field two is in data- or common-relocatable code. Field three of the X tag gives the common number. Field one of the tag characters contains the location of the last appearance of the symbol. The symbol in field two is the external reference. Both fields are used by the linker to provide the desired linking to the external reference.

For each external reference in a program, there is a tag character in the object code with a location or an absolute zero, and the symbol that is referenced. When field one of the tag character contains absolute zero, no location in the program requires the address that corresponds to the reference. When field one of the tag character contains a location, the address corresponding to the reference is placed by the linker in the location specified and the location's previous value is used to point to the next location or, if the previous value is absolute zero, linking is discontinued.

**Table 9-4. Object Record Format and Tags**

Tag	1st Field	2nd Field	3rd Field
<b>Module Definition</b>			
0	PSEG Length	Program ID(8)	
M	DSEG Length	\$Data	0000
M	Blank Common Length	\$Blank	Common #
M	CSEG Length	Common Name(6)	Common #
<b>Entry Point Definition</b>			
1	Absolute Address		
2	P-R Address		
<b>Load Address</b>			
9	Absolute Address		
A	P-R Address		
S	D-R Address		
P	C-R Address	Common or CBSEG #	
<b>Data</b>			
B	Absolute Value		
C	P-R Address		
T	D-R Address		
N	C-R Address	Common or CBSEG #	
<b>External Definitions</b>			
6	Absolute Value	Symbol(6)	
5	P-R Address	Symbol(6)	
W	D-R/C-R Address	Symbol(6)	Common #
<b>External References</b>			
3	P-R Address of Chain	Symbol(6)	
4	Absolute Address of Chain	Symbol(6)	
X	D-R/C-R Address of Chain	Symbol(6)	
E	Symbol Index Number	Absolute Offset	Common #
<b>Symbol Definitions</b>			
G	P-R Address	Symbol(6)	
H	Absolute Value	Symbol(6)	
J	D-R/C-R Address	Symbol(6)	Common #
<b>Force External Link</b>			
U	0000	Symbol(6)	
<b>Secondary External Reference</b>			
V	P-R Address of Chain Entry	Symbol(6)	
Y	Absolute Address of Chain	Symbol(6)	
Z	D-R/C-R Address of Chain	Symbol(6)	Common #
<b>Check Sum</b>			
7	Value		
<b>Ignore Check Sum</b>			
8	Any Value		

Table 9-4. Object Record Format and Tags (Continued)

Tag	1st Field	2nd Field	3rd Field
	Load Bias D Absolute Address		
	End of Record F		
	Repeat Count R Value	Repeat Count	
	Program ID (SYMT Option) I P-R Address	Program ID(8)	
	COBOL Segment Reference Q Record Offset	CBSEG #	
<b>Notes:</b>			
1. All field widths are four characters unless otherwise specified by numbers in parentheses.			
2. If the first tag is 01 (hex), the file is in compressed object format.			
3. P-R Program Segment Relative (Address) D-R Data Segment Relative (Address) C-R Common Segment Relative (Address)			

Figure 9-3 illustrates the chain of the external reference EXTR. The object code contains the following tag and fields:

4C00EEXTR

At location C00E, the address C00A points to the preceding appearance of the reference. The chain includes both absolute and relocatable addresses and consists of absolute addresses C00E, C00A, C006, and C002, relocatable addresses 029E, 029A, and 0298, absolute addresses B00E, B00A, B006, and B002, and relocatable addresses 0290 and 028E. Each location points to the preceding appearance, except for location 028E, which contains zero. The zero identifies location 028E as the first appearance of EXTR and the end of the chain.

Tag character E is also used for external references. An E tag is used when a nonzero quantity is to be added to a reference. Field 1 identifies the reference by occurrence in the object code (0, 1, 2, ...). Field 2 is the value to be added to the reference after the reference is resolved. The value in field one is an index into references identified by 3, 4, V, X, and Y tags in the object code. The list is maintained by order of occurrence; that is, the first entry in the list is the symbol located in field two of the first 3, 4, V, X, Y, or Z tag. The index to that reference in the E tag would be 0000.

```

0229          *
0230          *      DEMONSTRATE EXTERNAL REFERENCE LINKING
0231          *
0232          REF  EXTR
0233 028C      RORG
0234 028C  C820  MOV  @EXTR, @EXTR
          028E  0000
          0290  028E
0235 0292  28E0  XOR  @EXTR, 3
          0294  0290
0236 B000      AORG >B000
0237 B000  3220  LDCR @EXTR, 8
          B002  0294
0238 B004  0420  BLWP @EXTR
          B006  B002
0239 B008  0223  AI   3, EXTR
          B00A  B006
0240 B00C  38A0  MPY  @EXTR, 2
          B00E  B00A
0241 0296      RORG
0242 0296  C820  MOV  @EXTR, @EXTR
          0298  B00E
          029A  0298
0243 029C  28E0  XOR  @EXTR, 3
          029E  029A
0244 C000      AORG >C000
0245 C000  3220  LDCR @EXTR, 8
          C002  029E
0246 C004  0420  BLWP @EXTR
          C006  C002
0247 C008  0223  AI   3, EXTR
          C00A  C006
0248 C00C  38A0  MPY  @EXTR, 2
          C00E  C00A

```

**Figure 9-3. External Reference Example**

Tag characters 5, 6, and W are used for external definitions. Tag character 5 is used when the location is program-relocatable. Tag character 6 is used when the location is absolute. Tag character W is used when the location is data- or common-relocatable. The fields are used by the linker to provide the desired linking to the external definition. Field two contains the symbol of the external definition. Field three of tag character W contains the common number.

Tag character 7 precedes the checksum, which is an error detection word. The check sum is formed as the record is being written. It is the two's complement of the sum of the eight-bit ASCII values of the characters of the record from the first tag of the record through the checksum tag, 7.

Tag characters 9, A, S, and P are used with load addresses for data that follows. Tag character 9 is used when the load address is absolute. Tag character A is used when the load address is program-relocatable. Tag character two is used when the load address is data-relocatable. Tag character P is used when the load address is common-relocatable. Field one contains the address at which the following data word is to be loaded. A load address is required for a data word that is to be placed in memory at some address other than the next address. The load address is used by the linker. Field two of tag character P contains the common number.

Tag characters B, C, T, and N are used with data words. Tag character B is used when the data is absolute, that is, an instruction word or a word that contains text characters or absolute constants. Tag character C is used for a word that contains a program-relocatable address. Tag character T is used for a word that contains a data-relocatable address. Tag character N is used for a word that contains a common-relocatable address. Field one contains the data word. The linker places the data word in the memory location specified in the preceding load address field or in the memory location that follows the preceding data word. Field two of tag character N contains the common number.

Tag characters G, H, and J are used when the symbol table option is specified. Tag character G is used when the location or value of the symbol is program-relocatable, tag character H is used when the location or value of the symbol is absolute, and tag character J is used when the location or value of the symbol is data- or common- relocatable. Field one contains the location or value of the symbol, and field two contains the symbol to which the location is assigned. Field three of tag character J contains the common number.

Tag character U is generated by the LOAD directive. The symbol specified is treated as if it were the value specified in an INCLUDE command to the linker. Field one contains zeros. Field two contains the symbol for which the loader will search for a definition. Refer to the LOAD directive for further information.

Tag characters V, Y, and Z are used for secondary external references. Field two contains the symbol that is used as the secondary external reference. If the last appearance of the symbol is in program-relocatable code, a V tag is used. Tag character Y is used when the last appearance of the symbol is in absolute code. Tag character Z is used when the last appearance of the symbol is in data- or common-relocatable code. Field one of these tags contains the location of the last appearance of the symbol, field two contains the symbol, and field three of the Z tag gives the common number. The link editor uses these fields to provide linking to the secondary external reference.

Tag character 8 is used to ignore the checksum, which is contained in field one.

Tag character D is used to specify a load bias. Field one contains the absolute address that will be used by the loader to relocate the symbols when loaded. The link editor does not accept the D tag. Tag character D is described in detail in a subsequent paragraph.

Tag character F indicates the end of record. It may be followed by blanks.

The last record of an object module has a colon (:) in the first character position of the record, followed by blanks or a time and date identifying stamp.

### 9.5.2 Machine Language Format

Some of the data words preceded by tag character B represent machine instructions. Comparing the source listing with the object code fields identifies the data words that represent machine instructions. Figure 9-4 shows the manner in which the bits of the machine instructions relate to the operands in the source statements for each format of the machine instructions.

### 9.5.3 Symbol Table

When the SYMT option is specified, the symbol table is included in the object code file. One entry, using tag character G, H, or J as appropriate, is supplied for each symbol defined in the assembly.

### 9.5.4 Changing Object Code

In most cases, changing the object code is not the best way to correct errors in a program. All changes or corrections to a program should be made in the source code, then the program should be reassembled. Failure to follow this principle can make subsequent correction or maintenance of the program impossible. The information in the following paragraphs is intended for those rare instances when reassembly is not possible. Any changes made directly to the object code should be thoroughly documented so that the programmers who come later can see what the program actually does, not what the source code says that it does.

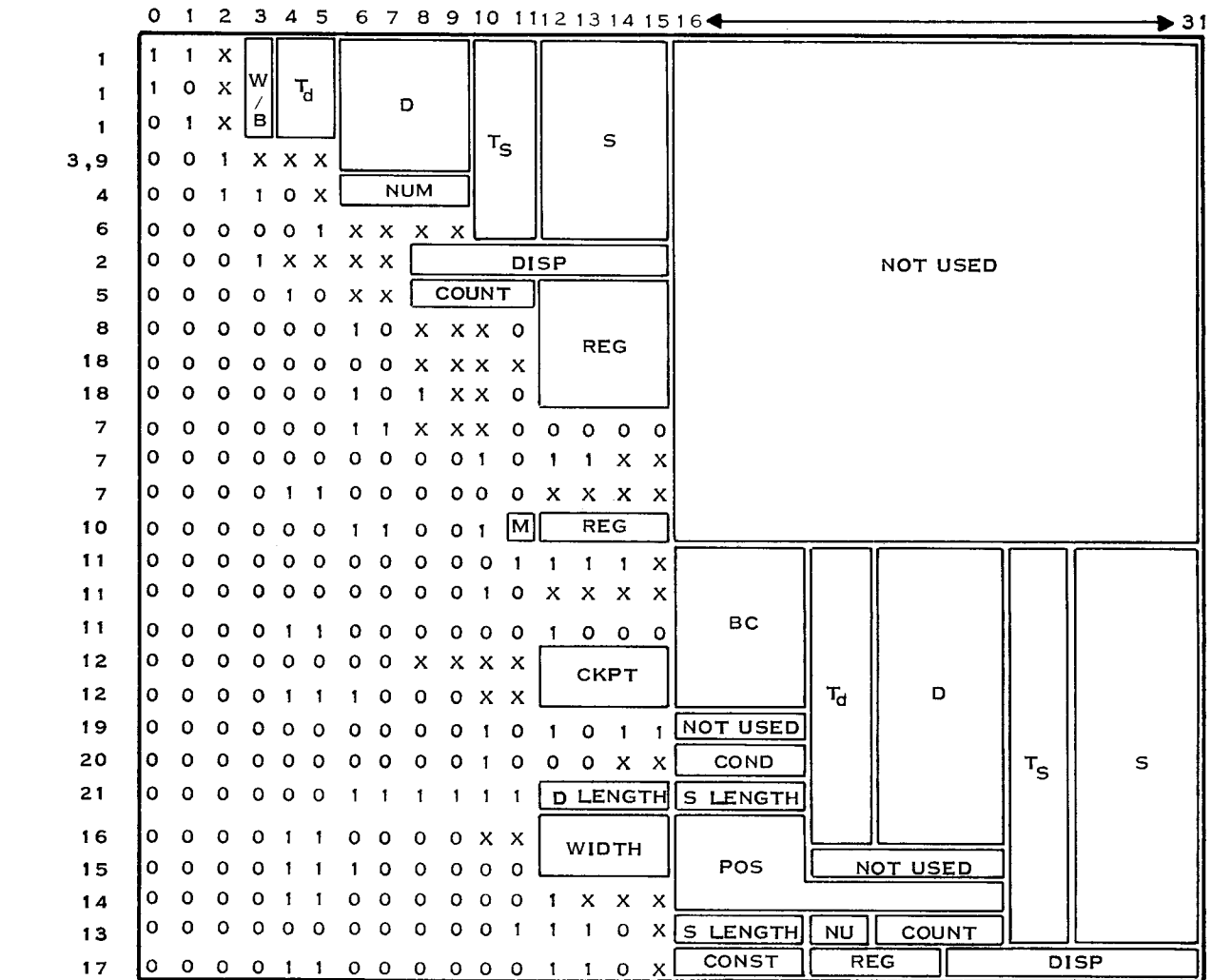
To correct the object code without reassembling a program, change the object code by changing or adding one or more records. One additional tag character is recognized by the loader to permit specifying a load point. The additional tag character, D, may be used in object records changed or added manually.

Tag character D is followed by a load bias (offset) value. The loader uses this value instead of the load bias computed by the loader itself. The loader adds the load bias to all relocatable entry addresses, external references, external definitions, load addresses, and data. The effect of the D tag character is to specify that area of memory into which the loader loads the program. The tag character D and the associated field must be placed ahead of the object code generated by the assembler.

Correction of the object code may require only changing a character or a word in an object code record. The user may duplicate the record up to the character or word in error, replace the incorrect data with the correct data, and duplicate the remainder of the record up to the seven tag character. Because the changes the user has made will cause a checksum error when the checksum is verified as the record is loaded, the user must change the seven tag character to eight.

When more extensive changes are required, the user may write an additional object code record or records. Begin each record with a tag character 9, A, S, or P, followed by an absolute load address or a relocatable load address. This may be an address into which an existing object code record places a different value. The new value on the new record will override the other value when the new record follows the other record in the loading sequence. Follow the load address with a tag character B, C, T, or N and an absolute data word or a relocatable data word. Additional data words preceded by appropriate tag characters may follow. When additional data is to be placed at a nonsequential address, write another load address tag character followed by the load address and data words preceded by tag characters. When the record is full, or all changes have been written, write tag character F to end the record.





- X IS A BIT OF THE OPERATION CODE THAT IS EITHER 0 OR 1 ACCORDING TO THE SPECIFIC INSTRUCTION IN THE FORMAT
- W/B IS A BIT IN THE OPERATION CODE THAT IS 0 IN INSTRUCTIONS THAT OPERATE ON WORDS, AND 1 IN INSTRUCTIONS THAT OPERATE ON BYTES
- T<sub>D</sub> IS A PAIR OF BITS THAT SPECIFY THE ADDRESSING MODE OF THE DESTINATION OPERAND, AS FOLLOWS:
  - 00 = WORKSPACE REGISTER ADDRESSING
  - 01 = WORKSPACE REGISTER INDIRECT ADDRESSING
  - 10 = SYMBOLIC MEMORY ADDRESSING WHEN D = 0
  - 10 = INDEXED MEMORY ADDRESSING WHEN D ≠ 0
  - 11 = WORKSPACE REGISTER INDIRECT AUTOINCREMENT ADDRESSING
- D IS THE WORKSPACE REGISTER FOR THE DESTINATION OPERAND
- T<sub>S</sub> IS A PAIR OF BITS THAT SPECIFY THE ADDRESSING MODE OF THE SOURCE OPERAND AS SHOWN FOR T<sub>D</sub>
- S IS THE WORKSPACE REGISTER FOR THE SOURCE OPERAND
- NUM IS THE NUMBER OF BITS TO BE TRANSFERRED
- DISP IS A TWO'S COMPLEMENT NUMBER THAT REPRESENTS A DISPLACEMENT
- REG IS A WORKSPACE REGISTER ADDRESS
- COUNT IS A SHIFT COUNT
- M IS A MAP REGISTER FILE NUMBER (0 OR 1)
- BC IS A BYTE COUNT
- CKPT IS A CHECKPOINT REGISTER ADDRESS
- COND IS A LOGICAL SEARCH CONDITION (EQ,GT,ETC.)
- D LENGTH IS A BYTE COUNT OF THE DESTINATION OPERAND
- S LENGTH IS A BYTE COUNT OF THE SOURCE OPERAND
- WIDTH IS THE NUMBER OF BITS CONTAINED IN THE OPERAND
- POS IS A BIT POSITION
- CONST IS A CONSTANT TO BE ADDED TO OR SUBTRACTED FROM A WORKSPACE REGISTER
- NOT USED IS A GROUP OF BITS NOT USED IN THE INSTRUCTION
- NU IS NOT USED

(A)141481A

Figure 9-4. Machine Instruction Formats

When additional memory locations are loaded as a result of changes, the user must change field one of tag character zero, which contains the number of bytes of relocatable code. For example, if the object field written by the assembler contained  $1000_{16}$  bytes of relocatable code and the user has added eight bytes in a new object record, additional memory locations will be loaded. The user must find the zero tag character in the object code file and change the value following the tag character from 1000 to 1008; he must also change the seven tag character to eight in that record.

When added records place corrected data in locations previously loaded, the added records must follow the incorrect records. The loader processes the records as they are read from the object medium, and the last record that affects a given memory location determines the contents of that location at execution time.

The object code records that contain the external definition fields, the external reference fields, the entry address field, and the final program start field must follow all other object records. An additional field or record may be added to include reference to a program identifier. The tag character is 4, and the hexadecimal field contains zeros. The second field contains the first six characters of the IDT character string. External definitions may be added using tag character 5 or 6 followed by the relocatable or absolute address, respectively. The second field contains the defined symbol, filled to the right with blanks when the symbol contains less than six characters.

#### NOTE

Both object code that will be linked and object code that will be loaded by the bootstrap loader can be changed without reassembling the program. The link editor, though, will not accept tag character D in changed or added object records.

# Appendix A

## Character Set

All of the 990 assemblers recognize the ASCII characters listed in Table A-1. The macro assembler, SDSMAC, also accepts the characters listed in Table A-2 if they occur within quoted strings or in comment fields. The special characters in Table A-3 are not accepted by the 990 assemblers but may be recognized and acted upon appropriately by other programs. The device service routine for the card reader accepts (and stores into the calling program's buffer) all the characters listed in Tables A-1, A-2, and A-3.

All of the tables include the ASCII code for each character represented as a hexadecimal value and a decimal value. The tables also include the Hollerith Code for each character. Table A-1 also lists the IBM Model 29 Keypunch Character for those characters in Table A-1 whose keypunch character representation differs from the character representation.

**Table A-1. Character Set**

Hexadecimal Value	Decimal Value	Character	Hollerith Code
20	32	Space	Blank
21	33	!	11-8-2
22	34	"	8-7
23	35	#	8-3
24	36	\$	11-8-3
25	37	%	0-8-4
26	38	&	12
27	39	'	8-5
28	40	(	12-8-5
29	41	)	11-8-5
2A	42	*	11-8-4
2B	43	+	12-8-6
2C	44	,	0-8-3
2D	45	-	11
2E	46	.	12-8-3
2F	47	/	0-1
30	48	0	0
31	49	1	1
32	50	2	2
33	51	3	3
34	52	4	4
35	53	5	5
36	54	6	6
37	55	7	7

Table A-1. Character Set (Continued)

Hexadecimal Value	Decimal Value	Character	Hollerith Code
38	56	8	8
39	57	9	9
3A	58	:	8-2
3B	59	;	11-8-6
3C	60	<	12-8-4
3D	61	=	8-6
3E	62	>	0-8-6
3F	63	?	0-8-7
40	64	@	8-4
41	65	A	12-1
42	66	B	12-2
43	67	C	12-3
44	68	D	12-4
45	69	E	12-5
46	70	F	12-6
47	71	G	12-7
48	72	H	12-8
49	73	I	12-9
4A	74	J	11-1
4B	75	K	11-2
4C	76	L	11-3
4D	77	M	11-4
4E	78	N	11-5
4F	79	O	11-6
50	80	P	11-7
51	81	Q	11-8
52	82	R	11-9
53	83	S	0-2
54	84	T	0-3
55	85	U	0-4
56	86	V	0-5
57	87	W	0-6
58	88	X	0-7
59	89	Y	0-8
5A	90	Z	0-9
5B	91	[	12-2-8*
5C	92	\	0-2-8 0-8-2*
5D	93	]	11-1-8 : (vertical bar)*
5E	94	^	11-7-8 (logical NOT)
5F	95	_	0-5-8

**Note:**

\*IBM 29 Keypunch Character

Table A-2. Special Characters Recognized by SDSMAC in Quoted Strings and Comment Fields

Hexadecimal Value	Decimal Value	Character	Hollerith Code
60	96	'	8-1
61	97	a	12-0-1
62	98	b	12-0-2
63	99	c	12-0-3
64	100	d	12-0-4
65	101	e	12-0-5
66	102	f	12-0-6
67	103	g	12-0-7
68	104	h	12-0-8
69	105	i	12-0-9
6A	106	j	12-11-1
6B	107	k	12-11-2
6C	108	l	12-11-3
6D	109	m	12-11-4
6E	110	n	12-11-5
6F	111	o	12-11-6
70	112	p	12-11-7
71	113	q	12-11-8
72	114	r	12-11-9
73	115	s	11-0-2
74	116	t	11-0-3
75	117	u	11-0-4
76	118	v	11-0-5
77	119	w	11-0-6
78	120	x	11-0-7
79	121	y	11-0-8
7A	122	z	11-0-9
7B	123	{	12-0
7C	124	:	12-11
7D	125	}	11-0
7E	126	~	11-0-1
7F	127	DFL	12-9-7
80	128		11-0-9-8-1
81	129		0-9-1
82	130		0-9-2
83	131		0-9-3
84	132		0-9-4
85	133		11-9-5
86	134		12-9-6
87	135		11-9-7
88	136		0-9-8
89	137		0-9-8-1
8A	138		0-9-8-2
8B	139		0-9-8-3
8C	140		0-9-8-4
8D	141		12-9-8-1
8E	142		12-9-8-2
8F	143		11-9-8-3
90	144		12-11-0-9-8-1
91	145		9-1
92	146		11-9-8-2

**Table A-2. Special Characters Recognized by SDSMAC in Quoted Strings and Comment Fields (Continued)**

Hexadecimal Value	Decimal Value	Character	Hollerith Code
93	147		9-3
94	148		9-4
95	149		9-5
96	150		9-6
97	151		12-9-8
98	152		9-8
99	153		9-8-1
9A	154		9-8-2
9B	155		9-8-3
9C	156		12-9-4
9D	157		11-9-4
9E	158		9-8-0
9F	159		11-0-9-1
A0	160		12-0-9-1
A1	161		12-0-9-2
A2	162		12-0-9-3
A3	163		12-0-9-4
A4	164		12-0-9-5
A5	165		12-0-9-6
A6	166		12-0-9-7
A7	167		12-0-9-8
A8	168		12-8-1
A9	169		12-11-9-1
AA	170		12-11-9-2
AB	171		12-11-9-3
AC	172		12-11-9-4
AD	173		12-11-9-5
AE	174		12-11-9-6
AF	175		12-11-9-7
B0	176		12-11-9-8
B1	177		11-8-1
B2	178		11-0-9-2
B3	179		11-0-9-3
B4	180		11-0-9-4
B5	181		11-0-9-5
B6	182		11-0-9-6
B7	183		11-0-9-7
B8	184		11-0-9-8
B9	185		0-8-1
BA	186		12-11-0
BB	187		12-11-0-9-1
BC	188		12-11-0-9-2
BD	189		12-11-0-9-3
BE	190		12-11-0-9-4
BF	191		12-11-0-9-5
C0	192		12-11-0-9-6
C1	193		12-11-0-9-7
C2	194		12-11-0-9-8
C3	195		12-0-8-1
C4	196		12-0-8-2
C5	197		12-0-8-3

**Table A-2. Special Characters Recognized by SDSMAC in Quoted Strings and Comment Fields (Continued)**

Hexadecimal Value	Decimal Value	Character	Hollerith Code
C6	198		12-0-8-4
C7	199		12-0-8-5
C8	200		12-0-8-6
C9	201		12-0-8-7
CA	202		12-11-8-1
CB	203		12-11-8-2
CC	204		12-11-8-3
CD	205		12-11-8-4
CE	206		12-11-8-5
CF	207		12-11-8-6
D0	208		12-11-8-7
D1	209		11-0-8-1
D2	210		11-0-8-2
D3	211		11-0-8-3
D4	212		11-0-8-4
D5	213		11-0-8-5
D6	214		11-0-8-6
D7	215		11-0-8-7
D8	216		12-11-0-8-1
D9	217		12-11-0-1
DA	218		12-11-0-2
DB	219		12-11-0-3
DC	220		12-11-0-4
DD	221		12-11-0-5
DE	222		12-11-0-6
DF	223		12-11-0-7

**Table A-3. Additional Characters Recognized by the Card Reader Device Service Routine**

Hexadecimal Value	Decimal Value	Character	Hollerith Code
00	0	NUL	12-0-9-8-1
01	1	SOH	12-9-1
02	2	STX	12-9-2
03	3	ETX	12-9-3
04	4	EOT	9-7
05	5	ENQ	0-9-8-5
06	6	ACK	0-9-8-6
07	7	BEL	0-9-8-7
08	8	BS	11-9-6
09	9	HT	12-9-5
0A	10	LF	0-9-5
0B	11	VT	12-9-8-3
0C	12	FF	12-9-8-4
0D	13	CR	12-9-8-5
0E	14	SO	12-9-8-6
0F	15	SI	12-9-8-7
10	16	DLE	12-11-9-8-1

**Table A-3. Additional Characters Recognized by the Card Reader Device Service Routine (Continued)**

Hexadecimal Value	Decimal Value	Character	Hollerith Code
11	17	DC1	11-9-1
12	18	DC2	11-9-2
13	19	DC3	11-9-3
14	20	DC4	9-8-4
15	21	NAK	9-8-5
16	22	SYN	9-2
17	23	ETB	0-9-6
18	24	CAN	11-9-8
19	25	EM	11-9-8-1
1A	26	SUB	9-8-7
1B	27	ESC	0-9-7
1C	28	FS	11-9-8-4
1D	29	GS	11-9-8-5
1E	30	RS	11-9-8-6
1F	31	US	11-9-8-7
7F	127	DEL	12-9-7



# Appendix B

## Instruction Tables

---

The source formats for the machine instructions are summarized in 11 tables. Refer to Section 3 for complete descriptions of the instructions.

The tables are organized as follows:

B-1	Arithmetic Instructions
B-2	Branch Instructions
B-3	Compare Instructions
B-4	Control and CRU Instructions
B-5	Load and Move Instructions
B-6	Logical Instructions
B-7	Shift Instructions
B-8	Extended Operation Instruction
B-9	Long Distance Addressing Instructions
B-10	Conversion Instructions
B-11	Pseudo-Instructions

The following symbols are used in Tables B-1 through B-11:

Sa	Source Address
Sb	Source Byte
Sw	Source Word
Sf	Source Single or Double Word (Field Instructions)
Si	Source Instruction (One to four words)
Sd	Source Double Word
Sq	Source Quad Word
Ss	Source String
Sm	Source Multiple Precision
St	Source Stack Descriptor
Sk	Source Stack String

SI	Source List Search Control Block Signed Offset to Link Signed Offset to Compare Value to Compare Mask for Comparison Terminal Link Value (End of List)
Da	Destination Address
Db	Destination Byte
Dw	Destination Word
Df	Destination Single or Double Word (Field Instruction)
Dd	Destination Double Word
Dq	Destination Quad Word
Ds	Destination String
Dm	Destination Multiple Precision
Dt	Destination Stack Descriptor
Dk	Destination Stack String
DI	Destination List Pointer Block Pointer to Current Node Pointer to Previous Node
FPAw	Floating Point Accumulator (R0) Integer
FPAd	Floating Point Accumulator (R0,R1) Extended Integer or Real
FPAq	Floating Point Accumulator (R0-R3) Double Precision Real
I	Immediate Operand
Cs	String Length
Cm	Multiple Precision Length
Cj	Amount to Add/Subtract from Register
Cp	Position
Cl	List Search Termination Code
Wr	Workspace Register
Ww	Width
Wc	Checkpoint Register
Wt	Top-of-Stack Pointer (in Workspace Register)
Wm	EP Destination Multiple Precision Length
SC	Shift Count
Pb	Bit Position
Dj	Jump Displacement
X	Extended Operation Number
T	Number of Bits to Transfer (CRU Instructions)
M	Map File Number
ST	Status Register
PC	Program Counter
WP	Workspace Pointer

- ( ) The contents of the address within the parentheses
- Indicates "replaces"
- :
- Indicates "is compared to"

The following example shows the use of the symbols in the format column:

XOR Sw, Wr.

The format entry means that the mnemonic opcode XOR requires a general source address and a workspace register address separated by a comma. In the effect column, the symbols are used as in the following example:

(Sw) XOR (Wr) (Wr)

This means that the result of the exclusive OR of the contents of the source address and the contents of the workspace register replaces the previous contents of the workspace register. In the status bits test column, the symbols are used as in the following example:

(Wr):0

This means that the result placed in the status register is compared to zero and the status bits contain the result of the comparison.

Table B-1. Arithmetic Instructions

Instruction	Format	Effect	Notes	Opcode	Status Bits Affected	Status Bits Test	Format Number
Add Words	A Sw,Dw	(Sw) + (Dw) → (Dw)		A000	0-4	(Dw):0	1
Add Bytes	AB Sb,Db	(Sb) + (Db) → (Db)		B000	0-5	(Db):0	1
Absolute Value	ABS Sw	Sw  → (Sw)	Note 1	0740	0-4	(Sw):0	6
Add Double Precision Real	AD Sq	(Sq) + (FPAq) → (FPAq)		0E40	0-4	(FPAq):0	6
Add Immediate	AI Wr,I	(Wr) + I → (Wr)		0220	0-4	(Wr):0	8
Add Multiple Precision Integer	AM Sm,Dm,Cm	(Sm) + (Dm) → (Dm)	Note 2	002A	0-4	(Dm):0	11
Add Real	AR Sd	(Sd) + (FPAq) → (FPAq)		0C40	0-4	(FPAq):0	6
Count Ones	CNT0 Sm,Dm,Cm	(Dm) + the number of ones in (Sm) → (Dm)	Note 2	0020	2	#1's in (Dm):0	11
Divide Double Precision Real	DD Sq	(FPAq) - (Sq) → (FPAq)		0F40	0-4	(FPAq):0	6
Decrement	DEC Sw	(Sw) - 1 → (Sw)		0600	0-4	(Sw):0	6
Decrement by Two	DECT Sw	(Sw) - 2 → (Sw)		0640	0-4	(Sw):0	6
Divide	DIV Sw,Wr	(Wr) - (Sw) → (Wr)	Note 3,4	3C00	4	(Sw) ≤ (Wr)	9
Divide Signed	DIVS Sw	(R0,R1) - (Sw) → (R0,R1)	Note 3	0180	0-2,4	R0:0 (Sw) ≤ R0	6
Divide Real	DR Sd	(FPAq) - (Sd) → (FPAq)		0D40	0-4	(FPAq):0 -63 ≤ (exponent FPAq) ≤ 63	6
Increment	INC Sw	(Sw) + 1 → (Sw)		0580	0-4	(Sw):0	6

**Table B-1. Arithmetic Instructions (Continued)**

Instruction	Format	Effect	Notes	Opcode	Status Bits Affected	Status Bits Test	Format Number
Increment by Two	INCT Sw	$(Sw) + 2 \rightarrow (Sw)$		05C0	0-4	(Sw):0	6
Multiply Double Precision Real	MD Sq	$(Sq) \times (FPAq) \rightarrow (FPAq)$		0F00	0-4	(FPAq):0 -63 ≤ (exponent FPAq) ≤ 63	6
Multiply	MPY Sw,Wr	$(Sw) \times (Wr) \rightarrow (Wr)$	Note 5	3800	none		9
Multiply Signed	MPYS Sw	$(Sw) \times (R0) \rightarrow (R0,R1)$	Note 5	01C0	0-2	R0,R1:0	6
Multiply Real	MR Sd	$(Sd) \times (FPAd) \rightarrow (FPAd)$		0D00	0-4	(FPAd):0 -63 ≤ (exponent FPAd) ≤ 63	6
Negate	NEG Sw	$-(Sw) \rightarrow (Sw)$		0500	0-4	(Sw):0 (Sw) = 8000 <sub>16</sub>	6
Negate Double Precision Real	NEGD	$-(FPAq) \rightarrow (FPAq)$		0C03	0-2	(FPAq):0	7
Negate Real	NEGR	$-(FPAd) \rightarrow (FPAd)$		0C02	0-2	(FPAd):0	7
Subtract Words	S Sw,Dw	$(Dw) - (Sw) \rightarrow (Dw)$		6000	0-4	(Dw):0	1
Subtract Bytes	SB Sb,Db	$(Db) - (Sb) \rightarrow (Db)$		7000	0-5	(Db):0	1
Subtract Double Precision Real	SD Sq	$(FPAq) - (Sq) \rightarrow (FPAq)$		0EC0	0-4	(FPAq):0 -63 ≤ (exponent FPAq) ≤ 63	6
Subtract Multiple Precision Integer	SM Sm,Dm,Om	$(Dm) - (Sm) \rightarrow (Dm)$	Note 2	0029	0-4	(Dw):0	11
Subtract Real	SR Sd	$(FPAd) - (Sd) \rightarrow (FPAd)$		0CC0	0-4	(FPAd):0 -63 ≤ (exponent FPAd) ≤ 63	6

**Table B-1. Arithmetic Instructions (Continued)**

- Note 1: The original value of *Sw* is compared to zero.
- Note 2: The *Cm* field specifies the number of bytes for the multiple precision operands.
- Note 3: The contents of *Wr* and the next consecutive register (32-bit magnitude) are divided by *Sw* (16-bit magnitude). The quotient (16-bit magnitude) is placed in *Wr* and the remainder is placed in the next consecutive register. If *Wr* is R15, the remainder is placed in the memory location immediately following the workspace. In the DIVS instruction, R0 and R1 are always used.
- Note 4: If the divisor is less than or equal to the left half of the dividend, the instruction is aborted and the overflow status bit (bit 4) is set.
- Note 5: *Sw* is multiplied by *Wr*. The result (32-bit magnitude) is placed in registers *Wr* and *Wr* + 1. If *Wr* = 15, the least significant half of the result is placed in the memory location immediately following the workspace. In the MPYS instruction, R0 and R1 are always used.

Table B-2. Branch Instructions

Instruction	Format	Effect	Notes	Necessary Status	Opcod	Format Number
Add to Register and Jump	ARJ Dj,Cj,Mr	Cj + (Mr) → (Mr) Conditionally Dj → (PC)	Note 1	Unconditional	0C0D	17
Branch	B Sw	Sw → (PC)		Unconditional	0440	6
Branch Indirect	BIND Sw	(Sw) → (PC)		Unconditional	0140	6
Branch and Link	BL Sw	(PC) → (R11) Sw → (PC)		Unconditional	0680	6
Branch Immediate and Push Link to Stack	BLSK Wt,I	(PC) → (Wt) I → (PC)		Unconditional	0080	8
Branch and Load Workspace Pointer	BLMP Sw		Note 2	Unconditional	0400	6
Jump if Equal	JEQ Dj	(PC) + Dj → (PC)		Bit 2 = 1	1300	2
Jump if Greater Than	JGT Dj	(PC) + Dj → (PC)		Bit 1 = 1	1500	2
Jump if Logical High	JH Dj	(PC) + Dj → (PC)		Bit 0 = 1 and Bit 2 = 0	1B00	2
Jump if High or Equal	JHE Dj	(PC) + Dj → (PC)		Bit 0 or Bit 2 = 1	1400	2
Jump if Logical Low	JL Dj	(PC) + Dj → (PC)		Bit 0 and Bit 2 = 0	1A00	2
Jump if Low or Equal	JLE Dj	(PC) + Dj → (PC)		Bit 0 = 0 or Bit 2 = 1	1200	2

Table B-2. Branch Instructions (Continued)

Instruction	Format	Effect	Notes	Necessary Status	Opcode	Format Number
Jump if Less Than	JLT Dj	(PC) + Dj → (PC)		Bit 1 and Bit 2 = 0	1100	2
Unconditional Jump	JMP Dj	(PC) + Dj → (PC)		Unconditional	1000	2
Jump if No Carry	JNC Dj	(PC) + Dj → (PC)		Bit 3 = 0	1700	2
Jump if Not Equal	JNE Dj	(PC) + Dj → (PC)		Bit 2 = 0	1600	2
Jump if No Overflow	JNO Dj	(PC) + Dj → (PC)		Bit 4 = 0	1900	2
Jump on Carry	JOC Dj	(PC) + Dj → (PC)		Bit 3 = 1	1800	2
Jump if Odd Parity	JOP Dj	(PC) + Dj → (PC)		Bit 5 = 1	1000	2
Return with Workspace Pointer	RTWP		Note 3	Unconditional	0380	7
Subtract from Register and Jump	SRJ Dj,Cj,Wr	(Wr) - (Cj) → (Wr) Conditionally Dj → (PC)	Note 1	Unconditional	0C0C	17
Execute	X Si		Note 4	Unconditional	0480	6

Note 1: If the value of the register does not equal zero, or has not passed through zero (sign change), the jump is performed.

Note 2: This instruction is explained fully in Section 3. The execution can be summarized as follows:

$$\begin{array}{l}
 (Sw) \rightarrow (WP) \rightarrow \text{Previous}(WP) \rightarrow R13 \\
 (Sw+2) \rightarrow (PC) \rightarrow \text{Previous}(PC) \rightarrow R14 \\
 0000_{16} \rightarrow (ST) \rightarrow \text{Previous}(ST) \rightarrow R15
 \end{array}$$



**Table B-2. Branch Instructions (Continued)**

Note 3: This instruction is explained fully in Section 3. The execution can be summarized as follows:

R13 → (WP)  
R14 → (PC)  
R15 → (ST)

Note 4: In the Execute instruction, an instruction at address Si is executed. If the instruction is more than one word long (that is, data or address), the word(s) following the Execute instruction are used, not the words following Si. The executed instruction affects the status register normally.

**Table B-3. Compare Instructions**

Instruction	Format	Effect	Notes	Opcode	Status Bits Affected	Status Bits Test	Format Number
Compare Words	C Sw,Dw	None		8000	0-2	(Sw):(Dw)	1
Compare Bytes	CB Sb,Db	None		9000	0-2,5	(Sb):(Db)	1
Compare Immediate	CI Wr,I	None		0280	0-2	(Wr):I	8
Compare Ones Corresponding	COC Sw,Wr	None	Note 1	2000	2		3
Compare Strings	CS Ss,Ds,Cs,Wc	Position of first unequal byte → (Wc)	Note 2	0040	0-2	(Ss):(Ds)	12
Compare Zeros Corresponding	CZC Sw,Wr	None	Note 3	2400	2		3
Left Test for Ones	LTO Sm,Dw,Cm	Position of leftmost one + (Dw) → (Dw)	Note 4	001F	2		11
Right Test for Ones	RTO Sm,Dw,Cm	Position of rightmost one + (Dw) → (Dw)	Note 4	001E	2		11
Search String for Equal Byte	SEQB Sw,Ds,Cs,Wc	Index to equal byte → (Wc)	Note 5	0050	0-2		12
Search List Logical Address	SLSL C1,S1,D1		Note 6	0021	2		20
Search List Physical Address	SLSP C1,S1,D1		Note 6	0022	2		20
Search String for Not Equal Byte	SNEB Sw,Ds,Cs,Wc	Index to non-equal byte → (Wc)	Note 5	0E10	0-2		12
Test Memory Bit	TMB Sw,Pb	Bit at (Sw) + Pb → ST Bit 2		0C09	2		14

**Table B-3. Compare Instructions (Continued)**

- Note 1: The bits in the destination operand that correspond to bits equal to one in the source operand are compared to one. If the corresponding bits are equal to one, status bit 2 is set to one. Otherwise, status bit 2 is set to zero.
- Note 2: The two strings are searched until an unequal byte is found. When the unequal byte is found, status bits 0 through 2 reflect the comparison of the bytes. This instruction can be re-executed, which will cause the comparison to continue from the point where the previous unequal byte was found.
- Note 3: The bits in the destination operand that correspond to bits equal to one in the source operand are compared to zero. If the corresponding bits are equal to zero, status bit 2 is set to one. Otherwise, status bit 2 is set to zero.
- Note 4: The Cm field specifies the number of bytes of precision. If a one bit is not found, status bit 2 is set to one.
- Note 5: The source operand is a 2-byte value containing a mask byte and a data byte. All bytes in the destination string are masked before being compared to the data byte. The status register bits 0 through 2 are affected by each comparison made. In the SEQB instruction, if no equal bytes are found in the destination string, the status register reflects the comparison of the data byte and the last byte in the destination string. In the SNEB instruction, if no non-equal byte is found, status bit 2 is set to one. When a non-equal byte is found, status bits 0 and 1 reflect the results of the comparison.

Note 6: The SLSL and SLSP instructions are explained fully in Section 3. The source operand is the list search control block. The destination operand is the list pointer block. The C1 field specifies which of the following conditions must be met for the search to terminate:

C1 Field	Condition	C1 Field	Condition
EQ	Equal	LT	Arithmetic Less Than
NE	Not Equal	LE	Logical Low or Equal
HE	Logical High or Equal	H	Logical High
L	Logical Low	LTE	Arithmetic Less Than or Equal
GE	Arithmetic Greater Than or Equal	GT	Arithmetic Greater Than

The SLSL instruction searches the processor's directly accessible 64K byte address space. The SLSP instruction searches anywhere in the available physical memory. When the search condition is met, the destination operand contains a pointer to the list element where the condition was met and a pointer to the previous element.

**Table B-4. Control and CRU Instructions**

Instruction	Format	Effect	Opcode	Status Bits Affected	Status Bits Test	Format Number
Clock Off	CKOF	Note 1	03C0	None		7
Clock On	CKON	Note 2	03A0	None		7
Disable Interrupts	DINT	Note 3	002F	None		7
Enable Interrupts	EINT	Note 4	002E	None		7
Execute Micro-diagnostics	EMD	Note 5	002D	0-15		7
Idle	IDLE	Note 6	0340	None		7
Load Writable Control Store	LCS Wr	Note 7	00A0	None		18
Load CRU	LDCR Sa,T	Note 8	3000	0-2,5	(Sw):0	4
Load or Restart Execution	LREX	Note 9	03E0	7-15		7
Reset	RSET	Note 10	0360	None		7
Set CRU Bit to Logic One	SB0 Pb	Note 11	1D00	None		2
Set CRU Bit to Logic Zero	SBZ Pb	Note 12	1E00	None		2
Store CRU	STCR Sa,T	Note 13	3400	0-2,5	(Sw):0	4
Test Bit	TB Pb	Note 14	1F00	2		2

**Table B-4. Control and CRU Instructions (Continued)**

- Note 1: Disables 120 Hz clock.
- Note 2: Enables 120 Hz clock. If interrupt level five is enabled, an interrupt occurs every 8.33 milliseconds.
- Note 3: All interrupts except level zero are disabled. The interrupt mask of the status register is not affected.
- Note 4: The interrupts are enabled according to the current level of the interrupt mask of the status register.
- Note 5: This instruction is explained fully in Section 3. The hardware microcoded diagnostic test is performed on the system. If the test fails, the system halts and the fault lamp is lit. If the test succeeds, the system is restarted automatically.
- Note 6: Places the computer in an idle state. An interrupt or start signal causes the computer to resume execution at the instruction following the IDLE instruction.
- Note 7: This instruction is explained fully in Section 3. The writable control store (user-implemented microcode) is loaded from memory for subsequent execution.
- Note 8: Transfers consecutive data bits from the address specified by Sw to the CRU. The number of bits to be transferred is specified by T. The CRU base address is specified in workspace register I2, bits 3-14. The least significant bit of the word specified by sw is placed in the CRU bit addressed by R12. One to sixteen bits can be transferred. When eight bits or less are transferred, status bit 5 reflects the parity of the transferred data.
- Note 9: Performs a context switch. The contents of memory location hex FFFC are loaded into the WP, the contents of memory location hex FFFE are loaded into the PC, and the status register is cleared.
- Note 10: Disables all interrupts. Resets all directly connected I/O devices.
- Note 11: Sets the CRU bit at the address in R12 plus Pb to one.
- Note 12: Sets the CRU bit at the address in R12 plus Pb to zero.
- Note 13: Transfers consecutive data bits from the CRU to the address specified by Sw. The number of bits transferred is specified by T. The CRU base address is specified in R12, bits 3-14. The CRU bit addressed by R12 is placed in the least significant bit of the word addressed by Sw. One to sixteen bits can be transferred. When eight bits or less are transferred, status bit 5 reflects the parity of the transferred data.
- Note 14: Tests CRU bit addressed by R12 plus Pb. Sets status bit 2 to the value of the CRU bit.

Table B-5. Load and Move Instructions

Instruction	Format	Effect	Notes	Opcode	Status Bits Affected	Status Bits Test	Format Number
Insert Field	INSF Sw,Dw,Cp,Ww	Field (Sw) → (Dw)	Note 1	0C10	0-2	Field (Sw):0	16
Load Double Precision Real	LD Sq	(Sq) → (FPAq)		0F80	0-2	(Sq):0	6
Load Immediate	LI Wr, I	I → (Wr)		0200	0-2	I:0	8
Load Interrupt Mask	LIM Wr	(Wr) → ST Bits 12-15	Note 2	0070	12-15	None	18
Load Interrupt Mask Immediate	LIMI I	I → ST Bits 12-15	Note 3	0300	12-15	None	8
Load Memory Map File	LMF Wr, M	((Wr)) → Map File M	Note 4	0320	None		10
Load Real	LR Sd	(Sd) → (FPAq)		0D80	0-2	(Sd):0	6
Load Status Register	LST Wr	(Wr) → (ST)		0080	0-15	None	18
Load Workspace Pointer	LWP Wr	(Wr) → (WP)		0090	None		18
Load Workspace Pointer Immediate	LWPI I	I → (WP)		02E0	None		8
Move Word	MOV Sw, Dw	(Sw) → (Dw)		C000	0-2	(Dw):0	1
Move Address	MOVA Sa, Dw	Sa → (Dw)	Note 5	002B	0-2	(Dw):0	19
Move Byte	MOVB Sb, Db	(Sb) → (Db)		D000	0-2,5	(Db):0	1

**Table B-5. Load and Move Instructions (Continued)**

Instruction	Format	Effect	Notes	Opcode	Status Bits Affected	Status Bits Test	Format Number
Move String	MOVS Ss,Ds,Cs,Wc	(Ss) → (Ds)	Note 6	0060	0-2	(Ds):0	12
Move String from Stack	MVSK St,Ds,Cs,Wc	((St)) → (Ds)	Note 7	0000	0-2	(Ds):0	12
Move String Reverse	MVSR Ss,Ds,Cs,Wc	(Ss) → (Ds)	Note 6	00C0	0-2	(Ds):0	12
Pop String from Stack	POPS St,Ds,Cs,Wc	((St)) → (Ds)	Note 7	00E0	0-2		12
Push String to Stack	PSHS Ss,Dt,Cs,Wc	(Ss) → ((Dt))	Note 7	00F0	0-2	(Ds):0	12
Store Double Precision Real	STD Sq	(FPAq) → (Sq)		0FC0	0-2	(Sq):0	6
Store Program Counter	STPC Wr	(PC) → (Wr)		0030	None		18
Store Real	STR Sd	(FPAAd) → (Sd)		0DC0	0-2	(Sd):0	6
Store Status Register	STST Wr	(ST) → (Wr)		02C0	None		18
Store Workspace Pointer	STWP Wr	(WP) → (Wr)		02A0	None		18
Swap Bytes	SWPB Sw	(Sw) → (Sw)	Note 8	06C0	None		6
Swap Multiple Precision	SWPM Sm,Dm,Cm	(Sm) → (Dm) (Dm) → (Sm)	Note 9	0025	0-2	(Sm):(Dm)	11
Translate String	TS Sb,Db	(Sb)+(Db) (Db)		0E30	0-2		12

**Table B-5. Load and Move Instructions (Continued)**

Instruction	Format	Effect	Notes	Opcode	Status Bits Affected	Status Bits Test	Format Number
Extract Field	XF Sw,Dw,Cp,Ww	Field (Sw) → (Dw)	Note 10	0C30	0-2	(Dw):0	16
Extract Value	XV Sw,Dw,Cp,Ww	Value (Sw) → (Dw)	Note 10	0C20	0-2	(Dw):0	16

Note 1: The field is the number of bits specified by Ww, extracted from the source operand starting at the least significant bit. The field is inserted in the destination operand, starting from bit position Cp and proceeding toward the least significant bit.

Note 2: The four least significant bits of Ww are used.

Note 3: The immediate operand should have the value 0-15. If not, only the four least significant bits are used.

Note 4: The Ww operand points to six consecutive words in memory. These six words are loaded into the map file specified by M. M must have a value of zero, one, or two.

Note 5: The binary value of address Sa is loaded into memory at the location specified by Dw. Any of the five general forms of addressing can be used.

Note 6: The Cs field specifies the length of the string. Workspace register zero may also be used to specify the length of the string, and the string may be a tagged string, where the string length is specified in the tag. If the string is tagged, and the tag specifies the string length is zero, status bits 0-2 are reset to zero. MOVs and MWSR perform the same functionally, but the MWSR instruction begins the move from the last byte of the source string, to the last byte of the destination string.

Note 7: The St and Dt operands are addresses to the stack control block. The stack control block contains the top-of-stack pointer (address of top element on the stack), the stack limit, and the bottom of stack. The string length is specified by the Cs field, by R0, or by the string tag.

Note 8: The least significant byte of the source operand is moved to the most significant byte, and the most significant byte is moved to the least significant byte.

Note 9: The two multiple precision operands are swapped byte for byte. The Cm field specifies the number of bytes of precision.

Note 10: The bit field of width Ww beginning at position Cp at the source address is stored, right justified, at the destination address. The bit at Cp is the most significant bit of the field. The position and/or width can be specified in R0. In the XF instruction, unused bits in the destination operand are filled with zeros. In the XV instruction, unused bits are filled with the most significant bit of the extracted value (sign extension).



Table B-6. Logical Instructions

Instruction	Format	Effect	Notes	Opcode	Status Bits Affected	Status Bits Test	Format Number
And Immediate	ANDI Wr, I	(Wr) AND I → (Wr)		0240	0-2	(Wr):0	8
And Multiple Precision	ANDM Sm, Dm, Cm	(Sm) AND (Dm) → (Dm)	Note 1	0028	0-2	(Dm):0	11
Clear	CLR Sw	0000 → (Sw)		04C0	None		6
Exit from Floating Point Interpreter	XIT	None	Note 2	0C0E 0C0F	None		7
Invert	INV Sw	complement(Sw) → (Sw)	Note 3	0540	0-2	(Sw):0	6
Invert Order of Field	IOF Sw, Cp, Ww		Note 4	0E00	None		15
Or Immediate	ORI Wr, I	(Wr) OR I → (Wr)		0260	0-2	(Wr):0	8
Or Multiple Precision	ORM Sm, Dm, Cm	(Sm) OR (Dm) → (Dm)	Note 1	0027	0-2	(Dm):0	11
Set to One	SET0 Sw	FFFF <sub>16</sub> → (Sw)		0700	None		6
Set Ones Corresponding	SOC Sw, Dw		Note 5	E000	0-2	(Dw):0	1
Set Ones Corresponding, Byte	SOCB Sb, Db		Note 5	F000	0-2, 5	(Db):0	1
Set Zeros Corresponding	SZC Sw, Dw		Note 6	4000	0-2	(Dw):0	1

**Table B-6. Logical Instructions (Continued)**

Instruction	Format	Effect	Notes	Opcode	Status Bits Affected	Status Bits Test	Format Number
Set Zeros Corresponding, Byte	SZCB Sb,Db		Note 6	5000	0-2,5	(Db):0	1
Test and Clear Memory Bit	TCMB Sw,Pb	0 → Bit (Sw) + Pb	Note 7	0COA	2		14
Test and Set Memory Bit	TSMB Sw,Pb	1 → Bit (Sw) + Pb	Note 7	0COB	2		14
Exclusive Or	XOR Sw, Wr	(Sw) XOR (Wr) → (Wr)		2800	0-2	(Wr):0	3
Exclusive Or Multiple Precision	XORM Sm,Dm,Cm	(Sm) XOR (Dm) → (Dm)	Note 1	0026	0-2	(Dm):0	11

Note 1: The Cm field specifies the number of bytes of precision of the operands. The precision may also be specified in R0, bits 12 through 15.

Note 2: Used to exit from the floating point interpreter. This instruction is effectively a no-op.

Note 3: The one's complement value is placed in the source operand.

Note 4: The order of the bits in the bit field of width Ww is reversed. Cp indicates the starting position of the field. The bit at Cp is the most significant bit.

Note 5: Set to a logic one the bits in the destination operand that correspond to the logic one bits in the source operand.

Note 6: Set to a logic zero the bits in the destination operand that correspond to the logic one bits in the source operand.

Note 7: Status bit two is set to the value of the memory bit previous to the instruction execution.

Table B-7. Shift Instructions

Instruction	Format	Value Placed in Vacated Bit Position on Each Shift	Notes	Opcode	Format Number
Normalize	NRM Sm,Dw,Cm	Bit one position to the right.	Note 1	0C08	11
Shift Left Arithmetic	SLA Wr,SC	Logic zero.	Note 2	0A00	5
Shift Left Arithmetic Multiple Precision	SLAM Sm,Cm,SC	Logic zero	Note 2	001D	13
Shift Right Arithmetic	SRA Wr,SC	Original value of leftmost bit.		0800	5
Shift Right Arithmetic Multiple Precision	SRAM Sm,Cm,SC	Original value of leftmost bit.	Note 3	001C	13
Shift Right Circular	SRC Wr,SC	Rightmost bit moves to leftmost bit; all other bits move one position to the right.		0B00	5
Shift Right Logical	SRL Wr,SC	Logic zero		0900	5

## General

Note: The shift count can be specified in R0, bits 12 through 15. If these bits are zero, the operand is shifted sixteen bits. The result is compared to zero and status bits 0 through 2 reflect the comparison. The carry bit (bit 3) contains the last bit shifted out of the operand.

Note 1: The source operand is shifted left until the two leftmost bits differ. The number of positions shifted is added to the destination operand. The Cm field specifies the number of bytes of precision. The precision can also be specified in R0, bits 4 through 7. If bits 4 through 7 equal zero, the precision is 16 bytes.

Note 2: The overflow bit is set if the sign bit (MSB) changes from its original value.

Note 3: The Cm field specifies the number of bytes of precision. The precision can also be specified in R0, bits 4 through 7. If bits 4 through 7 equal zero, the precision is 16 bytes.

**Table B-8. Extended Operation Instruction**

Instruction	Format	Effect	Opcode	Status Bits Affected	Status Bits Test	Format Number
Extended Operation	XOP Sw,X	Note 1	2C00	Note 1		9

Note 1: The extended operation instruction executes a software-implemented routine, or executes the writable control store (user-implemented microcode). If status bit 11 is set to zero, execution can be summarized as follows:

Memory Location	X x 4 + 40 <sub>16</sub>	→ (MP)	Sw → R11
"	X x 4 + 42 <sub>16</sub>	→ (PC)	(MP) → R13
"	X x 4 + 44 <sub>16</sub>	→ (PC)	(PC) → R14
"	X x 4 + 46 <sub>16</sub>	→ (PC)	(ST) → R15

Status bit 6 is set to one.

If status bit 11 is set to one, the source address is deposited in a hardware register. Control is then transferred to the writable control store, at the word specified by two times the X field.

**Table B-9. Long Distance Addressing Instructions**

Instruction	Format	Effect	Notes	Opcode	Status Bits Affected	Status Bits Test	Format Number
Long Distance Source	LDS Sa	(Sa) → M2	Note 1	0780	None		6
Long Distance Destination	LDD Sa	(Sa) → M2	Note 2	07C0	None		6

Note 1: Places the contents of six words in memory at the source address into memory map file two, to use for the source address for the following instruction.

Note 2: Places the contents of six words in memory at the source address into memory map file two, to use for the destination address for the following instruction.

**Table B-10. Conversion Instructions**

Instruction	Format	Effect	Notes	Opcode	Status Bits Affected	Status Bits Test	Format Number
Binary to Decimal ASCII Conversion	BDC Sm,Ds,Cm	(Sm) → (Ds)	Note 1	0023	0-2,4	(Dm):0	11
Convert Double Precision Real to Extended Integer	CDE	(FPAq) → (FPAq)	Note 2	0C05	0-4	(FPAq):0	7
Convert Double Precision Real to Integer	CDI	(FPAq) → (FPAq)	Note 2	0C01	0-4	(FPAq):0	7
Convert Extended Integer to Double Precision Real	CED	(FPAd) → (FPAq)	Note 2	0C07	0-4	(FPAq):0	7
Convert Extended Integer to Real	CER	(FPAd) → (FPAd)	Note 2	0C06	0-4	(FPAd):0	7
Convert Integer to Double Precision Real	CID Sw	(Sw) → (FPAq)		0E80	0-4	(FPAd):0	6
Convert Integer to Real	CIR Sw	(Sw) → (FPAd)		0C80	0-4	(FPAd):0	6
Cyclic Redundancy Code Calculation	CRC Sm,Dw,Cm,Wc		Note 3	0E20	2	(Dw):0	12
Convert Real to Extended Integer	CRE	(FPAd) → (FPAd)	Note 2	0C04	0-4	(FPAd):0	7

**Table B-10. Conversion Instructions (Continued)**

Instruction	Format	Effect	Notes	Opcode	Status Bits Affected	Status Bits Test	Format Number
Convert Real to Integer	CRI	(FPAd) → (FPAw)	Note 2	0C00	0-4	(FPAw):0	7
Decimal ASCII to Binary Conversion	DBC Ss,Dm,Cm	(Ss) → (Dm)	Note 1	0024	0-2,4	(Dm): 0	11
Extend Precision	EP Sm,Dm,Cm,Wm	(Sm) → (Dm)	Note 4	03F0	None		21

**Note 1:** The source string is converted from the existing format and placed in the destination string. The Cm field specifies the precision of the binary operand. The precision can also be defined by R0, bits 12 through 15. If bits 12 through 15 equal zero, the precision is sixteen bytes. The decimal operand string length is twice the number of bytes in the binary operand (Cm x 2).

**Note 2:** The floating point accumulator must be loaded with the value to be converted before execution of the instruction.

**Note 3:** The 16-bit CRC partial sum at the destination address is updated by the byte string at the source address. The length of the byte string is specified by the Cm field, by R0, or by the string tag for tagged strings. If the string length is zero, the update does not occur. This instruction is explained fully in Section 3.

**Note 4:** The value at the source address, specified by Cm, is placed right-justified at the destination address, of length Wm. Sign bits are used to fill in the unused bits.

**Table B-11. Pseudo-Instructions**

Instruction	Equivalent Instruction	Opcode
NOP	JMP \$ + 2	1000
RT	B *11	045B
XVEC	DATA,DATA,WPNT	None



# Appendix C

## Program Organization

---

### C.1 PROGRAM AREAS

The three types of areas in a program for the Model 990 Computer are the procedure, the workspace, and the data areas. The procedure area contains the computer instructions. The workspace area contains program linkage, high activity data, and addresses. As many workspaces as convenient may be allocated for a program. Data areas may be allocated as required.

The three previously described hardware registers—WP, PC and ST—control program execution. The workspace pointer contains the address of the first word of a 16-word area of memory called the workspace. The program workspace may be changed by changing the contents of the WP register. The ST contains condition bits set by instructions already performed and the interrupt level mask. These three registers completely control and define the context of a program.

The general environment of the Model 990 Computer is shown in Figure C-1. This arrangement of workspace, procedure, and data is the simplest approach to 990 programming. Though many application programs may be written in this manner, a more segregated approach with possibly several workspaces, data areas, and connected procedures would provide increased flexibility and applicability.

The programs execute in the environment provided by the executives. The areas may be combined in a single task, or the workspace and data areas may be combined in the data division of the task. The procedure area becomes the procedure division of the task in that case. The executives support writing a procedure division to be used with several data divisions to form tasks that perform the same functions on several sets of data. Refer to the applicable operating system reference manual for information about the environment the operating system provides for user programs.

### C.2 PROCEDURE

A procedure is the main body of a program and contains computer instructions. It is the action part of a program. Procedures could be coded to solve an equation, run a motor, determine status of a process, or condition a set of data that is to be processed by another procedure. Procedures in the Model 990 Computer may have workspaces and data as an integral part of the coding or may use workspaces and data passed from another procedure.

### C.3 WORKSPACE

The Model 990 Computer uses workspaces that may be anywhere in memory and that consist of 16 consecutive memory words. A context switch due to an interrupt, an XOP instruction, or a BLWP instruction changes the active workspace. A return from the subroutine provided for either of these context switches using an RTWP instruction restores the original workspace. Execution of an LWPI instruction makes a specified workspace active without changing the PC contents. When the data division is separate from the procedure division, any workspace that contains data that is unique to the task represented by the data division should be a part of the data division.

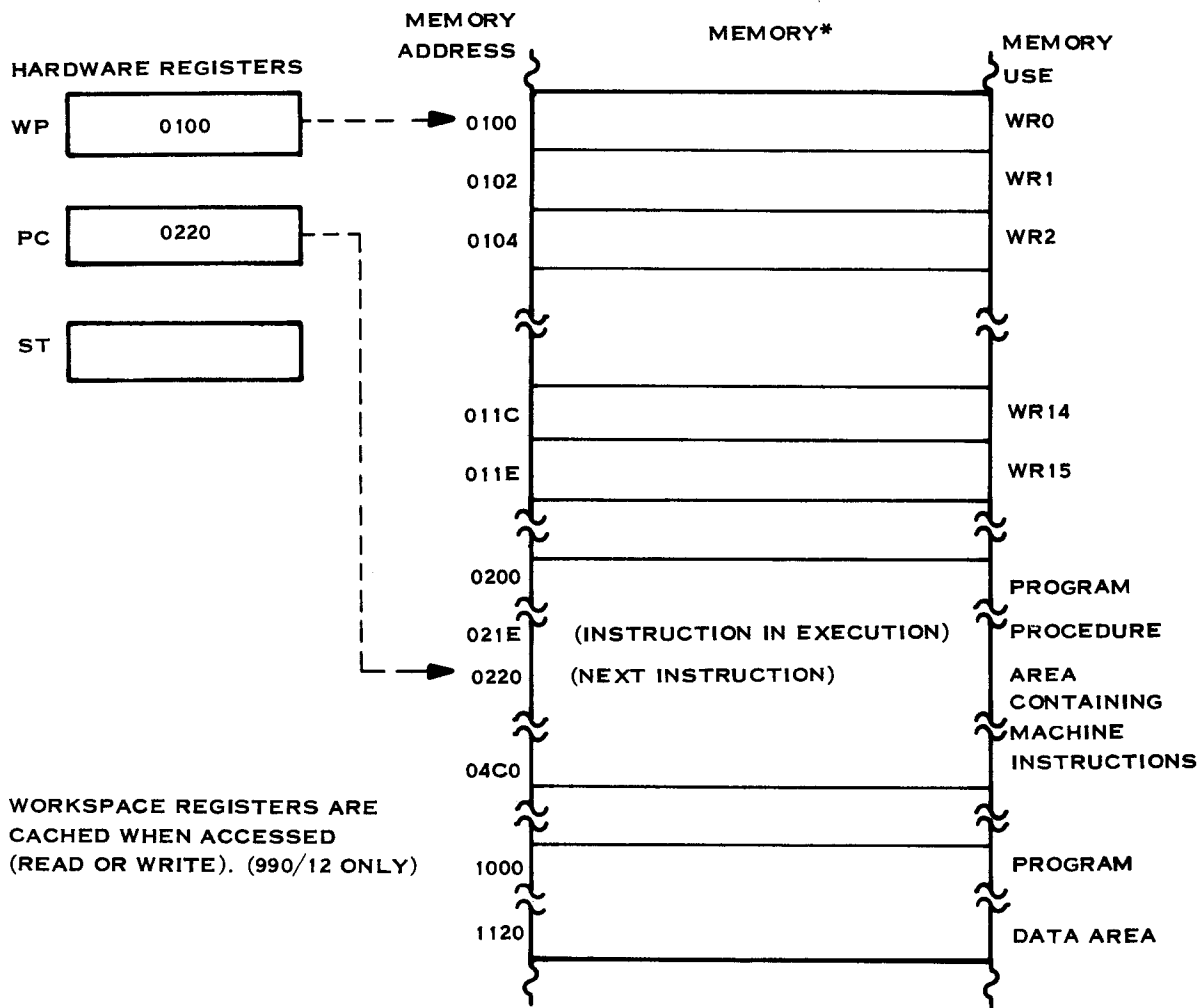


Figure C-1. Model 990 Computer Programming Environment

## C.4 DATA

Data for a procedure may appear in many forms. In assembly, there are three directives available to the programmer to initialize data within a program module. These directives are as follows:

- DATA — Initializes one or more consecutive words of memory to specific values that are input on this statement.
- BYTE — Initializes one or more consecutive bytes of memory as does the DATA statement, except that bytes are initialized.
- TEXT — Initializes a textual string of characters in consecutive bytes of memory. The characters are represented in USASCII code.

Also, data input from the data terminal or device attached to the CRU or TILINE is available to procedures in the 990 computer.

The executives for the Model 990 Computer support the user programs by executing supervisor calls to perform input and output operations, data conversions, and other functions. The user provides data in required formats for supervisor call blocks that define the supervisor call, and for other data blocks as appropriate. The assembler directives described previously may be used to provide this data. Details of the data requirements for interface with the executives are described in the operating system reference manuals.

# Appendix D

## Hexadecimal Instruction Table

---

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Paragraph
001C	SRAM	Shift Right Arithmetic Multiple Precision	13	3.124
001D	SLAM	Shift Left Arithmetic Multiple Precision	13	3.115
001E	RTO	Right Test for Ones	11	3.105
001F	LTO	Left Test for Ones	11	3.83
0020	CNT0	Count Ones	11	3.34
0021	SLSL	Search List Logical Address	20	3.116
0022	SLSP	Search List Physical Address	20	3.117
0023	BDC	Binary to Decimal ASCII Conversion	11	3.17
0024	DBC	Decimal to Binary ASCII Conversion	11	3.41
0025	SWPM	Swap Multiple Precision	11	3.135
0026	XORM	Exclusive OR Multiple Precision	11	3.148
0027	ORM	OR Multiple Precision	11	3.101
0028	ANDM	AND Multiple Precision	11	3.13
0029	SM	Subtract Multiple Precision Integer	11	3.118
002A	AM	Add Multiple Precision Integer	11	3.11
002B	MOVA	Move Address	19	3.88

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Paragraph
002D	EMD	Execute Micro-Diagnostic	7	3.50
002E	EINT	Enable Interrupts	7	3.49
002F	DINT	Disable Interrupts	7	3.45
0030	STPC	Store Program Counter	18	3.130
0040	CS	Compare Strings	12	3.39
0050	SEQB	Search String for Equal Byte	12	3.112
0060	MOVS	Move String	12	3.90
0070	LIM	Load Interrupt Mask	18	3.77
0080	LST	Load Status Register	18	3.82
0090	LWP	Load Workspace Pointer	18	3.84
00A0	LCS	Load Writable Control Store	18	3.71
00B0	BLSK	Branch Immediate and Push Link to Stack	8	3.20
00C0	MVSR	Move String Reverse	12	3.95
00D0	MVSK	Move String from Stack	12	3.94
00E0	POPS	Pop String from Stack	12	3.102
00F0	PSHS	Push String to Stack	12	3.103
0140	BIND	Branch Indirect	6	3.18
0180	DIVS	Divide Signed	6	3.47
01C0	MPYS	Multiply Signed	6	3.92
0200	LI	Load Immediate	8	3.76
0220	AI	Add Immediate	8	3.10
0240	ANDI	AND Immediate	8	3.12
0260	ORI	OR Immediate	8	3.100

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Paragraph
0280	CI	Compare Immediate	8	3.28
02A0	STWP	Store Workspace Pointer	18	3.135
02C0	STST	Store Status	18	3.132
02E0	LWPI	Load Workspace Pointer Immediate	8	3.85
0300	LIMI	Load Interrupt Mask Immediate	8	3.78
0320	LMF	Load Memory Map File	10	3.79
0340	IDLE	Idle	7	3.52
0360	RSET	Reset	7	3.104
0380	RTWP	Return with Workspace Pointer	7	3.106
03A0	CKON	Clock On	7	3.32
03C0	CKOF	Clock Off	7	3.31
03E0	LREX	Load or Restart Execution	7	3.81
03F0	EP	Extended Precision	21	3.51
0400	BLWP	Branch and Load Workspace Pointer	6	3.21
0440	B	Branch	6	3.16
0480	X	Execute	6	3.143
04C0	CLR	Clear	6	3.33
0500	NEG	Negate	6	3.96
0540	INV	Invert	6	3.56
0580	INC	Increment	6	3.53
05C0	INCT	Increment by Two	6	3.54
0600	DEC	Decrement	6	3.43

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Paragraph
0640	DECT	Decrement by Two	6	3.44
0680	BL	Branch and Link	6	3.19
06C0	SWPB	Swap Bytes	6	3.134
0700	SETO	Set to One	6	3.113
0740	ABS	Absolute Value	6	3.8
0780	LDS	Long Distance Source	6	3.75
07C0	LDD	Long Distance Destination	6	3.74
0800	SRA	Shift Right Arithmetic	5	3.123
0900	SRL	Shift Right Logical	5	3.127
0A00	SLA	Shift Left Arithmetic	5	3.114
0B00	SRC	Shift Right Circular	5	3.125
0C00	CRI	Convert Real to Integer	7	3.38
0C01	CDI	Convert Double Precision Real to Integer	7	3.25
0C02	NEGR	Negate Real	7	3.98
0C03	NEGD	Negate Double Precision Real	7	3.97
0C04	CRE	Convert Real to Extended Integer	7	3.37
0C05	CDE	Convert Double Precision Real to Extended Integer	7	3.24
0C06	CER	Convert Extended Integer to Real	7	3.27
0C07	CED	Convert Extended Integer to Double Precision Real	7	3.26
0C08	NRM	Normalize	11	3.99
0C09	TMB	Test Memory Bit	14	3.140
0C0A	TCMB	Test and Clear Memory Bit	14	3.139

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Paragraph
0C0B	TSMB	Test and Set Memory Bit	14	3.142
0C0C	SRJ	Subtract from Register and Jump	17	3.126
0C0D	ARJ	Add to Register and Jump	17	3.15
0C0E/ 0C0F	XIT	Exit from Floating Point Interpreter	7	3.145
0C10	INSF	Insert Field	16	3.55
0C20	XV	Extract Value	16	3.149
0C30	XF	Extract Field	16	3.144
0C40	AR	Add Real	6	3.14
0C80	CIR	Convert Integer to Real	6	3.30
0CC0	SR	Subtract Real	6	3.122
0D00	MR	Multiply Real	6	3.93
0D40	DR	Divide Real	6	3.48
0D80	LR	Load Real	6	3.80
0DC0	STR	Store Real	6	3.131
0E00	IOF	Invert Order of Field	15	3.57
0E10	SNEB	Search String for Not Equal Byte	12	3.119
0E20	CRC	Cyclic Redundancy Code Calculation	12	3.36
0E30	TS	Translate Strings	12	3.141
0E40	AD	Add Double Precision Real	6	3.9
0E80	CID	Convert Integer to Double Precision Real	6	3.29
0EC0	SD	Subtract Double Precision Real	6	3.111



Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Paragraph
0F00	MD	Multiply Double Precision Real	6	3.86
0F40	DD	Divide Double Precision Real	6	3.42
0F80	LD	Load Double Precision Real	6	3.72
0FC0	STD	Store Double Precision Real	6	3.129
1000	JMP	Unconditional Jump	2	3.65
1100	JLT	Jump if Less Than	2	3.64
1200	JLE	Jump if Low or Equal	2	3.63
1300	JEQ	Jump if Equal	2	3.58
1400	JHE	Jump if High Or Equal	2	3.61
1500	JGT	Jump if Greater Than	2	3.59
1600	JNE	Jump if Not Equal	2	3.67
1700	JNC	Jump if No Carry	2	3.66
1800	JOC	Jump on Carry	2	3.69
1900	JNO	Jump if No Overflow	2	3.68
1A00	JL	Jump if Logical Low	2	3.62
1B00	JH	Jump if Logical High	2	3.60
1C00	JOP	Jump if Odd Parity	2	3.70
1D00	SBO	Set CRU Bit to Logic One	2	3.109
1E00	SBZ	Set CRU Bit to Logic Zero	2	3.110
1F00	TB	Test Bit	2	3.138
2000	COC	Compare Ones Corresponding	3	3.35

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Paragraph
2400	CZC	Compare Zeros Corresponding	3	3.40
2800	XOR	Exclusive OR	3	3.147
2C00	XOP	Extended Operation	9	3.146
3000	LDCR	Load Communication Register	4	3.73
3400	STCR	Store Communication Register	4	3.128
3800	MPY	Multiply	9	3.91
3C00	DIV	Divide	9	3.46
4000	SZC	Set Zeros Corresponding	1	3.136
5000	SZCB	Set Zeros Corresponding, Byte	1	3.137
6000	S	Subtract Words	1	3.107
7000	SB	Subtract Bytes	1	3.108
8000	C	Compare Words	1	3.22
9000	CB	Compare Bytes	1	3.23
A000	A	Add Words	1	3.6
B000	AB	Add Bytes	1	3.7
C000	MOV	Move Word	1	3.87
D000	MOVB	Move Byte	1	3.89
E000	SOC	Set Ones Corresponding	1	3.120
F000	SOCB	Set Ones Corresponding, Byte	1	3.121

# Appendix E

## Assembler Directive Table

---

The assembler directives for the Model 990 Computer assembly language are listed in Table E-1. All directives may include a comment field following the operand field. Those directives that do not require an operand field may have a comment field following the operator field. Those directives that have optional operand fields (RORG) and (END) may have comment fields only when they have operand fields.

The following symbols and conventions are used in defining the syntax of assembler directives:

- Angle brackets (< >) enclose items supplied by the user.
- Brackets ([ ]) enclose optional items.
- An ellipsis (...) indicates that the preceding item may be repeated.
- Braces ({} ) enclose two or more items of which one must be chosen.

The following words are used in defining the items used in assembler directives:

- symbol.
- string — a character string of a length defined for each directive.
- exp — an expression.
- wd-exp — well-defined expression.
- wa — absolute expression in the range from 0 to 15.
- operation — mnemonic operation code, macro name, or previously defined operation or extended operation.

Table E-1. Assembler Directives

Directive	Syntax	Free Word Boundary	Note
Output Options	<code>b...OPTIONsb...&lt;keyword&gt;[,&lt;keyword&gt;] ...b...[&lt;comment&gt;]</code>	NA	1
Page Title	<code>[&lt;label&gt;]b...TITLb...&lt;string&gt;'b...[&lt;comment&gt;]</code>	NA	
Program Identifier	<code>[&lt;label&gt;]b...IDTb...&lt;string&gt;'b...[&lt;comment&gt;]</code>	NA	
Copy Source File	<code>[&lt;label&gt;]b...COPYb...&lt;file name&gt;b...[&lt;comment&gt;]</code>	NA	
External Definition	<code>[&lt;label&gt;]b...DEFb...&lt;symbol&gt;[,&lt;symbol&gt;] ...b...[&lt;comment&gt;]</code>	NA	
External Reference	<code>[&lt;label&gt;]b...REFb...&lt;symbol&gt;[,&lt;symbol&gt;] ...b...[&lt;comment&gt;]</code>	NA	
Secondary Reference	<code>[&lt;label&gt;]b...SREFb...&lt;symbol&gt;[,&lt;symbol&gt;] ...b...[&lt;comment&gt;]</code>	NA	
Force Load	<code>[&lt;label&gt;]b...LOADb...&lt;symbol&gt;[,&lt;symbol&gt;] ...b...[&lt;comment&gt;]</code>	NA	
Absolute Origin	<code>[&lt;label&gt;]b...AORgb...&lt;wd-exp&gt;b...[&lt;comment&gt;]</code>	No	3
Relocatable Origin	<code>[&lt;label&gt;]b...RORgb...&lt;exp&gt;b...[&lt;comment&gt;]</code>	No	2, 3
Dummy Origin	<code>[&lt;label&gt;]b...DORgb...&lt;exp&gt;b...[&lt;comment&gt;]</code>	No	2
Workspace Pointer	<code>[&lt;label&gt;]b...WPNTb...&lt;exp&gt;b...[&lt;comment&gt;]</code>	NA	
Block Starting With Symbol	<code>[&lt;label&gt;]b...BSSb...&lt;wd-exp&gt;b...[&lt;comment&gt;]</code>	No	
Block Ending With Symbol	<code>[&lt;label&gt;]b...BESb...&lt;wd-exp&gt;b...[&lt;comment&gt;]</code>	No	
Initialize Word	<code>[&lt;label&gt;]b...DATAb...&lt;exp&gt;[,&lt;exp&gt;] ...b...[&lt;comment&gt;]</code>	Yes	
Initialize Text	<code>[&lt;label&gt;]b...TEXTb...[']&lt;string&gt;'b...[&lt;comment&gt;]</code>	NA	4
Define Checkpoint Register	<code>[&lt;label&gt;]b...CKPTb...&lt;wa&gt;b...[&lt;comment&gt;]</code>	No	
Define Extended Operation	<code>[&lt;label&gt;]b...DXOPb...&lt;symbol&gt;[,&lt;wd-exp&gt;b...[&lt;comment&gt;]</code>	NA	
Define Operation	<code>[&lt;label&gt;]b...DFOPb...&lt;symbol&gt;[,&lt;operation&gt;b...[&lt;comment&gt;]</code>	NA	
Define Assembly-Time Constant	<code>[&lt;label&gt;]b...EQUb...&lt;exp&gt;b...[&lt;comment&gt;]</code>	NA	
Word Boundary	<code>[&lt;label&gt;]b...EVENb...[&lt;comment&gt;]</code>	Yes	
No Source List	<code>[&lt;label&gt;]b...UNLb...[&lt;comment&gt;]</code>	NA	
List Source	<code>[&lt;label&gt;]b...LISTb...[&lt;comment&gt;]</code>	NA	
Page Eject	<code>[&lt;label&gt;]b...PAGEb...[&lt;comment&gt;]</code>	NA	
Initialize Byte	<code>[&lt;label&gt;]b...BYTEb...&lt;exp&gt;[,&lt;exp&gt;] ...b...[&lt;comment&gt;]</code>	No	
Program End	<code>[&lt;label&gt;]b...ENDb...[&lt;symbol&gt;]b...[&lt;comment&gt;]</code>	NA	3, 5
Program Segment	<code>[&lt;label&gt;]b...PSEGb...[&lt;comment&gt;]</code>	No	
Program Segment End	<code>[&lt;label&gt;]b...PENDb...[&lt;comment&gt;]</code>	No	
Data Segment	<code>[&lt;label&gt;]b...DSEGb...[&lt;comment&gt;]</code>	No	
Data Segment End	<code>[&lt;label&gt;]b...DENDb...[&lt;comment&gt;]</code>	No	
Common Segment	<code>[&lt;label&gt;]b...CSEGb...[']&lt;string&gt;'b...[&lt;comment&gt;]</code>	No	3
Common Segment End	<code>[&lt;label&gt;]b...CENDb...[&lt;comment&gt;]</code>	No	
Assemble If	<code>[&lt;label&gt;]b...ASMIFb...&lt;wd-exp&gt;b...[&lt;comment&gt;]</code>	NA	
Assemble Else	<code>[&lt;label&gt;]b...ASMELSb...[&lt;comment&gt;]</code>	NA	
Assemble End	<code>[&lt;label&gt;]b...ASMENDb...[&lt;comment&gt;]</code>	NA	
Set Macro Nesting Level	<code>[&lt;label&gt;]b...SETMNLb...&lt;wd-exp&gt;</code>	NA	
Set Right Margin	<code>[&lt;label&gt;]b...SETRMb...&lt;wd-exp&gt;</code>	NA	

**Notes:**

1. One of the <keyword>s must be "12" to specify the 990/12 instruction set. Keywords supported by SDSMAC are XREF, RXREF, OBJ, SYMT, NOLIST, TUNLST, DUNLST, BUNLST, MUNLST, and FUNL.
2. Any symbols in the expression must have been previously defined.
3. The comment field may be used only when the operand field is used.
4. The minus sign causes the assembler to negate the rightmost character.
5. The symbol must be previously defined.

# Appendix F

## Macro Language Table

---

The syntax of the statements that contain the Macro Language verbs is shown in the following table.

Statement	Syntax
Macro	<macro name>\$.MACRO\$. [<parm>][<parm>]...\$. [<comment>]*
Variable	\$.VAR\$. <var>[, <var>]...\$. [<comment>]
Assign	\$.ASG\$. <expression>TO\$. <var>\$. [<comment>] <string>
Name	<label>\$.NAME\$. [<comment>]
Go to	\$.GOTO\$. <label>\$. [<comment>]
Exit	\$.EXIT\$. [<comment>]
Call	\$.CALL\$. <macro name>\$. [<comment>]
If	\$.IF\$. <expression>\$. [<comment>]
Else	\$.ELSE\$. [<comment>]
End if	\$.ENDIF\$. [<comment>]
End	[<label>]\$.END\$. [<comment>]
Substring	\$.SBSTG\$. <var>, <expr>, <expr>TO\$. <var>\$. [<comment>]

### Macro Variable Components

Qualifier	Meaning
S	The string component of the variable.
A	The attribute component of the variable.
V	The value component of the variable.
L	The length component of the variable.

### Symbol Components

Qualifier	Meaning
SS	String component of a linked symbol.
SV	Value component of a linked symbol.
SA	Attribute component of a symbol.
SL	Length component of a symbol.
SU	User attribute component of a linked symbol.
SG	Segment component of a linked symbol.

#### Note:

The comment field on the \$MACRO statement may be used only if at least one parameter is specified.

# Appendix G

## Example Program

---

### G.1 EXAMPLE PROGRAM

The source program shown in Figure G-1 is an example of a source program written on coding forms from which the source programs are prepared.

You can either transfer the information from the coding form, or create your source program directly on the system with the text editor. Once the source program is entered, assemble it using the Execute Macro Assembler (XMA) command.

### G.2 XMA (EXECUTE MACRO ASSEMBLER)

The XMA command assembles the source code of an assembly language program. You can assemble modules of the program separately and later link them together, or you can include all of the code in one module:

When the assembler completes, the following message appears:

```
MACRO ASSEMBLY COMPLETE, nnnn ERRORS, nnnn WARNINGS
```

where:

nnnn is the number of errors or warnings.

If the assembler detects any errors, you must reenter the text editor and correct your source file. When all the errors are corrected, reassemble the program.

#### G.2.1 XMA Command Format

```
SOURCE ACCESS NAME: <acnm>@
OBJECT ACCESS NAME: [acnm]@
LISTING ACCESS NAME: <acnm>@
ERROR ACCESS NAME: [acnm]@
                    OPTIONS: [string]@
MACRO LIBRARY PATHNAME: [acnm]@
                    PRINT WIDTH: <int>@
                    PAGE LENGTH: <int>@
```

**G.2.2 XMA User Responses**

<b>System Prompts</b>	<b>Response Required or Optional</b>	<b>User Responses</b>
SOURCE ACCESS NAME:	R	The pathname of the file that contains the source program to be assembled.
OBJECT ACCESS NAME:	O	The pathname in which you want the assembler to place the assembled output.
LISTING ACCESS NAME:	R	The pathname of the file for the results of the assembling operation, showing the program line-by-line, the address references, and any errors that occur.
ERROR ACCESS NAME:	O	The pathname of the file in which the error references are written.
OPTIONS:	O	See OPTIONS in the following prompt descriptions.
MACRO LIBRARY PATHNAME:	O	
PRINT WIDTH:	R	
PAGE LENGTH:	R	

**SOURCE ACCESS NAME**

The input file or device containing assembly language code to be assembled.

**OBJECT ACCESS NAME**

The object code output file or device. If this parameter is null, no object output is produced. This is useful for preliminary assemblies to check for errors; since the assembler produces no output, it operates faster.

**LISTING ACCESS NAME**

The assembly listing file or device. If DUMMY is entered, no assembly listing is produced.

**ERROR ACCESS NAME**

The assembly error output. This field may be viewed by entering the Show File (SF) command. If the ERROR ACCESS NAME is null, or if it is the same as the listing file, then errors are displayed on the terminal by the Show Background Status (SBS) command. If the device DUMMY is specified, no error listing is produced.

The error file contains a complete list of any source records which cause assembly errors along with the other errors. If a condition is sensed which prevents the assembler from continuing, a message is written to the error file stating what has occurred. Then the user must enter the SBS command to view the error messages output by the assembler. Appendix A contains a list of abnormal completion messages and possible causes.

## OPTIONS

Output and list options for the assembler. The user specifies any (or all) of the following options, separated by commas.

**XREF** — Prints a cross-reference listing at the end of the source and object listing file.

**RXREF** — Prints a reduced cross-reference listing at the end of the source and object listing file.

**SYMT** — Includes a symbol table with the output object code. This option must be specified to allow fully symbolic debugging.

**TUNLST** — Limits the listing for TEXT directives to a single line.

**BUNLST** — Limits the listing for BYTE directives to a single line.

**DUNLST** — Limits the listing for DATA directives to a single line.

**MUNLST** — Limits the listing for a macro expansion to a single line. TEXT, BYTE, and DATA statements and macro usage often expand to produce multiple lines of code. If these options are selected, the statements appear in the listing but the expansion does not. For example, the source statement TEXT 'ABCDEF' produces the listing:

```
41 TEXT 'ABCDEF'
42
43
44
45
46
```

With the TUNLST option specified, only the line 41 TEXT 'ABCDEF' is produced in the listing.

**FUNL** — Overrides the unlist directives.

**NOLIST** — Suppresses all listing output, except to the error file. Overrides other directives and keywords.

**12** — You must specify the 12 option if you intend to use instructions applicable to either processor set B or C, otherwise the macro assembler defaults to the instruction set for processor set A. See Table 3-2 for the applicability of each instruction.



Any of the OPTION keywords may be abbreviated; for example, any of the following may be used for the TUNLST option:

T  
TU  
TUN  
TUNL  
TUNLS  
TUNLST

To select more than one option, enter a list of keywords separated by commas. The keywords may appear in any order. To select all the options, one could enter the line:

OPTIONS: X,S,T,B,D,M

The options specified for this parameter are in addition to any options specified by OPTION directives in the source.

#### MACRO LIBRARY PATHNAME

A directory containing macro definitions for this assembly. This pathname specification is equivalent to specifying the same pathname in a LIBIN directive, except that this pathname becomes the system macro library and is retained through stacked assemblies. This pathname is printed on the cover sheet of the first module only.

#### PRINT WIDTH (CHARS)

Specifies the length of the lines to be written to the output file.

#### PAGE LENGTH (LINES)

Specifies the number of lines per page in the listing file.

Figure G-2 shows the listing produced by the assembler. The message printed as the program is executed is shown in Figure G-3. The program executes on a 990 computer with a 733 ASR and assumes that the CRU base address for the 733 ASR is 000<sub>16</sub>.



ASSEMBLY LANGUAGE CODING FORM												
LABEL	OPER	OPERAND					COMMENTS					
1	8	11	13	20	25	31	35	40	45	50	55	60
	MOV B		* 2 + , 8		GET A		CHARACTER.					
	JLT		LAST		LAST		CHARACTER?					
	BL		@PUTC		NO,		PUT IT OUT					
	JMP		LOOP									
LAST												
	BL		@PUTC		PUT IT OUT.							
	IDLE				STOP.							
*												
* OUTPUT ROUTINE.												
PUTC												
	TB		ASRID		CHECK		ASRID.					
	JEQ		OUT		GO		FOR TTY.					
PROGRAM												
					PROGRAMMED BY						PAGE	OF

228 3244 (2/4)

Figure G-1. Example Program (Part 2 of 4)

ASSEMBLY LANGUAGE CODING FORM

1	6	8	11	13	OPERAND					COMMENTS									
					20	25	31	35	40	45	50	55	60						
		MOV		11,5						SAVE	RETURN								
		BL		@OUT						SEND	CHARACTER								
		SBZ		RTS						TIMING	FOR ASR733								
		BL		@OUT															
		BL		@OUT															
		BL		@OUT															
		SBO		RTS															
		B		*5						RETURN	TO CALLER								
		OUT																	
		ONLINE	TB	DSR						ASR	ONLINE?								
			JNE	ONLINE						WAIT	UNTIL IT IS								
			LDCR	8,8						OUTPUT	CHARACTER								
PROGRAM				PROGRAMMED BY				CHARGE				PAGE				OF			

228 3244 (3/4)

Figure G-1. Example Program (Part 3 of 4)



HELLO PROGRAM

PAGE 0002

```

0002 0000
0003          * THIS PROGRAM PRINTS 'HELLO!' ON THE TELEPRINTER.
0004 0000
0005      0009 DTR   EQU   9           DATA TERMINAL READY.
0006 0000
0007      000B WRQ   EQU   11          WRITE REQUEST.
0008 0000
0009      000A RTS   EQU   10          REQUEST TO SEND.
0010 0000
0011      000A ASRID EQU   10          ASR733/33 ID.
0012      000E DSR   EQU   14          DATA SET READY.
0013          *
0014 0000
0015 0000      HELLO
0016 0000
0017 0000 02E0          LWPI REGS      INITIALIZE WORKSPACE POINTER.
0018 0004          0002 0054
0019 0004 0300          LIMI 0          DISABLE INTERRUPTS.
0020 0008          0006 0000
0021 0008 020C          LI    12,0      INITIALIZE CRU BASE.
0022 000C          000A 0000
0023 000C 0202          LI    2, TABLE LOAD TABLE ADDRESS
0024 0010 1D09          000E 004E
0025 0012 1D0A LOOP    SBO   DTR
0026 0014          SBO   RTS
0027 0014 D232          MOVB *2+,8      GET A CHARACTER.
0028 0016
0029 0016 1103          JLT   LAST      LAST CHARACTER?
0030 0018
0031 0018 06A0          BL    @PUTC     NO, PUT IT OUT.
0032 001C          001A 0024
0033 001C 10FA          JMP   LOOP
0034 001E
0035 001E          LAST
0036 001E
0037 001E 06A0          BL    @PUTC     PUT IT OUT.
0038 0022          0020 0024
0039 0022 0340          IDLE      STOP.
0040 0024
0041          *
0042 0024          * OUTPUT ROUTINE.
0043          *
0044 0024          PUTC
0045 0024
0046 0024
0047 0024 1FOA          TB    ASRID     CHECK ASR ID.
0048 0026
0049 0026 130C          JEQ   OUT      GO FOR TTY.
0050 0028
0051 0028 C14B          MOV   11,5     SAVE RETURN.
0052 002A
0053 002A 06A0          BL    @OUT     SEND CHARACTER.
0054 002E          002C 0040

```

(A)143977 (1/2)

Figure G-2. Assembly Listing of Example Program (Part 1 of 2)

```

HELLO PROGRAM
PAGE 0003

0055 002E 1E0A SBZ RTS TIMING FOR ASR733
0056 0030
0057 0030 06A0 BL @OUT
0032 0040'
0058 0034
0059 0034 06A0 BL @OUT
0036 0040'
0060 0038
0061 0038 06A0 BL @OUT
003A 0040'
0062 003C
0063 003C 1D0A SBO RTS
0064 003E
0065 003E 0455 B *5 RETURN TO CALLER.
0066 0040
0067 *
0068 0040
0069 0040 GUT
0070 0040
0071 0040 1F0E ONLINE TB DSR ASR ON LINE?
0072 0042
0073 0042 16FE JNE ONLINE WAIT UNTIL IT IS.
0074 0044
0075 0044 3208 LDCR 8,8 OUTPUT CHARACTER.
0076 0046
0077 *
0078 0046
0079 0046 1F0B WAIT TB WRQ WAIT ON IT.
0080 0048
0081 0048 16FE JNE WAIT
0082 004A
0083 004A 1D0B SBO WRQ
0084 004C
0085 004C 045B B *11 RETURN TO CALLER.
0086 004E
0087 *
0088 004E
0089 * MESSAGE TABLE.
0090 004E
0091 004E TABLE
0092 004E
0093 004E 48 TEXT 'HELLO'
004F 45
0050 4C
0051 4C
0052 4F
0094 0053
0095 0053 A1 BYTE '!'+>80 SET PARITY BIT.
0096 0054
0097 0054 EVEN
0098 0054
0099 0054 REGS BSS 32 WORKSPACE AREA.
0100 0074
0101 0000' END HELLO
NO ERRORS, NO WARNINGS

```

(A)143977 (2/2)

Figure G-2. Assembly Listing of Example Program (Part 2 of 2)

**HELLO!**

**Figure G-3. Example Program Message**



# Appendix H

## Numerical Tables

---

Table H-1 provides hexadecimal arithmetic. To use the addition section of the table, find one integer to be added in the first row, find the other integer to be added in the first column, and locate the sum where the row and column intersect. For example, to add  $F_{16}$  to  $9_{16}$ , find the 9 across the top and the F down the first column, and note that they intersect at 18. The multiplication section works the same way.

Table H-2 shows the powers of 16 converted to decimal. To find the value of  $16^9$  in decimal, go down the column labeled N until you find the 9 and note the values 68, 719, 476, 736 at the left. Table H-3 shows the powers of 10 in hexadecimal and works the same way as Table H-2.

The remaining tables serve the following functions: Table H-4 shows the powers of 2 in decimal, Table H-5 presents hexadecimal-decimal integer conversion, and Table H-6 provides hexadecimal-fraction conversions.

**Table H-1. Hexadecimal Arithmetic**

**ADDITION TABLE**

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10
2	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
3	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12
4	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
5	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14
6	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15
7	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16
8	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17
9	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18
A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19
B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A
C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

**MULTIPLICATION TABLE**

1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E
3	06	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D
4	08	0C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0A	0F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

**Table H-2. Table of Powers of 16<sub>16</sub>**

16 <sup>n</sup>				n	16 <sup>-n</sup>							
			1	0	0.10000	00000	00000	00000	x 10			
			16	1	0.62500	00000	00000	00000	x 10 <sup>-1</sup>			
			256	2	0.39062	50000	00000	00000	x 10 <sup>-2</sup>			
		4	096	3	0.24414	06250	00000	00000	x 10 <sup>-3</sup>			
		65	536	4	0.15258	78906	25000	00000	x 10 <sup>-4</sup>			
	1	048	576	5	0.95367	43164	06250	00000	x 10 <sup>-6</sup>			
	16	777	216	6	0.59604	64477	53906	25000	x 10 <sup>-7</sup>			
	268	435	456	7	0.37252	90298	46191	40625	x 10 <sup>-8</sup>			
	4	294	967	296	8	0.23283	06436	53869	62891	x 10 <sup>-9</sup>		
	68	719	476	736	9	0.14551	91522	83668	51807	x 10 <sup>-10</sup>		
	1	099	511	627	776	10	0.90949	47017	72928	23792	x 10 <sup>-12</sup>	
	17	592	186	044	416	11	0.56843	41886	08080	14870	x 10 <sup>-13</sup>	
	281	474	976	710	656	12	0.35527	13678	80050	09294	x 10 <sup>-14</sup>	
	4	503	599	627	370	496	13	0.22204	46049	25031	30808	x 10 <sup>-15</sup>
	72	057	594	037	927	936	14	0.13877	78780	78144	56755	x 10 <sup>-16</sup>
1	152	921	504	606	846	976	15	0.86736	17379	88403	54721	x 10 <sup>-18</sup>

**Table H-3. Table of Powers of 10<sub>16</sub>**

10 <sup>n</sup>				n	10 <sup>-n</sup>					
			1	0	1.0000	0000	0000	0000		
			A	1	0.1999	9999	9999	999A		
			64	2	0.28F5	C28F	5C28	F5C3	x 16 <sup>-1</sup>	
			3E8	3	0.4189	374B	C6A7	EF9E	x 16 <sup>-2</sup>	
			2710	4	0.68DB	8BAC	710C	B296	x 16 <sup>-3</sup>	
		1	86A0	5	0.A7C5	AC47	1B47	8423	x 16 <sup>-4</sup>	
		F	4240	6	0.10C6	F7A0	B5ED	8D37	x 16 <sup>-4</sup>	
		98	9680	7	0.1AD7	F29A	BCAF	4858	x 16 <sup>-5</sup>	
		5F5	E100	8	0.2AF3	1DC4	6118	73BF	x 16 <sup>-6</sup>	
		3B9A	CA00	9	0.44B8	2FA0	9B5A	52CC	x 16 <sup>-7</sup>	
	2	540B	E400	10	0.6DF3	7F67	5EF6	EADF	x 16 <sup>-8</sup>	
	17	4876	E800	11	0.AFEB	FF0B	CB24	AAFF	x 16 <sup>-9</sup>	
	E8	D4A5	1000	12	0.1197	9981	2DEA	1119	x 16 <sup>-9</sup>	
	918	4E72	A000	13	0.1C25	C268	4976	81C2	x 16 <sup>-10</sup>	
	5AF3	107A	4000	14	0.2D09	370D	4257	3604	x 16 <sup>-11</sup>	
	3	8D7E	A4C6	8000	15	0.480E	BE7B	9D58	566D	x 16 <sup>-12</sup>
	23	86F2	6FC1	0000	16	0.734A	CA5F	6226	F0AE	x 16 <sup>-13</sup>
	163	4578	5D8A	0000	17	0.B877	AA32	36A4	B449	x 16 <sup>-14</sup>
	DE0	B6B3	A764	0000	18	0.1272	5DD1	D243	ABA1	x 16 <sup>-14</sup>
	8AC7	2304	89E8	0000	19	0.1D83	C94F	B6D2	AC35	x 16 <sup>-15</sup>



The table appearing on the following pages provides a means for direct conversion of decimal integers in the range of 0 to 4096 and for hexadecimal integers in the range of 0 to FFF.

To convert numbers above those ranges, add table values to the figures below:

**Table H-5. Hexadecimal-Decimal Integer Conversion Table**

<b>Hexadecimal</b>	<b>Decimal</b>	<b>Hexadecimal</b>	<b>Decimal</b>
01 000	4 096	20 000	131 072
02 000	8 192	30 000	196 608
03 000	12 288	40 000	262 144
04 000	16 384	50 000	327 680
05 000	20 480	60 000	393 216
06 000	24 576	70 000	458 752
07 000	28 672	80 000	524 288
08 000	32 768	90 000	589 824
09 000	36 864	A0 000	655 360
0A 000	40 960	B0 000	720 896
0B 000	45 056	C0 000	786 432
0C 000	49 152	D0 000	851 968
0D 000	53 248	E0 000	917 504
0E 000	57 344	F0 000	983 040
0F 000	61 440	100 000	1 048 576
10 000	65 536	200 000	2 097 152
11 000	69 632	300 000	3 145 728
12 000	73 728	400 000	4 194 304
13 000	77 824	500 000	5 242 880
14 000	81 920	600 000	6 291 456
15 000	86 016	700 000	7 340 032
16 000	90 112	800 000	8 388 608
17 000	94 208	900 000	9 437 184
18 000	98 304	A00 000	10 485 760
19 000	102 400	B00 000	11 534 336
1A 000	106 496	C00 000	12 582 912
1B 000	110 592	D00 000	13 631 488
1C 000	114 688	E00 000	14 680 064
1D 000	118 784	F00 000	15 728 640
1E 000	122 880	1 000 000	16 777 216
1F 000	126 976	2 000 000	33 554 432

Table H-5. Hexadecimal-Decimal Integer Conversion Table (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
010	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
020	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
030	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
040	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
050	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
060	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
070	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
080	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
090	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0A0	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0B0	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0C0	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0D0	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0E0	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0F0	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255
100	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
110	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
120	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
130	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
140	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
150	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
160	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
170	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
180	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
190	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A0	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B0	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C0	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D0	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E0	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F0	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
200	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
210	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
220	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
230	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
240	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
250	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
260	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
270	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
280	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
290	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2A0	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2B0	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2C0	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2D0	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2E0	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2F0	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767

Table H-5. Hexadecimal-Decimal Integer Conversion Table (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
300	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
310	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
320	0800	0801	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
330	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
340	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
350	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
360	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
370	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
380	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
390	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3A0	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3B0	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3C0	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3D0	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3E0	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3F0	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023
400	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
410	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
420	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
430	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
440	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
450	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
460	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
470	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
480	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
490	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4A0	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B0	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C0	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4D0	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E0	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4F0	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
500	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1291	1293	1294	1295
510	1296	1297	1298	1299	1399	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
520	1312	1313	1314	1315	1316	1317	1318	1319	1329	1321	1322	1323	1324	1325	1326	1327
530	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
540	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1367	1358	1359
550	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
560	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
570	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
580	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1429	1421	1422	1423
590	1324	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5A0	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
3B0	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5C0	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5D0	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5E0	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5F0	1520	1521	1522	1523	1524	1515	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535

Table H-5. Hexadecimal-Decimal Integer Conversion Table (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
600	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
610	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
620	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
630	1584	1585	1586	1587	1588	1589	1590	1591	1592	1592	1594	1595	1596	1597	1598	1599
640	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
650	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
660	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
670	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
680	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
690	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6A0	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B0	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	17231	1724	1725	1726	1727
6C0	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D0	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E0	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F0	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791
700	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	8102	1803	1804	1805	1806	1807
710	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
720	1824	1825	1826	1827	1818	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
730	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
740	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
750	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
760	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1909	1902	1903
770	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
780	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
790	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A0	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B0	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C0	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D0	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7E0	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F0	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047
800	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
810	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
820	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
830	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
840	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
850	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
860	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
870	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
880	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
890	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A0	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B0	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C0	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D0	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E0	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F0	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303



Table H-5. Hexadecimal-Decimal Integer Conversion Table (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
900	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
910	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
920	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
930	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
940	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
950	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
960	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
970	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
980	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
990	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A0	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B0	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C0	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D0	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E0	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F0	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559
A00	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A10	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A20	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A30	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A40	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A50	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A60	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A70	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A80	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A90	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA0	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB0	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC0	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767
AD0	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE0	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF0	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B00	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B10	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B20	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B30	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B40	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895
B50	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B60	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B70	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B80	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B90	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA0	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB0	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC0	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD0	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE0	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF0	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071

Table H-5. Hexadecimal-Decimal Integer Conversion Table (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C00	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C10	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C20	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C30	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C40	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C50	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C60	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C70	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C80	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C90	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA0	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB0	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC0	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD0	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE0	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF0	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327
D00	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D10	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D20	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D30	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D40	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D50	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D60	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D70	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D80	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D90	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA0	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB0	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC0	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD0	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE0	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF0	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E00	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E10	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E20	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E30	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E40	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E50	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E60	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E70	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E80	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E90	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA0	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EBO	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775

Table H-5. Hexadecimal-Decimal Integer Conversion Table (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
EC0	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED0	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE0	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF0	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F00	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F10	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F20	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F30	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F40	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F50	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F60	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F70	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F80	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F90	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA0	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB0	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC0	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD0	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE0	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF0	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

Table H-6. Hexadecimal-Decimal Fraction Conversion Table

Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal
.00	00 00 00	.40	00 00 00	.80	00 00 00	.C0	00 00 00
.01	00 00 00	.41	00 00 00	.81	00 00 00	.C1	00 00 00
.02	00 00 00	.42	00 00 00	.82	00 00 00	.C2	00 00 00
.03	00 00 00	.43	00 00 00	.83	00 00 00	.C3	00 00 00
.04	00 00 00	.44	00 00 00	.84	00 00 00	.C4	00 00 00
.05	00 00 00	.45	00 00 00	.85	00 00 00	.C5	00 00 00
.06	00 00 00	.46	00 00 00	.86	00 00 00	.C6	00 00 00
.07	00 00 00	.47	00 00 00	.87	00 00 00	.C7	00 00 00
.08	00 00 00	.48	00 00 00	.88	00 00 00	.C8	00 00 00
.09	00 00 00	.49	00 00 00	.89	00 00 00	.C9	00 00 00
.0A	00 00 00	.4A	00 00 00	.8A	00 00 00	.CA	00 00 00
.0B	00 00 00	.4B	00 00 00	.8B	00 00 00	.CB	00 00 00
.0C	00 00 00	.4C	00 00 00	.8C	00 00 00	.CC	00 00 00
.0D	00 00 00	.4D	00 00 00	.8D	00 00 00	.CD	00 00 00
.0E	00 00 00	.4E	00 00 00	.8E	00 00 00	.CE	00 00 00
.0F	00 00 00	.4F	00 00 00	.8F	00 00 00	.CF	00 00 00
.10	00 00 00	.50	00 00 00	.90	00 00 00	.D0	00 00 00
.11	00 00 00	.51	00 00 00	.91	00 00 00	.D1	00 00 00
.12	00 00 00	.52	00 00 00	.92	00 00 00	.D2	00 00 00
.13	00 00 00	.53	00 00 00	.93	00 00 00	.D3	00 00 00
.14	00 00 00	.54	00 00 00	.94	00 00 00	.D4	00 00 00
.15	00 00 00	.55	00 00 00	.95	00 00 00	.D5	00 00 00
.16	00 00 00	.56	00 00 00	.96	00 00 00	.D6	00 00 00
.17	00 00 00	.57	00 00 00	.97	00 00 00	.D7	00 00 00
.18	00 00 00	.58	00 00 00	.98	00 00 00	.D8	00 00 00
.19	00 00 00	.59	00 00 00	.99	00 00 00	.D9	00 00 00
.1A	00 00 00	.5A	00 00 00	.9A	00 00 00	.DA	00 00 00
.1B	00 00 00	.5B	00 00 00	.9B	00 00 00	.DB	00 00 00
.1C	00 00 00	.5C	00 00 00	.9C	00 00 00	.DC	00 00 00
.1D	00 00 00	.5D	00 00 00	.9D	00 00 00	.DD	00 00 00
.1E	00 00 00	.5E	00 00 00	.9E	00 00 00	.DE	00 00 00
.1F	00 00 00	.5F	00 00 00	.9F	00 00 00	.DF	00 00 00
.20	00 00 00	.60	00 00 00	.A0	00 00 00	.E0	00 00 00
.21	00 00 00	.61	00 00 00	.A1	00 00 00	.E1	00 00 00
.22	00 00 00	.62	00 00 00	.A2	00 00 00	.E2	00 00 00
.23	00 00 00	.63	00 00 00	.A3	00 00 00	.E3	00 00 00
.24	00 00 00	.64	00 00 00	.A4	00 00 00	.E4	00 00 00
.25	00 00 00	.65	00 00 00	.A5	00 00 00	.E5	00 00 00
.26	00 00 00	.66	00 00 00	.A6	00 00 00	.E6	00 00 00
.27	00 00 00	.67	00 00 00	.A7	00 00 00	.E7	00 00 00
.28	00 00 00	.68	00 00 00	.A8	00 00 00	.E8	00 00 00
.29	00 00 00	.69	00 00 00	.A9	00 00 00	.E9	00 00 00
.2A	00 00 00	.6A	00 00 00	.AA	00 00 00	.EA	00 00 00
.2B	00 00 00	.6B	00 00 00	.AB	00 00 00	.EB	00 00 00
.2C	00 00 00	.6C	00 00 00	.AC	00 00 00	.EC	00 00 00
.2D	00 00 00	.6D	00 00 00	.AD	00 00 00	.ED	00 00 00
.2E	00 00 00	.6E	00 00 00	.AE	00 00 00	.EE	00 00 00
.2F	00 00 00	.6F	00 00 00	.AF	00 00 00	.EF	00 00 00
.30	00 00 00	.70	00 00 00	.B0	00 00 00	.F0	00 00 00
.31	00 00 00	.71	00 00 00	.B1	00 00 00	.F1	00 00 00
.32	00 00 00	.72	00 00 00	.B2	00 00 00	.F2	00 00 00
.33	00 00 00	.73	00 00 00	.B3	00 00 00	.F3	00 00 00
.34	00 00 00	.74	00 00 00	.B4	00 00 00	.F4	00 00 00
.35	00 00 00	.75	00 00 00	.B5	00 00 00	.F5	00 00 00
.36	00 00 00	.76	00 00 00	.B6	00 00 00	.F6	00 00 00
.37	00 00 00	.77	00 00 00	.B7	00 00 00	.F7	00 00 00
.38	00 00 00	.78	00 00 00	.B8	00 00 00	.F8	00 00 00
.39	00 00 00	.79	00 00 00	.B9	00 00 00	.F9	00 00 00
.3A	00 00 00	.7A	00 00 00	.BA	00 00 00	.FA	00 00 00
.3B	00 00 00	.7B	00 00 00	.BB	00 00 00	.FB	00 00 00
.3C	00 00 00	.7C	00 00 00	.BC	00 00 00	.FC	00 00 00
.3D	00 00 00	.7D	00 00 00	.BD	00 00 00	.FD	00 00 00
.3E	00 00 00	.7E	00 00 00	.BE	00 00 00	.FE	00 00 00
.3F	00 00 00	.7F	00 00 00	.BF	00 00 00	.FF	00 00 00

Table H-6. Hexadecimal-Decimal Fraction Conversion Table (Continued)

Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal
.00 00 00 00	.00000 00000	.00 40 00 00	.00097 65625	.00 80 00 00	.00195 31250	.00 C0 00 00	.00292 96875
.00 01 00 00	.00001 52587	.00 41 00 00	.00099 18212	.00 81 00 00	.00196 83837	.00 C1 00 00	.00294 49462
.00 02 00 00	.00003 05175	.00 42 00 00	.00100 70800	.00 82 00 00	.00198 36425	.00 C2 00 00	.00296 02050
.00 03 00 00	.00004 57763	.00 43 00 00	.00102 23388	.00 83 00 00	.00199 89013	.00 C3 00 00	.00297 54638
.00 04 00 00	.00006 10351	.00 44 00 00	.00103 75976	.00 84 00 00	.00201 41601	.00 C4 00 00	.00299 07226
.00 05 00 00	.00007 62939	.00 45 00 00	.00105 28564	.00 85 00 00	.00202 94189	.00 C5 00 00	.00300 59814
.00 06 00 00	.00009 15527	.00 46 00 00	.00106 81152	.00 86 00 00	.00204 46777	.00 C6 00 00	.00302 12402
.00 07 00 00	.00010 68115	.00 47 00 00	.00108 33740	.00 87 00 00	.00205 99365	.00 C7 00 00	.00303 64990
.00 08 00 00	.00012 20703	.00 48 00 00	.00109 86328	.00 88 00 00	.00207 51953	.00 C8 00 00	.00305 17578
.00 09 00 00	.00013 73291	.00 49 00 00	.00111 38916	.00 89 00 00	.00209 04541	.00 C9 00 00	.00306 70166
.00 0A 00 00	.00015 25878	.00 4A 00 00	.00112 91503	.00 8A 00 00	.00210 57128	.00 CA 00 00	.00308 22753
.00 0B 00 00	.00016 78466	.00 4B 00 00	.00114 44091	.00 8B 00 00	.00212 09716	.00 CB 00 00	.00309 75341
.00 0C 00 00	.00018 31054	.00 4C 00 00	.00115 96679	.00 8C 00 00	.00213 62304	.00 CC 00 00	.00311 27929
.00 0D 00 00	.00019 83642	.00 4D 00 00	.00117 49267	.00 8D 00 00	.00215 14892	.00 CD 00 00	.00312 80517
.00 0E 00 00	.00021 36230	.00 4E 00 00	.00119 01855	.00 8E 00 00	.00216 67480	.00 CE 00 00	.00314 33105
.00 0F 00 00	.00022 88818	.00 4F 00 00	.00120 54443	.00 8F 00 00	.00218 20068	.00 CF 00 00	.00315 85693
.00 10 00 00	.00024 41406	.00 50 00 00	.00122 07031	.00 90 00 00	.00219 72656	.00 D0 00 00	.00317 38281
.00 11 00 00	.00025 93994	.00 51 00 00	.00123 59619	.00 91 00 00	.00221 25244	.00 D1 00 00	.00318 90869
.00 12 00 00	.00027 46582	.00 52 00 00	.00125 12207	.00 92 00 00	.00222 77832	.00 D2 00 00	.00320 43457
.00 13 00 00	.00028 99169	.00 53 00 00	.00126 64794	.00 93 00 00	.00224 30419	.00 D3 00 00	.00321 96044
.00 14 00 00	.00030 51757	.00 54 00 00	.00128 17382	.00 94 00 00	.00225 83007	.00 D4 00 00	.00323 48632
.00 15 00 00	.00032 04345	.00 55 00 00	.00129 69970	.00 95 00 00	.00227 35595	.00 D5 00 00	.00325 01220
.00 16 00 00	.00033 56933	.00 56 00 00	.00131 22558	.00 96 00 00	.00228 88183	.00 D6 00 00	.00326 53808
.00 17 00 00	.00035 09521	.00 57 00 00	.00132 75146	.00 97 00 00	.00230 40771	.00 D7 00 00	.00328 06396
.00 18 00 00	.00036 62109	.00 58 00 00	.00134 27734	.00 98 00 00	.00231 93359	.00 D8 00 00	.00329 58984
.00 19 00 00	.00038 14697	.00 59 00 00	.00135 80322	.00 99 00 00	.00233 45947	.00 D9 00 00	.00331 11572
.00 1A 00 00	.00039 67285	.00 5A 00 00	.00137 32910	.00 9A 00 00	.00234 98535	.00 DA 00 00	.00332 64160
.00 1B 00 00	.00041 19873	.00 5B 00 00	.00138 85498	.00 9B 00 00	.00236 51123	.00 DB 00 00	.00334 16748
.00 1C 00 00	.00042 72460	.00 5C 00 00	.00140 38085	.00 9C 00 00	.00238 03710	.00 DC 00 00	.00335 69335
.00 1D 00 00	.00044 25048	.00 5D 00 00	.00141 90673	.00 9D 00 00	.00239 56298	.00 DD 00 00	.00337 21923
.00 1E 00 00	.00045 77636	.00 5E 00 00	.00143 43261	.00 9E 00 00	.00241 08886	.00 DE 00 00	.00338 74511
.00 1F 00 00	.00047 30224	.00 5F 00 00	.00144 95849	.00 9F 00 00	.00242 61474	.00 DF 00 00	.00340 27099
.00 20 00 00	.00048 82812	.00 60 00 00	.00146 48437	.00 A0 00 00	.00244 14062	.00 E0 00 00	.00341 79687
.00 21 00 00	.00050 35400	.00 61 00 00	.00148 01025	.00 A1 00 00	.00245 66650	.00 E1 00 00	.00343 32275
.00 22 00 00	.00051 87988	.00 62 00 00	.00149 53613	.00 A2 00 00	.00247 19238	.00 E2 00 00	.00344 84863
.00 23 00 00	.00053 40576	.00 63 00 00	.00151 06201	.00 A3 00 00	.00248 71826	.00 E3 00 00	.00346 37451
.00 24 00 00	.00054 93164	.00 64 00 00	.00152 58789	.00 A4 00 00	.00250 24414	.00 E4 00 00	.00347 90039
.00 25 00 00	.00056 45751	.00 65 00 00	.00154 11376	.00 A5 00 00	.00251 77001	.00 E5 00 00	.00349 42626
.00 26 00 00	.00057 98339	.00 66 00 00	.00155 63964	.00 A6 00 00	.00253 29589	.00 E6 00 00	.00350 95214
.00 27 00 00	.00059 50927	.00 67 00 00	.00157 16552	.00 A7 00 00	.00254 82177	.00 E7 00 00	.00352 47802
.00 28 00 00	.00061 03515	.00 68 00 00	.00158 69140	.00 A8 00 00	.00256 34765	.00 E8 00 00	.00354 00390
.00 29 00 00	.00062 56103	.00 69 00 00	.00160 21728	.00 A9 00 00	.00257 87353	.00 E9 00 00	.00355 52978
.00 2A 00 00	.00064 08691	.00 6A 00 00	.00161 74316	.00 AA 00 00	.00259 39941	.00 EA 00 00	.00357 05566
.00 2B 00 00	.00065 61279	.00 6B 00 00	.00163 26904	.00 AB 00 00	.00260 92529	.00 EB 00 00	.00358 58154
.00 2C 00 00	.00067 13867	.00 6C 00 00	.00164 79492	.00 AC 00 00	.00262 45117	.00 EC 00 00	.00360 10742
.00 2D 00 00	.00068 66455	.00 6D 00 00	.00166 32080	.00 AD 00 00	.00263 97705	.00 ED 00 00	.00361 63330
.00 2E 00 00	.00070 19042	.00 6E 00 00	.00167 84667	.00 AE 00 00	.00265 50292	.00 EE 00 00	.00363 15917
.00 2F 00 00	.00071 71630	.00 6F 00 00	.00169 37255	.00 AF 00 00	.00267 02880	.00 EF 00 00	.00364 68505
.00 30 00 00	.00073 24218	.00 70 00 00	.00170 89843	.00 B0 00 00	.00268 55468	.00 F0 00 00	.00366 21093
.00 31 00 00	.00074 76806	.00 71 00 00	.00172 42421	.00 B1 00 00	.00270 08056	.00 F1 00 00	.00367 73681
.00 32 00 00	.00076 29394	.00 72 00 00	.00173 95019	.00 B2 00 00	.00271 60644	.00 F2 00 00	.00369 26269
.00 33 00 00	.00077 81982	.00 73 00 00	.00175 47607	.00 B3 00 00	.00273 13232	.00 F3 00 00	.00370 78857
.00 34 00 00	.00079 34570	.00 74 00 00	.00177 00195	.00 B4 00 00	.00274 65820	.00 F4 00 00	.00372 31445
.00 35 00 00	.00080 87158	.00 75 00 00	.00178 52783	.00 B5 00 00	.00276 18408	.00 F5 00 00	.00373 84033
.00 36 00 00	.00082 39746	.00 76 00 00	.00180 05371	.00 B6 00 00	.00277 70996	.00 F6 00 00	.00375 36621
.00 37 00 00	.00083 92333	.00 77 00 00	.00181 57958	.00 B7 00 00	.00279 23583	.00 F7 00 00	.00376 89208
.00 38 00 00	.00085 44921	.00 78 00 00	.00183 10546	.00 B8 00 00	.00280 76171	.00 F8 00 00	.00378 41796
.00 39 00 00	.00086 97509	.00 79 00 00	.00184 63134	.00 B9 00 00	.00282 28759	.00 F9 00 00	.00379 94384
.00 3A 00 00	.00088 50097	.00 7A 00 00	.00186 15722	.00 BA 00 00	.00283 81347	.00 FA 00 00	.00381 46972
.00 3B 00 00	.00090 02685	.00 7B 00 00	.00187 68310	.00 BB 00 00	.00285 33935	.00 FB 00 00	.00382 99560
.00 3C 00 00	.00091 55273	.00 7C 00 00	.00189 20898	.00 BC 00 00	.00286 86523	.00 FC 00 00	.00384 52148
.00 3D 00 00	.00093 07861	.00 7D 00 00	.00190 73486	.00 BD 00 00	.00288 39111	.00 FD 00 00	.00386 04736
.00 3E 00 00	.00094 60449	.00 7E 00 00	.00192 26074	.00 BE 00 00	.00289 91699	.00 FE 00 00	.00387 57324
.00 3F 00 00	.00096 13037	.00 7F 00 00	.00193 78662	.00 BF 00 00	.00291 44287	.00 FF 00 00	.00389 09912

**Table H-6. Hexadecimal-Decimal Fraction Conversion Table (Continued)**

Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal
.00 00 00 00	.00000 00000	.00 00 40 00	.00000 38146	.00 00 80 00	.00000 76293	.00 00 C0 00	.00001 14440
.00 00 01 00	.00000 00596	.00 00 41 00	.00000 38743	.00 00 81 00	.00000 76889	.00 00 C1 00	.00001 15036
.00 00 02 00	.00000 01192	.00 00 42 00	.00000 39339	.00 00 82 00	.00000 77486	.00 00 C2 00	.00001 15633
.00 00 03 00	.00000 01788	.00 00 43 00	.00000 39935	.00 00 83 00	.00000 78082	.00 00 C3 00	.00001 16229
.00 00 04 00	.00000 02384	.00 00 44 00	.00000 40531	.00 00 84 00	.00000 78678	.00 00 C4 00	.00001 16825
.00 00 05 00	.00000 02980	.00 00 45 00	.00000 41127	.00 00 85 00	.00000 79274	.00 00 C5 00	.00001 17421
.00 00 06 00	.00000 03576	.00 00 46 00	.00000 41723	.00 00 86 00	.00000 79870	.00 00 C6 00	.00001 18017
.00 00 07 00	.00000 04172	.00 00 47 00	.00000 42319	.00 00 87 00	.00000 80466	.00 00 C7 00	.00001 18613
.00 00 08 00	.00000 04768	.00 00 48 00	.00000 42915	.00 00 88 00	.00000 81062	.00 00 C8 00	.00001 19209
.00 00 09 00	.00000 05364	.00 00 49 00	.00000 43511	.00 00 89 00	.00000 81658	.00 00 C9 00	.00001 19805
.00 00 0A 00	.00000 05960	.00 00 4A 00	.00000 44107	.00 00 8A 00	.00000 82254	.00 00 CA 00	.00001 20401
.00 00 0B 00	.00000 06556	.00 00 4B 00	.00000 44703	.00 00 8B 00	.00000 82850	.00 00 CB 00	.00001 20997
.00 00 0C 00	.00000 07152	.00 00 4C 00	.00000 45299	.00 00 8C 00	.00000 83446	.00 00 CC 00	.00001 21593
.00 00 0D 00	.00000 07748	.00 00 4D 00	.00000 45895	.00 00 8D 00	.00000 84042	.00 00 CD 00	.00001 22189
.00 00 0E 00	.00000 08344	.00 00 4E 00	.00000 46491	.00 00 8E 00	.00000 84638	.00 00 CE 00	.00001 22785
.00 00 0F 00	.00000 08940	.00 00 4F 00	.00000 47087	.00 00 8F 00	.00000 85234	.00 00 CF 00	.00001 23381
.00 00 10 00	.00000 09536	.00 00 50 00	.00000 47683	.00 00 90 00	.00000 85830	.00 00 D0 00	.00001 23977
.00 00 11 00	.00000 10132	.00 00 51 00	.00000 48279	.00 00 91 00	.00000 86426	.00 00 D1 00	.00001 24573
.00 00 12 00	.00000 10728	.00 00 52 00	.00000 48875	.00 00 92 00	.00000 87022	.00 00 D2 00	.00001 25169
.00 00 13 00	.00000 11324	.00 00 53 00	.00000 49471	.00 00 93 00	.00000 87618	.00 00 D3 00	.00001 25765
.00 00 14 00	.00000 11920	.00 00 54 00	.00000 50067	.00 00 94 00	.00000 88214	.00 00 D4 00	.00001 26361
.00 00 15 00	.00000 12516	.00 00 55 00	.00000 50663	.00 00 95 00	.00000 88810	.00 00 D5 00	.00001 26957
.00 00 16 00	.00000 13112	.00 00 56 00	.00000 51259	.00 00 96 00	.00000 89406	.00 00 D6 00	.00001 27553
.00 00 17 00	.00000 13708	.00 00 57 00	.00000 51855	.00 00 97 00	.00000 90002	.00 00 D7 00	.00001 28149
.00 00 18 00	.00000 14304	.00 00 58 00	.00000 52451	.00 00 98 00	.00000 90598	.00 00 D8 00	.00001 28745
.00 00 19 00	.00000 14900	.00 00 59 00	.00000 53047	.00 00 99 00	.00000 91194	.00 00 D9 00	.00001 29341
.00 00 1A 00	.00000 15496	.00 00 5A 00	.00000 53643	.00 00 9A 00	.00000 91790	.00 00 DA 00	.00001 29937
.00 00 1B 00	.00000 16092	.00 00 5B 00	.00000 54239	.00 00 9B 00	.00000 92386	.00 00 DB 00	.00001 30533
.00 00 1C 00	.00000 16688	.00 00 5C 00	.00000 54835	.00 00 9C 00	.00000 92982	.00 00 DC 00	.00001 31129
.00 00 1D 00	.00000 17284	.00 00 5D 00	.00000 55431	.00 00 9D 00	.00000 93578	.00 00 DD 00	.00001 31725
.00 00 1E 00	.00000 17880	.00 00 5E 00	.00000 56027	.00 00 9E 00	.00000 94174	.00 00 DE 00	.00001 32321
.00 00 1F 00	.00000 18476	.00 00 5F 00	.00000 56623	.00 00 9F 00	.00000 94770	.00 00 DF 00	.00001 32917
.00 00 20 00	.00000 19072	.00 00 60 00	.00000 57219	.00 00 A0 00	.00000 95366	.00 00 E0 00	.00001 33513
.00 00 21 00	.00000 19668	.00 00 61 00	.00000 57815	.00 00 A1 00	.00000 95962	.00 00 E1 00	.00001 34109
.00 00 22 00	.00000 20264	.00 00 62 00	.00000 58411	.00 00 A2 00	.00000 96558	.00 00 E2 00	.00001 34705
.00 00 23 00	.00000 20860	.00 00 63 00	.00000 59007	.00 00 A3 00	.00000 97154	.00 00 E3 00	.00001 35301
.00 00 24 00	.00000 21456	.00 00 64 00	.00000 59603	.00 00 A4 00	.00000 97750	.00 00 E4 00	.00001 35897
.00 00 25 00	.00000 22052	.00 00 65 00	.00000 60200	.00 00 A5 00	.00000 98346	.00 00 E5 00	.00001 36493
.00 00 26 00	.00000 22648	.00 00 66 00	.00000 60796	.00 00 A6 00	.00000 98942	.00 00 E6 00	.00001 37089
.00 00 27 00	.00000 23244	.00 00 67 00	.00000 61392	.00 00 A7 00	.00000 99538	.00 00 E7 00	.00001 37685
.00 00 28 00	.00000 23840	.00 00 68 00	.00000 61988	.00 00 A8 00	.00001 00134	.00 00 E8 00	.00001 38281
.00 00 29 00	.00000 24436	.00 00 69 00	.00000 62584	.00 00 A9 00	.00001 00730	.00 00 E9 00	.00001 38877
.00 00 2A 00	.00000 25032	.00 00 6A 00	.00000 63180	.00 00 AA 00	.00001 01326	.00 00 EA 00	.00001 39473
.00 00 2B 00	.00000 25628	.00 00 6B 00	.00000 63776	.00 00 AB 00	.00001 01922	.00 00 EB 00	.00001 40069
.00 00 2C 00	.00000 26224	.00 00 6C 00	.00000 64372	.00 00 AC 00	.00001 02518	.00 00 EC 00	.00001 40665
.00 00 2D 00	.00000 26820	.00 00 6D 00	.00000 64968	.00 00 AD 00	.00001 03114	.00 00 ED 00	.00001 41261
.00 00 2E 00	.00000 27416	.00 00 6E 00	.00000 65564	.00 00 AE 00	.00001 03710	.00 00 EE 00	.00001 41857
.00 00 2F 00	.00000 28012	.00 00 6F 00	.00000 66160	.00 00 AF 00	.00001 04306	.00 00 EF 00	.00001 42453
.00 00 30 00	.00000 28608	.00 00 70 00	.00000 66756	.00 00 B0 00	.00001 04902	.00 00 F0 00	.00001 43049
.00 00 31 00	.00000 29204	.00 00 71 00	.00000 67352	.00 00 B1 00	.00001 05500	.00 00 F1 00	.00001 43645
.00 00 32 00	.00000 29800	.00 00 72 00	.00000 67948	.00 00 B2 00	.00001 06096	.00 00 F2 00	.00001 44241
.00 00 33 00	.00000 30396	.00 00 73 00	.00000 68544	.00 00 B3 00	.00001 06692	.00 00 F3 00	.00001 44837
.00 00 34 00	.00000 30992	.00 00 74 00	.00000 69140	.00 00 B4 00	.00001 07288	.00 00 F4 00	.00001 45433
.00 00 35 00	.00000 31588	.00 00 75 00	.00000 69736	.00 00 B5 00	.00001 07884	.00 00 F5 00	.00001 46029
.00 00 36 00	.00000 32184	.00 00 76 00	.00000 70332	.00 00 B6 00	.00001 08480	.00 00 F6 00	.00001 46625
.00 00 37 00	.00000 32780	.00 00 77 00	.00000 70928	.00 00 B7 00	.00001 09076	.00 00 F7 00	.00001 47221
.00 00 38 00	.00000 33376	.00 00 78 00	.00000 71524	.00 00 B8 00	.00001 09672	.00 00 F8 00	.00001 47817
.00 00 39 00	.00000 33972	.00 00 79 00	.00000 72120	.00 00 B9 00	.00001 10268	.00 00 F9 00	.00001 48413
.00 00 3A 00	.00000 34568	.00 00 7A 00	.00000 72716	.00 00 BA 00	.00001 10864	.00 00 FA 00	.00001 49009
.00 00 3B 00	.00000 35164	.00 00 7B 00	.00000 73312	.00 00 BB 00	.00001 11460	.00 00 FB 00	.00001 49605
.00 00 3C 00	.00000 35760	.00 00 7C 00	.00000 73908	.00 00 BC 00	.00001 12056	.00 00 FC 00	.00001 50201
.00 00 3D 00	.00000 36356	.00 00 7D 00	.00000 74504	.00 00 BD 00	.00001 12652	.00 00 FD 00	.00001 50797
.00 00 3E 00	.00000 36952	.00 00 7E 00	.00000 75100	.00 00 BE 00	.00001 13248	.00 00 FE 00	.00001 51393
.00 00 3F 00	.00000 37548	.00 00 7F 00	.00000 75696	.00 00 BF 00	.00001 13844	.00 00 FF 00	.00001 51989



Table H-7. Common Mathematical Constants

Constant	Decimal Value			Hexadecimal Value	
$\pi$	3.14159	26535	89793	3.243F	6A89
$\pi^{-1}$	0.31830	98861	83790	0.517C	C1B7
$\sqrt{\pi}$	1.77245	38509	05516	1.C5BF	891C
$\ln \pi$	1.14472	98858	49400	1.250D	048F
$e$	2.71828	18284	59045	2.B7E1	5163
$e^{-1}$	0.36787	94411	71442	0.5E2D	58D9
$\sqrt{e}$	1.64872	12707	00128	1.A612	98E2
$\log_{10} e$	0.43429	44819	03252	0.6F2D	EC55
$\log_2 e$	1.44269	50408	88963	1.7154	7653
$\gamma$	0.57721	56649	01533	0.93C4	67E4
$\ln \gamma$	-0.54953	93129	81645	-0.8CAE	9BC1
$\sqrt{2}$	1.41421	35623	73095	1.6A09	E668
$\ln 2$	0.69314	71805	59945	0.B172	17F8
$\log_{10} 2$	0.30102	99956	63981	0.4D10	4D42
$\sqrt{10}$	3.16227	76601	68379	3.298B	075C
$\ln 10$	2.30258	40929	94046	2.4D76	3777



# Appendix I

## Instruction Usage Cross-Reference Table

---

This table shows the operations and operands of the 990 instruction set. The table is designed to tell the user all of the instructions that perform a type of operation (arithmetic, shift, etc.) and all of the instructions that operate on a given data type (word, multiple precision, etc.). The instructions are listed alphabetically by mnemonic. The terminology for the instruction categories is as follows:

ARITH	Arithmetic
JMP/BR	Jump and Branch
SRCH/CM	Search and/or Compare
CNT/CRU	Control/CRU
LD/MOVE	Load and Move
LOGICAL	Logical
SHIFT	Shift
LDA	Long Distance Addressing
CONVERT	Conversion
BYTE	Byte Operand(s)
WORD	Word Operand(s)
FLT PNT	Floating Point Number Operand(s)
MUL PRE	Multiple Precision Operand(s)
STRING	String Operand(s)
STACK	Stack Operand
LIST	List Operand
FIELD	Field Operand(s)
MISC	Miscellaneous Instructions
990/12	Instruction executes on 990/12 only

TABLE

M N E M O N I C	C A T E G O R Y																9 9 0 / 12		
	A R I T H	J M P / B R	S R C H / C M	C N T / C R U	L D / M O V E	L O G I C A L	S H I F T	L D A	C O N V E R T	B Y T E	W O R D	F L T P N T	M U L P R E	S T R I N G	S T A C K	L I S T		F I E L D	M I S C
A	X									X									
AB	X								X										
ABS	X									X									
AD	X										X								X
AI	X									X									
AM	X											X							X
ANDI					X					X									
ANDM					X							X							X
AR	X										X								X
ARJ	X	X								X									X
B		X																	
BDC								X				X							X
BIND		X																	X
BL		X																	
BLSK		X													X				X
BLWP		X																	
C			X							X									
CB			X						X										
CDE								X			X	X							X
CDI								X		X	X								X
CED								X			X	X							X
CER								X			X	X							X
CI			X							X									
CID								X		X	X								X
CIR								X		X	X								X
CKOF				X															
CKON				X															
CLR					X					X									
CNTO	X									X		X							X
COC			X							X									
CRC										X		X					X		X
CRE								X			X	X							X
CRI								X		X	X								X
CS			X											X					X
CZC			X							X									
DBC								X				X							X
DD	X										X								X
DEC	X									X									
DECT	X									X									
DINT				X															X

(B)143979

TABLE

M N E M O N I C	C A T E G O R Y														9 9 0 / 12			
	A R I T H	J M P / B R	S R C H / C M	C N T / C R U	L D / M O V E	L O G I C A L	S H I F T	L D A	C O N V E R T	B Y T E	W O R D	F L T P N T	M U L P R E	S T R I N G		S T A C K	L I S T	F I E L D
DIV	X									X								
DIVS	X											X						X
DR	X										X							X
EINT				X														X
EMD																	X	X
EP								X				X						X
IDLE				X														
INC	X										X							
INCT	X										X							
INSF																X		X
INV						X					X							
IOF																X		X
JEQ		X																
JGT		X																
JH		X																
JHE		X																
JL		X																
JLE		X																
JLT		X																
JMP		X																
JNC		X																
JNE		X																
JNO		X																
JOC		X																
JOP		X																
LCS				X	X													X
LD					X						X							X
LDCR				X														
LDD								X										
LDS								X										
LI					X						X							
LIM					X													X
LIMI					X													
LMF					X													
LR					X						X							X
LREX																	X	
LST					X													X
LTO			X									X						X
LWP					X						X							X
LWPI					X						X							X
MD	X										X							X

(B)143980

TABLE

M N E M O N I C	CATEGORY																9 9 0 / 12		
	A R I T H	J M P / B R	S R C H / C M	C N T / C R U	L D / M O V E	L O G I C A L	S H I F T	L D A	C O N V E R T	B Y T E	W O R D	F L T P N T	M U L P R E	S T R I N G	S T A C K	L I S T		F I E L D	M I S C
MOV				X						X									
MOVA				X						X									X
MOVB				X					X										
MOVS				X										X					X
MPY	X									X									
MPYS	X											X							X
MR	X										X								X
MVSK				X										X	X				X
MVSR				X										X					X
NEG	X									X									
NEGD	X										X								X
NEGR	X										X								X
NRM	X											X							X
ORI						X				X									
ORM						X						X							X
POPS				X										X	X				X
PSHS				X										X	X				X
RSET				X															
RTO			X										X						X
RTWP		X																	
S	X									X									
SB	X									X									
SBO				X															
SBZ				X															
SD	X										X								X
SEQB			X											X					X
SETO						X				X									
SLA							X			X									
SLAM							X					X							X
SLSL			X													X	X		X
SLSP			X					X								X	X		X
SM	X												X						X
SNEB			X											X					X
SOC						X				X									
SOCB						X				X									
SR	X										X								X
SRA							X			X									
SRAM							X					X							X
SRC							X			X									
SRJ	X	X								X									X

TABLE

M N E M O N I C	CATEGORY														9 9 0 / 12				
	A R I T H	J M P / B R	S R C H / C M	C N T / C R U	L D M O D E	L O G I C A L	S H I F T	L D A	C O N V E R T	B Y T E	W O R D	F L T P N T	M U L P R E	S T R I N G		S T A C K	L I S T	F I E L D	M I S C
SRL						X				X									
STCR				X															
STD					X						X								X
STPC					X					X									X
STR					X						X								X
STST					X					X									
STWP					X					X									
SWPB					X				X	X									
SWPM					X							X							X
SZC						X				X									
SZCB						X			X										
TB				X															
TCMB			X			X													X
TMB			X			X													X
TS								X					X						X
TSMB			X			X													X
X																		X	
XF					X					X						X			X
XIT																	X		X
XOP																	X		
XOR						X				X									
XORM						X						X							X
XV					X					X						X			X

(B)143982

# Appendix J

## Illegal Opcodes

---

0000-001B	0270-207F	0341-035F
002C	0290-029F	0361-037F
0100-013F	02B0-02BF	0381-039F
0210-021F	02D0-02DF	03A1-03BF
0230-023F	02E1-02FF	03C1-03DF
0250-025F	0301-031F	03E1-03FF

The following instructions will also cause an illegal operation interrupt:

1. SLSL instruction (opcode = 0021) or SLSP instruction (opcode = 0022) when the C field (bits 0-3 of the second word) is in the range A-F<sub>16</sub>.
2. LCS instruction (opcode = 00A0) when the specified microcode load addresses are in the ranges 0000-07FF or 0C00-FFFF.

# Alphabetical Index

## Introduction

---

### HOW TO USE INDEX

The index, table of contents, list of illustrations, and list of tables are used in conjunction to obtain the location of the desired subject. Once the subject or topic has been located in the index, use the appropriate paragraph number, figure number, or table number to obtain the corresponding page number from the table of contents, list of illustrations, or list of tables.

### INDEX ENTRIES

The following index lists key words and concepts from the subject material of the manual together with the area(s) in the manual that supply major coverage of the listed concept. The numbers along the right side of the listing reference the following manual areas:

- Sections — References to sections of the manual appear as “Sections x” with the symbol x representing any numeric quantity.
- Appendixes — References to appendixes of the manual appear as “Appendix y” with the symbol y representing any capital letter.
- Paragraphs — References to paragraphs of the manual appear as a series of alphanumeric or numeric characters punctuated with decimal points. Only the first character of the string may be a letter; all subsequent characters are numbers. The first character refers to the section or appendix of the manual in which the paragraph may be found.
- Tables — References to tables in the manual are represented by the capital letter T followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the table). The second character is followed by a dash (-) and a number.

Tx-yy

- Figures — References to figures in the manual are represented by the capital letter F followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the figure). The second character is followed by a dash (-) and a number.

Fx-yy

- Other entries in the index — References to other entries in the index are preceded by the word “See” followed by the referenced entry.

- A Instruction ..... 3.6
- AB Instruction ..... 3.7
- Abnormal Completion Messages ..... T9-1
- ABS Instruction ..... 3.8, 4.2.5.1, 4.3.2.1
- Absolute:
  - Code ..... 8.2
  - Origin (AORG) Directive ..... 5.3.1, 5.3.1.1
  - Symbols ..... 2.7
- Absolute Value ..... 1.4.5
  - Instruction ..... 3.8, 4.2.5.1, 4.3.2.1
- Access Name, Error ..... 9.3
- AD Instruction ..... 3.9
- Add Bytes Instruction ..... 3.7
- Add Double Precision Real Instruction ..... 3.9
- Add Immediate Instruction ..... 3.10
- Add Multiple Precision Integer
  - Instruction ..... 3.11
- Add Real Instruction ..... 3.14
- Add to Register and Jump Instruction ..... 3.15
- Add Words Instruction ..... 3.6
- Addition Operation ..... 1.4.5
- Address:
  - Byte ..... 1.2.1
  - Indexed Memory ..... 3.2.5
  - Indirect Workspace ..... 3.2.2
  - Memory ..... 1.2.1
  - Program Counter Relative ..... 3.2.4, 3.2.6
  - Symbolic Memory ..... 3.2.4
  - Trace Memory ..... 4.5.1.3
  - Word ..... 1.2.2
  - Workspace Register ..... 3.2.1
- Addresses, Symbolic ..... 2.6
- Addressing:
  - CRU Bit ..... 3.2.7
  - Immediate ..... 3.2.8
  - Indexed Memory ..... 3.2.5
  - Instruction ..... T3-2
  - Mode ..... 3.5.2
  - Modes ..... 3.2, T3-1
  - Program Counter Relative ..... 3.2.6
  - Symbolic ..... 5.4
  - Symbolic Memory ..... 3.2.4
  - Workspace Register ..... 3.2.1
    - Indirect ..... 3.2.2
    - Indirect Autoincrement ..... 3.2.3
- AI Instruction ..... 3.10
- Alter Registers and Jump
  - Instructions ..... 3.4.19
- AM Instruction ..... 3.11
- AND Immediate Instruction ..... 3.12
- AND Multiple Precision Instruction ..... 3.13
- ANDI Instruction ..... 3.12
- ANDM Instruction ..... 3.13
- AORG Directive ..... 5.3.1, 5.3.1.1
- AR Instruction ..... 3.14
- Arithmetic:
  - Error ..... 1.4.11
  - Expressions ..... 2.7.2
  - Greater Than Bit ..... 1.4.2
  - Left Shift ..... 1.4.5
- ARJ Instruction ..... 3.15
- ASMELS Directive ..... 5.3.5, 5.3.5.4
- ASMEND Directive ..... 5.3.5, 5.3.5.4
- ASMIF Directive ..... 5.3.5, 5.3.5.4
- Assembler:
  - Directives ..... 5.3, 5.3.1, 5.3.2
    - 5.3.3, 5.3.4, 5.3.5
  - Macro Library ..... 7.6
  - Execute Macro (XMA) ..... Appendix G
  - Output ..... 5.3.2
  - SDSMAC ..... 5.2
- Assembly language:
  - Instructions ..... 3.1
    - 990/12 ..... 2.3
  - Assembly-Time Constants ..... 2.5.4, 8.2.1
  - Attribute Component ..... 7.4.7.1
  - Attributes, Symbol ..... 9.4, T9-2
  - Autoincrement Addressing, Workspace
    - Register Indirect ..... 3.2.3
- B Instruction ..... 3.16
- BDC Instruction ..... 3.17
- BES Directive ..... 5.3.1, 5.3.1.5
- Bias Register ..... 3.74.1
- Binary to Decimal ASCII Conversion
  - Instruction ..... 3.17
- BIND Instruction ..... 3.18
- Bipolar Memory ..... 1.3
- Bit:
  - Addressing, CRU ..... 3.2.7
  - Arithmetic Greater Than ..... 1.4.2
  - Breakpoint ..... 4.5.1.4
  - Carry ..... 1.4.4
  - Equal ..... 1.4.3
  - Extended Operation ..... 1.4.7
  - Logical Greater Than ..... 1.4.1
  - Memory File ..... 1.4.9
  - Memory Management and Protection
    - Enabled ..... 1.4.10
    - Overflow ..... 1.4.5
    - Interrupt ..... 1.4.11
- Bit I/O Instructions ..... 3.4.3
- Bit Testing Instructions ..... 3.4.16
- BL Instruction ..... 3.19, 4.2.1
- Block:
  - Ending With Symbol (BES)
    - Directive ..... 5.3.1, 5.3.1.5
  - Starting With Symbol (BSS)
    - Directive ..... 5.3.1, 5.3.1.4
- BLSK Instruction ..... 3.20, 4.2.2
- BLWP Instruction ..... 1.2.4, 3.21, 4.2.3
- Branch and Link Instruction ..... 3.19, 4.2.1
- Branch and Load Workspace Pointer
  - Instruction ..... 3.21, 4.2.3
- Branch and Push Link to Stack
  - Instruction ..... 4.2.2
- Branch Immediate and Push Link to
  - Stack Instruction ..... 3.20
- Branch Indirect Instruction ..... 3.18
- Branch Instruction ..... 3.16
- Breakpoint:
  - Bit ..... 4.5.1.3, 4.5.1.4, 4.5.1.6
  - Register ..... 4.5.1.4, 4.5.1.6



- System ..... 4.5.1.4
- BSS Directive ..... 5.3.1, 5.3.1.4
- Byte:
  - Address ..... 1.2.1
  - Directive ..... 2.4.2, 5.3.3, 5.3.3.1
  - Even Address ..... 1.2.2
  - Instruction ..... 1.2.1
  - Odd Address ..... 1.2.2
  - String ..... 2.8.3
- Byte (BYTE) Directive,
  - Initialize ..... 5.3.3, 5.3.3.1
- C Instruction ..... 3.22
- Cache:
  - Register ..... 1.3
  - Workspace ..... 1.3, 4.3.1, 4.3.2.4, 4.3.2.9, 4.3.3.3, 4.3.3.4
- Carry Bit ..... 1.4.4
- CB Instruction ..... 3.23
- CDE Instruction ..... 3.24
- CDI Instruction ..... 3.25
- CED Instruction ..... 3.26
- CEND Directive ..... 5.3.1, 5.3.1.9, 5.3.1.10
- CER Instruction ..... 3.27
- Character Constants ..... 2.5.3
- Character Set, SDSMAC ..... 2.4.1
- Character String ..... 2.6.2
- Characters, Tag ..... 9.5, 9.5.1, T9-3
- Checkpoint Register (CKPT)
  - Directive ..... 5.3.3, 5.3.3.5
- CI Instruction ..... 3.28
- CID Instruction ..... 3.29
- CIR Instruction ..... 3.30
- CKOF Instruction ..... 3.31
- CKON Instruction ..... 3.32
- CKPT Directive ..... 5.3.3, 5.3.3.5
- Clear:
  - Instruction ..... 3.33
  - Status Register ..... 1.4.10
- Clock Off Instruction ..... 3.31
- Clock On Instruction ..... 3.32
- CLR Instruction ..... 3.33
- CNTO Instruction ..... 3.34
- COC Instruction ..... 3.35
- Code:
  - Absolute ..... 8.2
  - Object ..... 9.2, 9.5, 9.5.4
  - Relocatable ..... 8.2
- Comment:
  - Field ..... 2.4.5, 3.5.4
  - Statement ..... 2.4
- Common:
  - Segment ..... 5.3.9
  - Segment End (CEND)
    - Directive ..... 5.3.1, 5.3.1.9, 5.3.1.10
  - Segment (CSEG) Directive ..... 5.3.1, 5.3.1.9
- Communications Register Unit ..... 4.5.2
- Compare Bytes Instruction ..... 3.23
- Compare Immediate Instruction ..... 3.28
- Compare Ones Corresponding
  - Instruction ..... 3.35
- Compare Strings Instruction ..... 3.39
- Compare Words Instruction ..... 3.22
- Compare Zeros Corresponding
  - Instruction ..... 3.40
- Component:
  - Attribute ..... 7.4.7.1
  - Length ..... 7.4.5.2, 7.4.5.3
  - Parameter Attribute ..... 7.4.7.2
  - Segment ..... 7.4.5.3
  - String ..... 7.4.5.2, 7.4.5.3
  - Symbol Attribute ..... 7.4.6, 7.4.7.1
  - User Attribute ..... 7.4.5.3
  - Value ..... 7.4.5.2, 7.4.5.3
- Components of Variables ..... 7.4.4
- Computer, 990/12 ..... 1.3
- Conditional Assembly (ASMELS)
  - Directive ..... 5.3.5, 5.3.5.4
- Conditional Assembly (ASMEND)
  - Directive ..... 5.3.5, 5.3.5.4
- Conditional Assembly (ASMIF)
  - Directive ..... 5.3.5, 5.3.5.4
- Conditions:
  - Overflow ..... 1.4.5
  - System Error Interrupt ..... 4.5.1.2
- Constants ..... 2.5, 5.3.3
  - Assembly-Time ..... 2.5.4
  - Character ..... 2.5.3
  - Decimal Integer ..... 2.5.1
  - Hexadecimal Integer ..... 2.5.2
  - Relocatable ..... 2.5
- Context Switch ..... 1.2.4
- Control Space, TILINE Peripheral ..... 1.2.3
- Conventions:
  - Syntax ..... 2.4
  - Syntax Definition ..... 3.5.4
- Convert Double Precision Real to
  - Extended Integer Instruction ..... 3.24
- Convert Double Precision Real to
  - Integer Instruction ..... 3.25
- Convert Extended Integer to Double
  - Precision Real Instruction ..... 3.26
- Convert Extended Integer to Real
  - Instruction ..... 3.27
- Convert Integer to Double Precision
  - Real Instruction ..... 3.29
- Convert Integer to Real Instruction ..... 3.30
- Convert Real to Extended Integer
  - Instruction ..... 3.37
- Convert Real to Integer Instruction ..... 3.38
- COPY Directive ..... 5.3.5, 5.3.5.3
- Copy Source (COPY) Directive ..... 5.3.5, 5.3.5.3
- Count Ones Instruction ..... 3.34
- CRC Byte String Format ..... T3-3
- CRC Instruction ..... 3.36
- CRE Instruction ..... 3.37
- CRI Instruction ..... 3.38
- Cross-Reference Listing ..... 9.4
- CRU ..... 4.5.2
  - Bit Addressing ..... 3.2.7
  - Error Interrupt Status Register ..... 4.5.1.3
  - Instructions ..... 3.4.5

- CS Instruction ..... 3.39
- CSEG Directive ..... 5.3.1, 5.3.1.9
- Cyclic Redundancy Code Calculation
  - Instruction ..... 3.36
- CZC Instruction ..... 3.40
- DATA:
  - Directive ..... 2.4.2, 5.3.3, 5.3.3.2
  - Segment ..... 5.3.1.7
  - Segment (DSEG) Directive ... 5.3.1, 5.3.1.7
  - Segment End (DEND)
    - Directive ..... 5.3.1, 5.3.1.7, 5.3.1.8
    - Types ..... 2.8
- DBC Instruction ..... 3.41
- DD Instruction ..... 3.42
- DEC Instruction ..... 3.43, 4.2.4.2
- Decimal ASCII to Binary Conversion
  - Instruction ..... 3.41
- Decimal Integer Constants ..... 2.5.1
- Decrement by Two Instruction ... 3.44, 3.44.1
- Decrement Instruction ..... 3.43, 3.43.1
- DECT Instruction ..... 3.44, 3.44.1
- DEF Directive ..... 5.3.4, 5.3.4.1, 8.3, 8.3.2
- Define Assembly-Time
  - Constant (EQU) Directive ... 5.3.3, 5.3.3.4
- Define Extended Operation (DXOP)
  - Directive ..... 5.3.5, 5.3.5.1
- Define Operation (DFOP)
  - Directive ..... 5.3.5, 5.3.5.5
- Definition (DEF) Directive,
  - External ..... 5.3.4, 5.3.4.1, 8.3, 8.3.2
- DEND Directive ..... 5.3.1, 5.3.1.7, 5.3.1.8
- DFOP Directive ..... 5.3.5, 5.3.5.5, 7.6.1
- Differences:
  - Execution ..... 4.3.2
  - Performance ..... 4.3.3
- DINT Instruction ..... 3.45
- Directive:
  - Absolute Origin (AORG) .... 5.3.1, 5.3.1.1
  - AORG ..... 5.3.1, 5.3.1.1
  - ASMELS ..... 5.3.5, 5.3.5.4
  - ASMEND ..... 5.3.5, 5.3.5.4
  - ASMIF ..... 5.3.5, 5.3.5.4
  - BES ..... 5.3.1, 5.3.1.5
- Block:
  - Ending With Symbol
    - (BES) ..... 5.3.1, 5.3.1.5
  - Starting With Symbol
    - (BSS) ..... 5.3.1, 5.3.1.4
- BYTE ..... 2.4.2, 5.3.3, 5.3.3.5
- CEND ..... 5.3.1, 5.3.1.9, 5.3.1.10
- Checkpoint Register (CKPT) .. 5.3.3, 5.3.3.5
- CKPT ..... 5.3.3, 5.3.3.5
- Common:
  - Segment (CSEG) ..... 5.3.1, 5.3.1.9
  - Segment End
    - (CEND) ..... 5.3.1, 5.3.1.9, 5.3.1.10
- Conditional Assembly
  - (ASMELS) ..... 5.3.5, 5.3.5.4
- Conditional Assembly
  - (ASMEND) ..... 5.3.5, 5.3.5.4
- Conditional Assembly
  - (ASMIF) ..... 5.3.5, 5.3.5.4
- COPY ..... 5.3.5, 5.3.5.3
- Copy Source (COPY) ..... 5.3.5, 5.3.5.3
- CSEG ..... 5.3.1, 5.3.1.9
- DATA ..... 2.4.2, 5.3.3, 5.3.3.2
  - Segment (DSEG) ..... 5.3.1, 5.3.1.7
  - Segment End
    - (DEND) ..... 5.3.1, 5.3.1.7, 5.3.1.8
- DEF ..... 5.3.4, 5.3.4.1, 8.3, 8.3.2
- Define Assembly-Time Constant
  - (EQU) ..... 5.3.3, 5.3.3.4
- Define Extended Operation
  - (DXOP) ..... 5.3.5, 5.3.5.1
- Define Operation (DFOP) .... 5.3.3, 5.3.5.5
- DEND ..... 5.3.1, 5.3.1.7, 5.3.1.8
- DFOP ..... 5.3.5, 5.3.5.5, 7.6.1
- DORG ..... 5.3.1, 5.3.1.3
- DSEG ..... 5.3.1, 5.3.1.7
- Dummy Origin (DORG) ..... 5.3.1, 5.3.1.3
- DXOP ..... 2.6, 5.3.5, 5.3.5.1, 7.6.1
- EQU ..... 2.5.4, 5.3.3, 5.3.3.4
- EVEN ..... 2.4.2, 5.3.1, 5.3.1.6
- External:
  - Definition (DEF) .. 5.3.4, 5.3.4.1, 8.3, 8.3.2
  - Reference (REF) .. 5.3.4, 5.3.4.2, 8.3, 8.3.1
- Force Load (LOAD) ..... 5.3.4, 5.3.4.4
- IDT ..... 5.3.2, 5.3.2.2, 8.3, 8.3.3
- Initialize:
  - Byte (BYTE) ..... 5.3.3, 5.3.3.1
  - Text (TEXT) ..... 5.3.3, 5.3.3.3
  - Word (DATA) ..... 5.3.3, 5.3.3.2
- LIBIN ..... 7.6, 7.6.2
- LIBOUT ..... 7.6, 7.6.1
- LIST ..... 5.3.2, 5.3.2.4
  - Source (LIST) ..... 5.3.2, 5.3.2.4
- LOAD ..... 5.3.4, 5.3.4.4
- Macro:
  - Library Attachment (LIBIN) .... 7.6, 7.6.2
  - Library Declaration (LIBOUT) .. 7.6, 7.6.1
- No Source List (UNL) ..... 5.3.2, 5.3.2.5
- OPTION ..... 5.3.2, 5.3.2.1
- Output Options (OPTION) .... 5.3.2, 5.3.2.1
- PAGE ..... 5.3.2, 5.3.2.6
- Page Eject (PAGE) ..... 5.3.2, 5.3.2.6
- Page Title (TITL) ..... 5.3.2, 5.3.2.3
- PEND ..... 5.3.1, 5.3.1.11, 5.3.1.12, 5.3.5, 5.3.5.2
- Program:
  - End (PEND) ..... 5.3.5, 5.3.5.2
  - Identifier (IDT) ... 5.3.2, 5.3.2.2, 8.3, 8.3.3
  - Segment End
    - (PEND) ..... 5.3.1, 5.3.1.11, 5.3.1.12
    - Segment (PSEG) ..... 5.3.1, 5.3.1.11
- PSEG ..... 5.3.1, 5.3.1.11
- REF ..... 2.7.1, 5.3.4, 5.3.4.2, 8.3, 8.3.1
- Relocatable Origin (RORG) ... 5.3.1, 5.3.1.2
- RORG ..... 5.3.1, 5.3.1.2

- Secondary External Reference (SREF) . . . . . 5.3.4, 5.3.4.3
- Set Maximum Macro Nesting Level (SETMNL) . . . . . 5.3.5, 5.3.5.6
- SETMNL . . . . . 5.3.5, 5.3.5.6
- SREF . . . . . 5.3.4, 5.3.4.3
- TEXT . . . . . 2.4.2, 5.3.3, 5.3.3.3
- TITL . . . . . 5.3.2, 5.3.2.3
- UNL . . . . . 5.3.2, 5.3.2.5
- Word Boundary EVEN . . . . . 5.3.1, 5.3.1.6
- Workspace Pointer (WPNT) . . . . . 5.3.3, 5.3.3.6
- WPNT . . . . . 5.3.3, 5.3.3.6
- Directives:
  - Assembler . . . . . 5.3, 5.3.1, 5.3.2, 5.3.3, 5.3.4, 5.3.5
  - Macro Library Assembler . . . . . 7.6
  - Miscellaneous . . . . . 5.3.5
  - Disable Interrupts Instruction . . . . . 3.45
  - DIV Instruction . . . . . 3.46
  - Divide Double Precision Real Instruction . . . . . 3.42
  - Divide:
    - Instruction . . . . . 3.46
    - Operation . . . . . 1.4.5
  - Divide Real Instruction . . . . . 3.48
  - Divide Signed Instruction . . . . . 3.47
  - DIVS Instruction . . . . . 3.47
  - DORG Directive . . . . . 5.3.1, 5.3.1.3
  - Double Precision Real Numbers . . . . . 2.8.7
  - DR Instruction . . . . . 3.48
  - DSEG Directive . . . . . 5.3.1, 5.3.1.7
  - Dummy Origin (DORG)
    - Directive . . . . . 5.3.1, 5.3.1.3
  - DXOP Directive . . . . . 2.6, 5.3.5, 5.3.5.1, 7.6.1
  - D\$DFX\$ . . . . . 7.6, 7.6.1
- Editor, Link . . . . . 8.3, 8.3.4
- EINT Instruction . . . . . 3.49
- Elements, Macro Language . . . . . 7.5
- EMD Instruction . . . . . 3.50
- Enable Interrupts Instruction . . . . . 3.49
- Enabled Bit Memory Management and Protection . . . . . 1.4.10
- End-of-File Statement . . . . . 2.4
- Entry Point, Subroutine . . . . . 1.2.4, 1.2.5.2
- EP Instruction . . . . . 3.51
- EQU Directive . . . . . 2.5.4, 5.3.3, 5.3.3.4
- Equal Bit . . . . . 1.4.3
- Error:
  - Access Name . . . . . 9.3
  - Arithmetic . . . . . 1.4.11
  - Interrupt:
    - Conditions, System . . . . . 4.5.1.2
    - Handling Routine . . . . . 4.5.1.2
    - Mapping . . . . . 4.3.2.5
    - Messages, SDSMAC . . . . . 9.3, T9-1
    - Status Register, CRU . . . . . 4.5.1.3, 4.5.1.4
    - System . . . . . 4.5.1.2, 4.5.1.3
    - Trace Control and Map
      - Control Register . . . . . 4.5.1.4, 4.5.1.5, 4.5.1.6, 4.5.1.7
      - Trace Memory . . . . . 4.5.1.4, 4.5.1.5
- Error Interrupt Status Register . . . . . 4.3.2.6
- Evaluating Expressions . . . . . 2.5.1, 2.5.2, 2.5.4
- Even Address Byte . . . . . 1.2.2
- EVEN Directive . . . . . 2.4.2, 5.3.1, 5.3.1.6
- Examples:
  - Macro . . . . . 7.7
  - Programming . . . . . 4.2
- Exclusive OR Instruction . . . . . 3.147
- Exclusive OR Multiple Precision Instruction . . . . . 3.148
- Execute Instruction . . . . . 3.143, 4.3.7.4
- Execute Micro-Diagnostic Instruction . . . . . 3.50
- Execution Differences . . . . . 4.3.2
- EXIT . . . . . 7.7.2
- Exit From Floating Point Interpreter
  - Instruction . . . . . 3.145
- Expander, Macro . . . . . 7.2, 7.5.4.2, 7.5.6.1
- Expressions . . . . . 2.7
  - Arithmetic . . . . . 2.7.2
  - Evaluating . . . . . 2.7.2, 2.7.3, 2.7.5
  - Well-Defined . . . . . 2.7
- Extend Precision Instruction . . . . . 3.4.23, 3.51
- Extended Operation . . . . . 1.2.4, 1.4.12
  - Bit . . . . . 1.4.7
  - Instruction . . . . . 3.4.10, 3.146
- Extended Operation (DXOP) Directive, Define . . . . . 5.3.5, 5.3.5.1
- Extended Operations . . . . . 3.146.1
- External:
  - Definition (DEF)
    - Directive . . . . . 5.3.4, 5.3.4.1, 8.3, 8.3.2
  - Reference (REF)
    - Directive . . . . . 5.3.4, 5.3.4.2, 8.3, 8.3.1
    - Reference (SREF) Directive, Secondary . . . . . 5.3.4, 5.3.4.3
- Extract Field Instruction . . . . . 3.144
- Extract Value Instruction . . . . . 3.149
- Field:
  - Comment . . . . . 2.4.5, 3.5.4
  - Instructions . . . . . 3.4.18
  - KEY . . . . . 9.2
  - Label . . . . . 2.4.2, 3.5.4
  - LINE . . . . . 9.2
  - NAME . . . . . 9.2
  - Operand . . . . . 2.4.4
  - Operation . . . . . 2.4.3
- Floating Point Accumulator FPA . . . . . 2.8.8
- Force Load (LOAD) Directive . . . . . 5.3.4, 5.3.4.4
- Forced Interrupt . . . . . 4.5.1.6
- Format:
  - Machine Language . . . . . 9.5, 9.5.2
  - Source Statement . . . . . 2.4
- Format 1 Instructions . . . . . 3.4.1
- Format 2 Instructions . . . . . 3.4.2, 3.4.3
- Format 3 Instructions . . . . . 3.4.4
- Format 4 Instructions . . . . . 3.4.5
- Format 5 Instructions . . . . . 3.4.6
- Format 6 Instructions . . . . . 3.4.7
- Format 7 Instructions . . . . . 3.4.8

- Format 8 Instructions . . . . . 3.4.9
- Format 9 Instructions . . . . . 3.4.10, 3.4.11
- Format 10 Instructions . . . . . 3.4.12
- Format 11 Instructions . . . . . 3.4.13
- Format 12 Instructions . . . . . 3.4.14
- Format 13 Instructions . . . . . 3.4.15
- Format 14 Instructions . . . . . 3.4.16
- Format 15 Instructions . . . . . 3.4.17
- Format 16 Instructions . . . . . 3.4.18
- Format 17 Instructions . . . . . 3.4.19
- Format 18 Instructions . . . . . 3.4.20
- Format 19 Instructions . . . . . 3.4.21
- Format 20 Instructions . . . . . 3.4.22
- Format 21 Instructions . . . . . 3.4.23
- Formats, Instruction . . . . . 3.4
- FPA . . . . . 2.8.8
  - Floating Point Accumulator . . . . . 2.8.8
- GENCMT . . . . . 7.7.5
- GOSUB . . . . . 7.7.1
- Greater Than:
  - Bit:
    - Arithmetic . . . . . 1.4.2
    - Logical . . . . . 1.4.1
- Handling Routine, Error Interrupt . . . . 4.5.1.2
- Hexadecimal Integer Constants . . . . . 2.5.2
- ID . . . . . 7.7.3
- Identifier (IDT) Directive,
  - Program . . . . . 5.3.2, 5.3.2.2, 8.3, 8.3.3
- Idle Instruction . . . . . 3.52
- IDT Directive . . . . . 5.3.2, 5.3.2.2, 8.3, 8.3.3
- Illegal Opcodes . . . . . 4.3.2.3, Appendix J
- Immediate:
  - Addressing . . . . . 3.2.8
  - Instructions . . . . . 3.4.9
- INC Instruction . . . . . 3.53, 3.53.1
- Increment by Two Instruction . . . . . 3.54
- Increment Instruction . . . . . 3.53, 3.53.1
- INCT Instruction . . . . . 3.54
- Indexed Memory Addressing . . . . . 3.2.5
- Indirect:
  - Addressing, Workspace Register . . . 3.2.2
  - Autoincrement Addressing, Workspace Register . . . . . 3.2.3
- Initialize:
  - Byte (BYTE) Directive . . . . . 5.3.3, 5.3.3.1
  - Text (TEXT) Directive . . . . . 5.3.3, 5.3.3.3
  - Word (DATA) Directive . . . . . 5.3.3, 5.3.3.2
- Insert Field Instruction . . . . . 3.55
- INSF Instruction . . . . . 3.55
- Instruction:
  - A . . . . . 3.6
  - AB . . . . . 3.7
  - ABS . . . . . 3.8, 4.2.5.1
  - Absolute Value . . . . . 3.8, 4.2.5.1
  - AD . . . . . 3.9
  - Add Bytes . . . . . 3.7
  - Add Double Precision Real . . . . . 3.9
  - Add Immediate . . . . . 3.10
  - Add Multiple Precision Integer . . . . . 3.11
  - Add Real . . . . . 3.14
  - Add to Register and Jump . . . . . 3.15
  - Add Words . . . . . 3.6
  - Addressing . . . . . T3-2
  - AI . . . . . 3.10
  - AM . . . . . 3.11
  - And Immediate . . . . . 3.12
  - And Multiple Precision . . . . . 3.13
  - ANDI . . . . . 3.12
  - ANDM . . . . . 3.13
  - AR . . . . . 3.14
  - ARJ . . . . . 3.15
  - B . . . . . 3.16
  - BDC . . . . . 3.17
  - Binary to Decimal ASCII Conversion . . 3.17
  - BIND . . . . . 3.18
  - BL . . . . . 3.19, 4.2.1
  - BLSK . . . . . 3.20, 4.2.2
  - BLWP . . . . . 2.3, 3.21, 4.2.3
  - Branch . . . . . 3.16
  - Branch and Link . . . . . 3.19, 4.2.1
  - Branch and Load Workspace Pointer . . . . . 3.21, 4.2.3
  - Branch Immediate and Push Link to Stack . . . . . 3.20, 4.2.2
  - Branch Indirect . . . . . 3.18
  - Byte . . . . . 1.2.1
  - C . . . . . 3.22
  - CB . . . . . 3.23
  - CDE . . . . . 3.24
  - CDI . . . . . 3.25
  - CED . . . . . 3.26
  - CER . . . . . 3.27
  - CI . . . . . 3.28
  - CID . . . . . 3.29
  - CIR . . . . . 3.30
  - CKOF . . . . . 3.31
  - CKON . . . . . 3.32
  - Clear . . . . . 3.33
  - Clock Off . . . . . 3.31
  - Clock On . . . . . 3.32
  - CLR . . . . . 3.33
  - CNTO . . . . . 3.34
  - COC . . . . . 3.35
  - Compare Bytes . . . . . 3.23
  - Compare Immediate . . . . . 3.28
  - Compare Ones Corresponding . . . . . 3.35
  - Compare Strings . . . . . 3.39
  - Compare Words . . . . . 3.22
  - Compare Zeros Corresponding . . . . . 3.40
  - Convert Double Precision Real to Extended Integer . . . . . 3.24
  - Convert Double Precision Real to Integer . . . . . 3.25
  - Convert Extended Integer to Real . . . 3.27
  - Convert Integer to Double Precision Real . . . . . 3.29
  - Convert Integer to Real . . . . . 3.30
  - Convert Real to Extended Integer . . . 3.37
  - Convert Real to Integer . . . . . 3.38

Count Ones	3.34	Jump If Equal	3.58
CRC	3.36	Jump If Greater Than	3.59
CRE	3.37	Jump If High or Equal	3.61
CRI	3.38	Jump If Less Than	3.64
CS	3.39	Jump If Logical High	3.60
Cyclic Redundancy Code		Jump If Logical Low	3.62
Calculation	3.36	Jump If Low or Equal	3.63
CZC	3.40	Jump If No Carry	3.66
DD	3.42	Jump If No Overflow	3.68
DEC	3.43, 3.43.1	Jump If Not Equal	3.67
Decimal ASCII to Binary Conversion	3.41	Jump If Odd Parity	3.70
Decrement	3.43, 3.43.1	Jump On Carry	3.69
Decrement by Two	3.44, 3.44.1	LCS	3.71
DECT	3.44, 3.44.1	LD	3.72
DINT	3.45	LDCR	3.73, 4.5.2.6
Disable Interrupts	3.45	LDD	3.74
DIV	3.46	LDS	3.75
Divide	3.46	Left Test for One	3.83
Divide Double Precision Real	3.42	LI	3.76
Divide Real	3.48	LIM	1.4.13, 3.77
Divide Signed	3.47	LIMI	1.4.13, 3.78
DIVS	3.47	LMF	3.74.1, 3.79
DR	3.48	Load CRU	3.73, 4.5.2
EINT	3.49	Load Double Precision Real	3.72
EMD	3.50	Load Immediate	3.76
Enable Interrupts	3.49	Load Interrupt Mask	3.77
EP	3.51	Load Interrupt Mask Immediate	3.78
Exclusive OR	3.147	Load Memory Map File	3.79
Exclusive OR Multiple Precision	3.148	Load or Restart Execution	3.81
Execute	3.143, 3.143.1	Load Real	3.80
Execute Micro-Diagnostic	3.50	Load Status Register	3.82
Exit from Floating Point Interpreter	3.145	Load Workspace Pointer	
Extend Precision	3.4.23, 3.51	Immediate	3.85, 4.3.3.2
Extended Operation	3.146, 3.146	Load Workspace Pointer Register	3.84
Extract Field	3.144	Load Writable Control Store	3.71
Extract Value	3.149	Long Distance Destination	3.74
Format	3.4	Long Distance Source	3.75
Idle	3.52	LR	3.80
INC	3.53, 3.53.1	LREX	1.2.4, 3.81
Increment	3.53, 3.53.1	LST	3.82
Increment by Two	3.54	LTO	3.83
INCT	3.54	LWP	3.84
Insert Field	3.55	LWPI	3.85, 4.3.3.2
INSF	3.55	MD	3.86
INV	3.56	Memory Map File	3.4.12, 3.74.1
Invert	3.56	MOV	3.87
Invert Order of Field	3.4.17, 3.57	MOVA	3.88
IOF	3.57	MOVB	3.89
JEQ	3.58	Move Address	3.4.21, 3.88
JGT	3.59	Move Byte	3.89
JH	3.60	Move String	3.90
JHE	3.61	Move String from Stack	3.94
JL	3.62	Move String Reverse	3.95
JLE	3.63	Move Word	3.87
JLT	3.64	MOVS	3.90
JMP	3.65	MPY	3.91
JNC	3.66	MPYS	3.92
JNE	3.67	MR	3.93
JNO	3.68	Multiply	3.91
JOC	3.69	Multiply Double Precision Real	3.86
JOP	3.70	Multiply Real	3.93

Multiply Signed	3.92
MVSK	3.94
MVSR	3.95
NEG	3.96
Negate	3.96
Negate Double Precision Real	3.97
Negate Real	3.98
NEGD	3.97
NEGR	3.98
No Operation (NOP)	6.1, 6.2
NOP	6.1, 6.2
Normalize	3.99
NRM	3.99
OR Immediate	3.100
OR Multiple Precision	3.101
ORI	3.100
ORM	3.101
Pop String From Stack	3.102
POPS	3.102
PSHS	3.103
Push String to Stack	3.103
Reset	3.104
Return With Workspace Pointer	3.106
Return Workspace Pointer	4.2.3, 4.3.3.2
Return (RT)	6.1, 6.3
Right Test for One	3.105
RSET	3.104
RT	6.1, 6.3
RTO	3.105
RTWP	3.106, 4.2.3, 4.3.3.2
S	3.107
SB	3.108
SBO	3.109, 4.5.2.3
SBZ	3.110, 4.5.2.4
SD	3.111
Search List Logical Address	3.116
Search List Physical Address	3.117
Search String for Equal Byte	3.112
Search String for Not Equal Byte	3.119
SEQB	3.112
Set CRU Bit to Logic One	3.109
Set CRU Bit to Logic Zero	3.110
Set CRU Bit to One	4.2.5.3
Set CRU Bit to Zero	4.2.5.4
Set Ones Corresponding	3.120
Set Ones Corresponding (BYTE)	3.121
Set to One	3.113
Set Zeros Corresponding	3.136
Set Zeros Corresponding (BYTE)	3.137
SETO	3.113
Shift Left Arithmetic	3.114
Shift Left Arithmetic Multiple	
Precision	3.115
Shift Right Arithmetic	3.123
Shift Right Arithmetic Multiple	
Precision	3.124
Shift Right Circular	3.125
Shift Right Logical	3.127
SLA	3.114
SLAM	3.115
SLSL	3.116
SLSP	3.117
SM	3.118
SNEB	3.119
SOC	3.120
SOCB	3.121
SR	3.122
SRA	3.123
SRAM	3.124
SRC	3.125
SRJ	3.126
SRL	3.127
STCR	3.128, 4.5.2.7
STD	3.129
Store CRU	3.128, 4.5.2.7
Store Double Precision Real	3.129
Store Program Counter	3.130
Store Real	3.131
Store Status	3.132
Store Workspace Pointer	3.133
STPC	3.130
STR	3.131
STST	3.132
STWP	3.133
Subtract Bytes	3.108
Subtract Double Precision Real	3.111
Subtract From Register and Jump	3.126
Subtract Multiple Precision Integer	3.118
Subtract Real	3.122
Subtract Words	3.107
Swap Bytes	3.134
Swap Multiple Precision	3.135
SWPB	3.134
SWPM	3.135
Symbol Attributes	7.4.6, 9.4, T9-2
SZC	3.136
SZCB	3.137
TB	3.138, 4.5.2.5
TCMB	3.139, 4.2.5.2
Test and Clear Memory Bit	3.139, 4.2.5.2
Test and Set Memory Bit	3.142, 4.2.5.2
Test Bit	3.138, 4.5.2.5
Test Memory Bit	3.140
TMB	3.140
Transfer Vector (XVEC)	1.2.4, 6.1, 6.4
Translate String	3.141
TS	3.141
TSMB	3.142, 4.2.5.2
Unconditional Jump	3.65
X	3.143, 3.143.1
XF	3.144
XIT	3.145
XOP	3.146, 3.146.1
XOR	3.147
XORM	3.148
XV	3.149
XVEC	6.1, 6.4
Instructions:	
Alter Registers and Jump	3.4.19
Assembly Language	3.1
Bit I/O	3.4.3
Bit Testing	3.4.16

CRU	3.4.5	12 Millisecond Test Clock	4.5.1.5
Extended Operation	3.4.10	Interrupts	1.2.5, 4.5.1
Field	3.4.18	INV Instruction	3.56
Format 1	3.4.1	Invert Instruction	3.56
Format 2	3.4.2, 3.4.3	Invert Order of Field Instruction	3.4.17, 3.57
Format 3	3.4.4	IOF Instruction	3.57
Format 4	3.4.5		
Format 5	3.4.6	JEQ Instruction	3.58
Format 6	3.4.7	JGT Instruction	3.59
Format 7	3.4.8	JH Instruction	3.60
Format 8	3.4.9	JHE Instruction	3.61
Format 9	3.4.10, 3.4.11	JL Instruction	3.62
Format 10	3.4.12	JLE Instruction	3.63
Format 11	3.4.13	JLT Instruction	3.64
Format 12	3.4.14	JMP Instruction	3.65
Format 13	3.4.15	JNC Instruction	3.66
Format 14	3.4.16	JNE Instruction	3.67
Format 15	3.4.17	JNO Instruction	3.68
Format 16	3.4.18	JOC Instruction	3.69
Format 17	3.4.19	JOP Instruction	3.70
Format 18	3.4.20	Jump If Equal Instruction	3.58
Format 19	3.4.21	Jump If Greater Than Instruction	3.59
Format 20	3.4.22	Jump If High or Equal Instruction	3.61
Format 21	3.4.23	Jump If Less Than Instruction	3.64
Immediate	3.4.9	Jump If Logical High Instruction	3.60
Jump	3.4.2	Jump If Logical Low Instruction	3.62
List Search	3.4.22	Jump If No Carry Instruction	3.66
Logical	3.4.4	Jump If No Overflow Instruction	3.68
Multiple Precision	3.4.13	Jump If Not Equal Instruction	3.67
Multiple Precision Shift	3.4.15	Jump If Odd Parity Instruction	3.70
Multiply and Divide	3.4.11	Jump Instructions	3.4.2
Register Shift	3.4.6	Jump On Carry Instruction	3.69
Single Address	3.4.7		
Single Register Operand	3.4.20	KEY Field	9.2
String	3.4.14	Keywords	7.4.7
Two Address	3.4.1	Parameter Attribute	7.4.7.1
Without Operands	3.4.8	Symbol Attribute	7.4.7.2
Integer Constants:			
Decimal	2.5.1	Label	2.4
Hexadecimal	2.5.2	Field	2.4.2
Integers, Multiple Precision	2.8.2	Labels	7.4.1
Interrupt	1.2.4, 1.2.5	Language:	
Bit, Overflow	1.4.11	Elements, Macro	7.4
Conditions, System Error	4.5.1.2	Format, Machine	9.5, 9.5.2
Forced	4.5.1.7	LCS Instruction	3.71
Handling Routine, Error	4.5.1.2	LD Instruction	3.72
Level	1.2.5, 1.4.13, 4.5.1.1	LDCR Instruction	3.73, 4.5.2
Line Frequency Clock	4.5.1.1	LDD Instruction	3.74
Mask	1.4.13	LDS Instruction	3.75
Power Failing	1.4.13	Least Significant Bit (LSB)	1.2.1, 1.2.2
Power-On	1.4.13	Left Shift, Arithmetic	1.4.5
Predefined	4.5.1.1	Left Test for One Instruction	3.83
Priority	1.4.13	Length:	
Sequence	1.2.5.1	Component	7.4.5.1
Status Register, CRU Error	4.3.2.6	Source Word	2.4
Subroutine	4.5.1.2, 4.5.3.6	Level, Interrupt	1.2.5, 1.4.13, 4.5.1
System Error	4.5.1.2	LI Instruction	3.76
Trace Control and Map Control		LIBIN Directive	7.6, 7.6.2
Register, Error	4.5.1.3	LIBOUT Directive	7.6, 7.6.1
Trace Memory, Error	4.5.1.3	Library:	
Transfer Vector	1.2.4	Assembler Directives, Macro	7.6

- Attachment (LIBIN) Directive,
  - Macro ..... 7.6, 7.6.2
- Declaration (LIBOUT) Directive,
  - Macro ..... 7.6, 7.6.1
  - Macro ..... 7.3, 7.4, 7.6
  - Management, Macro ..... 7.6, 7.6.3
- LIM Instruction ..... 1.4.13, 3.77
- LIMI Instruction ..... 1.4.13, 3.78
- Limit Register ..... 3.74.1
- LINE Field ..... 9.2
- Line-Frequency-Clock Interrupt ..... 4.5.1
- Link Editor ..... 8.3, 8.3.4
- Linkage, Program ..... 5.3.4
- Linking, Program ..... 8.3
- List ..... 2.8.5, 7.7.8
  - Directive ..... 5.3.2, 5.3.2.4
  - Search Control Block (LSCB) ..... 2.8.4
- List Search Instructions ..... 3.4.22
- List Source (LIST) Directive ..... 5.3.2, 5.3.2.4
- Listing:
  - Cross Reference ..... 9.4
  - Source ..... 9.2
- LMF Instruction ..... 3.74.1, 3.79
- LOAD ..... 7.7.6
- Load CRU Instruction ..... 3.73, 4.5.2.1
- LOAD Directive ..... 5.3.4, 5.3.4.4
- Load Double Precision Real
  - Instruction ..... 3.72
- Load Immediate Instruction ..... 3.76
- Load Interrupt Mask Immediate
  - Instruction ..... 3.78
- Load Interrupt Mask Instruction ..... 3.77
- Load (LOAD) Directive, Force ..... 5.3.4, 5.3.4.4
- Load Memory Map File Instruction ..... 3.79
- Load or Restart Execution Instruction ..... 3.81
- Load Workspace Pointer Immediate
  - Instruction ..... 3.85, 4.2.3
- Load Workspace Pointer Register
  - Instruction ..... 3.84
- Load Writable Control Store
  - Instruction ..... 3.71
- Location Counter ..... 5.3.1, 10.2
- Logical:
  - Greater Than Bit ..... 1.4.1
  - Instructions ..... 3.4.4
  - Operators ..... 2.5.3
- Logical Operators in Expressions ..... 2.5.3
- Long Distance Destination Instruction ..... 3.74
- Long Distance Source Instruction ..... 3.75
- LR Instruction ..... 3.80
- LREX Instruction ..... 3.81
- LSB, Least Significant Bit ..... 1.2.1, 1.2.2
- LSCB, List Search Control Block ..... 2.8.5
- LST Instruction ..... 3.82
- LTO Instruction ..... 3.83
- LWP Instruction ..... 3.84
- LWPI Instruction ..... 3.85, 4.2.3
  
- Machine Language Format ..... 9.5, 9.5.2
- Macro:
  - Assembler ..... Appendix G
  - Examples ..... 7.7
  - Expander ..... 7.2
  - Language Elements ..... 7.5
  - Library ..... 7.3, 7.4, 7.6
    - Assembler Directives ..... 7.6
    - Attachment (LIBIN) Directive ..... 7.6, 7.6.2
    - Declaration (LIBOUT)
      - Directive ..... 7.6, 7.6.1
      - Management ..... 7.6, 7.6.3
  - Macro Instructions ..... 5.2
  - Macro Language ..... 7.1
  - Macro Processing ..... 7.2
  - Macro Symbol Table ..... 7.4.5.1
  - Macro Translator ..... 7.3
- Management:
  - and Protection Enabled Bit,
    - Memory ..... 1.4.10
  - Macro Library ..... 7.6, 7.6.3
- Map Diagnostic Hardware ..... 4.3.2.8
- Mapping:
  - Error ..... 4.3.2.5
  - Memory ..... 3.74.1
  - Registers ..... 3.74.1
- Mask, Interrupt ..... 1.4.13
- MD Instruction ..... 3.86
- Memory:
  - Address ..... 1.2.1
  - Trace ..... 4.5.1.3
  - Bipolar ..... 1.3
  - Mapping ..... 3.74.1
  - Organization ..... 1.2.3
  - Protection ..... 3.74.1
  - Word ..... 1.2.2
- Memory Cache ..... 4.3.1
- Memory, Error Interrupt Trace ..... 4.5.1.3
- Memory File Bit ..... 1.4.9
- Memory Management and Protection
  - Enabled Bit ..... 1.4.10
- Memory Map File Instruction ..... 3.4.12
- Messages:
  - Abnormal Completion ..... T9-1
  - SDSMAC:
    - Error ..... 9.3, T9-1
    - Warning ..... 9.3, T9-1
  - Warning ..... 9.2
- Miscellaneous Directives ..... 5.3.5
- Mode:
  - Addressing ..... 3.5.2
  - Privileged ..... 1.4.8
- Model Statements ..... 7.4.13
- Modes, Addressing ..... 3.2, T3-1
- Most Significant Bit (MSB) ..... 1.2.1, 1.2.2
- MOV Instruction ..... 3.87
- MOVA Instruction ..... 3.88
- MOVB Instruction ..... 3.89
- Move Address Instruction ..... 3.4.21, 3.88
- Move Byte Instruction ..... 3.89
- Move String From Stack Instruction ..... 3.94
- Move String Instruction ..... 3.90
- Move String Reverse Instruction ..... 3.95
- Move Word Instruction ..... 3.87



- MOVS Instruction ..... 3.90
- MPY Instruction ..... 3.91
- MPYS Instruction ..... 3.92
- MR Instruction ..... 3.93
- MSB, Most Significant Bit ..... 1.2.1, 1.2.2
- Multiple Precision:
  - Instructions ..... 3.4.13
  - Integers ..... 2.8.2
- Multiple Precision Shift Instructions .. 3.4.15
- Multiply and Divide Instructions ..... 3.4.11
- Multiply Double Precision Real
  - Instruction ..... 3.86
- Multiply Instruction ..... 3.91
- Multiply Real Instruction ..... 3.93
- Multiply Signed Instruction ..... 3.92
- MVSK Instruction ..... 3.94
- MVSR Instruction ..... 3.95
  
- NAME Field ..... 9.2
- NEG Instruction ..... 3.96
- Negate:
  - Instruction ..... 3.96
  - Operation ..... 1.4.5
- Negate Double Precision Real
  - Instruction ..... 3.97
- Negate Real Instruction ..... 3.98
- NEGD Instruction ..... 3.97
- NEGR Instruction ..... 3.98
- No Operation (NOP) Instruction ..... 6.1, 6.2
- No Source List (UNL) Directive .. 5.3.2, 5.3.2.5
- NOP Instruction ..... 6.1, 6.2
- Normalization ..... 2.8.7
- Normalize Instruction ..... 3.99
- NRM Instruction ..... 3.99
- Number, Source Statement ..... 9.2
- Numbers, Real ..... 2.8.6, 2.8.7
  
- Object Code ..... 9.2, 9.5, 9.5.4
- Odd Address Byte ..... 1.2.2
- Odd Parity Bit ..... 1.4.6
- Opcode ..... 2.4, 3.5.1, 3.5.4
- Operand
  - Field ..... 2.4.4
- Operand List ..... 7.5.1
- Operation:
  - Addition ..... 1.4.5
  - Divide ..... 1.4.5
  - Field ..... 2.4.3
  - Negate ..... 1.4.5
  - Subtraction ..... 1.4.5
- Operators ..... 7.5, 7.5.3
  - Logical ..... 2.7.3
  - Relational ..... 2.7.4
- OPTION Directive ..... 5.3.2, 5.3.2.1
- Options (OPTION) Directive,
  - Output ..... 5.3.2, 5.3.2.1
- Or Immediate Instruction ..... 3.100
- Or Multiple Precision Instruction ..... 3.101
- Organization, Memory ..... 1.2.3
- ORI Instruction ..... 3.100
  
- Origin:
  - (AORG) Directive, Absolute ... 5.3.1, 5.3.1.1
  - (DORG) Directive, Dummy ... 5.3.1, 5.3.1.3
  - (RORG) Directive,
    - Relocatable ..... 5.3.1, 5.3.1.2
- ORM Instruction ..... 3.101
- Output:
  - Assembler ..... 5.3.2
  - Options (OPTION) Directive .. 5.3.2, 5.3.2.1
- Overflow:
  - Bit ..... 1.4.5
  - Conditions ..... 2.7.2
  - Interrupt Bit ..... 1.4.11
  
- PAGE Directive ..... 5.3.2, 5.3.2.6
- Page Eject (PAGE) Directive .... 5.3.2, 5.3.2.6
- Page Title (TITL) Directive ..... 5.3.2, 5.3.2.3
- Parameter Attribute:
  - Component ..... 7.4.7.1
  - Keywords ..... 7.4.7.2
  - Parameters ..... 7.4.5, 7.4.7.2
- PEND Directive ..... 5.3.1, 5.3.1.11, 5.3.1.12, 5.3.5, 5.3.5.2
- Performance Differences ..... 4.3.3
- Pop String From Stack Instruction ..... 3.102
- POPS Instruction ..... 3.102
- Power Failing Interrupt ..... 4.5.1.1
- Power-On Interrupt ..... 4.5.1.1
- Predefined:
  - Interrupt ..... 4.5.1.1
  - Symbols ..... 2.7.1
- Priority, Interrupt ..... 1.2.5.1, 1.2.5.2, 4.5.1
- Privileged Mode ..... 1.4.8
- Privileged Mode Bit ..... 1.4.8
- Program:
  - End (PEND) Directive ..... 5.3.5, 5.3.5.2
  - Identifier (IDT)
    - Directive ..... 5.3.2, 5.3.2.2, 8.3, 8.3.3
  - Linkage ..... 5.3.4
  - Linking ..... 8.3
  - Segment ..... 5.3.1.11
  - Segment End (PEND)
    - Directive ..... 5.3.1, 5.3.1.11, 5.3.1.12
  - Segment (PSEG) Directive ... 5.3.1, 5.3.1.11
  - Program Counter ..... 1.2.4, 1.2.5
  - Relative:
    - Address ..... 3.2.4, 3.2.6
    - Addressing ..... 3.2.6
- Programming Examples ..... 4.2
- Protection:
  - Enabled Bit, Memory Management
    - and ..... 1.4.10
    - Memory ..... 3.74.1
  - PSEG Directive ..... 5.3.1, 5.3.1.11
  - Pseudo-Instructions ..... 6.1
  - PSHS Instruction ..... 3.103
  - PSHS or POPS Representation ..... F3-3
  - Push String to Stack Instruction ..... 3.103
- Qualifiers, Variable ..... 7.4.5.3

- Real Numbers ..... 2.8.6, 2.8.7
  - Double Precision ..... 2.8.7
  - Single Precision ..... 2.8.6
- Reentrant Programming ..... 4.4
- Reexecutable Instructions ..... 2.9
- REF Directive ..... 2.5.1, 5.3.4, 5.3.4.2, 8.3, 8.3.1
- Reference Listing, Cross ..... 9.4
- Reference (REF) Directive,
  - External ..... 5.3.4, 5.3.4.2, 8.3, 8.3.1
- Reference (SREF) Directive,
  - Secondary External ..... 5.3.4, 5.3.4.3
- Register:
  - Bias ..... 3.74.1
  - Breakpoint ..... 4.5.1.4
  - Cache, Workspace ..... 1.3
  - Clear Status ..... 1.4.10
  - CRU Error Interrupt Status ..... 4.5.1.2
  - Error Interrupt Status ..... 4.3.2.6
  - Error Interrupt Trace Control and
    - Map Control ..... 4.5.1.3, 4.5.1.4, 4.5.1.5, 4.5.1.6
  - Limit ..... 3.74.1
  - Status ..... 1.4
  - Workspace Pointer ..... 1.2.7
- Register Shift Instructions ..... 3.4.6
- Registers, Mapping ..... 3.74.1
- Registers, Mapping ..... 3.74.1
- Relational Operators ..... 2.7.4
- Relational Operators in Expressions ..... 2.7.4
- Relocatability:
  - of Expressions ..... 8.2.1
  - of Source Statement Elements ..... 8.2.1
- Relocatable:
  - Code ..... 8.2
  - Constants ..... 2.5
  - Origin (RORG) Directive ..... 5.3.1, 5.3.1.2
  - Symbols ..... 2.6
- Relocation Capability ..... 8.2
- Reset Instruction ..... 3.104
- Return With Workspace Pointer
  - Instruction ..... 3.106
- Return Workspace Pointer Instruction ..... 4.2.3
- Return (RT) Instruction ..... 6.1, 6.3
- Right Test for One Instruction ..... 3.105
- RORG Directive ..... 5.3.1, 5.3.1.2
- RSET Instruction ..... 3.104
- RT Instruction ..... 6.1, 6.3
- RTO Instruction ..... 3.105
- RTWP Instruction ..... 3.106, 4.2.3
- S Instruction ..... 3.107
- SB Instruction ..... 3.108
- SBO Instruction ..... 3.109, 4.5.2.3
- SBZ Instruction ..... 3.110, 4.5.2.4
- SD Instruction ..... 3.111
- SDSMAC:
  - Assembler ..... 5.2
  - Character Set ..... 2.4.1
  - Error Messages ..... 9.3, T9-1
  - Warning Messages ..... 9.3, T9-1
- Search Control Block (LSCB) List ..... 2.8.5
- Search List Logical Address
  - Instruction ..... 3.116
- Search List Physical Address
  - Instruction ..... 3.117
- Search String for Equal Byte
  - Instruction ..... 3.112
- Search String for Not Equal Byte
  - Instruction ..... 3.119
- Search Termination Conditions ..... T3-6
- Second Word Modification ..... 4.3.2.2
- Secondary External Reference (SREF)
  - Directive ..... 5.3.4, 5.3.4.3
- Segment:
  - Common ..... 5.3.9
  - Component ..... 7.4.5.3
  - Data ..... 5.3.1.7
- Segment (CSEG) Directive,
  - Common ..... 5.3.1, 5.3.1.9
- Segment (DSEG) Directive,
  - Data ..... 5.3.1, 5.3.1.7
- Segment End (CEND) Directive,
  - Common ..... 5.3.1, 5.3.1.9, 5.3.1.10
- Segment End (DEND) Directive,
  - Data ..... 5.3.1, 5.3.1.7, 5.3.1.8
- Segment End (PEND) Directive,
  - Program ..... 5.3.1, 5.3.1.11, 5.3.1.12
- Segment, Program ..... 5.3.1.11
- Segment (PSEG) Directive,
  - Program ..... 5.3.1, 5.3.1.11
- SEQB Instruction ..... 3.112
- Sequence, Interrupt ..... 1.2.5.1
- Set CRU Bit to Logic One Instruction ..... 3.109
- Set CRU Bit to Logic Zero Instruction ..... 3.110
- Set CRU Bit to One Instruction ..... 4.5.2.3
- Set CRU Bit to Zero Instruction ..... 4.5.2.4
- Set Maximum Macro Nesting Level
  - (SETMNL) Directive ..... 5.3.5, 5.3.5.6
- Set Ones Corresponding Instruction ..... 3.120
- Set Ones Corresponding (Byte)
  - Instruction ..... 3.121
- Set to One Instruction ..... 3.113
- Set Zeros Corresponding (BYTE)
  - Instruction ..... 3.137
- Set Zeros Corresponding Instruction ..... 3.136
- SETMNL Directive ..... 5.3.5, 5.3.5.6
- SETO Instruction ..... 3.113
- Shift, Arithmetic Left ..... 1.4.5
- Shift Left Arithmetic Instruction ..... 3.114
- Shift Left Arithmetic Multiple Precision
  - Instruction ..... 3.115
- Shift Right Arithmetic Instruction ..... 3.123
- Shift Right Arithmetic Multiple
  - Precision Instruction ..... 3.124
- Shift Right Circular Instruction ..... 3.125
- Shift Right Logical Instruction ..... 3.127
- Single Address Instructions ..... 3.4.7
- Single Precision Real Numbers ..... 2.8.6
- Single Register Operand
  - Instructions ..... 3.4.20
- SLA Instruction ..... 3.114
- SLAM Instruction ..... 3.115

- SLSL Instruction .....3.116
- SLSP Instruction .....3.117
- SM Instruction .....3.118
- SNEB Instruction .....3.119
- SOC Instruction .....3.120
- SOCB Instruction .....3.121
- Source:
  - Listing .....9.2
  - Statement .....9.2
    - Format .....2.4
  - Source (LIST) Directive, List ....5.3.2, 5.3.2.4
  - Source Statement Number .....9.2
  - Source Word Length .....2.4
- SR Instruction .....3.122
- SRA Instruction .....3.123
- SRAM Instruction .....3.124
- SRC Instruction .....3.125
- SREF Directive .....5.3.4, 5.3.4.3
- SRJ Instruction .....3.126
- SRL Instruction .....3.127
- Stack .....2.8.4
- Starting with Symbol (BSS) Directive,
  - Block .....5.3.1, 5.3.1.4
- Statement:
  - Comment .....2.4
  - End-of-File .....2.4
  - Format, Source .....2.4
  - Source .....9.2
- Status:
  - Register .....1.4
    - Clear .....1.4.10
    - CRU Error Interrupt .....4.5.1.3
- STCR Instruction .....3.128, 4.5.2.7
- STD Instruction .....3.129
- Store CRU Instruction .....3.128, 4.5.2.7
- Store Double Precision Real
  - Instruction .....3.129
- Store Program Counter Instruction ....3.130
- Store Real Instruction .....3.131
- Store Status Instruction .....3.132
- Store Workspace Pointer Instruction ..3.133
- STPC Instruction .....3.130
- STR Instruction .....3.131
- String:
  - Byte .....2.8.3
  - Character .....2.6.2
  - Component .....7.4.5.1, 7.4.6.1
  - Instructions .....3.4.14
  - Tagged .....2.8.3
- Strings .....7.4.2
- STST Instruction .....3.132
- STWP Instruction .....3.133
- Subroutine:
  - Entry Point .....1.2.4, 1.2.5.2
  - Interrupt .....1.2.4, 1.2.5.2
- Subroutines .....4.2
- Subtract Bytes Instruction .....3.108
- Subtract Double Precision Real
  - Instruction .....3.111
- Subtract From Register and Jump
  - Instruction .....3.126
- Subtract Multiple Precision Integer
  - Instruction .....3.118
- Subtract Real Instruction .....3.122
- Subtract Words Instruction .....3.107
- Subtraction Operation .....1.4.5
- Swap Bytes Instruction .....3.134
- Swap Multiple Precision Instruction ...3.135
- SWPB Instruction .....3.134
- SWPM Instruction .....3.135
- Symbol:
  - Attributes .....9.4, T9-2
  - Table .....7.4.6, 9.5, 9.5.3
    - Macro .....7.4.5.1, 7.4.6
- Symbol Attribute:
  - Component .....7.4.6, 7.4.7.1
  - Keywords .....7.4.7.1
- Symbolic:
  - Addresses .....2.6
  - Addressing .....5.4
- Symbolic Memory:
  - Addressing .....3.2.4
- Symbols .....2.6
  - Absolute .....2.6
  - Predefined .....2.6.1
  - Relocatable .....2.6
  - User-Defined .....2.6
- Syntax Conventions .....2.4
- Syntax Definition Conventions .....3.5.4
- System:
  - Breakpoint .....4.5.1.4
  - Error:
    - Interrupt .....4.5.1.2, 4.5.1.3
    - Interrupt Conditions .....4.5.1.3
- SZC Instructions .....3.136
- SZCB Instruction .....3.137
- TABLE .....7.7.7
  - Macro Symbol .....7.4.5.1
  - Symbol .....7.4.6, 9.5, 9.5.3
- Tag Characters .....9.5, 9.5.1, T9-3
- Tagged String .....2.8.3
- TB Instruction .....3.138, 4.5.2.5
- TCMB Instruction .....3.139, 4.2.5.7
- Test and Clear Memory Bit
  - Instruction .....3.139, 4.2.5.1
- Test and Set Memory Bit
  - Instruction .....3.142, 4.2.5.1
- Test Bit Instruction .....3.138, 4.5.2.5
- Test Memory Bit Instruction .....3.140
- TEXT Directive .....2.4.2, 5.3.3, 5.3.3.3
- Text (TEXT) Directive,
  - Initialize .....5.3.3, 5.3.3.3
- TILINE .....1.2.3, 4.5.4
- Timing Loops .....4.3.3.1
- TITL Directive .....5.3.2, 5.3.2.3
- TMB Instruction .....3.140

- Trace:
  - Memory:
    - Address ..... 4.5.1.2
    - Error Interrupt ..... 4.5.1.2
  - Trace Control and Map Control
    - Register, Error Interrupt ..... 4.5.1.2
  - Transfer Vector, Interrupt ..... 1.2.4
  - Transfer Vector (XVEC) Instruction ... 6.1, 6.4
  - Transfer Vectors ..... 1.2.4
  - TS Instruction ..... 3.141
  - TSMB Instruction ..... 3.142, 4.2.5.2
  - Two Address Instructions ..... 3.4.1
  - Two's Complement ..... 1.2.1, 1.2.2.2, 1.4.2
  - Types, Data ..... 2.8
- Unconditional Jump Instruction ..... 3.65
- UNIQUE ..... 7.7.4
- UNL Directive ..... 5.3.2, 5.3.2.5
- Upgrade Considerations, 990/10 to 990/12 ..... 4.3
- Use of Parentheses in Expressions .... 2.7.5
- User Attribute Component ..... 7.4.7.1
- User Defined Symbols ..... 2.6
- Value Component ..... 7.4.7.1, 7.4.7.2
- Variable Qualifiers ..... 7.4.5.3
- Variables ..... 7.4.4
- Vector, Interrupt Transfer ..... 1.2.5
- Vectors, Transfer ..... 1.2.4, 1.2.5
- Verb:
  - \$ASG ..... 7.5.3
  - \$CALL ..... 7.5.8
  - \$ELSE ..... 7.5.10
  - \$END ..... 7.5.12
  - \$ENDIF ..... 7.5.11
  - \$EXIT ..... 7.5.7
  - \$GOTO ..... 7.5.6
  - \$IF ..... 7.5.9
  - \$MACRO ..... 7.5.1
  - \$NAME ..... 7.5.5
  - \$SBSTG ..... 7.5.4
  - \$VAR ..... 7.5.2
- Verbs ..... 7.5
- Warning:
  - Messages ..... 9.2
  - SDSMAC ..... 9.3, T9-1
- Well-Defined Expressions ..... 2.5
- Word Address ..... 1.2.1
- Word Boundary ..... 1.2.2
- Word Boundary (EVEN)
  - Directive ..... 5.3.1, 5.3.1.6
- Word:
  - Length, Source ..... 2.4
  - Memory ..... 1.2.2
  - Word (DATA) Directive,
    - Initialize ..... 5.3.3, 5.3.3.2
  - Workspace ..... 1.2.6
  - Cache ..... 4.3.1
  - Workspace Pointer Register ..... 1.2.7
  - Workspace Pointer (WPNT)
    - Directive ..... 5.3.3, 5.3.3.6
  - Workspace Register:
    - Address ..... 3.2.1
    - Indirect ..... 3.2.2
    - Addressing ..... 3.2.1
  - Workspace Register:
    - Indirect:
      - Addressing ..... 3.2.2
      - Autoincrement Addressing ..... 3.2.3
  - Workspace Register Cache ..... 1.3
  - WPNT Directive ..... 5.3.3, 5.3.3.6
  - Writable Control Store ..... 1.3
  - Bit ..... 1.4.12
- X Instruction ..... 3.143, 3.143.1
- XF Instruction ..... 3.144
- XIT Instruction ..... 3.145
- XMA Command ..... G-2
- XOP Instruction ..... 3.146, 3.146.1
- XOR Instruction ..... 3.147
- XORM Instruction ..... 3.148
- XV Instruction ..... 3.149
- XVEC Instruction ..... 6.1, 6.4
- 12-Millisecond Test Clock Interrupt ... 4.5.1.5
- 990/10 to 990/12 Upgrade
  - Considerations ..... 4.3
- 990/12:
  - Assembly Language ..... 2.3
  - Computer ..... 1.3
- \$ASG Verb ..... 7.5.3
- \$CALL Verb ..... 7.5.8
- \$ELSE Verb ..... 7.5.10
- \$END Verb ..... 7.5.12
- \$ENDIF Verb ..... 7.5.11
- \$EXIT Verb ..... 7.5.7
- \$GOTO Verb ..... 7.5.6
- \$IF Verb ..... 7.5.9
- \$MACRO Verb ..... 7.5.1
- \$NAME Verb ..... 7.5.5
- \$SBSTG Verb ..... 7.5.4
- \$VAR Verb ..... 7.5.2

