PROGRAMMER'S GUIDE TO THE CENTRAL PROCESSOR

TEXAS INSTRUMENTS
INCORPORATED

ASC

PROGRAMMER'S GUIDE TO THE

CENTRAL PROCESSOR

TEXAS INSTRUMENTS
INCORPORATED

930039-2
MAY 1976

LIST OF EFFECTIVE PAGES

INSERT LATEST CHANGED PAGES DESTROY SUPERSEDED PAGES

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

Programmer's Guide to the Central Processor

Original . . . . . May 1973
Revised and Reissued . . . . . May 1976

Total number of pages in this publication is 427 consisting of the following:

PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.
Title.	0				
Effec. Pages . . .	0				
i - xvi	0				
1-1 - 1-4	0				
2-1 - 2-20	0				
3-1 - 3-5.	0				
4-1 - 4-25	0				
5-1 - 5-7.	0				
6-1 - 6-54	0				
7-1 - 7-176	0				
8-1 - 8-65/8-66 .	0				
A-1 - A-10.	0				
B-1 - B-10.	0				
C-1 - C-10.	0				
D-1 - D-4	0				
E-1 - E-4	0				
F-1 - F-4	0				
G-1 - G-5	0				
H-1 - H-2	0				
User's Resp . . .	0				
Bus. Reply	0				

TEXAS INSTRUMENTS
INCORPORATED
EQUIPMENT GROUP
AUSTIN, TEXAS

Application For Automatic Update

	REF NO. (1–5)	NAME (9–12)	LAST, INITIAL (41–56)	
CARD 1		NAME		
CARD 2		PART NUMBER	73654	
CARD 3		MAIL STATION		ENTER 0 USM IF US MAIL
CARD 4		QUANTITY		ENTER QUANTITY OF MANUALS

RIGHT JUSTIFIED

COMPLETE ONLY IF US MAIL ADDRESS

	NAME (41–60)
ADD1	LAST, INITIAL
ADD2	ADDRESS
ADD3	
ADD4	
ADD5	
ADD6	

USE NUMBER OF LINES REQUIRED FOR COMPLETE ADDRESS

I AM PRESENTLY ON DISTRIBUTION FOR OTHER DOCUMENTS ☐ YES ☐ NO

FOLD AND STAPLE THIS SHEET. RETURN ADDRESS IS ON REVERSE SIDE.

TEXAS INSTRUMENTS
INCORPORATED

EQUIPMENT GROUP
P.O. BOX 2909
AUSTIN, TEXAS 78767

ATTENTION
TECHNICAL DATA BRANCH
MAIL STATION 2146

TABLE OF CONTENTS

Section			Page
I	GENERAL DESCRIPTION		
	1-1	The Central Processor	1-1
	1-2	Central Processor-Peripheral Processor Relationship	1-1
	1-3	Central Processor Resources	1-1
	1-8	The Assembler	1-2
	1-9	Coding Media	1-3
	1-10	Punched Card	1-10
	1-11	Coding Form	1-11
II	LANGUAGE ELEMENTS		
	2-1	Character Set for the ASC	2-1
	2-2	Printable Characters	2-1
	2-3	Special Characters	2-1
	2-4	Items	2-1
	2-5	Symbol	2-3
	2-6	Character String	2-3
	2-7	Decimal Integer	2-3
	2-8	Hexadecimal Integer	2-4
	2-9	Floating Point Item	2-4
	2-10	Fixed Point Decimal Item	2-5
	2-11	Location Counter	2-6
	2-12	Literal	2-6
	2-13	Intrinsic Function	2-6

TABLE OF CONTENTS (Continued)

Section		Page
2-14	Operators	2-7
2-15	Operator Types	2-7
2-16	Expressions	2-7
2-17	Subexpressions	2-10
2-18	Assumed Parentheses	2-11
2-19	Literals	2-12
2-20	Lists	2-13
2-21	Intrinsic Functions	2-14
2-22	Global Attribute Functions	2-14
2-26	Location Intrinsic Functions	2-16
2-27	Program Sections	2-16
2-28	Relocation	2-17
2-29	Constants	2-17
2-32	Location Counter	2-17
2-33	Relocatability of Symbols	2-18
2-34	Relocatability of Expressions	2-18
III	LANGUAGE STRUCTURE	
3-1	Statement Format	3-1
3-2	Conventions for Describing Language Statements	3-2
3-3	Continuation Lines	3-2
3-4	Label Field	3-3
3-6	Command Field	3-3
3-7	Operand Field	3-4
3-8	Remark Field	3-5

TABLE OF CONTENTS (Continued)

Section			Page
	3-9	Comment Lines	3-5
	3-10	Blank Lines	3-5
IV	DIRECTIVES		
	4-1	Introduction	4-1
	4-2	Definition Directives	4-1
	4-3	Equate Directive (EQU)	4-1
	4-4	Set Directive (SET)	4-2
	4-5	External Name Directive (EXTRN)	4-3
	4-6	Entry Name Directive (ENTRY)	4-4
	4-7	Data Directive (DATA)	4-5
	4-8	Format Directive (FORM)	4-6
	4-9	Using Directive (USING)	4-7
	4-10	Drop Directive (DROP)	4-8
	4-11	Origin Directive (ORG)	4-8
	4-12	Control Directives	4-9
	4-13	Literal Origin Directive (LITORG)	4-9
	4-14	End Assembly Directive (END)	4-10
	4-15	Section Directive (SEC)	4-11
	4-16	Common Module Directive (COM)	4-12
	4-17	Dummy Section Directive (DUM)	4-13
	4-18	Dummy Common Module Directive (COMD)	4-14
	4-19	Copy Directive (COPY)	4-15
	4-20	Reserve Directive (RES)	4-16
	4-21	Align Directive (ALIGN)	4-16
	4-22	Do Directive (DO)	4-17
	4-23	Pseudo Directives	4-21

TABLE OF CONTENTS (Continued)

Section			Page
	4-24	Indirect Address Constant Directive (IND)	4-21
	4-25	Branch Address Constant Directive (BCON)	
	4-26	Data Halfword Directive (DATAH)	4-22
	4-27	Listing Directives	4-22
	4-28	Skip Directive (SKIP)	4-23
	4-29	List Directive (LIST)	4-24
	4-30	Nolist Directive (NOLIST)	4-25
V		ASSEMBLER OUTPUT	
	5-1	Assembler Output	5-1
	5-2	Source Program Listing	5-1
	5-3	Messages	5-3
	5-5	Cross-Reference Listing	5-5
VI		ASSEMBLER-CENTRAL PROCESSOR INTERFACE	
	6-1	Introduction	6-1
	6-2	Instruction Formats	6-1
	6-3	Label	6-3
	6-4	Command	6-3
	6-5	Operands	6-3
	6-6	R, N, X Operand List	6-3
	6-10	R, R, N Operand List	6-7
	6-11	Register Addressing	6-8
	6-12	Register Operand-R Field Addresses	6-8
	6-13	Address Operand Register Addresses	6-11
	6-14	Address Development	6-12

TABLE OF CONTENTS (Continued)

Section		Page
6-15	Assembler Translation	6-12
6-18	Machine Translation	6-15
6-29	Immediate Operands	6-31
6-30	Assembler Translation	6-31
6-33	Machine Translation	6-32
6-40	Branch Address Development	6-36
6-41	Assembler Translation	6-37
6-47	Machine Translation	6-39
6-51	Data Formats	6-43
6-52	Fixed Point Data	6-44
6-53	Floating Point Data	6-44
6-56	Program Status Doubleword	6-49
6-57	Branch or Skip Register	6-50
6-58	Compare Code	6-50
6-59	Result Code	6-51
6-60	Arithmetic Exception Condition Code	6-51
6-62	Arithmetic Exception Mask	6-53
6-64	Program Counter	6-54
VII	THE SCALAR INSTRUCTIONS FOR THE CENTRAL PROCESSOR	
7-1	Introduction	7-1
7-2	Load Register Instructions	7-2
7-3	Load, Word (L)	7-3
7-4	Load, Left Half From Left (LLL)	7-4
7-5	Load, Right Halfword From Right (LRR)	7-5
7-5.1	Load, Right Halfword From Left (LRL)	7-5A
7-6	Load, Left Halfword From Right (LLR)	7-6

TABLE OF CONTENTS (Continued)

Section		Page
7-7	Load, Doubleword (LD)	7-7
7-8	Load Immediate, Word (LI)	7-8
7-9	Load Immediate, Halfword (LIH)	7-9
7-10	Load Negative, Fixed Point Word (LN)	7-10
7-11	Load Negative, Fixed Point Halfword (LNH)	7-11
7-12	Load Negative, Floating Point Word (LNF)	7-12
7-13	Load Negative, Floating Point Doubleword (LND)	7-13
7-14	Load Magnitude, Fixed Point Word (LM)	7-14
7-15	Load Magnitude, Fixed Point Halfword (LMH)	7-15
7-16	Load Magnitude, Floating Point Word (LMF)	7-16
7-17	Load Magnitude, Floating Point Doubleword (LMD)	7-17
7-18	Load Negative Magnitude, Fixed Point Word (LNM)	7-18
7-19	Load Negative Magnitude, Fixed Point Halfword (LNMH)	7-19
7-20	Load Negative Magnitude, Floating Point Word (LNMF)	7-20
7-21	Load Negative Magnitude, Floating Point Doubleword (LNMD)	7-21
7-22	Load One's Complement, Word (LD)	7-22
7-23	Load Register File (LF)	7-23
7-24	Load Register Files, Multiple (LFM)	7-24
7-25	Store Instructions	7-25
7-26	Store Word (ST)	7-26
7-27	Store Halfword (STLL)	7-27
7-28	Store Right Halfword Into Right (STRR)	7-28
7-28.1	Store Right Halfword Into Left (STRL)	7-28A
7-29	Store Left Halfword Into Right (STLR)	7-29
7-30	Store Doubleword (STD)	7-30
7-31	Store Zero, Word (STZ)	7-31

TABLE OF CONTENTS (Continued)

Section		Page
7-32	Store Zero, Halfword (STZH)	7-32
7-33	Store Zero, Doubleword (STZD)	7-33
7-34	Store Negative, Fixed Point Word (STN)	7-34
7-35	Store Negative, Fixed Point Halfword (STNH)	7-35
7-36	Store Negative, Floating Point Word (STNF)	7-36
7-37	Store Negative, Floating Point Doubleword (STND)	7-37
7-38	Store One's Complement, Word (STO)	7-38
7-39	Store One's Complement, Halfword (STDH)	7-39
7-40	Store Register File (STF)	7-40
7-41	Store Register Files, Multiple (STFM)	7-41
7-42	Arithmetic Instructions	7-42
7-43	Add, Fixed Point Word (A)	7-43
7-44	Add, Fixed Point Halfword (AH)	7-44
7-45	Add, Floating Point Word (AF)	7-45
7-46	Add, Floating Point Doubleword (AFD)	7-46
7-47	Add Immediate, Fixed Point Word (AI)	7-47
7-48	Add Immediate, Fixed Point Halfword (AIH)	7-48
7-49	Add Magnitude, Fixed Point Word (AM)	7-49
7-50	Add Magnitude, Fixed Point Halfword (AMH)	7-50
7-51	Add Magnitude, Floating Point Word (AMF)	7-51
7-52	Add Magnitude, Floating Point Doubleword (AMFD)	7-52
7-53	Subtract, Fixed Point Word (S)	7-53
7-54	Subtract, Fixed Point Halfword (SH)	7-54
7-55	Subtract, Floating Point Word (SF)	7-55
7-56	Subtract, Floating Point Doubleword (SFD)	7-56
7-57	Subtract Immediate, Fixed Point Word (SI)	7-57
7-58	Subtract Immediate, Fixed Point Halfword (SIH)	7-58

TABLE OF CONTENTS (Continued)

Section		Page
7-59	Subtract Magnitude, Fixed Point Word (SM)	7-59
7-60	Subtract Magnitude, Fixed Point Halfword (SMH)	7-60
7-61	Subtract Magnitude, Floating Point Word (SMF)	7-61
7-62	Subtract Magnitude, Floating Point Doubleword (SMFD)	7-62
7-63	Multiply, Fixed Point Word (M)	7-63
7-64	Multiply, Fixed Point Halfword (MH)	7-64
7-65	Multiply, Floating Point Word (MF)	7-65
7-66	Multiply, Floating Point Doubleword (MFD)	7-66
7-67	Multiply Immediate, Fixed Point Word (MI)	7-67
7-68	Multiply Immediate, Fixed Point Halfword (MIH)	7-68
7-69	Divide, Fixed Point Word (D)	7-69
7-70	Divide, Fixed Point Halfword (DH)	7-70
7-71	Divide, Floating Point Word (DF)	7-71
7-72	Divide, Floating Point Doubleword (DFD)	7-72
7-73	Divide Immediate, Fixed Point Word (DI)	7-73
7-74	Divide Immediate, Fixed Point Halfword (DIH)	7-74
7-75	Logical Instructions	7-75
7-76	AND, Word (AND)	7-76
7-77	AND, Doubleword (ANDD)	7-77
7-78	AND Immediate, Word (ANDI)	7-78
7-79	OR, Word (OR)	7-79
7-80	OR, Doubleword (ORD)	7-80
7-81	OR Immediate, Word (ORI)	7-81
7-82	Exclusive OR, Word (XOR)	7-82
7-83	Exclusive OR, Doubleword (XORD)	7-83
7-84	Exclusive OR Immediate, Word (XORI)	7-84
7-85	Equivalence, Word (EQC)	7-85

TABLE OF CONTENTS (Continued)

Section		Page
7-86	Equivalence, Doubleword (EQCD)	7-86
7-87	Equivalence Immediate, Word (EQCI)	7-87
7-88	Shift Instructions	7-88
7-89	Arithmetic Shifts	7-89
7-90	Logical Shifts	7-89
7-91	Circular Shifts	7-90
7-92	Algorithm for Bit Reversal	7-91
7-93	Arithmetic Shift, Word (SA)	7-92
7-94	Arithmetic Shift, Halfword (SAH)	7-93
7-95	Arithmetic Shift, Doubleword (SAD)	7-94
7-96	Logical Shift, Word (SL)	7-95
7-97	Logical Shift, Halfword (SLH)	7-96
7-98	Logical Shift, Doubleword (SLD)	7-97
7-99	Circular Shift, Word (SC)	7-98
7-100	Circular Shift, Halfword (SCH)	7-99
7-101	Circular Shift, Doubleword (SCD)	7-100
7-102	Bit Reversal, Word (RVS)	7-101
7-103	Compare Instructions	7-102
7-104	Compare, Fixed Point Word (C)	7-103
7-105	Compare, Fixed Point Halfword (CH)	7-104
7-106	Compare, Floating Point Word (CF)	7-105
7-107	Compare, Floating Point Doubleword (CFD)	7-106
7-108	Compare Immediate, Fixed Point Word (CI)	7-107
7-109	Compare Immediate, Fixed Point Halfword (CIH)	7-108
7-110	Compare Logical AND, Word (CAND)	7-109
7-111	Compare Logical AND, Doubleword (CANDD)	7-110
7-112	Compare Logical AND Immediate, Word (CANDI)	7-111

TABLE OF CONTENTS (Continued)

Section		Page
7-113	Compare Logical OR, Word (COR)	7-112
7-114	Compare Logical OR, Doubleword (CORD)	7-113
7-115	Compare Logical OR Immediate, Word (CORI)	7-114
7-116	Increment or Decrement, Test and Skip Instructions	7-115
7-117	Increment, Test and Skip on Equal (ISE)	7-116
7-118	Increment, Test and Skip on Not Equal (ISNE)	7-117
7-119	Decrement, Test and Skip on Equal (DSE)	7-118
7-120	Decrement, Test and Skip on Not Equal (DSNE)	7-119
7-121	Increment or Decrement, Test and Branch Instructions	7-120
7-122	Increment, Test and Branch on Zero (IBZ)	7-121
7-123	Increment, Test and Branch on Not Zero (IBNZ)	7-122
7-124	Decrement, Test and Branch on Zero (DBZ)	7-123
7-125	Decrement, Test and Branch on Not Zero (DBNZ)	7-124
7-126	Index, Test and Branch Instructions	7-125
7-127	Algorithm for Index Test and Branch	7-126
7-128	Branch on Less Than or Equal (BCLE)	7-127
7-129	Branch on Greater Than (BCG)	7-128
7-130	Conditional Branch Instructions	7-129
7-131	Condition Algorithms for Conditional Branches	7-132
7-132	Branch on Comparison Code True (BCC)	7-133
7-133	Branch on Result Code True (BRC)	7-135
7-134	Branch on Arithmetic Exception (BAE)	7-137
7-135	Branch on Execute Branch Condition True (BXEC)	7-140
7-136	Unconditional Branch Instructions	7-141
7-137	Branch and Load Base Register With Program Counter (BLB)	7-142

TABLE OF CONTENTS (Continued)

Section		Page
7-138	Branch and Load Index or Vector Register With Program Counter (BLX)	7-143
7-139	Stack Instructions	7-144
7-140	Stack Instruction Definition	7-145
7-141	Push Word Into Last-In-First-Out Stack (PSH)	7-146
7-142	Pull Word From Last-In-First-Out Stack (PUL)	7-147
7-143	Modify Stack Parameter Doubleword (MOD)	7-148
7-144	Conversion and Normalization Instructions	7-150
7-145	Algorithm for Floating Point to Fixed Point Conversions	7-151
7-146	Algorithm for Fixed Point to Floating Point Conversions	7-152
7-147	Fixed Point Normalization	7-154
7-148	Convert Floating Point Word to Fixed Point Word (FLFX)	7-155
7-149	Convert Floating Point Word to Fixed Point Halfword (FLFH)	7-156
7-150	Convert Floating Point Doubleword to Fixed Point Word (FDFX)	7-157
7-151	Convert Fixed Point Word to Floating Point Word (FXFL)	7-158
7-152	Convert Fixed Point Halfword to Floating Point Word (FHFL)	7-159
7-153	Convert Fixed Point Word to Floating Point Doubleword (FXFD)	7-160
7-154	Convert Fixed Point Halfword to Floating Point Doubleword (FHFD)	7-160
7-155	Normalize Fixed Point Word (NFX)	7-162
7-156	Normalize Fixed Point Halfword (NFH)	7-163
7-157	Miscellaneous Instructions	7-164

TABLE OF CONTENTS (Continued)

Section		Page
7-158	Exchange Words (XCH)	7-165
7-159	Load Look Ahead (LLA)	7-166
7-159.1	Prepare to Branch (PB)	7-167A
7-160	Load Effective Address (LEA)	7-168
7-161	Execute (XEC)	7-169
7-162	Interpret (INT)	7-170
7-162.1	Fork (FORK)	7-170A
7-162.2	Join (JOIN)	7-170A
7-163	Monitor Call and Proceed (MCP)	7-171
7-164	Monitor Call and Wait (MCW)	7-172
7-165	Program Status Instructions	7-173
7-166	Load Arithmetic Exception Mask (LAM)	7-174
7-167	Load Arithmetic Exception Condition (LAC)	7-175
7-167.1	Load Arithmetic Exception Mask and Condition (LEM)	7-175A
7-168	Store Program Status Word (SPS)	7-176
VIII	THE VECTOR INSTRUCTIONS FOR THE CENTRAL PROCESSOR	
8-1	Introduction	8-1
8-2	Definition	8-1
8-3	Execute Vector Parameter File Instructions	8-4
8-4	Vector Load and Execute (VECTL)	8-4
8-5	Vector Execute (VECT)	8-5
8-6	The Vector Parameter File	8-6
8-7	Vector Operation Specification	8-6
8-8	Arithmetic and Logical Comparison Condition Specification	8-6
8-9	Vector Length (Self Loop Count) Specification	8-8

TABLE OF CONTENTS (Continued)

Section		Page
8-10	Single-Valued Vector and Word Size Specification	8-8
8-11	Single-Valued Vectors	8-8
8-12	Immediate Vectors	8-11
8-13	Vector Address Development	8-14
8-14	Directly Addressed Vectors	8-14
8-13	Vector Address Development	8-14
8-15	Halfword Index Start Specification	8-15
8-16	Self Loop Increment Direction	8-16
8-17	Inner Loop Specification	8-18
8-20	Outer Loop Specification	8-19
8-23	Program Interrupts	8-19
8-24	Vector Hazard	8-20
8-25	Vector Arithmetic Instructions	8-20
8-26	Vector Add Instructions	8-21
8-27	Vector Add Magnitude Instructions	8-22
8-28	Vector Subtract Instructions	8-23
8-29	Vector Subtract Magnitude Instructions	8-24
8-30	Vector Multiply Instructions	8-25
8-31	Vector Dot Product Instructions	8-26
8-32	Vector Divide Instructions	8-28
8-33	Vector Logical Instructions	8-30
8-34	Vector Shift Instructions	8-31
8-35	Vector Merge Instructions	8-33
8-36	Vector Order Instructions	8-34
8-37	Vector Compare Instructions	8-36

TABLE OF CONTENTS (Continued)

Section		Page
8-38	Vector Compare Arithmetic Instructions	8-38
8-39	Vector Compare Logical Instructions	8-38
8-40	Vector Peak Picking Instructions	8-39
8-41	Vector Search Instructions	8-40
8-42	Vector Search for Largest Element Instructions	8-41
8-43	Vector Search for Largest Magnitude Instructions	8-42
8-44	Vector Search for Smallest Element Instructions	8-43
8-45	Vector Search for Smallest Magnitude Instructions	8-44
8-46	Vector Conversion Instructions	8-45
8-47	Convert Floating Point Elements to Fixed Point Elements	8-45
8-48	Convert Fixed Point Elements to Floating Point Elements	8-46
8-49	Vector Normalize Instructions	8-47
8-50	Vector Map Instructions	8-49
8-51	Vector Select Boolean Instructions	8-52
8-52	Vector Replace Boolean Instructions	8-54
8-53	Vector Map Boolean Instructions	8-56
8-54	Vector Maximum/Minimum Instructions	8-58
8-55	Vector Compare Boolean Instructions	8-60
8-56	Vector Compare and/or Boolean Instructions	8-61
8-57	Vector Select	8-62
8-58	Vector Replace	8-63

APPENDIXES

Appendix		
A	SCALAR INSTRUCTIONS BY LOGICAL GROUPING	A-1
B	SCALAR INSTRUCTIONS IN ALPHABETICAL ORDER BY ASSEMBLER CODE	B-1
C	SCALAR INSTRUCTIONS IN NUMERIC ORDER BY MACHINE CODE	C-1
D	VECTOR INSTRUCTIONS BY LOGICAL GROUPING	D-1
E	VECTOR INSTRUCTIONS IN ALPHABETICAL ORDER BY ASSEMBLER CODE	E-1
F	VECTOR INSTRUCTIONS IN NUMERIC ORDER BY MACHINE CODE	F-1
G	SCALAR INSTRUCTION TIME REQUIREMENTS	G-1
	G-1 Scalar Instruction Timing Groups	G-1
H	VECTOR INSTRUCTION TIME REQUIREMENTS	H-1
	H-1 Time Requirements for Complete Vector Operation	H-1

LIST OF ILLUSTRATIONS

Figure	Title	Page
1-1	Coding Form	1-4
5-1	Sample Source Program Listing	5-2
5-2	Cross-Reference Listing Example	5-6
6-1	Assembler Statement Translations into Machine Code	6-2
6-2	Register File Specifications	6-9
6-3	Development of Singleword Effective Addresses	6-17
6-4	Development of Halfword Effective Addresses	6-20
6-5	Development of Doubleword Effective Addresses	6-23
6-6	Indirect Address Cell Format	6-28
6-7	Development of Singleword Effective Immediate Operands	6-33
6-8	Development of Halfword Effective Immediate Operands	6-34
6-9	Development of Singleword Logical Immediate Operands	6-36

LIST OF ILLUSTRATIONS (Continued)

Figure	Title	Page
6-10	Program Counter Relative Branch Address Development	6-40
6-11	Development of Base Relative Branch Addresses	6-42
6-12	Algebraic Data Formats	6-43
8-1	The Vector Parameter File	8-3
8-2	Flow of Execution of a Vector Parameter File	8-7
8-3	Flow of Execution with B Single-valved	8-10
8-4	Flow of Execution with B Single-valved	8-13

LIST OF TABLES

Table	Title	Page
2-1	Printable Characters	2-2
2-2	Special Characters	2-2
2-3	Operator Hierarchies and Descriptions	2-8
2-4	Use of Operators	2-9
2-5	Results of Operations on Absolute and Relocatable Items in Expressions	2-19
5-1	Assembler Generated Messages	5-4
5-2	Procedure Processing Message Symbols	5-5
6-1	General Forms and Variations of the Operand Lists	6-4
6-2	Register Addressing Symbols	6-10
6-3	Development of Singleword Addresses (Direct)	6-18
6-4	Development of Branch Addresses (Direct)	6-41
6-5	Value Ranges of Fixed Point Data	6-44
6-6	Specifications for Arithmetic Exception Mask Data Constants	6-54
8-1	Specifications of the SV Field	8-9
8-2	Specifications of the HS Field Valves	8-17
8-3	Specifications of the VI Field Valves	8-17
8-4	Specifications of the VI, HS, and ALCT Fields	8-37
8-5	Specifications of the ALCT Valves	8-37
H-1	Vector Execution Rates in Clocks/Element	H-1

SECTION I
GENERAL DESCRIPTION

1-1. THE CENTRAL PROCESSOR

The Central Processor is that unit of the ASC dedicated to processing the user's raw data. It is particularly oriented toward the processing of numerical data that is typical of scientific data processing.

1-2. CENTRAL PROCESSOR-PERIPHERAL PROCESSOR RELATIONSHIP

The Central Processor operates under control of the Peripheral Processor in which the operating system resides. The resources of the Central Processor are time-shared among users through this system which can cause a current program's status to be saved, the program to be removed from the Central Processor, and another program to be given control of the Central Processor.

All communication, either to or from the Central Processor, takes place through the Peripheral Processor.

1-3. CENTRAL PROCESSOR RESOURCES

Once a program is given control of the Central Processor, both program and data are streamed directly from central memory to the Central Processor and results streamed back. Streaming is accomplished by double buffering of both program and data.

1-4. Pipeline Instruction Processing

The instructions fetched from memory are decoded in a pipeline made of four levels. This method permits four instructions to be in the process of decoding at any given time, and, unless a branch instruction requires the discard of some of the instructions, each instruction is ready for execution in the arithmetic unit as soon as its resources are available.

1-5. Vector Operations

The Central Processor performs operations on ordered sets of data without requiring additional instruction decoding. Once a vector operation has been initiated, the data upon which it operates is streamed directly to the arithmetic unit from central memory, and the results streamed back to central memory.

1-6. Instruction Set

The Central Processor has 177 scalar instructions and 70 vector instructions that provide a large range of programming ploys.

1-7. Data Formats

The Central Processor performs operations on fixed point, floating point, or binary logical data.

Fixed point data may be either 32-bit singlewords or 16-bit halfwords, and in either case the values are represented with any negative numbers in two's complement notation.

Floating point data may be either 32-bit singlewords or 64-bit doublewords. In either case, the biased hexadecimal exponent method of representation is used.

1-8. THE ASSEMBLER

The assembler as implemented for the Central Processor provides for symbolic coding of programs to be executed in the Central Processor.

There are directives which are commands to the assembler itself. These directives are used to inform the assembler of conditions to be expected at assembly time, of conditions to be expected at object program execution time, and of the nature of the symbols used by the programmer.

The assembler mnemonics for actual machine codes are the names of procedures built into the assembler. These procedures translate the mnemonics and the operands associated with them into object code that the machine can execute.

There are also procedures built into the assembler which translate data that is in a form more convenient to the programmer into the object data forms usable by the machine.

1-9. CODING MEDIA

A source program is a sequence of source statements that are punched into cards and entered into the computer by a card reader.

1-10. PUNCHED CARD

The card format is a standard 80-column punched card.

1-11. CODING FORM

Assembler source statements may be written on the standard coding form, shown in Figure 1-1. One line of code on the form is punched into one card; vertical columns on the form correspond to card columns.

Space is provided for program identification and for instructions to keypunch operators. The body of the coding form consists of the statement field, columns 1 through 72, and the identification sequence field, columns 73 through 80.

CODING FORM

PROGRAM		PUNCHING INSTRUCTIONS	GRAPHIC								PAGE OF
PROGRAMMER	DATE		PUNCH								

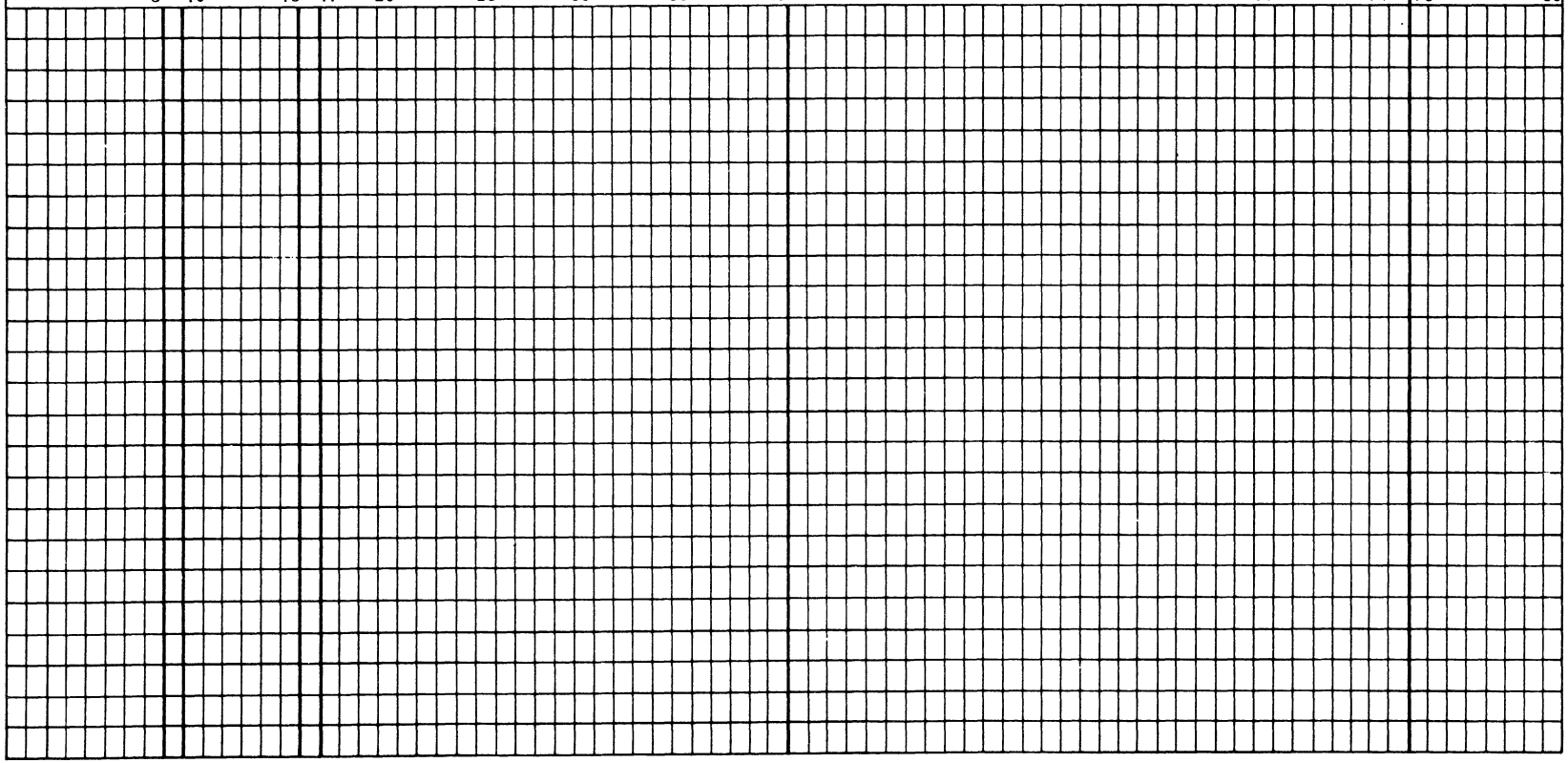


Figure 1-1. Coding Form

SECTION II
LANGUAGE ELEMENTS

2-1. CHARACTER SET FOR THE ASC

The ASC Assembler recognizes the EBCDIC character set as standard notation. That is, characters are interpreted as punched on the IBM 029 keypunch. References in this manual are made to alphabetic characters (A through Z), numeric characters (0 through 9), and special characters (all the rest).

All characters except the double quotation mark (") and the semicolon (;) may be used in character strings, and these also may be used freely in the remark field and in comments. The period (or decimal point), dollar sign ($), and question mark (?) may be used in symbols along with alphabetic and numeric characters. Most of the special characters have unique meanings to the assembler.

The double quotation mark (") inside a character string will terminate the string. The semicolon (;) when used inside character strings will terminate the card image, and the string will be continued on the next line beginning with the first non-blank character.

2-2. PRINTABLE CHARACTERS

Table 2-1 contains a list of the non-alphanumeric printable characters and their names without regard to their special meanings to the assembler.

2-3. SPECIAL CHARACTERS

Table 2-2 lists the special characters which have unique meaning to the ASC Assembler.

2-4. ITEMS

Any item consists of a combination of one or more characters. An item may be a symbol, decimal integer, character string, hexadecimal integer, location counter, floating point item, fixed point item, literal, or intrinsic function.

Table 2-1. Printable Characters

CHAR-ACTER	NAME	CARD CODE	CHAR-ACTER	NAME	CARD CODE
	blank	blank	-	hyphen, or minus sign	11
¢	cent sign	12-8-2	/	slash (virgule)	0-1
.	period	12-8-3	,	comma	0-8-3
<	less than	12-8-4	%	percent sign	0-8-4
(	left parenthesis	12-8-5	—	horizontal bar	0-8-5
+	plus sign	12-8-6	>	greater than	0-8-6
|	vertical bar	12-8-7	?	question mark	0-8-7
&	ampersand	12	↑	vertical arrow	8-1
!	exclamation point	11-8-2	:	colon	8-2
			#	number	8-3
$	dollar sign	11-8-3	@	at	8-4
*	asterisk	11-8-4	'	apostrophe	8-5
)	right parenthesis	11-8-5	=	equals	8-6
;	semicolon	11-8-6	"	quotation marks	8-7
¬	not sign	11-8-7			

Table 2-2. Special Characters

CHARACTER	MEANING	CHARACTER	MEANING
#	hexadecimal	)	right parenthesis
@	indirect addressing	¢	augment indicator
,	separator	>	greater than
$	location counter	;	continuation
*	multiply and comments	¬	not
.	period or decimal point	"	EBCDIC string indicator
<	less than	=	equals or literal indicator
-	subtract		
/	divide	(	left parenthesis
+	add	blank	separator or space

2-5. SYMBOL

A symbol is represented as a string of from one to eight EBCDIC characters, the first of which must be alphabetic. The remaining characters may be alphabetic, numeric, . , $, # , or any other special characters not used by the Assembler for unique purposes. (See Table 2-2 for characters having unique meaning to the Assembler.)

VALUE: The value of a symbol is the value of the item to which the symbol is assigned.

Examples: AABBCCDD

Q.J$P?

2-6. CHARACTER STRING

A character string is any string of characters surrounded by double quotation marks (not to be confused with two single quotation marks). Semicolons (;) or double quotation marks ('') cannot be parts of a character string because they operate on the string. Character strings which are assigned to symbols as values cannot exceed 8 characters in length. Other character strings are restricted to 256 characters.

VALUE: The value of a character string is the EBCDIC representation of the characters found between the quotation marks. Each character string is converted into an even multiple of 4 characters (32 bits). Strings which do not contain a multiple of 4 characters are filled to the right with blanks.

Example: "AB*C"

2-7. DECIMAL INTEGER

A decimal integer is a string of unsigned decimal digits (0 through 9).

VALUE: The value of a decimal integer is the 32-bit (binary representation) base 10 value of the string of digits.

Examples: 19

5440

2-8. HEXADECIMAL INTEGER

A hexadecimal integer is a string of unsigned hexadecimal digits (0 through F) preceded by a #. The maximum number of characters after the # is 16.

VALUE: The value of a hexadecimal integer is the 32 or 64-bit (binary representation) base 16 value of the string of digits.

Example: #3B8FE5

2-9. FLOATING POINT ITEM

A floating point item is a string of decimal digits with a decimal point and optionally followed by a decimal exponent. The exponent is written as the letter E or the letter D followed by an integer constant. The item may be positive, zero, or negative. If either the initial string of decimal digits or the integral exponent are unsigned, the assembler assumes the respective part to be positive. If a decimal exponent is given, the decimal point is not required in the initial string of digits. The item may assume one of three forms:

1. A string of decimal digits with a decimal point, and without an exponent. This form is assumed by the assembler, to be single precision representation.

2. A string of decimal digits, optionally with a decimal point, followed by the letter E and an integral decimal exponent. The E specifies single precision representation.

3. A string of decimal digits, optionally with a decimal point, followed by the letter D and an integral decimal exponent. The D specifies double precision representation.

For both single and double precision representation, the value of the exponent, n, has the range : $-64 \leq n \leq +63$. The range of values M, a floating point item, may have is: (1) in single precision (32-bit representation), $16^{-65} \leq M \leq (1 - 16^{-6}) \times 16^{63}$ and true zero; and (2) in double precision (64-bit representation), $16^{-65} \leq M \leq (1 - 16^{-14}) \times 16^{63}$ and true zero; or, approximately, $5.4 \times 10^{-79} \leq M \leq 7.2 \times 10^{75}$. The number of bits used in the representation of the fractional part of a

floating point item does not significantly affect its range of values, but affects the precision of the values that may be represented.

If the maximum exponent value is exceeded, a syntax error is returned; but, if the significance is exceeded, truncation of the least significant bits occurs and no error message is returned.

VALUE: The value of a floating point item is the 32 or 64-bit binary representation of the string of digits with 8 bits reserved for the exponent and with the remaining 24 or 56 bits left for the fraction. The exponent is represented in excess 64 notation. The fraction is normalized in its area.

Examples: 5.321E+6
6D-26
5.3
-5.2E6
2.718

2-10. FIXED POINT DECIMAL ITEM

A fixed point decimal item is a string of decimal digits, which may have a decimal point, followed by (1) a B or a BB, and by (2) a binary scale factor. The item may be positive, zero, or negative. If either the initial string of decimal digits or the binary scale factor is unsigned, the assembler assumes the respective part to be positive. A positive binary scale factor shifts the binary representation of the quantity to the left by the specified number of binary units, and a negative binary scale factor shifts the binary representation of the quantity to the right by the specified number of binary units. Any part of the decimal numeral which would result in a binary fraction, when converted to binary and scaled, will be truncated. A single B specifies single precision, and a double B (i.e., BB) specifies double precision.

The range of values of a fixed point item, F, is restricted to:
$-2,147,483,648 \leq F \leq +2,147,483,647$ (i.e., $-2^{31} \leq F \leq 2^{31} - 1$).

VALUE: The value of a fixed point decimal item is the 32 or 64-bit binary representation of the string of digits with the representation determined by converting integer and fraction portions of the string separately and placing the result in either 32 or 64 bits as determined by the precision designator, B or BB, respectively.

Examples: 3.21B+5 3.21BB+5
6B2 6BB2

2-11. LOCATION COUNTER

The coding symbol for the value of the location counter is $.

VALUE: The value of $ is the 32-bit current value at assembly time of the location counter.

Example: $+6

2-12. LITERAL

A literal is a constant which is the relative location of the start of one or more words of data. A literal is expressed in the form of an equals sign followed by the data to be contained in the relative location (see Topic 2-19).

VALUE: The value of a literal is the location of a constant.

Examples: =A
=6
=A+6

2-13. INTRINSIC FUNCTION

An intrinsic function is an item used to produce substitution of another item, expression, or list in its place. See Topic 2-21.

VALUE: The value of an intrinsic function is the identity of the particular parameter operated on by the function, or is the value assigned to the condition of the parameter operated on by the function.

Example: T(RHO)

2-14. OPERATORS

Items may be combined using the special character operators defined in Table 2-3. The table also gives hierarchy numbers for determining the sequence in which the value of an expression is computed. Operations with higher hierarchies are performed before operations having lower hierarchies. Operations with the same hierarchy are performed from left to right.

2-15. OPERATOR TYPES

Each operator falls under two type classifications: every operator is either a unary operator or a binary operator, and every operator is one of the following: an arithmetic operator, a relational operator, or a logical operator. See Tables 2-3 and 2-4 for the operator symbols and their uses.

UNARY OPERATION: A unary operation is one that involves only one operand.

BINARY OPERATION: A binary operation is one that involves two operands.

ARITHMETIC OPERATION: An arithmetic operation is one that yields algebraic quantities.

RELATIONAL OPERATION: A relational operation is one that yields a "TRUE" or "FALSE" quantity; i.e., 1 or 0, respectively.

LOGICAL OPERATION: A logical operation is one that yields a Boolean quantity.

2-16. EXPRESSIONS

An expression is an item, or it is a series of items, connected by operators. The sequence of operations performed in evaluating an expression is determined by the hierarchy of the operators in the expression. The hierarchy of operators is shown in Table 2-3. Operations with higher hierarchy numbers are performed first; operations with the same hierarchy are performed from left to right.

Table 2-3. Operator Hierarchies and Descriptions

HIER-ARCHY	SYMBOL	TYPE	DESCRIPTION
7	+	Unary Arithmetic	Plus
7	-	Unary Arithmetic	Minus (two's complement)
7	¬	Unary Arithmetic	Not (one's complement)
6	//	Binary Logical	Logical Binary Operator
5	*	Binary Arithmetic	Arithmetic Product
5	/	Binary Arithmetic	Arithmetic Quotient
4	+	Binary Arithmetic	Arithmetic Sum
4	-	Binary Arithmetic	Arithmetic Difference
3	<	Binary Relational	Arithmetic Less Than
3	¬<	Binary Relational	Not Less Than
3	=	Binary Relational	Arithmetic Equals
3	¬=	Binary Relational	Not Equals
3	<=	Binary Relational	Less Than or Equal
3	>	Binary Relational	Arithmetic Greater Than
3	¬>	Binary Relational	Not Greater Than
3	>=	Binary Relational	Greater Than or Equal
2	**	Binary Logical	Logical Product (AND)
1	++	Binary Logical	Logical Sum (OR)
1	--	Binary Logical	Logical Difference (Exclusive OR)
1	==	Binary Logical	Logical Equivalence

The length of an expression is limited by the number of continuation lines over which the statement may extend. The value of an arithmetic expression, E, is restricted to the range: $-2,147,483,648 \leq E \leq +2,147,483,647$ ($-2^{31} \leq E \leq 2^{31} - 1$). The value of an expression, E, containing an external symbol or symbols is restricted to the range: $-8,388,608 \leq E \leq +8,388,617$, ($-2^{23} \leq E \leq 2^{23} - 1$).

Floating point numbers are not valid in expressions which contain more than one item. That is, floating point arithmetic will not be performed at assembly

Table 2-4. Use of Operators

SYMBOL	GENERAL FORM	WHERE	RESULTS
+	+a	a is an algebraic expression	a
-	-a	a is an algebraic expression	two's complement of a
¬	¬a	a is an algebraic or logical expression	one's complement of a
//	a//i	a is a logical expression; i is an integer expression	shift a left i binary digits if i is positive; shift a right i binary digits if i is negative
*	a*b	a and b are algebraic expressions	the product of a and b
/	a/b	the numerator a is an algebraic expression; the denominator b is an algebraic expression	the quotient of a divided by b
+	a+b	a and b are algebraic expressions	the sum of a and b
-	a-b	a and b are algebraic expressions	the difference of a and b
<	a<b	a and b are algebraic expressions	true if a is less than b
¬<	a¬<b	a and b are algebraic expressions	true if a is not less than b
=	a=b	a and b are algebraic expressions	true if a is equal to b
¬=	a¬=b	a and b are algebraic expressions	true if a is not equal to b
<=	a<=b	a and b are algebraic expressions	true if a is less than or equal to b
>	a>b	a and b are algebraic expressions	true if a is greater than b
¬>	a¬>b	a and b are algebraic expressions	true if a is not greater than b
>=	a>=b	a and b are algebraic expressions	true if a is greater than or equal to b
**	a**b	a and b are logical expressions	logical product of a and b (AND)
++	a++b	a and b are logical expressions	logical sum of a and b (OR)
--	a--b	a and b are logical expressions	logical difference of a and b (exclusive OR)
==	a==b	a and b are logical expressions	logical equivalence of a and b

time. The assembler will denote as an error any attempts to do arithmetic operations on double length floating point numbers in expressions, or on character strings longer than four characters.

Certain logical operations (**, ++, --, and ==) and all relational operations may be performed on values that require more than four characters (32 bits) to represent them.

2-17. SUBEXPRESSIONS

An expression may contain subexpressions, and subexpressions may contain other subexpressions. A subexpression is an expression enclosed in parentheses, and it may appear wherever an item is valid. Subexpressions are evaluated before other items in an expression, and the innermost subexpression is evaluated first.

The value of an item or expression is right-justified in its generated result field, and unspecified leading bit positions will contain zeros; character strings are left-justified with blanks filled to the right in the last word for unjustified characters.

Note: Character strings used in immediate operands (see Topic 6-32) are not left-justified in a fullword, but justified in the 16-bit N field. Thus, a character string immediate operand has a maximum of two characterters, and the rightmost byte is blank filled if there is only one character in the string.

The value of the part of the expression or subexpression containing and affected by a relational operator (e.g., >, <, or =) is equated to one if the relation is true and equated to zero if the relation is false. For example, if E is an expression of the form:

X >Y

then, E is evaluated as a one (1) if the relation is true, or zero (0) if the relation is false. Also, if the assigned section of expression X is not the same as the assigned section of expression Y, then the expression E cannot be completed and is evaluated as false (zero).

Examples:

1. The following expression is evaluated as zero if R is a relocatable item:

   R-4 >37

2. The following expression is evaluated as zero if the subexpression (X >Y) is false and equal to A if the subexpression (X >Y) is true:

   A*(X >Y)

2-18. ASSUMED PARENTHESES

The following examples denote how parentheses are assumed, the results being governed by the hierarchies in Table 2-3.

Expression: -A//-I*2

Method:
1. Two's complement A
2. Two's complement I
3. Shift two's complement of A by value of two's complement of I
4. Multiply result by two

Assumes: ((-A)//(-I))*2

Expression: -A//(-I*2)

Method:
1. Two's complement I
2. Multiply result of two's complement of I by two
3. Two's complement A
4. Shift result of two's complement of A by result obtained in step 2

Assumes: (-A)//((-I)*2)

Expression: -A//-(I*2)

Method:
1. Multiply I by two
2. Two's complement A
3. Two's complement the result of I multiplied by two
4. Shift the result of the two's complement of A by the result obtained in step 3.

Assumes: (-A)//(-(I*2))

2-19. LITERALS

The value of a literal is a constant. The constant is the relative location of the start of one or more words of data. The relative location is reserved by the assemb and the contents of the location are set to the value of the expression which specifies the data. An expression which is to be a literal is identified by being preceded by an equal sign (=). The assembler reserves sufficient contiguous words to contain the value of the expression. The number of words reserved for expressions which do not contain forward references is determined by the number of bit positions required to specify the value. Expressions that contain forward references are assumed to require no more than one word to specify their respective values.

Literals which have the same value are stored only once, whenever possible. Reaching the end of an assembly or using the LITORG directive (see Topic 4-13) causes all literals identified, since the last LITORG directive or since the start of the assembly, to be assigned locations and to be output. Literals appearing after a LITORG directive that are duplicates of values occurring before that LITORG directive will be assigned at least two separate locations. Further duplication will occur if the expression composing the literal is not a single item and all of the quantities composing the expression are not defined prior to their appearance in the expression.

The literal table is adjusted so that multiple-word literals are output first.

Limitations and Restrictions: The initial literal location assignment occurring after a LITORG directive or at the end of an assembly will always start at an even-word boundary (a location whose value is a multiple of two).

Multiple word literal values will be assigned locations beginning on even-word boundaries. Words that are skipped to achieve even-word alignment will not be cleared.

The value of an expression that identifies a literal is restricted to 28 characters in length.

Subexpressions and lists (see Topic 2-20) will not be made into literals.

Creation of Address Constants: Address constants will be placed in the literal table when symbols with relocatable values are used as literals. See Topics 2-27 and 2-28.

Examples:

LITERAL	VALUE
=ETA	Address of a word that contains the address of symbol ETA, if ETA is relocatable; address of a word that contains the value of ETA, if ETA is absolute
=50	Address of a word that contains the absolute value 50
=ETA+50	Address of a word that contains the value of the expression ETA+50. If this literal is used before ETA is defined, more than one constant with this value will be allocated.
=TAU+50=RHO	Address of a constant that contains zero or one (the value of the expression TAU+50=RHO)
NU+=PI	Error
=(MU, NU, XI)	Error

2-20. LISTS

A list is a set of items, expressions, or sublists separated by commas. In the most trivial case a list may be a single item. A sublist is a list enclosed in parentheses. Lists are used in the operand field of a statement.

If a list of parameters is enclosed by a single set of parentheses, these parameters are considered to be second level parameters. In the list

A, (B, C), D

B and C are second level parameters, whereas A and D are first level. Parameter 2 (at first level) is a sublist.

Restrictions: The maximum number of expressions in a list at one level is 15. The maximum number of levels of parentheses in a list is five.

The value of a parameter which is nonexistent or uncoded is always zero; e.g., for a general list: expa,(expd, expb), expx that is coded: A1, (A4,) the expb and expx would both be evaluated as zero.

2-21. INTRINSIC FUNCTIONS

An intrinsic function is an operation performed on or applied to an expression or a list. Some intrinsic functions (global intrinsic functions) may be used outside or inside procedures, whereas others (local intrinsic functions) may be used only in procedures. (This manual does not include procedure programming.)

Intrinsic function usages may be nested.

2-22. GLOBAL ATTRIBUTE FUNCTIONS

A global intrinsic function is one that may be used either outside or inside procedures.

An attribute of a parameter is the characteristic, or the value of the characteristic of the parameter; e.g., the fact that the parameter is a literal is a characteristic, or the value of the base of the parameter is the value of a characteristic.

An attribute function either determines the value of the characteristic of a specified parameter or determines on a true or false basis whether a specified parameter has a certain characteristic.

Some attribute functions are global and others are local. All global functions are also attribute functions.

A global attribute function is one that may be used either inside or outside procedure definitions and that determines the value of some characteristic of the specified parameter.

When using an assembler directive that produces multiple code (e.g., the DO directive), the argument (exp) of the intrinsic function cannot be a forward reference

2-23. Base Function - B(exp)

The base intrinsic function, B(exp), is replaced by the number of the base which yields the smallest non-negative result (displacement) when the value of that base is subtracted from the value of the expression, exp.

If two or more bases yield the same least result, the highest numbered base is selected to replace B(exp). See Topic 6-15.

Limits and Restrictions: If all bases yield a displacement greater than 4095, an error message is generated by the assembler.

The base function is replaced by:

Expression (exp)	B(exp)
external reference	error message generated by assembler
absolute	zero
relocatable	absolute

Example: RSRU ST B(TOTAL), BSV, X2

2-24. Displacement Function-D(exp)

The displacement intrinsic function, D(exp), is replaced by the displacement value of the expression, exp.

The displacement value of exp is the smallest non-negative difference between the value of the expression and the values of the bases (if any) in the table of applicable bases. See Topic 6-15.

If all bases have a value of zero, the displacement of the expression is the displacement of the expression relative to the beginning of the control section.

Limits and Restrictions: If a displacement is greater than 4095, an error message is generated by the assembler.

The displacement function is replaced by:

Expression (exp)	D(exp)
external reference	error message generated by assembler
absolute	zero
relocatable	absolute

Example: INX L X2, D(RHO)

2-25. Section Function-T(exp)

The section intrinsic function is replaced by the section number of symbol exp, if a section number is valid. See Topic 2-27.

Limits and Restrictions: If exp is an external reference, T(exp) is greater than 256 and the value of T(exp) is the sum of 256 plus the external symbol number. See Topic 4-5.

If (exp) is absolute, T(exp) is replaced by zero. If (exp) is relocatable, T(exp) is replaced by an absolute value. If (exp) is in a dummy section (see Topic 4-17), T (exp) is replaced by the negative of the dummy section number.

2-26. LOCATION INTRINSIC FUNCTION

The location counter symbol, $, when it is processed during evaluation of expressions, causes the current relative location in the assembly of the instruction procedure call to be inserted in place of the symbol. In this sense, it acts somewhat like an intrinsic function. See paragraph 2-32.

2-27. PROGRAM SECTIONS

An assembly may be divided into logical subdivisions called sections. Each section has a protection key for use in regrouping the various sections of an assembly at link edit time. A section is defined by the SEC directive; see Topic 4-15.

Sections provide the basis for addressing memory locations during an assembly. Memory locations are identified in the assembler as relative locations from the start of the section.

Section numbers are assigned to each section by the assembler. Section numbers 1 through 63 may be assigned in one assembly; i.e., any given assembly may have a maximum of 63 sections.

2-28. RELOCATION

Since the assembler does not actually place object statements in fixed Central Memory locations, the relative locations assigned by the assembler must be relocatable to available memory locations. Thus the relative location of a statement within a section is a relocatable value, and the value of a symbol, or an expression that refers to a relative location, is a section identification and the relative location within that section.

2-29. CONSTANTS

Two types of constants are identifiable during an assembly: (1) the actual value of a numeral, and (2) the relative location of a symbol within its section. The relative location of a symbol in its section is referred to as an address constant.

2-30. Address Constants

There are two classifications of address constants: (1) an address constant that is an internally relocatable value; i.e., a value whose section and relative location within its section is defined in the current assembly; and (2) an address constant that is an externally relocatable value; i.e., a value whose section and whose relative location within that section is defined in another assembly.

2-31. Numeral Constants

The value of a numeral is not relocatable. An absolute value cannot be defined as belonging to a section or to a relative location within a section. An absolute value may result from the use of relocatable items in an expression which produces loss of identity of the items within their sections (see Table 2-5).

2-32. LOCATION COUNTER

The location counter is a relocatable variable whose value is the current section number and current relative location within that section. The value of the location counter is positioned at the statement being assembled. The character, $, represents the value of the location counter symbolically. Use of $ in the operand of various control directives (see Topic 4-12) permits the value of the location counter to be changed so that assembly control may be changed to different sections or to other positions within the same section.

2-33. RELOCATABILITY OF SYMBOLS

The section to which a symbol belongs is determined by either of the following: (1) the symbol may be equated to a procedure reference statement, or, (2) it may be equated to the value in the location counter. Such symbols are relocatable since the statement's location will be relocated with the section itself. Symbols defined in other assemblies and identified by use of the EXTRN directive (see Topic 4-5) are relocatable.

Symbols which are equated to absolute expressions or items are absolute. Symbols equated to the T(a) intrinsic function are absolute.

2-34. RELOCATABILITY OF EXPRESSIONS

Expressions, because they contain symbols, may be evaluated as absolute or relocatable. An expression that would be relocatable to more than one section because the symbols in the expression are defined in different sections is illegal; e.g., A+B where A and B are relocatable and belong to separate sections.

Table 2-5 shows, for each type of operator, the relocatability of the result. The result may be relocatable, absolute, or illegal. If the result is relocatable, its section is the section of the relocatable item or items.

2-35. Effect of Relational Operators

The effect of relational operations (i.e., initiated by the operators: <, ¬<, =, ¬=, <=, >, ¬>, >=) is as follows:

A	B	A REL B
ABS	ABS	will compare, if same length
ABS	RELOC	will not compare, evaluated as false
RELOC	ABS	will not compare, evaluated as false
RELOC	RELOC	will compare if in same section

Note: Since the result of a relational operation is always zero (false) or one (true), the result is always absolute.

Table 2-5. Results of Operations on Absolute and Relocatable Items in Expressions

A	B	A+B	A-B	A*B	A/B
ABS	ABS	ABS	ABS	ABS	ABS(B≠0)
ABS	RELOC	RELOC	illegal	Note1	illegal
RELOC	ABS	RELOC	RELOC	Note2	Note3
RELOC	RELOC	illegal	Note4	illegal	illegal

A	B	A++B	A--B	A**B	A==B
ABS	ABS	ABS	ABS	ABS	ABS
ABS	RELOC	illegal	illegal	illegal	illegal
RELOC	ABS	illegal	illegal	illegal	illegal
RELOC	RELOC	illegal	illegal	illegal	illegal

Note 1: Illegal unless A equals zero or one. If A is one, the result is relocatable; if A is zero the result is an absolute zero.

Note 2: Illegal unless B equals zero or one. If B is one, the result is relocatable; if B is zero, the result is an absolute zero.

Note 3: Illegal unless B equals one. If B equals one, the result is relocatable.

Note 4: Illegal unless A and B are in the same section. If A and B are in the same section, the result is absolute.

SECTION III
LANGUAGE STRUCTURE

3-1. STATEMENT FORMAT

A program consists of a sequence of coded lines, each line containing from 1 to 80 characters. However, only the first 72 characters of a line are processed by the assembler. A line may contain a statement or a comment. Statement columns 73 through 80 can be used for program identification or for sequencing.

A statement generally consists of three coding fields: a label field, a command field, and an operand field. These three fields are of variable length and are terminated by one or more blanks; i. e., no embedded blanks are permitted in these fields. Any statement columns to the right of the operand field may be used as a remark field which contains text.

GENERAL FORM: The general form of an assembler statement is:

LABEL		COMMAND		OPERANDS		REMARKS
[symbol]	⍽	symbol	⍽	[exp1[,exp2[, ...,expn]]]	⍽	[text]
TEST		SEC		0		
START		LF		#10, INIT, X1		

3-2. CONVENTIONS FOR DESCRIBING LANGUAGE STATEMENTS

The following conventions are used to illustrate the language statements:

1. Upper case letters and punctuation marks (except those explained in items 3 and 4 below) represent information that must be coded exactly as shown.
2. Lower case letters and words are generic terms that represent information that must be supplied; i. e., a substitution must be made when coding a parameter or option so represented.

3. Information within brackets [ ] is optional. It may be included or omitted entirely, depending upon program requirements.

4. When several choices are enclosed in braces { }, one of the enclosed alternatives must be selected by the programmer. If one of the alternatives is underlined, the parameter may be omitted and the system assumes the underlined alternative.

5. Mandatory blanks are represented by a slashed, lower-case letter "b" (b). This symbol is not used to represent permissible blanks.

3-3. CONTINUATION LINES

A semicolon (;) appearing in the operand field is a line terminator which signals that the following line is to be treated as a continuation of the current one. That is, the semicolon is considered to be followed immediately by the first non-blank on the following line, and the information following a semicolon on the line on which the semicolon appeared is ignored.

Restrictions: No more than two continuation lines are permitted for each statement

A semicolon (;) cannot be used to terminate an item; such usage will be treated as an error.

Character strings and intrinsic functions are the only types of items that can be divided by a semicolon. Character strings can be divided anywhere within the string, but leading blanks on the continuation line are not treated as part of the character string. Intrinsic functions can be divided by a semicolon only after the open parenthesis.

Examples: The following lines of code illustrate use of continuation lines:

```
CMPREG L (A1*(SUM1=SUM2))++(A2*(SUM1>SUM2))++(A3*(SUM1< SUM2)), (D(;
SUM1), B2), X5 COMPUTE REGISTER TO LOAD

MSSG DATA "THIS COMPUTATION EXTENDS INTO AN UN;
DEFINED REGION."
```

3-4. LABEL FIELD

A statement may be given a name by the programmer to permit references to be made to the statement from other points within the program. The use of a name is normally optional, but some directives do require a symbol in the label field. The label must start in column 1. If no label is used, column 1 must be blank. The symbol is normally equated to the current value of the Assembler's location counter.

Examples: The following are valid labels:

A$QED
BCD345
AABBCCDD
Y

3-5. Reserved Symbols

The following symbols are reserved and may not be used as labels:

1. Symbolic register names; viz., B0 through B15, A0 through A15, X0 through X7, and V0 through V7.
2. The names of any of the directives; e.g., DATA, FORM, etc.
3. The names of any of the built-in procedures for the machine instructions, i.e., the assembler mnemonics for the machine operations.

3-6. COMMAND FIELD

Each statement has a command. The command begins with the first non-blank following the label field and is terminated by one or more blanks. The command must be a symbol.

The command field dictates the operation to be performed and may call an assembler directive or a previously defined or built-in procedure. Thus the command field contains a mnemonic which is the name of a directive or a label of a procedure. New operations may be introduced by defining new procedures. (The ASC Central Processor instruction set is represented by built-in procedures.)

Any error occurring in the command field will result in an illegal instruction. The assembler will generate one word of zeros (absolute) of loader text and will process the operand field for general syntax errors only.

Examples: DATA, SET, LR, and BLB are representative commands.

3-7. OPERAND FIELD

Most commands require operands. If a line is to include operands, the operand field begins with the first non-blank following the command field.

The operand field is composed of a list of elements. Elements are composed of one or more expressions (often referred to as parameters). The last element in the operand list is terminated by a blank; all other elements in lists are terminated by a comma. Sublists, which are elements in the form of lists enclosed in parentheses, may exist. Intrinsic functions may also be elements in an operand.

Elements omitted from the right end of an operand list are assumed to have a value of zero, (0, 0, 0 may be written as 0). If the operand field is left vacant, the remark field must also be left vacant (blank). The number of blanks between fields is not limited.

The assembler will check each expression in the operand field for valid syntax. If a syntax error is found, the assembler will print a diagnostic flag and supply zero in place of the expression found to be in error. The object code generated for the remainder of the statement depends upon the use of the expression; i.e., the command produced may or may not be correct. In some cases, a word of zeros is generated.

3-8. REMARK FIELD

The optional remark field is allowed as a convenience for documentation and has no effect upon the nature of the assembled object code. A remark must be isolated from the end of the operand field by at least one blank. The remark field may not be continued to the next line. It cannot be used if the operand field is omitted.

3-9. COMMENT LINES

A line (which is not a statement continuation line) whose first column contains an asterisk (*) is treated as entirely commentary. No loader text is generated. The line will be listed in context.

3-10. BLANK LINES

A line consisting of only spaces (blanks) in character positions 1 through 71 is treated as commentary. A blank line will be printed on the assembly listing as a result.

SECTION IV
DIRECTIVES

4-1. INTRODUCTION

Assembler directives supply special types of information to the assembler. A reference to any symbolic item in an expression on a directive line must have previously appeared as a label; e.g., it must be possible to immediately evaluate the expression(s) in the operand field of the directive. If the operand expression can not be evaluated, it will be assigned a value of zero, and an error message will be printed.

Exceptions to this rule are the symbols in the operands of the EXTRN, the ENTRY, the DATA, and the USING directives, symbols which may be forward references. The SET directive has conditional exceptions. All forward references may be satisfied with values which do not exceed one word. The value of a forward reference may be relocatable or absolute, and, if n is the value, $-2^{31} \leq n < 2^{31}$.

Note: In the general forms of the directive statements, items enclosed in brackets are optional.

4-2. DEFINITION DIRECTIVES

4-3. EQUATE DIRECTIVE (EQU)

The EQU directive is used to assign a permanent value to its symbolic label.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
symbol	↯	EQU	↯	exp

The symbolic label is defined to have the value of the expression, exp.

The value of exp may be absolute or relocatable, positive or negative, and is expanded to an integral multiple of fullwords.

Restrictions: An EQU statement must have a label.

Symbols in the expression must be defined prior to their use in the EQU directive; i.e., the expression cannot be a forward reference.

Symbols defined in the label of this directive cannot be redefined.

Limitations: The maximum number of characters in a character string named by an EQU statement is 28.

Examples:

LABEL	COMMAND	OPERANDS
ALPHA	EQU	$+3
BETA	EQU	ALPHA
NO	EQU	0

4-4. SET DIRECTIVE (SET)

The SET directive is used to assign a temporary value to its symbolic label.

GENERAL FORM:

LABEL	COMMAND	OPERANDS
symbol	SET	exp

The symbolic label is defined to have the value of the expression, exp.

The value of the symbol can be changed by redefining it as the label of another SET directive.

The value of exp which may be absolute or relocatable, positive or negative, is expanded to an integral multiple of fullwords, and the expression may contain a symbol that is a forward reference.

Restrictions: A SET statement must have a label.

If the value of the symbolic label is changed by being used as the label of another SET directive, no subsequent values can exceed the length of the first.

Limitations: If exp contains a forward reference, n, the value of n is within the range: $-2^{31} \leq n < 2^{31} - 1$.

The maximum number of characters in a character string named by a SET statement is 28.

Examples:

LABEL	⊬	COMMAND	⊬	OPERANDS
ALPHA		SET		3
YES		SET		BETA-ALPHA
BETA		SET		$

4-5. EXTERNAL NAME DIRECTIVE (EXTRN)

The EXTRN directive is used to identify every symbol that is used but not defined in the current assembly.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
blank	⊬	EXTRN	⊬	symbol[,...[,symbol]]

Each name appearing in the operand field of the EXTRN directive will be output to the Link Editor, provided that reference to that name is within the current assembly. Any declared external names to which references are not made will not be output to the Link Editor. Reference to an external name requires the use of an address constant.

WARNING: Any reference within the current assembly to an external name not declared by an EXTRN directive will be treated as being undefined and an error flag will appear on the assembly listing.

Restrictions: An EXTRN statement cannot have a label.

Limitations: A limit of 255 external symbols may be declared for a module.

A limit of 15 external symbols may be listed in the operand field of one EXTRN statement.

Examples:

LABEL	ß	COMMAND	ß	OPERANDS
		EXTRN		ALPHA
		EXTRN		SQRT, SIN

4-6. ENTRY NAME DIRECTIVE (ENTRY)

The ENTRY directive is used to establish linkages between programs that have been assembled separately but that are to be loaded and executed together.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
blank	ß	ENTRY	ß	symbol[, . . . [, symbol]]

Each name appearing in the operand field of the ENTRY directive declares an entry point into the current assembly to which external programs may refer. Any name declared to be an entry point that is not defined within the assembly will cause an error message to be output in the assembly listing.

Control section names can be used as entry points. Entry points are generated automatically for them.

Restrictions: An ENTRY statement cannot have a label.

Each name appearing in the operand field of the ENTRY directive must also appear as the label of a statement in the body of the assembly and must have a relocatable value defined in a control section.

Limitations: A limit of 255 entry names may be declared for a module.

A limit of 15 entry names may be listed in the operand field of one ENTRY statement.

Examples:

LABEL	ß	COMMAND	ß	OPERANDS
		ENTRY		ALPHA
		ENTRY		SQRT, SIN

4-7. DATA DIRECTIVE (DATA)

The DATA directive generates enough fullword data units to contain the information in the operand field.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	DATA	ß	exp[, . . . [, exp]]

The label symbol is the location of the first expression in the operand list. The symbol is given the current value of the location counter.

Each expression is expanded to a multiple of fullword units.

An address constant is generated for any expression that is not absolute.

If a generated address constant refers to an external symbol, the output module indicates that the value of the external is to be added, at link edit (or simulation) time, to the constant displacement derived from the expression in which the external symbol is used; e.g., for the statement:

```
DATA    EXTRN1 + 10, EXTRN2+(T< S)
```

in which EXTRN1 and EXTRN2 are the first and second external symbols defined in the assembly, ten will be added to the location of EXTRN1 at link edit time, and either zero or one will be added to the location of EXTRN2, depending upon whether (T< S) is false or true, respectively.

Limitations: A limit of 15 expressions may be listed in the operand field of one DATA statement.

A symbol in the operand field of the data statement that is defined by an EQU or SET statement is assumed to have a singleword value. If a symbol is set (EQU or

SET) to a value (e.g., a character string) greater than one word in length, only the rightmost word of the value will be generated as data.

Examples:

LABEL	COMMAND	OPERANDS
	DATA	BETA
ALPHA	DATA	0
FLTPNT	DATA	1.0,2.0
	DATA	"LITERAL"
	DATA	1.0D0

4-8. FORMAT DIRECTIVE (FORM)

The FORM directive is used to specify arbitrary data formats.

GENERAL FORM:

LABEL	COMMAND	OPERANDS
symbol	FORM	exp1, exp2[, exp3, • • • , expn]

The values of the absolute expressions in the operand field of the FORM directive give the bit lengths of successive fields in the resultant data word.

Reference may be made to a format definition by using its label as the command in any succeeding statement with an operand field composed of values to be placed in the object word fields defined in the FORM statement.

Restrictions: A FORM statement must have a label.

The sum of the values of the expressions must be a multiple of fullword (32-bit) units.

Limitations: A limit of 15 fields may be defined in the operand list of a FORM statement.

Examples:

LABEL	COMMAND	OPERANDS
FSTART	FORM	8, 4, 4, 4, 12
	FSTART	#C4, 2, 8, #E, 31

produces the hexadecimal code:

BYTE	0		1		2		3	
	C	4	2	8	E	0	1	F
HEX	0	1	2	3	4	5	6	7

4-9. USING DIRECTIVE (USING)

The USING directive indicates to the assembler that the specified base register contains the value of the relocatable expression.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
blank	⊬	USING	⊬	exp, register

The assembler will select a base and compute a displacement from the specified base value for each relocatable expression that follows the USING statement. The base selected will be that base (for the section) which produces the smallest displacement.

WARNING: Failure to specify a base register or registers for each section of an assembly will result in addressability errors. All relocatable values in the relocatable expression following the using directive must be previously defined or be the relocatable value.

Restrictions: A USING statement cannot have a label.

Base register zero (symbol, B0) cannot be specified as the register operand of the USING directive.

The USING directive does not produce code to place the value of exp in the base register. The programmer must include code to actually place the value in the base register.

Note: Refer to Topics 2-23, 2-24, and 6-16.

Examples:

LABEL	COMMAND	OPERANDS
	USING	ALPHA, B2
	USING	$+1, B14

4-10. DROP DIRECTIVE (DROP)

The DROP directive indicates to the assembler that the specified base register is no longer available for base selection. The base will not be considered available until another USING directive declares it to be available.

GENERAL FORM:

LABEL	COMMAND	OPERANDS
blank	DROP	register

Restrictions: A DROP statement cannot have a label. A DROP of a register for which no previous USING directive was encountered will generate an error flag.

Examples:

LABEL	COMMAND	OPERANDS
	DROP	B14

4-11. ORIGIN DIRECTIVE (ORG)

The ORG directive is used to set or reset the location of the origin for all or a portion of the section being assembled.

GENERAL FORM:

LABEL	COMMAND	OPERANDS
[symbol]	ORG	exp

The location counter is set to the value of the expression, exp. All code generated following the ORG directive will begin at the location whose value is that of the expression.

If the expression is blank, it directs the assembler to use the highest value previously assigned to the location counter of the section being assembled as the present value of the location counter.

The label, if present, is assigned the value of the location counter before the location counter is reset.

Restrictions: The expression must have a relocatable value which must be within the same control section as the ORG statement.

Examples:

LABEL	COMMAND	OPERANDS
ALPHA	ORG	$+50
	ORG	ALPHA

4-12. CONTROL DIRECTIVES

4-13. LITERAL ORIGIN DIRECTIVE (LITORG)

The LITORG directive sets the location of the origin for all literals (regardless of the referring section) defined since the previous LITORG or the beginning of the assembly directive and places the locations of the literals in their respective object code statements.

GENERAL FORM:

LABEL	COMMAND	OPERANDS
[symbol]	LITORG	[exp]

The origin is determined by incrementing the value of the expression, if necessary, to a doubleword boundary. The literals will be generated beginning at the aligned location.

The label, if present, is assigned the value of the location counter before the location counter is reset.

After the literals have been generated, the location counter will remain set to the first location following the last generated literal.

If the expression is blank, the current location counter value becomes the literal origin before alignment.

Restrictions: The expression must have a relocatable value and must be within the same control section as the LITORG statement.

Examples:

LABEL	⊬	COMMAND	⊬	OPERANDS
ALPHA		LITORG		$+10
		LITORG		

4-14. END ASSEMBLY DIRECTIVE (END)

The END directive signals termination of the assembly.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
blank	⊬	END	⊬	[exp]

The value of the expression represents the beginning execution address of the assembly when it is loaded and run (unless otherwise overridden). If the operand field is blank, no address for beginning execution of the program is output to the loader.

Whenever an END statement is encountered, it will be recognized as the end of the assembly.

Restrictions: An END statement cannot have a label.

The END directive cannot be used in a procedure.

The expression, exp, must be relocatable.

Examples:

LABEL	⊬	COMMAND	⊬	OPERANDS
		END		FIRST
		END		

4-15. SECTION DIRECTIVE (SEC)

The SEC directive defines a control section and asserts assembly control to that section for the generation of any subsequent code that is to have the same protection conditions.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	SEC	⊬	[exp1][, exp2]

The label is the symbolic name applied to the control section. Subsequent uses of a SEC statement with the same label will return control to the section at the location immediately following the highest location count used previously within the section. Sections may be resumed as desired.

The assembler assigns a control section number to the section when the defining statement is used for the first time. Section numbers are assigned sequentially.

Initially, the location counter is set to zero. An ORG directive (see Topic 4-11) may be used to adjust the location counter values.

The first operand expression, exp1, specifies the hardware protection of all code generated under control of the defined section. Expression 1 need not be used in subsequent returns to a defined section; the original protection will be assumed. Expression 1 must be absolute with a value of 0, 1, 2, or 3. An exp1 value of 0 specifies read, write, or execute (i.e., no protection); an exp1 value of 1 specifies read only; an exp1 value of 2 specifies read or write; and an exp1 value of 3 specifies execute only. If exp1 is blank, the value 0 is assumed.

The second operand expression, exp2, specifies the memory alignment for the beginning of the section. Expression2 must be absolute and the value is considered to be an exponent of 2, i.e., 2^{exp2}. If exp2 is not present, a value of 3 is assumed, and, thus, the section will be aligned on an octet boundary.

The assembler will assign a control section to the module name if no SEC directive is used. No protection (i.e., read, write, or execute) will be assumed for such a section.

Restrictions: Both expressions must be absolute.

The protection condition cannot be changed after the initial defining SEC statement; i.e., a given section has only one protection condition.

Limitations: Only one unlabeled SEC statement is allowed.

WARNING: The assembler will create absolute literals in a section with execute only protection even though they cannot be read and, therefore, cannot be used.

Examples:

LABEL	COMMAND	OPERANDS
ALPHA	SEC	1,2
BETA	SEC	0
	SEC	3,3

4-16. COMMON MODULE DIRECTIVE (COM)

The COM directive defines a common module.

GENERAL FORM:

LABEL	COMMAND	OPERANDS
[symbol]	COM	[exp1][,exp2]

The label is the symbolic name applied to the common section. Subsequent uses of a COM statement with the same label will return control to the common section at the location immediately following the highest location count used previously within the section. Common sections may be resumed as desired. If no label is present, "blank" common is defined; i.e., it is an unlabeled common section.

The protection condition of the common section is specified by exp1. The absolute value of exp1 may be 0, 1, 2, or 3 with the same protection interpretation as for the SEC directive; viz., no protection, read only, read or write only, and execute only, respectively. If exp1 is blank, zero is assumed.

The beginning boundary alignment is specified by exp2 where exp2 is an exponent of 2; i.e., the alignment will be 2^{exp2}. If exp2 is blank, three (an octet boundary) is assumed.

The module definition output generated for a COM statement has the same format as that for a SEC statement; the setting of a reserved bit within the format distinguishes the defined module as a common module.

Restrictions: Both expressions must be absolute.

The protection condition cannot be changed after the initial defining COM statement.

Examples:

LABEL	⌀	COMMAND	⌀	OPERANDS
ALPHA		COM		0,3
BETA		COM		
		COM		1

4-17. DUMMY SECTION DIRECTIVE (DUM)

The DUM directive defines an absolute dummy section.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⌀	DUM	⌀	[exp1][,exp2]

Any reference to the absolute dummy section name or to any symbol defined within the dummy section is treated as an absolute reference to a section. The symbolic name has a value of zero since it is the first location in the dummy section. The values of any other symbols defined (as labels of statements) within the section have the values of their respective displacements from the beginning of the dummy section.

A dummy section produces no object text output and no evidence will exist in the object "deck" that the DUM statement appeared in the source file.

The operand expressions have no significance other than as a comment for the protection and boundary conditions of the actual section for which the dummy section substitutes.

Examples:

LABEL	ß	COMMAND	ß	OPERANDS
ALPHA		DUM		2, 3
BETA		DUM		0, 0
		DUM		

4-18. DUMMY COMMON MODULE DIRECTIVE (COMD)

The COMD directive defines a relocatable dummy section.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
symbol	ß	COMD	ß	[exp1][, exp2]

Any reference to the dummy common module name or to any symbol defined within the dummy common section is treated as a relocatable value to which the value of the symbolic label is to be added at link-edit time. At assembly time the symbolic label has the value of relative zero and symbols defined (as labels of statements) within the dummy common section have values that are the relocatable displacements relative to the beginning of the module.

The symbolic label is assumed to be the name of a common section.

A dummy common module produces no object text output.

The operand expressions have no significance other than as a comment for the protection and boundary conditions of the actual common section for which the dummy common section substitutes.

Restrictions: A COMD statement must have a label.

Examples:

LABEL	COMMAND	OPERANDS
ALPHA	COMD	0, 3
BETA	COMD	1, 1

4-19. COPY DIRECTIVE (COPY)

The COPY directive causes the specified file, "sourcefilename," to be copied inline as source text to the assembler.

GENERAL FORM:

LABEL	COMMAND	OPERANDS
blank	COPY	sourcefilename

The source statements from the file, sourcefilename, are merged into the assembly after the COPY statement and before any later source statements in the assembly. The file may exist on an indicated user library or on the system procedure library.

The COPY function is processed during PASS 1 of the Assembler without regard to level of assembly.

Restrictions: A COPY statement cannot have a label.

The occurrence of an END statement in the copied file will cause termination of the assembly.

Limitations: The COPY function cannot be used to copy part of a source file; all card images in the file will be copied.

Examples:

LABEL	COMMAND	OPERANDS
	COPY	SOURCE
	COPY	PROC1

4-20. RESERVE DIRECTIVE (RES)

The RES directive is used to reserve space within the assembly.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↓	RES	↓	exp

The present value of the current location counter is modified by the value of the expression, exp. The expression may be positive or negative, but must be absolute.

The value of the symbolic label is the value of the current location counter before it is modified.

Restrictions: The expression, exp, must be absolute.

Limitations: The maximum value of exp is 65536.

Examples:

LABEL	↓	COMMAND	↓	OPERANDS
ALPHA		RES		10
BETA		RES		#16
		RES		100

4-21. ALIGN DIRECTIVE (ALIGN)

The ALIGN directive causes the location counter value to be incremented, if necessary, to place the next statement on a specified word boundary.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
blank	↓	ALIGN	↓	exp1, exp2

The second operand expression, exp2, specifies a basic boundary alignment and the first operand specifies a number of words past that basic alignment; e.g., ALIGN 2, 8 would specify the second word past an octet boundary.

If the current value of the location counter is not on a word boundary as specified by exp2 and expl, the location counter value will be incremented by the least value sufficient to place the new location count on the specified boundary. The contents of any locations skipped are not modified.

Mathematically, the location count must meet the criterion: $C \equiv x \bmod (y)$, where C is the location count, x is the value of expl, and y is the value of exp2. Illustratively, this means that ALIGN 2, 8 will force the location counter value to a member of the set $\{2, 10, 18, 26, \cdots\}$.

Restrictions: An ALIGN statement cannot have a label.

The expressions, expl and exp2, must be absolute.

Expl must be less than exp2.

Examples:

LABEL	COMMAND	OPERANDS
	ALIGN	0, 8
	ALIGN	1, 2
	ALIGN	0, 16

4-22. DO DIRECTIVE (DO)

The DO directive provides control of assembly by including, excluding, or repeating a variable number of statements. The result in the assembly is the same as if the "DO-controlled" statements had been included, excluded or repeated in the source input stream.

GENERAL FORM:

LABEL	COMMAND	OPERANDS
[symbol]	DO	[expl][,[exp2][,[exp3]]]

ACTION SUMMARY: The DO directive is restricted in its interpretation when used outside of procedures. It does, however, have limited use. Letting the DO parameters, expl, exp2, exp3, be represented by x, y, and z, respectively, the action of the DO directive outside of procedures may be summarized as:

1. If x > 1, assemble the next statement x times; ignore y and z.
2. If x = 1, assemble the next y statements once, and skip z statements.
3. If x < 1, skip the next y statements; ignore z.

Note: See PROGRAMMERS' GUIDE TO PROCEDURE PROGRAMMING for the interpretation of the DO directive when it is used within procedures.

Restrictions: Of the intrinsic functions only B(exp), D(exp), and T(exp) are valid as parameters of a DO statement used outside of a procedure.

Outside of procedures, DO directives can be nested only to exclude the nested DO statement; they cannot be used to cause repetition of the nested DO statement's control range.

Outside of procedures, no statements within the range of a given DO statement other than a nested DO statement or a SET statement may have labels.

Default Values for Parameters: The following table illustrates the Assembler's interpretation of the DO parameters for the various cases of coding:

	expl < 1	expl = 1	expl >1
CODED	ASSUMES	ASSUMES	ASSUMES
DO x	DO x, 1, 0	DO 1, 1, 0	DO x, 1, 0
DO x,	DO x, 1, 0	DO 1, 1, 0	DO x, 1, 0
DO x, ,	DO x, 0, 0	DO 1, 0, 0	DO x, 1, 0
DO x, , z	DO x, 0, 0	DO 1, 0, z	DO x, 1, 0
DO x, y	DO x, y, 0	DO 1, y, 0	DO x, 1, 0
DO x, y,	DO x, y, 0	DO 1, y, 0	DO x, 1, 0
DO x, y, z	DO x, y, 0	DO 1, y, z	DO x, 1, 0

Note: If expl is defaulted, it will be assumed to be zero.

ITERATION COUNT: The value of expl specifies the number of times the iteration group is to be assembled. This value is called the iteration count.

An expl value of less than one causes the iteration group to be skipped without action. An expl value of n, where n is greater than or equal to one, causes the iteration group to be assembled n consecutive times.

Restrictions: Outside of procedures, if expl is greater than one, exp2 defaults to one; if expl is equal to or less than one, exp2 may be greater than one.

Limitations: The effective value of expl is limited to the range: $0 \leq \text{expl} \leq 255$.

ITERATION GROUP: The value of exp2 specifies the number of statements to be assembled as a group. This group, called the iteration group, begins with the statement immediately following the DO statement.

Restrictions: Outside of procedures, if expl is greater than one, exp2 defaults to one.

Note: Commentary lines are not statements and are, therefore, ignored in DO directive iterations.

SKIP COUNT: The value of exp3 specifies the number of contiguous statements in the source stream that are to be excluded when the iterations are complete. This value is called the skip count.

The statements skipped are those that immediately follow the last statement iterated.

If expl is less than one, the DO execution is complete after the iteration group is skipped and exp3 is ignored; i.e., only the number of statements equal to exp2 will be skipped.

Restrictions: Outside of procedures, exp3 is defaulted to zero for all cases other than those in which expl is equal to one.

SATISFACTION OF PARAMETERS: When a number of statements equal to (or greater than) the value of exp2 have been assembled in one iteration, exp2 is said to be satisfied for that iteration.

When a number of iterations equal to the value of expl has been completed, expl is said to be satisfied for that DO statement.

The DO directive is said to be satisfied when the iterations and/or skips specified by all three parameters have been completed.

DO LABEL: A label on the DO directive provides symbolic access to the number of the iteration the DO directive is performing (or has performed) at the time reference is made to the label.

When the DO statement has been encountered in the source stream, the label is initially given a value of one; thus, if the DO label is used as one of the parameters of its own DO statement, that parameter will always be evaluated as one. This initial assignment of the label value overrides any previous assignment of a value to that symbol by a previous SET or DO statement.

The value of the DO label is incremented at the beginning of each iteration of the DO directive; thus, the value of the label is always the number of the iteration being performed, or is the number of the last iteration performed once the DO directive is satisfied. Any attempts to modify the value of the DO label by a SET statement or as the label of a nested DO statement within the range of the DO directive will cause an anomalous assembly. In summary, if a DO statement has an iteration count, (value of expl) of n and has a label, the value of the label will be incremented through the series $\{1, 2, 3, \ldots, n\}$ in successive iterations. If the value of expl is less than one, the label will always have the value of one.

Once the DO directive is satisfied, the label retains its last value unless or until its value is modified by a SET statement or it is used as the label of another DO statement, or the DO statement is reaccessed. Such modifications must occur outside the range of the subject DO directive.

Restrictions: The value of a DO label cannot be preset to a value other than one.

The value of a DO label cannot be modified within the range of the subject DO directive, other than by its own iteration incrementation.

NESTED DO DIRECTIVES: A DO directive is said to be nested when it is one of the statements included in the iteration group of another DO directive which is called the parent DO directive. DO directives may be nested up to 32 levels.

Restrictions: An error message will be returned if the range of a nested DO directive exceeds the range of its parent DO directive even through the resultant assembly may be the desired result.

Nested DO directives are not permitted outside of procedures. The only exception is a parent DO directive which never assembles its iteration group and thus always excludes the nested DO directive from the assembly. All other attempts to nest DO directives outside of procedures will produce errors.

4-23. PSEUDO DIRECTIVES

4-24. INDIRECT ADDRESS CONSTANT DIRECTIVE (IND)

The IND directive is used to generate an indirect address constant.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	IND	ß	exp1[,exp2]

Expression1 represents the address value and expression2 represents the second level index. Expression1 may be preceded by an at sign (@) to indicate another level of indirectness.

Examples:

LABEL	ß	COMMAND	ß	OPERANDS
GAMMA		IND		@BETA
		IND		SIGMA

Restrictions: Access to indirect address constants are execute requests; therefore, indirect address constants must be in control sections with execute permited protection codes.

4-25. BRANCH ADDRESS CONSTANT DIRECTIVE (BCON)

The BCON directive is used to generate a branch address constant to be use by the link editor for automatic overlaying. Indirect branches thru BCON addre constants will be trapped by the link editor to invoke the overlay supervisor to o lay the segment containing the target address.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↓	BCON	↓	expl

EXPl represents the address value to be trapped.

Examples:

LABEL	↓	COMMAND	↓	OPERANDS
CALL SCAN		BCON		SCANENT
		BCON		OVLY3

Restrictions: Access to branch constants are execution requests and therefore must be in execute permit control sections.

4-26. DATA HALFWORD DIRECTIVE (DATAH)

The DATAH directive will place the values of expressionl and expression2 into the left and right halves, respectively, of the word generated by the statemen Both expressions must be absolute.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↓	DATAH	↓	expl, exp2

Examples:

LABEL	↓	COMMAND	↓	OPERANDS
ALPHA		DATAH		0, 1
		DATAH		-3, 2

4-27. LISTING DIRECTIVES

-28. SKIP DIRECTIVE (SKIP)

The SKIP directive permits control of the assembly listing. This directive auses the assembler to skip print lines or to eject the page of the assembly listing. he contents of the operand field also control the printing of the heading at the top f the new page. The directive itself is not printed on the listing.

ENERAL FORM:

LABEL		COMMAND		OPERANDS
blank	⊬	SKIP	⊬	[exp[, character string]]

If the expression is zero or blank, the page will be ejected. Otherwise, a umber of print lines equal to the value of the expression will be skipped. The ex- ression must be absolute.

SKIP directives are ignored if the NOLIST directive is used.

The operand field of the SKIP directive may have any of the following formats:

FORMAT	FUNCTION
SKIP	Eject the page; new page number equals old page number plus one; print previous title.
SKIP n	Skip n lines; if this causes page ejection, start at top of page as in previous format.
SKIP 0, "TITLE"	Eject the old page; new page number equals old page number plus one; new title is the character string, TITLE.
SKIP n, "TITLE"	Skip n lines; this causes page ejection, start at top of new page; new title for the next page is the character string, TITLE, regardless of whether page is ejected now; new page number equals old page number plus one.

estrictions: No label is allowed with the SKIP directive.

The expression must be absolute.

Limitations: The character string may not exceed 100 positions.

Examples:

LABEL	∅	COMMAND	∅	OPERANDS
		SKIP		3
		SKIP		
		SKIP		0,"TITLE"
		SKIP		6,"TITLE"

4-29. LIST DIRECTIVE (LIST)

The LIST directive is used to cause the object code listing to resume.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
blank	∅	LIST	∅	blank

The combination of NOLIST and LIST directives can be used when only a po tion of the assembly listing is desired. The directive is not printed on the listi

Restrictions: No label is allowed with the LIST directive.

Example:

LABEL	∅	COMMAND	∅	OPERANDS
		LIST		

4-30. NOLIST DIRECTIVE (NOLIST)

The NOLIST directive is used to suppress the listing.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
blank	∅	NOLIST	∅	blank

When the assembler encounters this directive, it stops the listing. The dir tive is not printed on the listing.

Restrictions: No label is allowed with the NOLIST directive.

Example:

LABEL	↯	COMMAND	↯	OPERANDS
NOLIST				

SECTION V
ASSEMBLER OUTPUT

5-1. ASSEMBLER OUTPUT

The Assembler output consists of object modules and a listing with messages.

5-2. SOURCE PROGRAM LISTING

An assembly listing will be output in the format shown in Figure 5-1. A double space listing may be requested by CPSTEP option, or listing may be suppressed by CPSTEP option.

Columns with headings are provided for error flags, the location of each statement, the object text generated, the statement number, and each source statement. If program sequencing has been requested, an unheaded column of sequence numbers will be printed.

Error Column (ERRORS): The ERROR column contains the flags for assembler error messages and procedure processing error messages. These error flags and their meanings are described in the section on Messages.

Statement Location Column (LOCATION): The LOCATION column contains the location (in hexadecimal numerals) relative to the beginning of its section for each source statement.

Object Code Column (OBJECT TEXT GENERATED): The OBJECT TEXT GENERATED column contains the hexadecimal object code generated by each statement in the source program.

Statement Number Column (STMT): The STMT column contains decimal statement numbers for each source statement in an assembly.

Note: The user should note that the Central Processor Procedure Library statements which precede each program are not listed; therefore, statement numbers begin at some number greater than zero.

```
TEXAS INSTRUMENTS -- CPU ASSEMBLER LISTING                    PAGE    1

ERRORS LOCATION OBJECT TEXT GENERATED          STMT              SOURCE STATEMENT                    02/05/70


                                               669

                                               670 BRCHTST  SEG    0
                                               671          USING BRCHTST,B1
                                               672          ORG   BRCHTST+#F
      404040                                   673 SUM      AF    A1,(A5)
      00000F  42 1 0 0 0 015                   674          BCLF  X1,A6,SUM
      000010  86 1 6 0 FFF                     675          CF    A1,(A2)
      000011  CA 1 0 0 0 012                   676          BG    TOTAL
      000012  91 2 0 0 1 8A0                   677          ORG   BRCHTST+#8A0
      00000F                                   678 TOTAL    ST    A1,OUT,X5
      000840  24 1 0 5 1 A10                   679          ORG   BRCHTST+#A10
      000840                                   680 OUT      RES   5
      000A10                                   681          END   BRCHTST
      000A15  00000000

                                                     E N T R Y   N A M E S

                                                     BRCHTST   01 000000


ASSEMBLY COMPLETE.    NO STATEMENTS HAVE ERRORS.
```

Figure 5-1. Example Source Program Listing

Source Statement Column (SOURCE STATEMENT): The SOURCE STATEMENT column contains each symbolic source statement in the assembly input.

SUMMARY OF SPECIAL LISTS: Following the listing of source statements for an assembly is a summary of special lists and the number of statements in error for each section. The format is as follows:

LITERALS ASSIGNED TO SECTION hh

location counter value literal

ENTRY NAMES

entry name location counter value

EXTERNAL NAMES

external name

ASSEMBLY COMPLETE xxxx STATEMENTS IN ERROR

5-3. MESSAGES

An assembly listing line consists of the hexadecimal representation of the location counter and machine language instruction followed by the number and image of the original source statement. Message flags are indicated to the left of the location counter value.

For each line upon which an error condition is discovered (which results in a message flag being listed) a word containing an error count will be incremented by one. At the end of the assembly run, this count is set in the specified word on the listing. A maximum of six flags will appear on the listing for each line of generated listing.

Table 5-1. Assembler Generated Messages

FLAG	ERROR CONDITION
A	Addressability Error
B	Invalid Use of Base Intrinsic
C	Too Many Continuation Cards
D	Duplicate Label Assignment
E	General Syntax Error
F	Intrinsic Function Invalidly Used
G	Invalid Use of a List
H	Invalid Use of a Directive
I	Undefined Instruction
J	Invalid Use of Displacement Intrinsic
K	Invalid Use of Control Section Intrinsic
L	Error in the Label Field
M	Magnitude Error
N	No END Card on Deck
O	Too Many Operands on Statement
P	Parentheses Are Unbalanced
Q	Invalid Arithmetic Operation
R	Relocation Error
S	Truncation Has Occurred
T	Assembler Table Overflow
U	Undefined Symbol
V	Invalid Forward Reference
W	Warning, Possible Error
X	Reserved for Future Use
Y	Reserved for Future Use
Z	Disagreement in Location Counter between Pass One and Pass Two.

5-4. Procedure Processing Generated Messages

The message flags following are generated by the Central Processor and the Peripheral Processor procedures that the assembler processes. User generated procedures may generate additional flags, or the same flag with a different meaning.

Table 5-2. Procedure Processing Message Symbols

FLAG	ERROR CONDITION
1	Invalid use of a register, or insufficient number of parameters.
2	Questionable use of a register, or insufficient number of parameters.
3	Invalid use of a literal for the ASC.
4	Invalid use of an @ for the ASC.
5	Invalid use of a ¢ for the ASC.

5-5. CROSS-REFERENCE LISTING

A cross-reference listing will be output in the format appearing in Figure 5-2. The cross-reference listing can be suppressed by control card option.

Each new symbol encountered is entered in the SYMBOL column and its definition and/or cross-references are listed to the right through the TYPE, SEC, VALUE, DEFN, and REFERENCES columns. Each new symbol will have a SYMBOL and TYPE entry in the cross-reference listing. If the symbol is defined, the SEC, VALUE, and DEFN fields will be filled; if the symbol is not defined, these three fields will contain hyphens. A series of hyphens in any field indicates that, for that symbol, the field is not applicable, not available, or not known.

Symbol Column (SYMBOL): The SYMBOL column contains the symbol whose cross-references are listed in the succeeding columns and lines of the listing format. All entries prior to the next entry in the SYMBOL column refer to the given symbol.

```
                                        CROSS-REFERENCE LISTING                                    PAGE    2

 SYMBOL   TYPE SEC   VALUE    DEFN     REFERENCES                                                  02/05/70

A1        VAR  -- 00000011     29      673   675   678
A2        VAR  -- 00000012     30      675
A5        VAR  -- 00000015     33      673
A6        VAR  -- 00000016     34      674
BRCHTST   ENT  01 00000000    670      671   672   677   679   681
B1        VAR  -- 00000001     13      671
I$N       ---  -- -------- UNDEF      273   276   320   337   354   371   412   415   505   508
OUT       REL  01 00000A10    680      678
SUM       REL  01 0000000F    673      674
TOTAL     REL  01 000008A0    678      676
X1        VAR  -- 00000021     45      674
X5        VAR  -- 00000025     49      678
 STEP-ASMC     DATE-02/05/70  EXECUTION TIME-00HRS 00MINS 52.67SECS
```

Figure 5-2. Cross-Reference Listing Example

Type Column (TYPE): The TYPE column contains an abbreviation specifying the type of the symbol being cross-referenced. The meanings of the abbreviations in the TYPE column are as follows:

ABBREVIATION	MEANING
REL	internally relocatable
ABS	absolute
EXT	external
ENT	entry point
VAR	variable, section and value may change; the first section and value is displayed

The VAR type symbol results from the definition of a SET or a DO directive outside of all procedures. Symbols defined by SET and DO directives inside a procedure have definition information only.

Section Column (SEC): The SEC column contains the number of the section to which the symbol belongs and in which it is defined.

Value Column (VALUE): The VALUE column contains the value assigned to or the evaluation of the symbol being cross-referenced. Depending upon the type of the symbol, its value may be an address constant, a value set by the programmer, or the evaluation of the symbol via operations. If the VALUE column contains eight asterisks, the value of the symbol cannot be represented in eight hexadecimal digits.

Definition Column (DEFN): The DEFN column contains the number of the statement in which the symbol is defined.

References Column (REFERENCES): The REFERENCES column contains a complete listing of the statement numbers of all the statements in which the symbol being cross-referenced appears. This will not include its definition statement number that appears in the DEFN column.

SECTION VI
ASSEMBLER-CENTRAL PROCESSOR INTERFACE

6-1. INTRODUCTION

The assembler produces machine code statements by interpretation of symbolic coding through procedures that are defined in the assembler. The names by which the procedures may be called (i.e., the assembler mnemonics for the central processor instructions) are described in Sections VII and VIII of this manual.

This section of the manual describes the modes of interpretation of symbolic code into machine code for scalar instructions. Since the vector operations have special characteristics, both in the assembler procedures and in the machine, they are described in Section VIII, which is devoted entirely to the vector operations.

6-2. INSTRUCTION FORMATS

The machine instruction has an eight-bit operation code field, a four-bit R field which either specifies a register whose contents are used or altered in the operation or is a condition mask for the operation, a four-bit T field whose most significant bit specifies indirect addressing and whose other three bits specify an index register, a four-bit M field which specifies the base register to be used in address development, and a 12-bit N field which specifies the displacement to be used in address development. For instructions which use immediate operands, the M and N fields compose a 16-bit immediate operand field. For index, test, and branch instructions, the T field addresses arithmetic registers. All three machine formats are illustrated in Figure 6-1.

The assembler statement of a scalar instruction may have a label, must have a command, and may have from one to three operands.

In general:

LABEL		COMMAND		OPERANDS
[symbol]	↓	mnemonic	↓	[operand1,]operand2[, operand3]

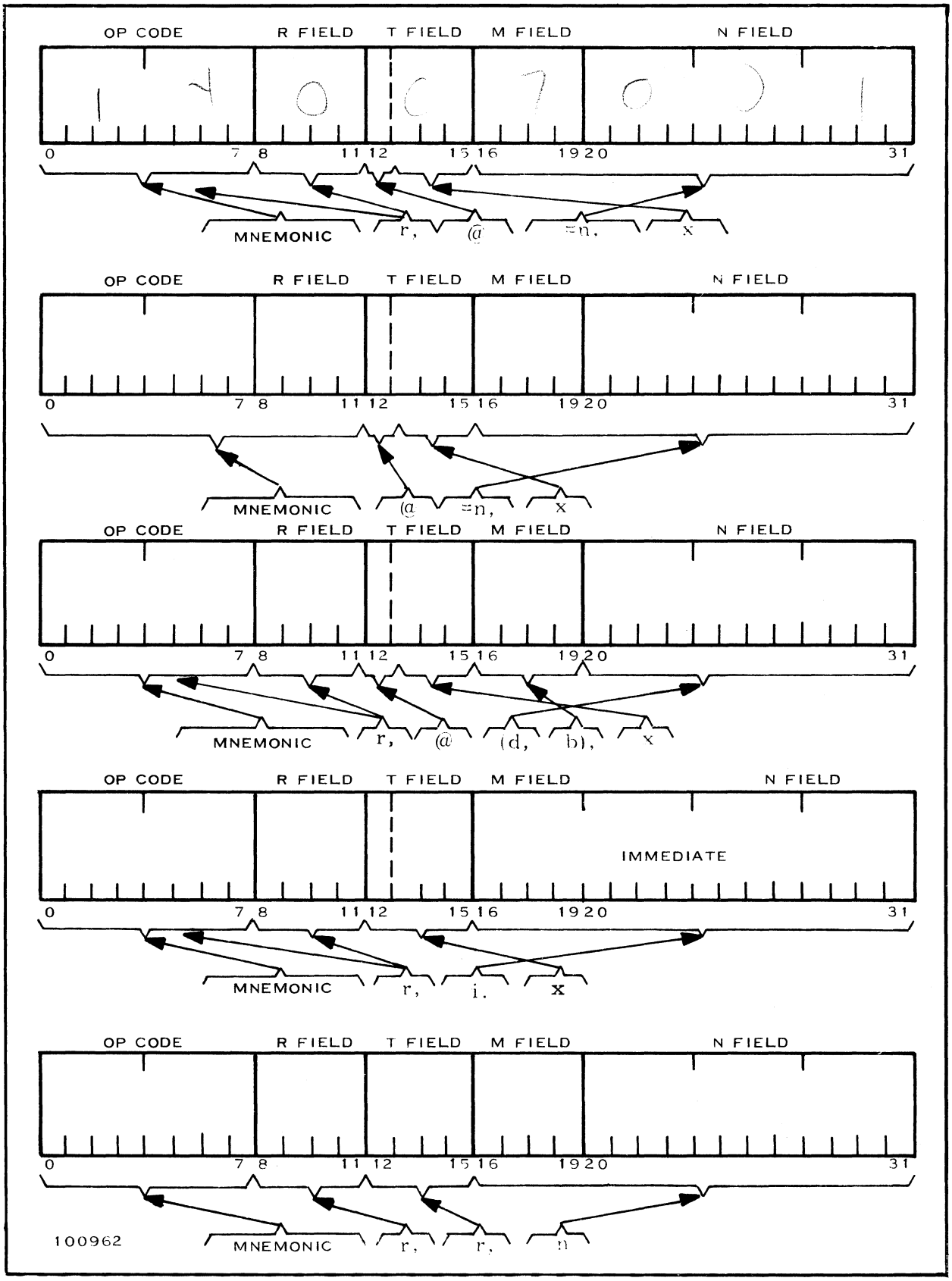


Figure 6-1. Assembler Statement Translations into Machine Code

6-3. LABEL

A label on an assembler instruction is optional. When it is present, it is a symbol (see Topic 2-5) and it is assigned a relocatable value which is its relative position in the assembly. If a label is not present, its field must be represented by one or more blanks.

The label is not interpreted into any part of the machine instruction.

6-4. COMMAND

The command in the assembler instruction is mandatory. The command is one of the mnemonics defined by procedures in the assembler. This mnemonic, in conjunction with information from the operands, determines the operation code field of the machine code instruction (see Figure 6-1). Note that the machine operation code may not be determined from the command mnemonic alone; e.g., the mnemonic L for load instructions may be translated into any one of three machine codes depending upon the register operand included in the statement.

6-5. OPERANDS

The operand list varies in number of operands and in operand interpretation from instruction to instruction, but there are only two basically different operand lists. They may be called the R, N, X list and the R, R, N list, respectively. Table 6-1 is a definitive list of all operand combinations.

6-6. R, N, X OPERAND LIST

The R, N, X operand list is basic to all except two of the scalar instructions.

6-7. First Operand

The first operand is that operand which is translated into the R field of the machine instruction and also helps determine the operation code for some instructions. See Figure 6-1.

Table 6-1. General Forms and Variations of the Operand Lists

GENERAL FORMS	DEFINITIVE VARIATIONS			
	SYMBOLIC N		EXPLICIT BASE AND DISP	
R, N, X Lists:				
r, [@][=]n[, x]	r,@=n, x r,@n, x r, =n, x r, n, x	r,@=n r,@n r, =n r, n	r,@(d, b), x r, (d, b), x	r,@(d, b) r, (d, b)
r, [@[=]]n[, x]	r,@=n, x r,@n, x r, n, x	r,@=n r,@n r, n	r,@(d, b), x r, (d, b), x	r,@(d, b) r, (d, b)
r, [@]n[, x]	r,@n, x r, n, x	r,@n r, n	r,@(d, b), x r, (d, b), x	r,@(d, b) r, (d, b)
m, [@[=]n[, x]	m,@=n, x m,@n, x m, n, x	m,@=n m,@n m, n	m,@(d, b), x m, (d, b), x	m,@(d, b) m, (d, b)
m, [@]n[, x]	m,@n, x m, n, x	m,@n m, n	m,@(d, b), x m, (d, b), x	m,@(d, b) m, (d, b)
[@[=]]n[x,]	@=n, x @n, x n, x	@=n @n n	@(d, b), x (d, b), x	@(d, b) (d, b)
[@]n[, x]	@n, x n, x	@n n	@(d, b), x (d, b), x	@(d, b) (d, b)
r, i[, x]	r, i, x	r, i	—	—
i[, x]	i, x	i	—	—
i		i	—	—
R, R, N Lists: r, r, n		r, r, n		r, r, (d, b)

The first operand may be a register symbol or value, a mask, or supplied by the assembler. For those instructions where the assembler supplies the R field of the object instruction, no "first" operand is coded; therefore, it is convenient to refer to it as the register operand or the mask operand, as appropriate.

GENERAL FORM SYMBOLS: In general forms, the first operand, when present, will be represented by symbols as follows:

SYMBOL	MEANING
r	replace r with a register symbol or register value
m	replace m with an absolute expression
neither	no "first" operand, the R field is supplied by the assembler

6-8. Second Operand

The second operand is that operand which is translated into the M and N fields of the machine instruction and, also, specifies whether the most significant bit of the T field is to be set to one or zero (i.e., indicates whether to use indirect addressing.) See Figure 6-1.

The second operand may be a relocatable expression, a sublist specifying base and displacement, or an absolute expression which is an immediate operand. Instructions which use relocatable symbols also permit base and displacement sublists and vice versa, but absolute expressions and relocatable values are not interchangeable.

Since the "first" operand is not always coded, it is convenient to refer to the "second" operand as the address operand (when a relocatable symbol or a base and displacement sublist) or as the immediate operand (when an absolute expression).

An address operand may be preceded by an @ sign to indicate indirect addressing (see Topic 6-27), and its presence causes the most significant bit of the T field of the object instruction to be set to one.

BASE AND DISPLACEMENT SUBLIST: When a relocatable expression is coded in the second operand, it is provided by the assembler with a base and displacement. When it is desired to code the base and displacement explicitly, the address operand is coded as a sublist (i. e., a list enclosed in parentheses) with the displacement first, a comma, and then the base register symbol or value. The displacement is translated into the N field and the base into the M field of the object instruction. See Figure 6-1. Both the displacement and the base must be absolute expressions.

RESTRICTIONS ON LITERALS: In all instructions in which the data flow is from the location specified by the address operand, a literal may be used as an address operand since it is given a relocatable value.

In those instructions in which the data flow or the flow of control (i. e., a branch) is to the location specified by the address operand, use of a literal is restricted and may be prohibited. For some branch instructions a literal which will create an indirect address is permitted in conjunction with the indirect address symbol.

Note that, in any case where a literal is conjuncted with indirect addressing, the literal must be assembled into an indirect address.

A base and displacement sublist, since it is a list, cannot be a literal.

GENERAL FORM SYMBOLS: In general forms, the second operand will be represented by symbols as follows:

SYMBOL	MEANING
n	replace n with a relocatable expression or a base and displacement sublist (d, b)
@ n	use the indirect address cell at location n + index to develop a terminal effective address
=n	assembler give =n a relocatable value and store the value of n at that location (n may be absolute)
@=n	assembler give =n a relocatable value and store the indirect address, n, at that location
(d, b)	explicit base and displacement address (Continued)

SYMBOL	MEANING
d	replace d with a positive absolute expression, or a register symbol or value
b	replace b with an absolute expression, or a base register symbol or value
i	replace i with an absolute expression

6-9. Third Operand

The third operand is that operand which is translated into the three least significant bits of the T field of the object instruction. See Figure 6-1.

The third operand will always be an index register symbol or value. It is convenient to refer to it as the index operand. In general forms, the index operand will be represented by the symbol x.

It is always optional to leave off the index operand, but there is one instruction, LLA, in which an index operand is prohibited.

6-10. R, R, N OPERAND LIST

There are only two instructions, BCLE and BCG, which use the R, R, N operand list. It differs from the R, N, X list in that it is the second operand that is translated into the T field of the object instruction. See Figure 6-1.

The first operand is always an arithmetic register symbol or value, the second operand is always an even arithmetic register symbol or value, and the third operand is always a relocatable address (branch) symbol. The third operand does not permit use of indirect addressing or literals.

The algorithm of these two instructions is discussed in Topic 7-127.

For convenience, the first operand may be called the register operand, the second operand may be called the test operand, and the third operand may be called the address operand.

6-11. REGISTER ADDRESSING

The bank of 48 registers in the Central Processor are separated into six eight-word files: the two base register files, A and B, the two arithmetic files, C and D, the index register file, X, and the vector register file, V.

Symbolic addressing of the registers may be performed by their reserved symbols built into the assembler procedures, by their decimal positions in the register bank, by their hexadecimal positions in the register bank, or by any absolute expression which is equated to any of the previous addresses.

Figure 6-2 illustrates the division of the register bank into files, and Table 6-2 lists the symbols by which the individual registers may be addressed.

Note that all register addressing modes are absolute values.

6-12. REGISTER OPERAND - R FIELD ADDRESSES

Register operands are translated by the assembler into the R field of the object code.

Since there are 48 registers but only four bits in the R field, the translation is performed modulo 16. For those assembly instructions which may refer to any register in the bank (e.g., L), the modulus number (i.e., first 16, second 16, third 16) or the symbol letter (i.e., Bx, Ax, Xx, or Vx) determines which object operation code is selected. In the object code, it is the operation code that specifies which group of 16 registers is accessed, whereas in the assembler code the register operand specifies which operation code to select. Note that the index and vector register files are accessed in the same modulus, and, therefore, instructions whose names specify an index register in the R field also may use vector registers.

SYMBOLIC ADDRESS	DECIMAL ADDRESS	HEXADECIMAL ADDRESS	REGISTER BANK	FILE	
B0	0	#0	WIRED TO ZERO	A	BASE REGISTERS (BR)
B1	1	#1			
⋮	⋮	⋮			
B7	7	#1			
B8	8	#8		B	
⋮	⋮	⋮			
B15	15	#F			
A0	16	#10		C	ARITHMETIC REGISTERS (AR)
⋮	⋮	⋮			
A7	23	#17			
A8	24	#18		D	
⋮	⋮	⋮			
A15	31	#1F			
X0	32	#20		X	INDEX REGISTERS (XR)
⋮	⋮	⋮			
X7	39	#27			
V0	40	#28		V	VECTOR REGISTERS (VR)
⋮	⋮	⋮			
V7	47	#2F			

Figure 6-2. Register File Specifications

Table 6-2. Register Addressing Symbols

SYMBOLIC ADDRESS	NUMERIC ADDRESS		SYMBOLIC ADDRESS	NUMERIC ADDRESS	
	DECIMAL	HEXADECIMAL		DECIMAL	HEXADECIMAL
B0*	0	# 0	A8	24	# 18
B1	1	# 1	A9	25	# 19
B2	2	# 2	A10	26	# 1A
B3	3	# 3	A11	27	# 1B
B4	4	# 4	A12	28	# 1C
B5	5	# 5	A13	29	# 1D
B6	6	# 6	A14	30	# 1E
B7	7	# 7	A15	31	# 1F
B8	8	# 8	X0**	32	# 20
B9	9	# 9	X1	33	# 21
B10	10	# A	X2	34	# 22
B11	11	# B	X3	35	# 23
B12	12	# C	X4	36	# 24
B13	13	# D	X5	37	# 25
B14	14	# E	X6	38	# 26
B15	15	# F	X7	39	# 27
A0	16	# 10	V0	40	# 28
A1	17	# 11	V1	41	# 29
A2	18	# 12	V2	42	# 2A
A3	19	# 13	V3	43	# 2B
A4	20	# 14	V4	44	# 2C
A5	21	# 15	V5	45	# 2D
A6	22	# 16	V6	46	# 2E
A7	23	# 17	V7	47	# 2F

*When these values are used in the b parameter of the explicit address sublist, (d, b), they specify that there is no base.

**When these values are used in the index operand, they specify that there is no index.

Examples: The following assembler statements would translate into the illustrated hexadecimal object code:

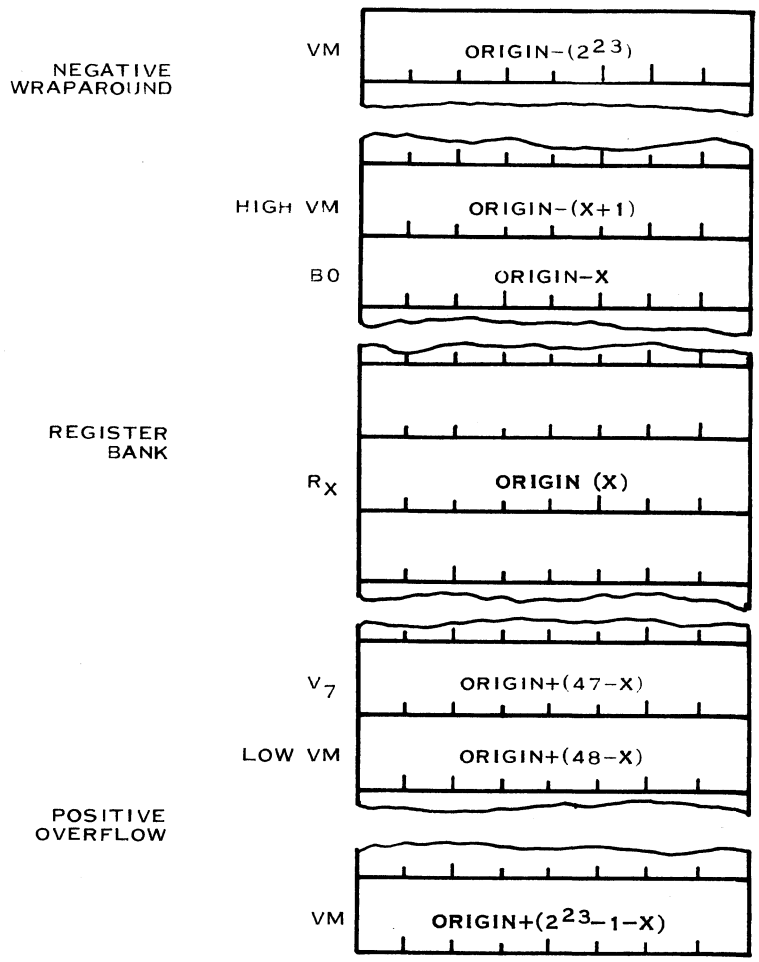
LABEL	ß	CMMND	ß	OPERANDS	OP	ß	R	ß	T	ß	M	ß	N
		L		B2,(X1),X2	18		2		2		0		021
		L		A3,(A4)	14		3		0		0		014
		L		#13,(20)	14		3		0		0		014
		L		X2,(X3)	1C		2		0		0		023
		L		V3,(X1)	1C		B		0		0		021

6-13. ADDRESS OPERAND REGISTER ADDRESSES

Registers may be addressed in the address operand (N field) of most instructions by coding a base and displacement sublist with base register specification of zero (implied or explicit).

Since the N field is 12 bits, references to registers in the N field are by their hexadecimal positions from the beginning of the register bank. Refer to the example in Topic 6-12.

Indexing address operand register references may produce effective addresses in virtual memory since the index word sets are greater than 48 words. Refer to Topic 6-23. Illustratively, the index word set for a register origin, x, is:



Note: VM = Virtual Memory

6-14. ADDRESS DEVELOPMENT

From the viewpoint of the user, address development occurs in two phases: (1) the assembler's interpretation of symbolic addresses into object code, and (2) the machine's interpretation of object code. The user must be able to anticipate the final result of his coding. Table 6-3 gives the specifications for direct single-word addresses.

6-15. ASSEMBLER TRANSLATION

The general translation of an assembler statement into an object statement is described in Topics 6-6 through 6-10. Translation of an address operand into the base and displacement fields is discussed in greater detail in the following topics.

6-16. Symbolic Addresses

Since Central Processor instructions use base and displacement addressing (M and N fields), the programmer must provide the assembler with information about the use of base registers. This is accomplished with the USING directive, Topic 4-9.

In developing the base and displacement fields for a symbolic address, the assembler determines which base registers are in use, and selects that base registe whose contents when subtracted from the program counter relative displacement of the symbol will give the smallest positive N field displacement. Then that base regi ter's object address is assembled into the M field of the object instruction, and the derived N displacement is assembled into the N field of the object instruction.
Limitations: The N displacement must be within the range: $0 \leq N \leq 4095$. The N displacement is translated into the 12-bit N field as a positive number.

Default: The base register addressed in the M field will be base register zero if no base registers are specified by the USING directive to be in use. This produces program counter relative displacements, and for $0 \leq N \leq 47$ produces register addressing in the address operand.

Note: Note that there are three distinct displacements involved in programming the Central Processor in assembler language. First, there is the displacement of a symbolic location from the beginning of the program (section) in which it is named; this is called program counter relative displacement. Second, there is the displacement of a symbolic location from some specified base position within the program in which it is named; this, for convenience, will be called the N displacement. Third, there is the index displacement specified by the contents of an index register addressed in the index operand.

Example: The following assembler code would produce the illustrated object code (note that SUM is addressed by base register 2 because a smaller displacement is possible) and results on execution:

LABEL	ß	CMMND	ß	OPERANDS	COUNT	OP	ß	R	ß	T	ß	M	ß	N
EXAMP		SEC		0										
		USING		EXAMP, B1										
		USING		EXAMP+50, B2										
		BLB		B1, $+1	0	98		1		0		0		001
		AI		B1, -1	1	70		1		0		F		FFF
		LI		X1, 50	2	5C		1		0		0		032
		L		B2, (X1)	3	18		2		0		0		021
		⋮		⋮	⋮									
		LI		X1, 2	(6)	5C		1		0		0		002
		⋮		⋮	⋮									
		ST		A3, SUM, X1	(F)	24		3		1		2		00A
		⋮		⋮	⋮									
SUM		DATA		0, 0, 0, 0, 0	(3C)	00		0		0		0		000
					(3D)	00		0		0		0		000
					(3E)	00		0		0		0		000
					(3F)	00		0		0		0		000
					(40)	00		0		0		0		000

On execution:

COUNT	REGISTER CONTENTS		VIRTUAL MEMORY CONTENTS
0	B1	0000 0001	
1	B1	0000 0000	
2	X1	0000 0032	
3	B2	0000 0032	
⋮			
6	X1	0000 0002	
⋮			
F	A3	40E0 0000	3E 40E0 0000

6-17. Explicit Base and Displacement Addresses

Explicit base and displacement address interpretation is described in Topic 6-8.

Base and displacement as calculated by the assembler can be avoided, whenever desirable, by explicit base and displacement addressing in the address operand. Use of explicit base and displacement requires that the programmer know both the contents of the base register he specifies and the exact displacement from that base location.

The principal use of the base and displacement sublist in the address operand is to develop a register address in the M and N fields of the object code.

Limitations: Any expression used as the d parameter must have a value within the range: $0 \leq d \leq 4095$. Any expression used as the b parameter must have a value within the range: $0 \leq b \leq 15$.

Note: As within any list, an empty paramter is provided, by the assembler, with the value of zero; thus, if (d, b) is coded (A2), the result is an M field of zero and an N field of 12 (base 16).

Example: The following assembler code would translate into the illustrated hexadecimal object code:

LABEL	∅	CMMND	∅	OPERANDS	OP	∅	R	∅	T	∅	M	∅	N
		L		X5,(#7F,B1)	1C		5		0		1		07F
		AF		A3,(A8),X5	42		3		5		0		018

6-18. MACHINE TRANSLATION

There are four factors that determine the mode by which the Central Processor develops the effective address: first, the operation code specifies whether the effective address is that of a singleword, halfword or doubleword; second, the indirect (most significant) bit of the T field specifies whether the address is to be obtained directly or indirectly; third the M field specifies whether a base value is to be added to the N field displacement; and fourth, the three least significant bits of the T field specify whether the address is to be modified by an index value.

6-19. Direct Address Development

Direct addresses are those that are developed from the N field, the contents of the base register addressed by the M field, and the contents of the index register addressed by the T field.

Restrictions: An M field of zero specifies no base value and any effective address obtained that is within the range: $0 \leq EA \leq 47$, will be the address of a register.

If the M field is any value other than zero, effective addresses within the range $0 \leq EA \leq 47$ will address low virtual momory even though the contents of the base register addressed might be zero.

The first index register (index register 0, X0) cannot be used for indexing. It can be addressed in any other field of an instruction, but a value of zero in the T field specifies no indexing.

6-20. Singleword Addresses

Singleword addressing is specified by the operation code (assembler mnemonic) and the addresses are developed as follows:

1. Consider the 12-bit N field to be the positive N displacement.
2. Examine the M field and,
   a. if it is not zero, find the base register it addresses and add the least significant 24 bits of the contents (as a positive value) to the N deplacement, or
   b. if it is zero, treat an effective address within the range of the register bank addresses as an indexable register address.
3. Examine the three least significant bits of the T field and,
   a. if it is not zero, find the index register it addresses and add the least significant 24 bits of the contents (as a two's complement value) to the result of step 2, or
   b. if it is zero, ignore the index unit.
4. Use the result of step 3 as the effective address of the instruction.

Figure 6-3 illustrates the full process, and Table 6-3 gives the full specification for singleword addressing.

Note: This development takes place in the arithmetic unit; neither the instruction nor the contents of the base and index registers are altered.

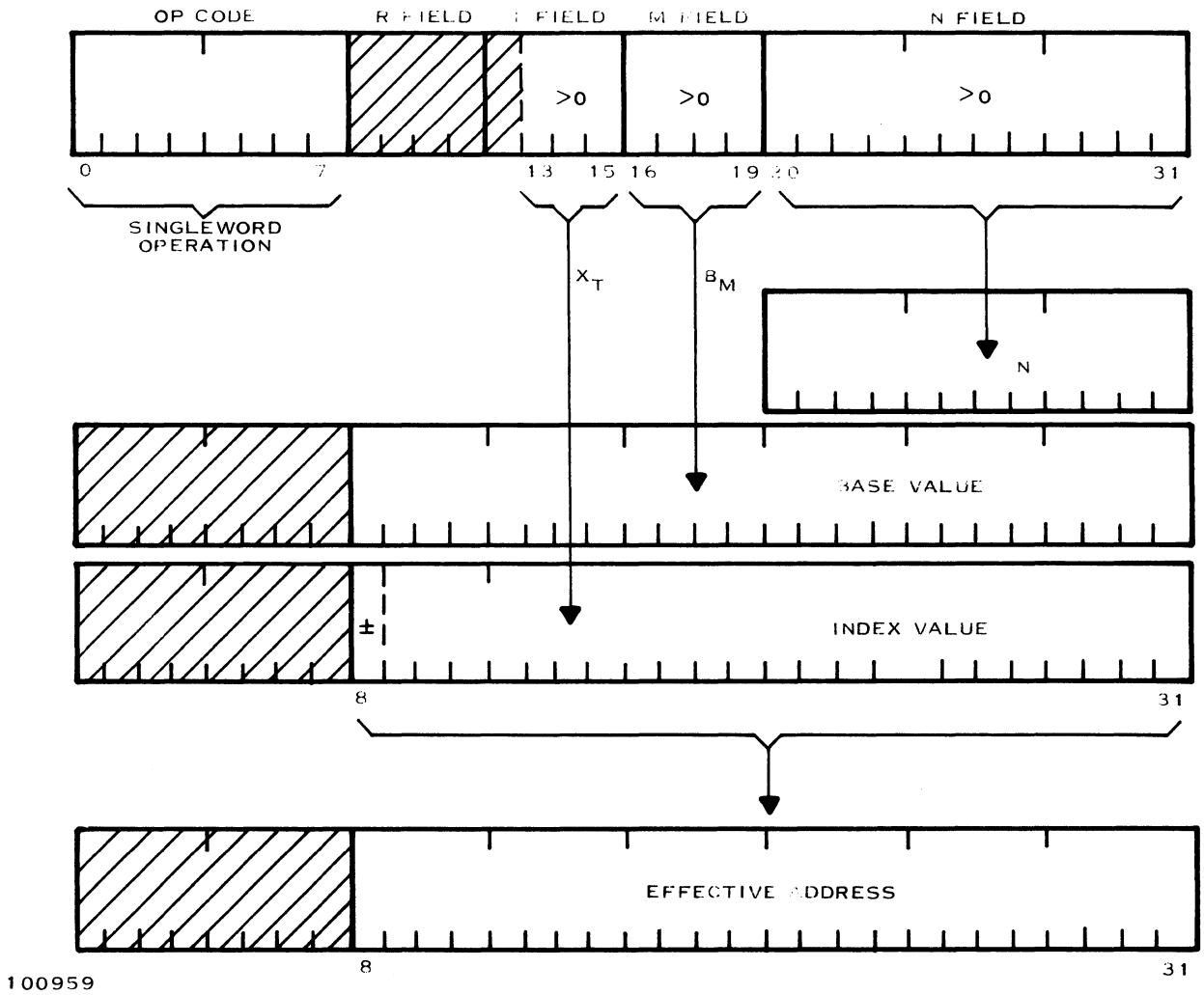


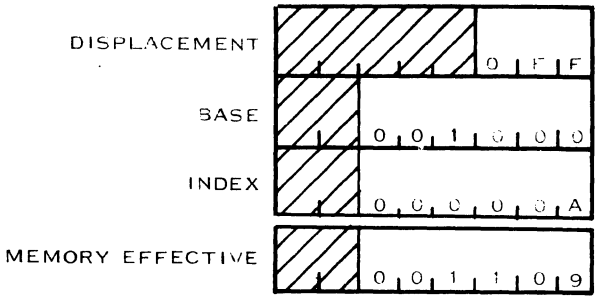
Figure 6-3. Development of Singleword Effective Addresses

Table 6-3. Development of Singleword Addresses (Direct)

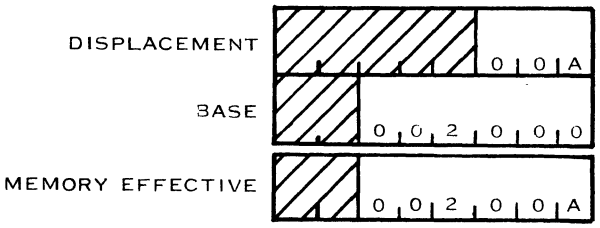
ADDRESS AND INDEX OPERANDS		$0 \le N \le 2^{12}-1$					EFFECTIVE ADDRESS
SYMBOLIC	EXPLICIT	M FIELD	(M)+N	ORIGIN	T FIELD	(M)+N+(T)	
expr	(expa, expa)	M=0	N≤47	REG	T=0	N/A	REG
			N>47	VM	T=0	N/A	VM
		0<M≤15	≥0	VM	T=0	≥0	VM
expr, expa	(expa, expa), expa	M=0	N≤47	REG	T=0	N/A	REG
					0<T≤7	EA≤47	REG
						EA>47	VM
			N>47	VM	T=0	N/A	VM
					0<T≤7	EA≤47	REG
						EA>47	VM
		0<M≤15	≥0	VM	0≤T≤7	≥0	VM
	(expa)	M=0	N≤47	REG	T=0	N/A	REG
			N>47	VM	T=0	N/A	VM
	(expa), expa	M=0	N≤47	REG	T=0	N/A	REG
					0<T≤7	EA≤47	REG
						EA>47	VM
			N>47	VM	T=0	N/A	VM
					0<T≤7	EA≤47	REG
						EA>47	VM

Where expr is a relocatable expression (symbol), expa is an absolute expression (symbol), (M) is the content of the base register, (T) is the content of the index register, REG is a register, and VM is a virtual memory location.

Examples: Given that the assembler statement:
L A4, SUM, X5 translates into object code: 14 4 5 1 0FF, that B1 contains: 0000 1000, and X5 contains: 0000 000A; then the address will be developed as:



Given that the assembler statement:
A X4, (#A, B2) translates into object code: 62 4 0 2 00A, and that B2 contains: 0000 2000; the address will be developed as:



6-21. Halfword Addresses

Halfword addressing is specified by the operation code (assembler mnemonic), and the halfword specified may be either the left or the right halfword of a single-word depending upon the code. In either case, the addresses are developed as follows:

1. Consider the 12-bit N field to be the positive N displacement.
2. Examine the M Field and,
   a. if it is not zero, find the base register it addresses and add the least significant 24 bits of the contents (as a positive value) to the N displacement, or
   b. if it is zero, treat any effective address within the range of the register bank addresses as an indexable register address.

3. Examine the three least significant bits of the T field and,
   a. if it is not zero, find the index register it addresses and,
      (1) displace its 25 least significant bits to the right by one bit (arithmetic shift) and,
      (2) add the value as a two's complement signed number to the result of step 2, or
   b. if it is zero, ignore the index unit.
4. Use the result of step 3 as the effective address of the instruction.

Figure 6-4 illustrates the full process.

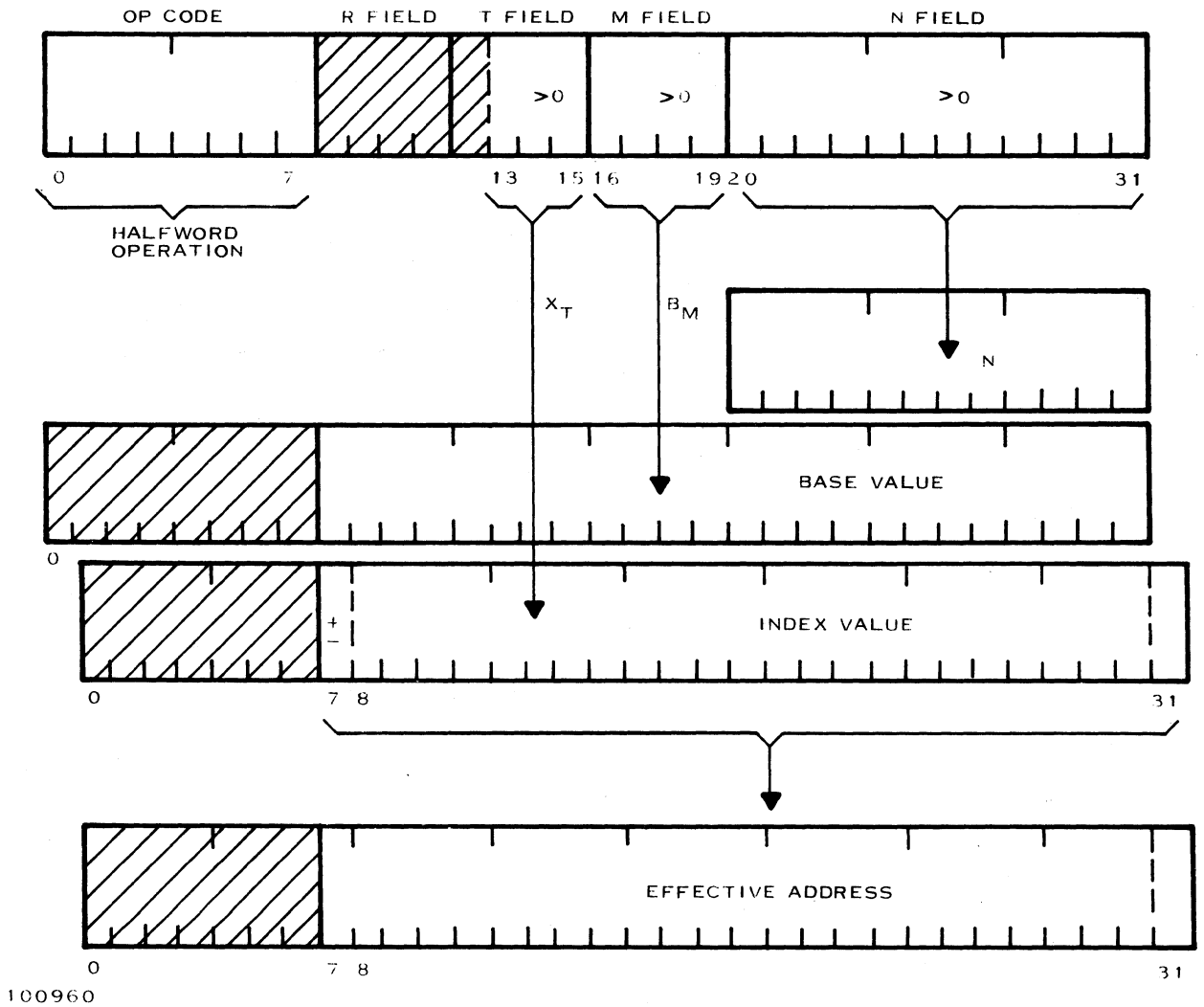


Figure 6-4. Development of Halfword Effective Addresses

Restrictions: The address developed from steps 1 and 2 is actually a word address; it is the operation code that specifies the use of a halfword. Note then that any symbolic address is a word address; e.g., in the following instructions LH X1, IMMOD and LR X1, IMMOD is the same word, but one instruction accesses the left half and the other accesses the right half.

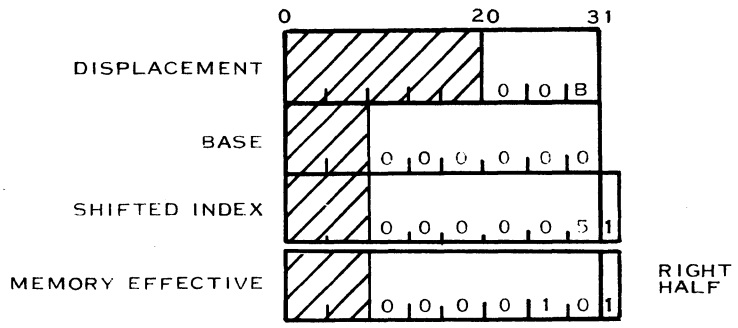
It is only the index parameter that produces halfword displacements. See Topic 6-25, for a description of halfword index word sets.

Note: This development takes place in the arithmetic unit; neither the instruction nor the contents of the base and index registers are altered.

Note: Table 6-3 is valid for halfword addressing up to determination of the origin, but when indexing is specified, the singleword displacement is only one-half the index value; e.g., in an instruction with base register 0, an N displacement of 48 would be the first virtual memory address; but with an N displacement of 0 it would require an index value of 96 to exit from the register bank (for H halfword operations).

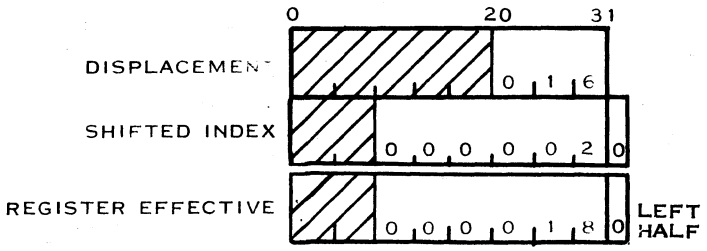
The register exit point for base register 0 addresses will be:
exit = N + X, where the index value X = 2(47-N) + 2 for "H" halfword instructions and X = 2(47-N) + 1 for "L" or "R" halfword instructions.

Examples: Given that the assembler statement:
LH A4,(#B, B2), X1 translates into: 15 4 1 2 00B, and that B2 contains: 0000 0000, and X1 contains: 0000 000B; then the address will be developed as:



and the contents of the right half of virtual memory location #10 would be loaded into the left half of A4.

Given that the assembler statement:
LH A4, (A6), X1 translates into: 15 4 1 0 016, and that X1 contains: 0000 0004; then the address will be developed as:



and the contents of the left half of A8 will be loaded into the left half of A4.

6-22. Doubleword Addresses

Doubleword addressing is specified by the operation code (assembler mnemonic) and the addresses are developed as follows:

1. Consider the 12-bit N field to be the positive N displacement.
2. Examine the M field and,
   a. if it is not zero, find the base register it addresses and add the least significant 24 bits of the contents (as a positive value) to the N displacement, or
   b. if it is zero, treat any effective address within the range of the register bank addresses as an indexable register address.
3. Examine the three least significant bits of the T field and,
   a. if it is not zero, find the index register it addresses and,
      (1) displace its 23 least significant bits to the left by one bit and,
      (2) add the value as a two's complement signed number to the result of step 2, or
   b. if it is zero, ignore the index unit.
4. Use the result of step 3 as the effective address of the instruction.

Figure 6-5 illustrates the full process.

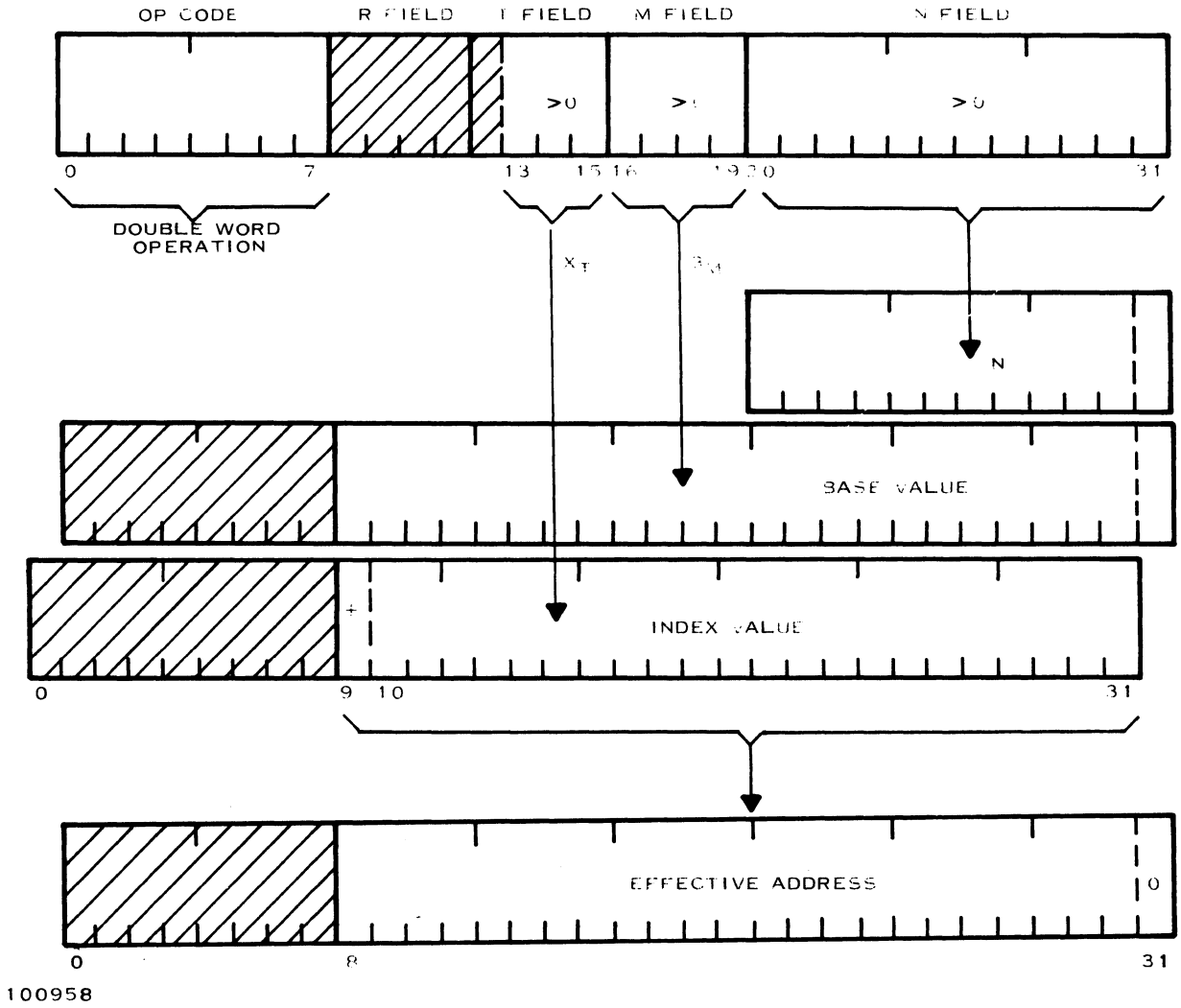


Figure 6-5. Development of Doubleword Effective Addresses

Restrictions: The address developed from steps 1 and 2 must be an even number so that the least significant bit will be zero; otherwise, the bit is zeroed automatically. No doubleword instruction will accept as its first word of data an odd word location; i.e., doubleword addressing is by even-odd word pairs only.

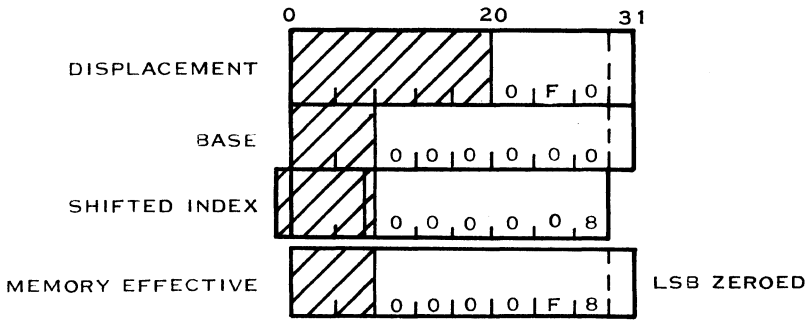
The index parameter, due to the register shift, produces automatic doubleword displacements. See Topic 6-26.

Note: This development takes place in the arithmetic unit; neither the instruction nor the contents of the base and index registers are altered.

Note: Table 6-3 is valid for doubleword addressing up to determination of the origin, but when indexing is specified, the singleword displacement is twice the index value; e.g., in an instruction with base register 0, an N displacement of 48 would be the first virtual memory address, but with an N displacement of 0 it would require only an index value of 24 to exit from the register bank.

The register exit point for base register 0 addresses will be:
exit = N + X, where the index value X = 1/2 (48-N) and N must be even.

Example: Given that Assembler statement:
LD A6, SUMD, X1 translates into 17 6 1 1 0F0, and that B1 contains: 0000 0000, and X1 contains: 0000 0004; then the address will be developed as:



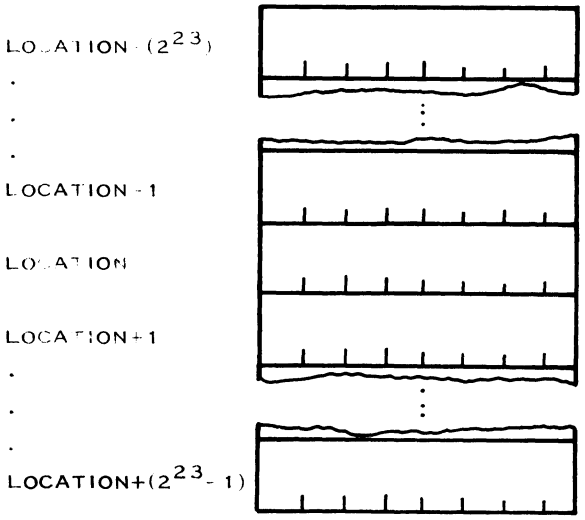
and the contents of the virtual memory location F8 and F9 will be loaded into registers A6 and A7.

6-23. Index Word Sets

Symbolic addresses are all effectively singleword addresses as can be seen from Topics 6-16 and 6-19 through 6-22. The index value is displaced left or right or not at all to produce the proper index displacement units; viz., singleword units, halfword units, and doubleword units. This produces a set of units which can be accessed by a single symbol plus an index value; for convenience they are called index word sets.

6-24. Singleword Index Word Sets

Singleword index word sets are straightforward; all addressable locations within the set fall within the interval:
location $-(2^{23})$, location + $(2^{23}-1)$. Illustratively



Limitations: The index values (i. e., contents of the index register addressed by the index operand) must be within the range: $-2^{23} \leq x \leq 2^{23}-1$

6-25. Halfword Index Word Sets

There are two types of halfword index word sets: those with left halfword origins and those with right halfword origins.

Those halfword instructions whose assembler mnemonics end with the letter "H" (e. g., LH, STH) all access, without index, the left half of the location addressed; i. e., they set left halfword origins. Even index values access left halfwords, and odd index values access right halfwords.

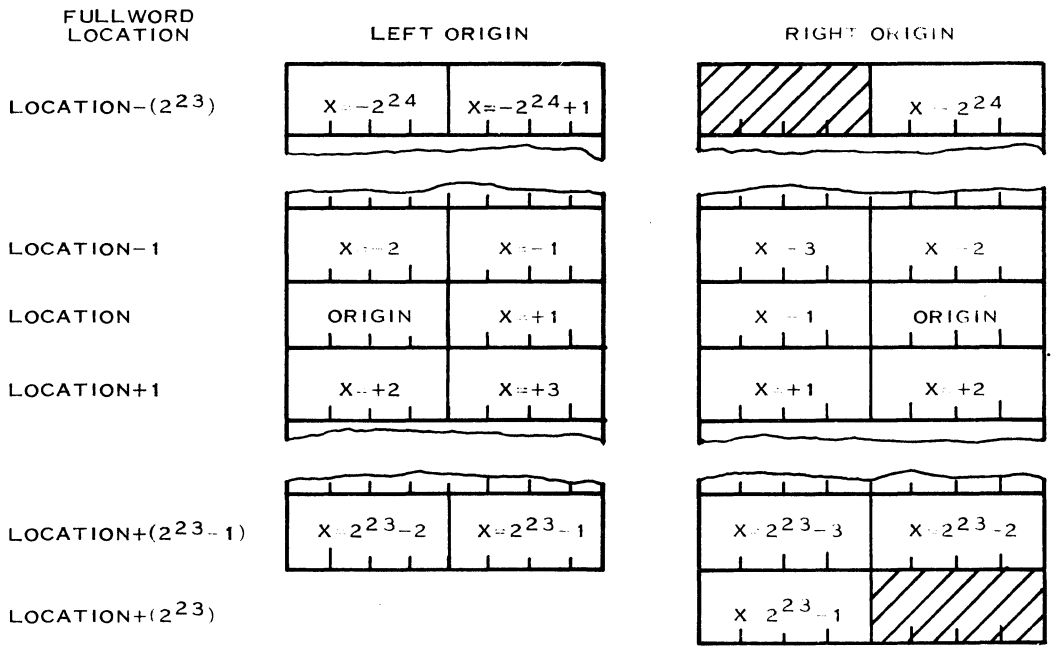
Those halfword instructions whose assembler mnemonics end with the letter "L" or the letter "R" (e. g., LL, LR) all access, without index, the right half of the location addressed; i. e., they set right halfword origins. Even index values access right halfwords, and odd index values access left halfwords.

Limitations: The index values (i.e., contents of the index register addressed by the index operand) must be within the range: $-2^{24} \leq x \leq 2^{24}-1$.

The fullword interval spanned by a left index word set is: location $-(2^{23})$, location $+(2^{23}-1)$. Note that the fullword interval of the set is the same as that for the singleword index word set, but that there are twice as many addressable units.

The fullword interval spanned by a right halfword index word set is: location $-(2^{23})$, location $+(2^{23})$. The right half displacement extends it one word farther than any of the other index word sets.

Illustratively:



6.26. Doubleword Index Word Sets

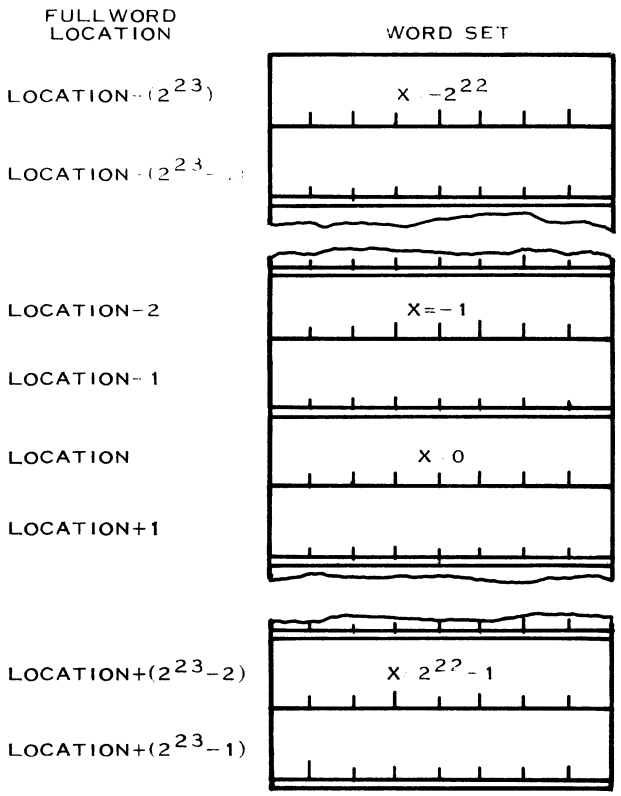
Doubleword index word sets originate at an even-odd doubleword location and are indexed by doubleword incremental units.

Restrictions: The location name in the address operand must be the name of an even numbered location.

Limitations: The index values (i.e., contents of the index register addressed by the index operand) must be within the range: $-2^{22} \leq x \leq 2^{22}-1$.

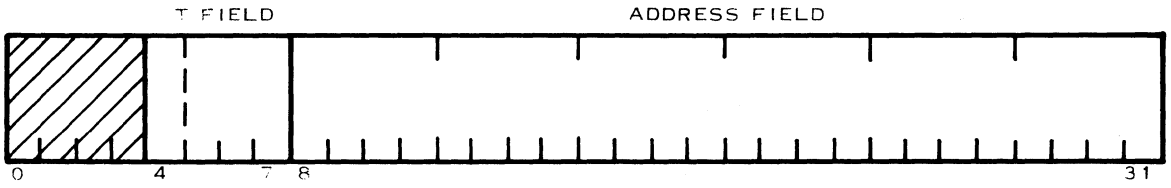
The singleword interval spanned by a doubleword index word set is: location $-(2^{23})$, location + $(2^{23}-1)$. Note that the singleword interval of the set is the same as for the singleword index word set, but there are only half as many addressable units.

Illustratively:



6-27. Indirect Address Development

Indirect addresses are those that are developed from indirect address cells which are originally addressed by the T, M, and N fields of the instruction. The indirect address cells may or may not specify additional indirection and/or indexing. Figure 6-6 illustrates the indirect address cell format.



100937

Figure 6-6. Indirect Address Cell Format

The indirect addresses are developed as follows:

1. Examine the most significant bit of the T field and,
   a. if it is not set to one, this is not an indirect address; develop the appropriate direct address (Topic 6-19 through 6-22).
   b. if it is set to one, proceed to step 2.
2. Develop a singleword address (Topic 6-20).
3. Examine the indirect address cell in the location found in step 2 and,
   a. if the most significant bit (bit 4) of its T field is set to one,
      (1) develop an address from its address field (a full 24-bit virtual memory address) and the contents of the index register specified (if any) in its T field, and
      (2) use the contents of the location obtained to repeat step 3.
   b. if the most significant bit of its T field is zero, develop an appropriate 24-bit singleword, halfword, or doubleword virtual memory address.
4. Use the terminal address obtained as the location from which or to which data is to be moved.

Restrictions: All initial addresses in the T, M, and N fields (address and index operands) will be developed as singleword addresses.

Only the first level of indirection can refer to a register; there is no M field in an indirect address cell and all addresses refer to virtual memory.

The terminal address index increment will always be appropriate to the word size specified by the operation code.

Indirect address development is in the execute mode; indirect address cells must be in execution permitted control sections.

Program Interruptions: If any intermediately accessed location contains a one in any bit position zero through three, an illegal operation interrupt will occur.

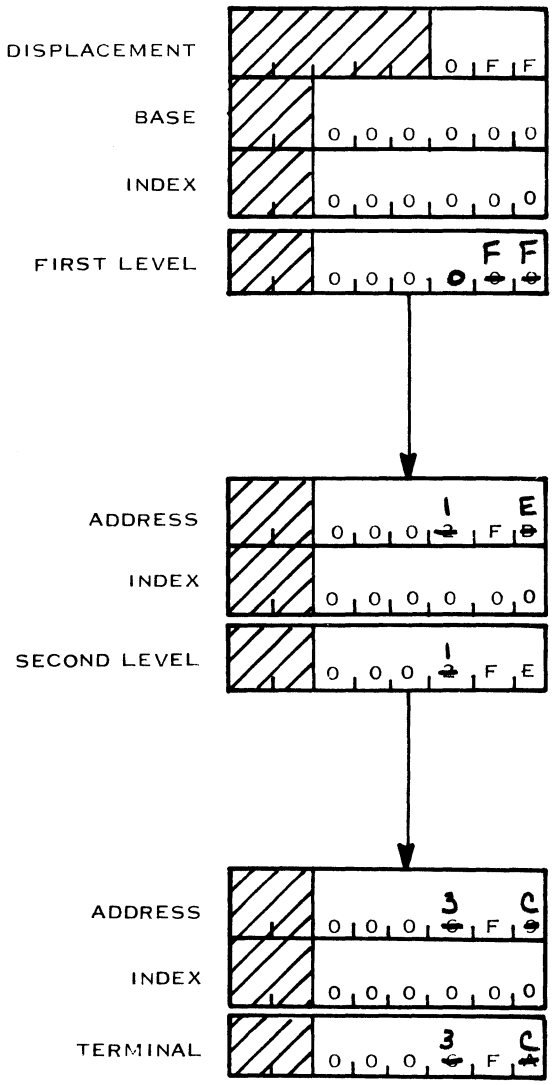
Note: Indirect addressing will not be inadvertently introduced into immediate operand instructions by introduction of a one into the most significant bit of the T field because those instructions do not examine the indirect bit.

Note: Table 6-3 is valid for development of the first level of indirect address development. The table need only be altered by coding the operands with an @ sign on the left as they are expressed in the table. The T field values would then always be either: $T = 8$, or $8 < T \leq 15$ in place of $T = 0$ and $0 < T \leq 7$, respectively.

Example: Given that base register B1 is in use with contents of: 0000 0000, and registers X1, X2, X3, X4, X5, X6, and X7 each contains: 0000 0000, and the following assembler code transtated as illustrated:

LABEL ␣ CMMND ␣ OPERANDS			COUNT	OP ␣ R ␣ T ␣ M ␣ N
	⋮		⋮	
	L	A2,@NODE10,X1	F	14 2 9 1 0FF
	⋮		⋮	- ␣ T ␣ ADDRESS
NODE10	IND	@NODE11,X2	FF	0 A 0001FE
	IND	@NODE12,X3	100	0 B 0002FD
	⋮		⋮	
NODE11	IND	D111,X4	1FE	0 4 0003FC
	IND	D112,X5	1FF	0 5 0004FB
	⋮		⋮	
NODE12	IND	D121,X6	2FD	0 6 0005FA
	IND	D122,X7	2FE	0 7 0006F9
	⋮		⋮	WORD 1 WORD 2
D111	DATA	2,4	3FC	0000 0002 0000 0004
	⋮		⋮	
D112	DATA	3,5	4FB	0000 0003 0000 0005
	⋮		⋮	
D121	DATA	2,3	5FA	0000 0002 0000 0003
	⋮		⋮	
D122	DATA	4,5	6F9	0000 0004 0000 0005

then on execution of instruction F the address will be developed as:



and 0000 0002 will be loaded into A2. ✓

6-28. Creating Indirect Address Cells

Indirect address cells with the full power of indirect addressing are normally programmed with the IND directive (see Topic 4-24). This directive provides for building T fields in the indirect address cells. This produces indirect address indexing which is convenient for the creation of tree structures of addresses.

The load effective address instruction also creates effective address cells (in the base, index or vector registers), but these cells will have no T field and, thus, no intermediate or terminal indexing.

Refer to the example in Topic 6-27.

6-29. IMMEDIATE OPERANDS

If the "second" operand of an instruction is specified by the instruction to be an immediate operand, it is developed by the assembler into a 16-bit absolute value that occupies the M and N fields of the object instruction. See Figure 6-1.

All assembler mnemonics for instructions which treat the M-N field as immediate data end with the letter "I", e.g., LI, AI, SI.

6-30. ASSEMBLER TRANSLATION

The assembler will translate either numeric expressions or character strings into immediate M-N fields.

6-31. Numeric Immediate Operands

Numeric immediate operands are right-justified in the right half of the instruction word and unspecified bit positions are filled with zeros. Negative numbers are expressed in two's complement form with the sign bit in bit 16 of the instruction word.

Limitations: The value of an expression used as an immediate operand must be within the range: $-32,768 \leq i \leq +32,767$ ($-2^{15} \leq i \leq 2^{15}-1$).

6-32. Character String Immediate Operands

Character strings used as immediate operands are left justified in the right half of the instruction word and unspecified bit positions are filled with zeros.

Restrictions: Since EBCDIC character representation is used in the ASC and EBCDIC representation requires one byte per character, a character string immediate operand is restricted to two characters in length. A single character operand will have a blank represented in the right byte.

6-33. MACHINE TRANSLATION

The Central Processor processes the object code representation of an immediate operand as a numeric or as a logical value, depending upon the operation code, and modifies the 16-bit immediate value by the appropriate index value, depending upon the operation code and the presence of an index operand.

6-34. Numeric Immediate Development

Numeric immediate operands are developed as signed numbers with negative values represented in two's complement form and may, if so specified or permitted, be modified by an index value. The effective immediate operand will be of the word size appropraite the operation code, i. e., will be a singleword or a halfword.

6-35. Singleword Numeric Immediates

Singleword immediates are specified by the operation code and the effective immediate is developed as follows:

1. Extend the sign of the 16-bit value in the M-N field to the left to produce a full 32-bit signed value.
2. Examine the T field of the instruction (ignoring the indirect bit) and,
   a. if it is zero, ignore the index unit, or
   b. if it is greater than zero,
      (1) add the 24-bit signed value in the index register addressed to the value obtained in step 1 (the index unit is only 24 bits in length), and
      (2) extend the sign of the result to a full 32-bit singleword.
3. Use the full singleword result as the effective immediate operand.

Figure 6-7 illustrates the full process.

Limitations: The index value in the index register addressed, if any, must be within the range: $-2^{23} \leq X \leq 2^{23}-1$.

The effective immediate operand produced will be within the range: $-2^{23} \leq ei \leq 2^{23}-1$.

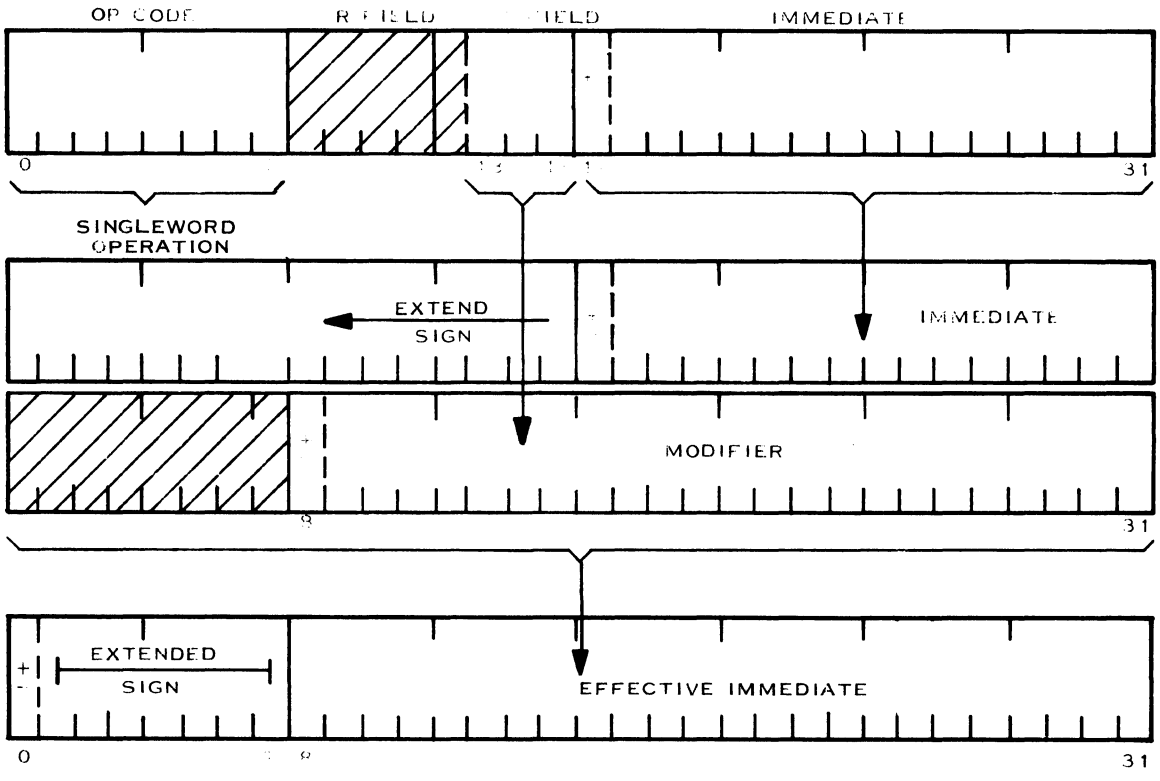


Figure 6-7. Development of Singleword Effective Immediate Operands

Note: The load look ahead (LLA) instruction does not permit modification of its immediate operand, and the shift instructions place closer limitations on the possible values of the effective immediate operand. These instructions are described in Section VII.

Note: The development of an effective immediate operand takes place in the arithmetic unit; the original instruction word is not modified.

6-36. Halfword Numeric Immediates

Halfword immediates are specified by the operation code and the effective halfword immediate value is developed as follows:

1. Consider the data in the M-N field to be a 16-bit signed value with negative numbers in two's complement form.
2. Examine the T field of the instruction (ignoring the indirect bit) and,
   a. if it is zero, ignore the index unit, or
   b. if it is greater than zero,

(1) extract the 16-bit signed value from the right half of the index register addressed, and

(2) add it to the value obtained in step 1.

3. Use the value obtained in step 2 as an effective halfword immediate operand.

Figure 6-8 illustrates the full process.

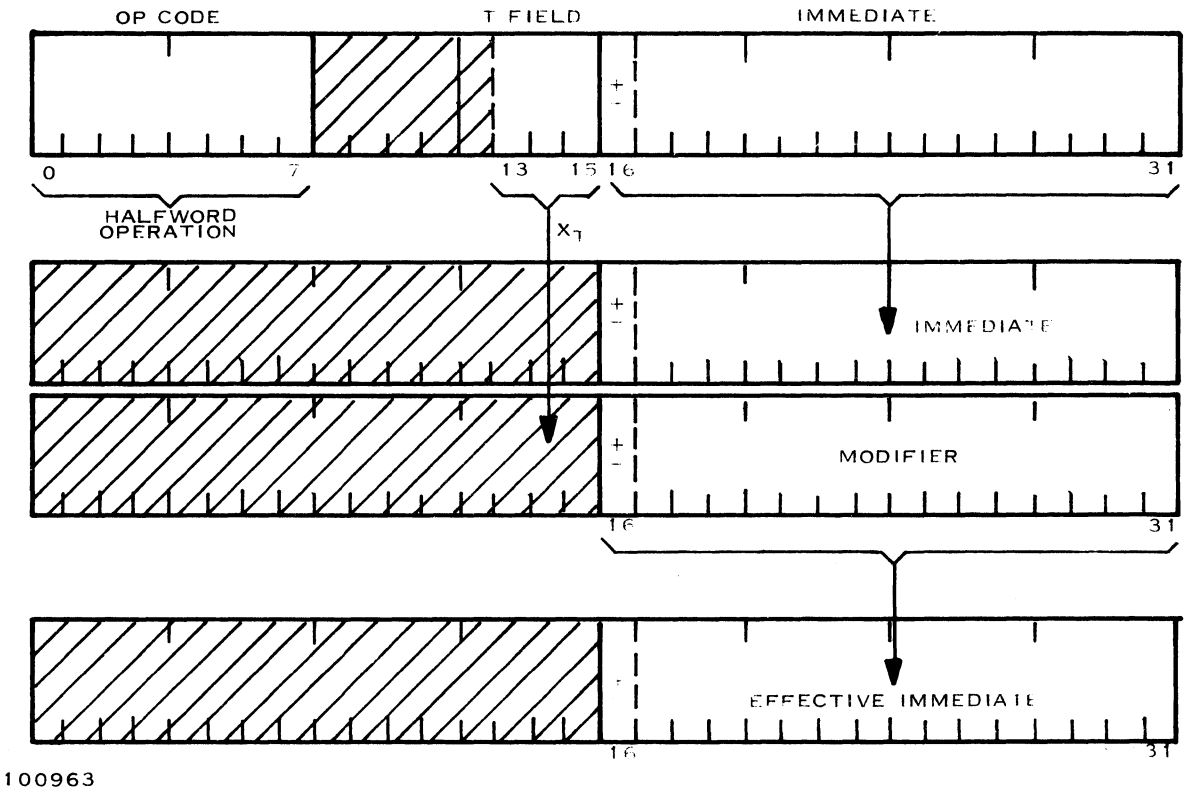


Figure 6-8. Development of Halfword Effective Immediate Operands

Restrictions: Only the right half of an index register is accessed by the index operand (T field).

Limitations: The index value in the right half of the index register addressed, if any, will be within the range: $-2^{15} \leq X \leq 2^{15}-1$.

The effective immediate operand produced will be within the range: $-2^{15} \leq ei \leq 2^{15}-1$.

6-37. Logical Immediate Development

Logical immediate operands are developed as pure binary values and may, if so specified, be modified by an index value. The effective immediate operand will be of the word size appropriate to the instruction; i. e., will be a singleword or halfword.

6-38. Singleword Logical Immediates

Singleword immediates are specified by the operation code and the effective immediate is developed as follows:

1. Extend zeros into the left of the logical value in the M-N field to produce a full 32-bit logical value.
2. Examine the T field of the instruction (ignoring the indirect bit) and,
   a. if it is zero, ignore the index unit, or
   b. if it is greater than zero,
      (1) add the 24-bit logical value in the index register addressed to the value obtained in step 1 (the index unit is only 24 bits in length), and
      (2) extend zeros to the left to produce a full 32-bit logical value.
3. Use the full singleword result as the effective immediate operand.

Figure 6-9 illustrates the process.

Restrictions: Index modification is by one's complement addition only; there is no overflow into the eight most significant bits of the singleword and no end-around-carry.

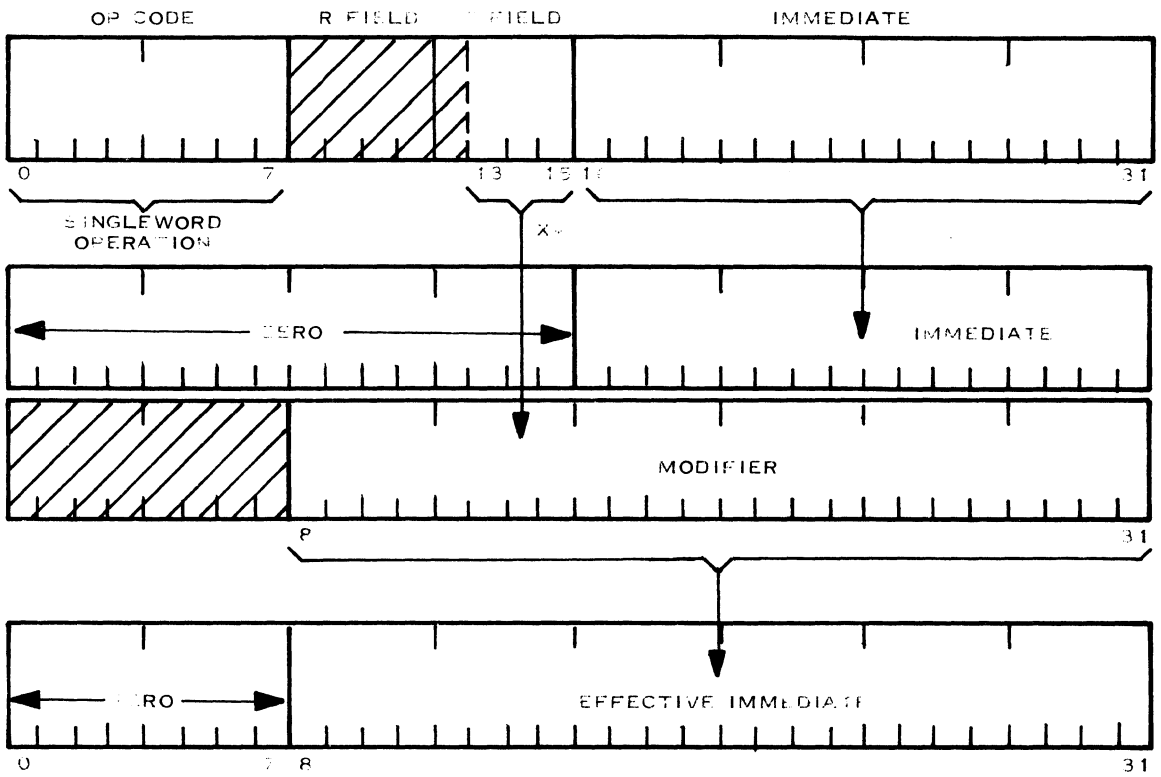


Figure 6-9. Development of Singleword Logical Immediate Operands

6-39. Halfword Logical Immediates

Halfword immediates are specified by the operation code and are developed under the same restrictions as singleword logical immediates; i. e., index modification is by one's complement addition only.

Halfword logical immediates are developed under the additional restraints that index modification is restricted to the 16 bits of the right half of the index register, and that the result is restricted to a 16-bit logical value. No extension of zeros is required.

6-40. BRANCH ADDRESS DEVELOPMENT

Branch addresses may be either program counter relative or base relative. The branch addresses developed are always singleword addresses. Table 6-4 gives the specifications for direct branch addresses.

6-41. ASSEMBLER TRANSLATION

The assembler translates all symbolic addresses into program counter relative branches if possible. Base relative branches can be forced by coding the address operand as an explicit base and displacement sublist.

6-42. Symbolic Branch Addresses

A symbolic address, which in other types of instructions would be translated into base and displacement fields, is translated into program counter relative branches if the displacement from the current location counter is within the interval: location -2048, location +2047.

For those branches which fall outside the program counter relative interval, the assembler produces the typical base and displacement values on the basis of the information supplied by the USING directive (see Topic 4-9).

Restrictions: Branch addresses will be program counter relative whenever possible.

Branch addresses never refer to the register bank.

Limitations: For branch addresses to be program counter relative, their displacement from the current location must be within the range: $-2048 \leq d \leq 2047$ (i.e., $-2^{12} \leq d \leq 2^{12}-1$).

Example: Given that base register B1 is in use with contents of: 0000 0000, and the following assembler statements translated as illustrated.

LABEL	CMMND	OPERANDS	COUNT	OP	R	T	M	N
	⋮		⋮					
SUM	AF	A1,(A5)	F	42	1	0	0	015
	BCLE	X1,A6,SUM	10	86	1	6	0	FFF
	CF	A1,(A2)	11	CA	1	0	0	012
	BG	TOTAL	12	91	2	0	1	8A0
	⋮		⋮					
TOTAL	ST	A1,OUT,X5	8A0	24	1	5	1	A10
	⋮		⋮					
OUT	RES	5	A10	five words reserved				

the BCLE branch is program counter relative with a negative displacement because the location to which it branches, SUM, is within -2048 words of its own location, whereas the BG branch is base relative because the location to which it branches is greater than 2047 words from its own location (viz., 8A0 = 2208 base 10, 12=18 base 10, and 2208 - 18 = 2190).

6-43. Explicit Base and Displacement Branch Address

Either program counter relative or base relative branches can be coded explicitl with a base and displacement sublist as the address operand.

6-44. Program Counter Relative Branch

Since a base register specification of zero specifies a program counter relative branch, the sublist can be used to produce an explicit branch displacement by coding a displacement only. This requires that the precise displacement be known.

Limitations: For an explicit program counter relative branch, the expression used as the d parameter must have a value within the range: $-2048 \leq d \leq 2047$. If the b parameter is coded, the expression used must have a value of zero: $b = 0$.

6-45. Base Relative Branch

When a branch address is coded explicitly with a base register other than zero, the branch will be base relative. The contents of the base register and the precise displacement from that base location must be known. The displacement in such an instruction is always positive.

Limitations: To produce an explicit base relative branch, the expression used as the d parameter must have a value within the range: $0 \leq d \leq 4095$ (i.e., $0 \leq d \leq 2^{12}-1$ and the expression used as the b parameter must have a value within the range: $0 < b \leq 15$.

6-46. Indirect Branch Addresses

When the assembler statement specifies that the branch address is to be develop by indirect addressing, the address of the first level indirect address cell is

obtained by the standard direct address development as described in Topics 6-16, 6-20, and 6-24.

The indirect address cell format is the same as that for non-branch instructions i.e., the terminal branch address will always be an absolute virtual memory address and never program counter relative. Refer to Topic 6-27.

Restrictions: The first level of indirection cannot access the register bank; branch address development, regardless of the T and M fields, will never address a register.

6-47. MACHINE TRANSLATION

There are three factors that determine the mode by which the Central Processor develops the effective branch address: first, the indirect (most significant) bit of the T field specifies whether the branch address is to be developed directly or indirectly; second, the M field specifies whether a direct branch address is to be program counter or base relative; and third, the three least significant bits of the T field specify whether the address is to be modified by an index value.

6-48. Program Counter Relative Branch Address Development

Direct program counter relative branch addresses are developed when the indirect bit of the T field and the M field are both zero. The address is developed by the following procedure:

1. Consider the 12-bit N field to be a signed value with negative numbers in two's complement form.
2. Add the value from step 1 to the current value of the program counter.
3. Examine the three least significant bits of the T field and,
   a. if they are zero, ignore the index unit, or
   b. if they are greater than zero, add the 24-bit signed value in the index register addressed to the value obtained in step 2.
4. If the branch condition is true or if the instruction is an unconditional branch, enter the result into the program counter; otherwise, go to the next instruction.

Figure 6-10 illustrates the process and Table 6-4 gives the specifications for branch addressing.

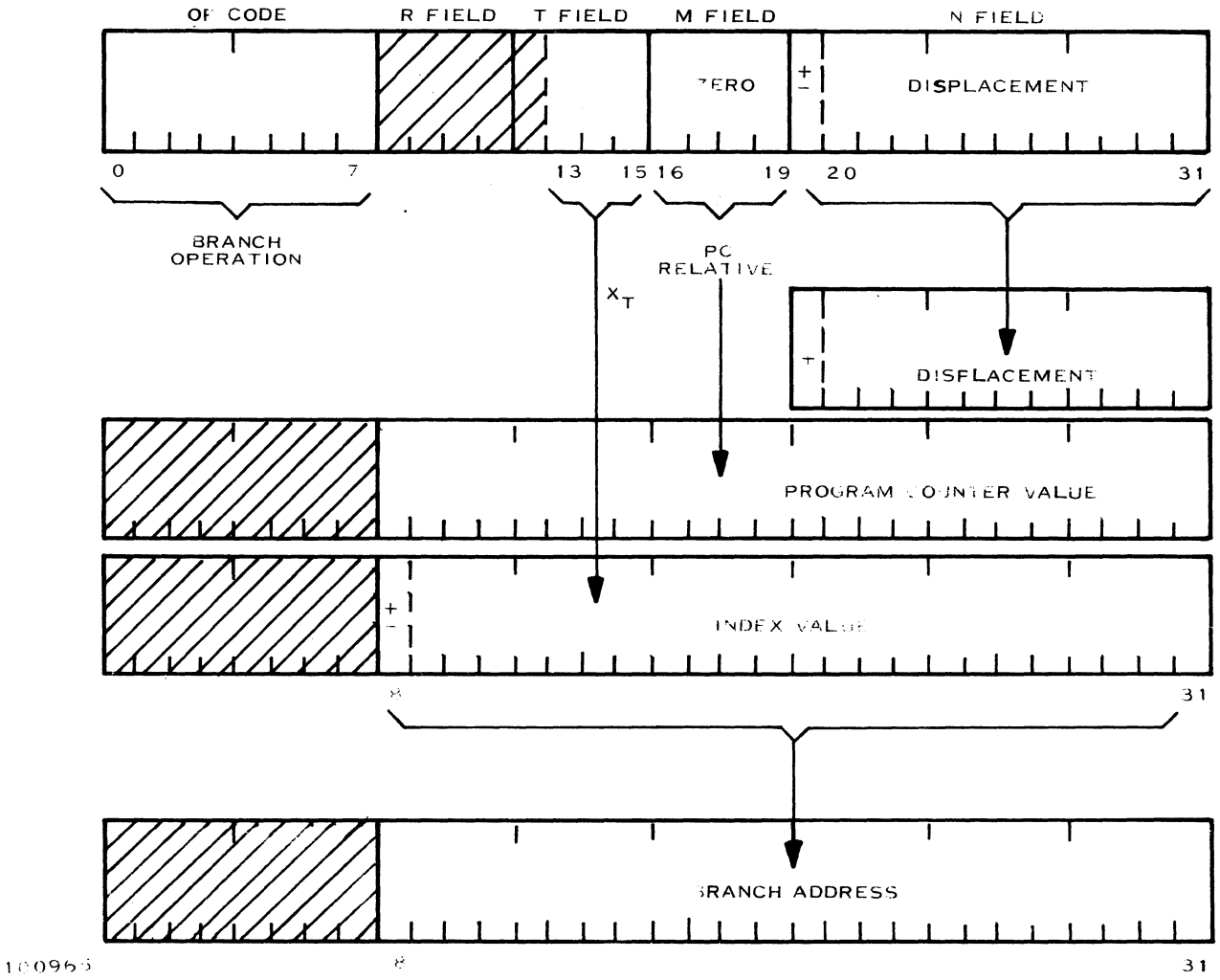


Figure 6-10. Program Counter Relative Branch Address Development

6-49. Base Relative Branch Address Development

Direct base relative branch addresses are developed when the indirect bit of the T field is zero and the M field is not zero. The effective address is developed by the following procedure:

1. Consider the 12-bit N field to be a positive value.
2. Add the value from step 1 to the positive 24-bit value in the base register addressed by the M field
3. Examine the three least significant bits of the T field and,
   a. if they are zero, ignore the index unit, or
   b. if they are not zero, add the signed 24-bit value in the index register addressed by the T field to the value from step 2.

Table 6-4. Development of Branch Addresses (Direct)

ADDRESS & INDEX OPERANDS		M FIELD	N FIELD	T FIELD	DEVELOPMENT
SYMBOLIC	EXPLICIT				
expr	(expa, expa)	M=0	$-2^{11} \leq N \leq 2^{11}-1$	T=0	(PC)+N
*		0<M≤15	$0 \leq N \leq 2^{12}-1$	T=0	N+(M)
expr, expa	(expa, expa), expa	M=0	$-2^{11} \leq N \leq 2^{11}-1$	T=0	(PC)+N
				0<T≤7	(PC)+N+(T)
*		0<M≤15	$0 \leq N \leq 2^{12}-1$	T=0	N+(M)
				0<T≤7	N+(M)+(T)
	(expa)	M=0	$-2^{11} \leq N \leq 2^{11}-1$	T=0	(PC)+N
	(expa), expa	M=0	$-2^{11} \leq N \leq 2^{11}-1$	T=0	(PC)+N
				0<T≤7	(PC)+N+(T)
		0<M≤15	$0 \leq N \leq 2^{12}-1$	T=0	N+(M)
				0<T≤7	N+(M)+(T)

Where expr is a relocatable expression (symbol), expa is an absolute expression (symbol), (PC) is the present value of the program counter, (M) is the content of the base register, and (T) is the content of the index register.

*expr will not be translated with M > 0 unless the value of N would fall outside the range: $-2^{11} \leq N \leq 2^{11}-1$.

4. If the branch condition is true or if the instruction is an unconditional branch, enter the result into the program counter; otherwise go to the next instruction.

Figure 6-11 illustrates the process.

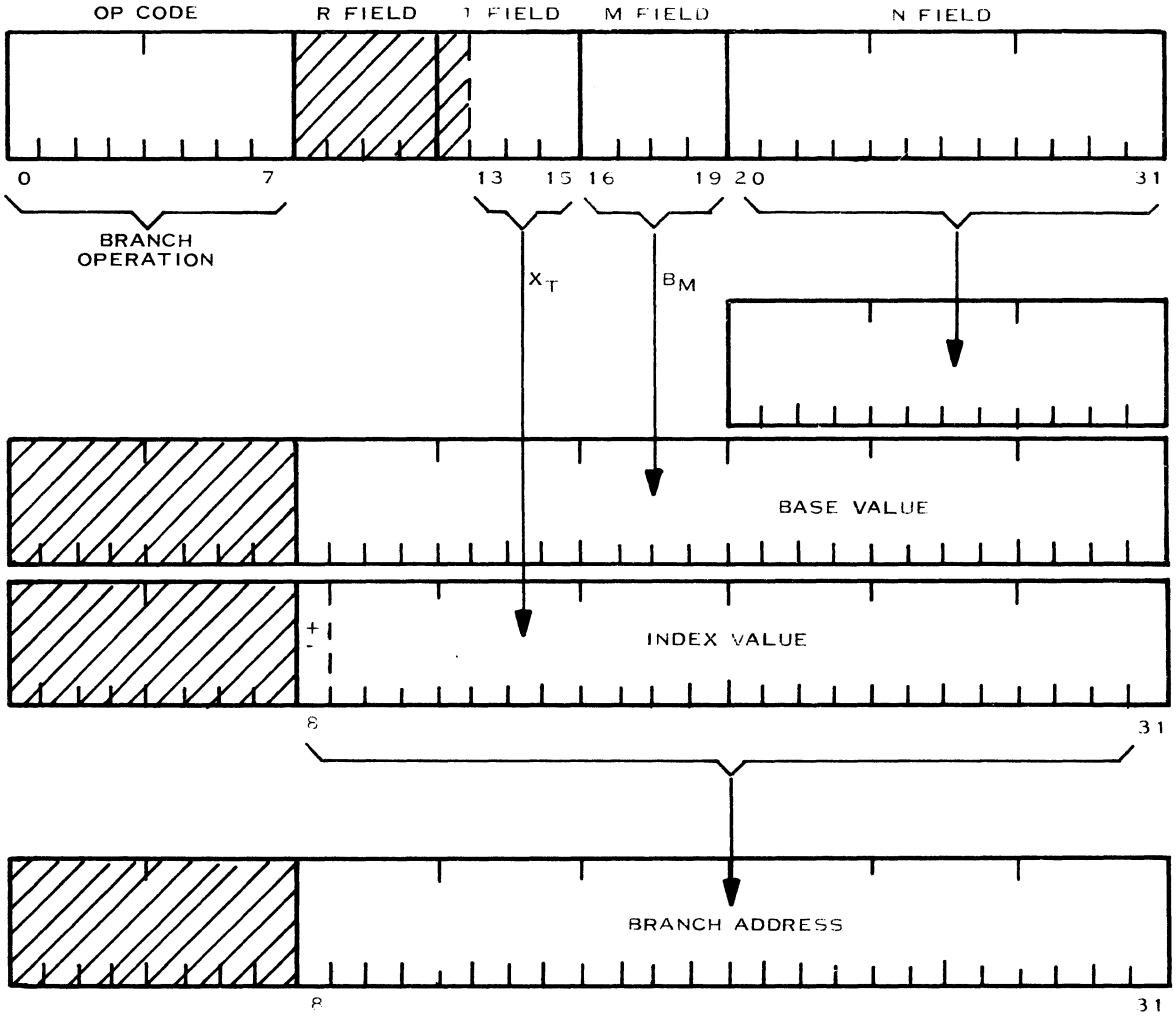


Figure 6-11. Development of Base Relative Branch Addresses

6-50. Indirect Branch Address Development

If the most significant bit of the T field is set to a one, the effective branch address will be developed indirectly.

Indirect branch address development proceeds the same as that for other instructions with the exception that an M field of zero at the first level will not

produce a reference to a register when $0 \leq N \leq 47$. Branch address development never accesses the register bank. See Topic 6-27 for a description of the development process.

6-51. DATA FORMATS

The ASC uses two algebraic data formats: fixed point with two's complement representation for negative numbers, and floating point with excess 64 (biased) exponent representation. Fixed point values may occur in either singleword or halfword lengths. Floating point values may occur in either singleword or doubleword lengths. Figure 6-12 illustrates the machine formats of these data forms.

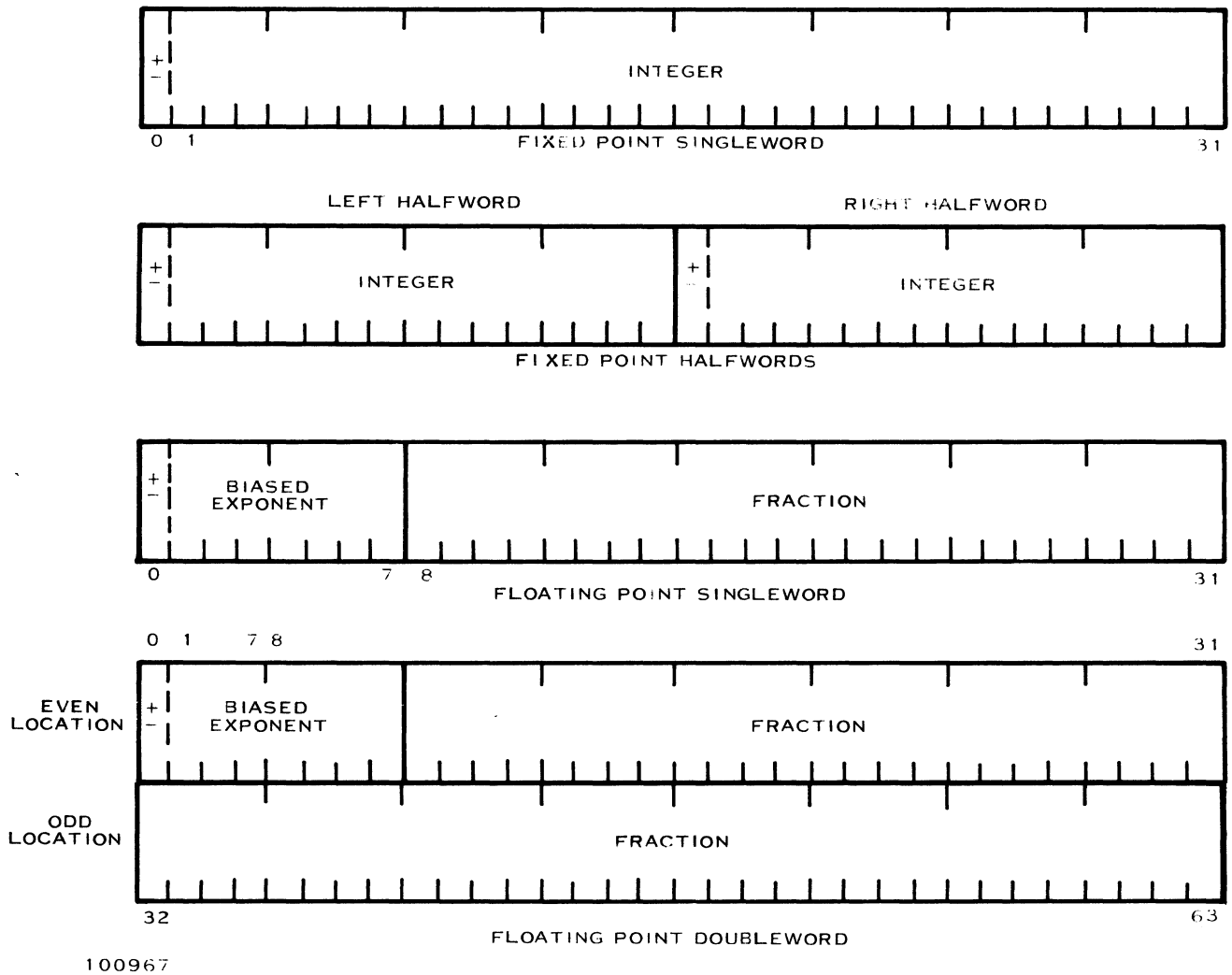


Figure 6-12. Algebraic Data Formats

6-52. FIXED POINT DATA

The assembler format for fixed point data is described in Topic 2-10.

The range of values that any given fixed point data constant may have depends upon its proposed usage. Table 6-5 lists the ranges for some typical uses:

Table 6-5. Value Ranges of Fixed Point Data

PURPOSE	VALUE RANGE
GENERAL ALGEBRAIC:	
Singleword	$-2^{31} \leq fx \leq 2^{31}-1$
Halfword	$-2^{15} \leq fx \leq 2^{15}-1$
INDEXES:	
Singleword	$-2^{23} \leq fx \leq 2^{23}-1$
Halfword	$-2^{24} \leq fx \leq 2^{24}-1$
Doubleword	$-2^{22} \leq fx \leq 2^{22}-1$
BASES:	
Singleword	$0 \leq fx \leq 2^{24}-1$

6-53. FLOATING POINT DATA

The assembler format for floating point data is described in Topic 2-9.

6-54. Normalized Floating Point Values

A normalized floating point value is one in which at least one of the four most significant bits of the fraction is set to one. All floating point data created by the assembler is normalized as is required by many floating point instructions.

WARNING: Although all floating point data constants created by the assembler will be normalized, there is no guarantee that all input data will be normalized.

6-55. Infinite and Indefinite Floating Point Values

Infinite (+∞ or -∞) floating point values are output from the arithmetic unit when a floating point operation would have a resultant value that, if normalized, would require a biased exponent greater than 127. Such operations produce a floating point overflow condition (see Topic 6-60), and attempts to use these values in subsequent operations other than division will also cause the overflow condition.

Indefinite floating point values are output from the arithmetic unit when input to the unit is either an indefinite form or a "dirty zero". A "dirty zero" is a floating point value with a zero fraction, but a non-zero exponent. Overflow also occurs whenever an indefinite value is input to the arithmetic unit for any operation.

The hexadecimal representations of the infinite and indefinite forms are as follows:

VALUE	HEXADECIMAL FORM	
	SINGLEWORD	DOUBLEWORD
+∞	7FFF FFFF	7FFF FFFF FFFF FFFF
-∞	FFFF FFFF	FFFF FFFF FFFF FFFF
IND	7F00 0000	7F00 0000 0000 0000

DATA FORMS

INFINITE FORMS AND INDEFINITE FORMS:

FLOATING ADD	OUTPUT	FLOATING POINT OVERFLOW
(+ ∞) + (+ ∞)	+ ∞	Yes
(+ ∞) + (- ∞)	IND	Yes
(- ∞) + (+ ∞)	IND	Yes
(- ∞) + (- ∞)	- ∞	Yes
(+ ∞) + (± N)	+ ∞	Yes
(- ∞) + (± N)	- ∞	Yes
(DZ) + (± N)	IND	Yes
(DZ) + (± ∞)	IND	Yes

FLOATING POINT SINGLE LENGTH FORMS ARE:

+ ∞	7FFF	FFFF	Positive infinite form.
- ∞	FFFF	FFFF	Negative infinite form.
IND	7F00	0000	Indefinite form.

FLOATING POINT DOUBLE LENGTH FORMS ARE:

+ ∞	7FFF	FFFF	FFFF	FFFF
- ∞	FFFF	FFFF	FFFF	FFFF
IND	7F00	0000	0000	0000

The indefinite form, 7F00 . . . 00, is generated by the Arithmetic Unit when an indefinite form or a "dirty zero" appears as input to the Arithmetic Unit during a floating point arithmetic operation.

A "dirty zero" is a floating point form consisting of a zero mantissa and a non-zero exponent. It has the form XX00 . . . 00, where at least one X is not equal to zero.

FLOATING ADD MAGNITUDE	OUTPUT	FLOATING POINT OVERFLOW
(+ ∞) + |(± ∞)|	+ ∞	Yes
(- ∞) + |(± ∞)|	IND	Yes
(+ ∞) + |(± N)|	+ ∞	Yes
(- ∞) + |(± N)|	- ∞	Yes
(± N) + |(± ∞)|	+ ∞	Yes
(DZ) + |(± N)|	IND	Yes
(DZ) + |(± ∞)|	IND	Yes
(± N) + |(DZ)|	IND	Yes
(± ∞) + |(DZ)|	IND	Yes

FLOATING SUBTRACT	OUTPUT	FLOATING POINT OVERFLOW
(+∞) - (+∞)	IND	Yes
(+∞) - (-∞)	+∞	Yes
(-∞) - (+∞)	-∞	Yes
(-∞) - (-∞)	IND	Yes
(+∞) - (±N)	+∞	Yes
(-∞) - (±N)	-∞	Yes
(±N) - (+∞)	-∞	Yes
(±N) - (-∞)	+∞	Yes
(DZ) - (±N)	IND	Yes
(DZ) - (±∞)	IND	Yes
(±N) - (DZ)	IND	Yes
(±∞) - (DZ)	IND	Yes

FLOATING SUBTRACT MAGNITUDE	OUTPUT	FLOATING POINT OVERFLOW
(+∞) - |(±∞)|	IND	Yes
(-∞) - |(±∞)|	-∞	Yes
(+∞) - |(±N)|	+∞	Yes
(-∞) - |(±N)|	-∞	Yes
(±N) - |(±∞)|	-∞	Yes
(DZ) - |(±N)|	IND	Yes
(DZ) - |(±∞)|	IND	Yes
(±N) - |(DZ)|	IND	Yes
(±∞) - |(DZ)|	IND	Yes

FLOATING MULTIPLY OR FLOATING VECTOR DOT PRODUCT	OUTPUT	FLOATING POINT OVERFLOW
$(+\infty) \cdot (+\infty)$	$+\infty$	Yes
$(+\infty) \cdot (-\infty)$	$-\infty$	Yes
$(-\infty) \cdot (+\infty)$	$-\infty$	Yes
$(-\infty) \cdot (-\infty)$	$+\infty$	Yes
$(+\infty) \cdot (\pm N)$	$\pm\infty$	Yes
$(-\infty) \cdot (\pm N)$	$\pm\infty$	Yes
$(\pm\infty) \cdot (0)$	IND	Yes
$(\pm N) \cdot (0)$	0	No
$(0) \cdot (0)$	0	No
$(DZ) \cdot (\pm\infty)$	IND	Yes
$(DZ) \cdot (\pm N)$	IND	Yes
$(DZ) \cdot (0)$	IND	Yes

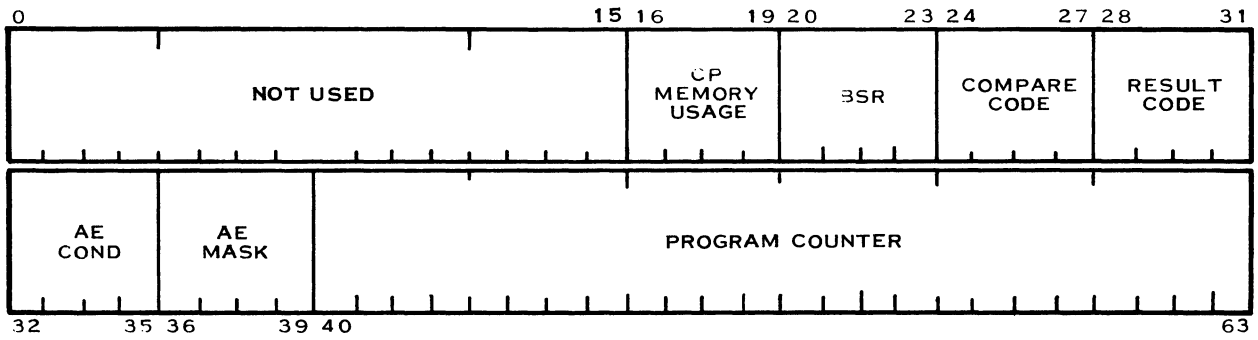
FLOATING DIVIDE	OUTPUT	FLOATING POINT OVERFLOW	DIVIDE CHECK
$(\pm\infty) \div (\pm\infty)$	IND	Yes	No
$(+\infty) \div (N)$	$+\infty$	Yes	No
$(+\infty) \div (-N)$	$-\infty$	Yes	No
$(-\infty) \div (N)$	$-\infty$	Yes	No
$(-\infty) \div (-N)$	$+\infty$	Yes	No
$(\pm\infty) \div (0)$	$\pm\infty$	Yes	Yes
$(\pm N) \div (\pm\infty)$	0	No	No
$(0) \div (\pm\infty)$	0	No	No
$(0) \div (\pm N)$	0	No	No
$(0) \div (0)$	IND	Yes	Yes
$(N) \div (0)$	$+\infty$	Yes	Yes
$(-N) \div (0)$	$-\infty$	Yes	Yes

FLOATING DIVIDE	OUTPUT	FLOATING POINT OVERFLOW	DIVIDE CHECK
(DZ) ÷ (± ∞)	IND	Yes	No
(DZ) ÷ (± N)	IND	Yes	No
(DZ) ÷ (0)	IND	Yes	Yes
(± ∞) ÷ (DZ)	IND	Yes	No
(± N) ÷ (DZ)	IND	Yes	No
(0) ÷ (DZ)	IND	Yes	No

6-56. PROGRAM STATUS DOUBLEWORD

The program status doubleword is a set of controls and registers internal to the Central Processor. They are accessible only in part to the programmer through the branch and load instructions, BLB and BLX, and the program status instructions, LAM, LAC, and SPS.

For access purposes, the program status doubleword may be said to have the following format:



The control state and Central Processor memory usage fields are of no interest to the Central Processor programmer, but rather to the system programmer. All the other fields are affected by and/or affect the result of one or more Central Processor instructions.

6-57. BRANCH OR SKIP REGISTER

The branch or skip register is a four-bit field in which only the two least significant bits are used.

The least significant bit is set (to one) or reset (to zero) depending upon whether, when an execute instruction, XEC, executes a branch or skip instruction, the condition for branching or skipping is true or false, respectively. See Topic 7-161.

The setting of this bit is used by the branch on execute condition instruction, BXEC. The BXEC can be coded to branch on either condition true or condition false. See Topics 7-135 and 7-131.

6-58. COMPARE CODE

The compare code is a four-bit field that is set upon execution of an arithmetic or a logical compare instruction to indicate the nature of the comparison result. Only the three least significant bits of the field are used.

The compare code is used by the branch on compare code true instructions to determine whether a previously executed comparison meets the condition for branching.

The specifications of the lists are as follows:

ARITHMETIC COMPARISON RESULT	LOGICAL COMPARISON RESULT	COMPARE CODE 0 cl cg ce
x<y	mixed ones and zeros	0 1 0 0
x>y	all bits ones	0 0 1 0
x=y	all bits zeros	0 0 0 1

6-59. RESULT CODE

The result code is a four-bit field that is set according to the arithmetic or logical properties of a result emerging from the arithmetic unit to be entered into a register. The setting is changed only when a new result emerges from the arithmetic unit; thus, the result code reflects the properties of the data in the most recently modified register. Only the three least significant bits of the field are used.

The result code is used by the branch on result code true instructions to determine whether the properties of the most recently acquired datum meets the condition for branching.

The specifications of the bits are as follows:

ARITHMETIC RESULT	LOGICAL RESULT	RESULT CODE 0 rl rg re
x<0	mixed ones and zeros	0 1 0 0
x>0	all bits are ones	0 0 1 0
x=0	all bits are zero	0 0 0 1

6-60. ARITHMETIC EXCEPTION CONDITION CODE

The arithmetic exception condition code is a four-bit field whose bits are set whenever the arithmetic unit detects one of the arithmetic exceptions: divide check, fixed point overflow, floating point exponent underflow, or floating point exponent overflow. See Topic 6-62. Illustratively:

ARITHMETIC EXCEPTION CONDITION CODE D X O U.

Divide Check, D: The divide check bit, D, is set to one when the arithmetic unit encounters an attempt to divide by zero in either fixed or floating point operations.

Fixed Point Overflow, X: The fixed point overflow bit, X, is set to one when a fixed point arithmetic or arithmetic shift operation produces a result in which a high order bit or bits would be lost (i.e., move out the left end of the data word). The operation will be completed by ignoring the lost bits.

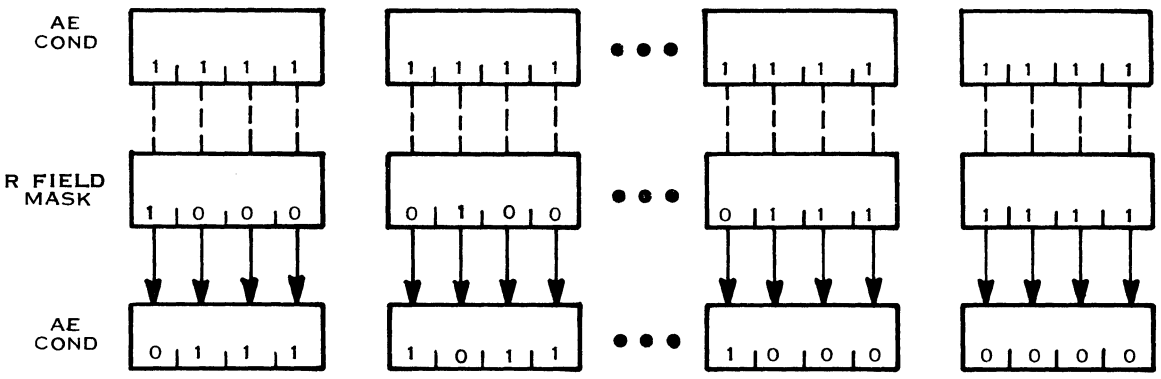
Floating Point Exponent Overflow, O: The floating point exponent overflow bit, O, will be set to one when a floating point operation produces a result in which the biased exponent would exceed 127. The operation is completed by entering a result of $+\infty$ for positive values and $-\infty$ for negative values.

Floating Point Exponent Underflow, U: The floating point exponent underflow bit, U, is set to one when a floating point operation produces a result in which the biased exponent would be less than zero. The operation is completed by entering a result of true zero.

6-61. Resetting the Arithmetic Exception Code

All bits set by detection of an arithmetic exception condition remain set until reset by the execution of a branch on arithmetic exception condition (BAE) instruction in which the corresponding bit of the R field (mask operand) of the instruction contains a one. Thus, a branch on arithmetic exception of divide check will not remove the record of a previous fixed point overflow, and so on.

Illustratively:



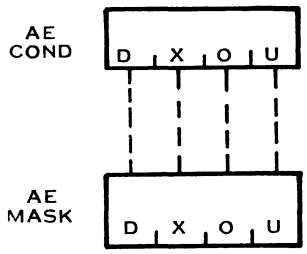
Refer to Topics 7-131 and 7-134.

The arithmetic exception condition code can be reset in its entirety by the LAC instruction, Topic 7-167.

6-62. ARITHMETIC EXCEPTION MASK

The arithmetic exception mask is a four-bit field that is used to specify whether detection of any given arithmetic exception (or combination of exceptions) is to cause a program interruption. When a given bit is set to a zero, detection of an arithmetic exception condition corresponding to that bit will not cause program interruption; when the bit is set to one, detection of the corresponding arithmetic exception will cause program interruption.

The bits of the arithmetic exception mask correspond on a one-to-one basis with those of the arithmetic exception condition code; illustratively:



The arithmetic exception condition code and the arithmetic exception mask are continuously compared (within the arithmetic unit), and, at any time a bit setting of one occurs in both corresponding bits, a program interrupt signal is issued to the peripheral processor for system action.

6-63. Setting the Arithmetic Exception Mask

The arithmetic exception mask is set by the load arithmetic exception mask (LAM) instruction. Refer to Topic 7-166.

Since only bits four through seven of the word accessed by the LAM instruction are loaded and all other bits of the word are ignored, the data constant which specifies the desired interrupt conditions can be built as either a fullword or a left halfword. Illustratively, the singleword accessed by LAM appears as if it were:

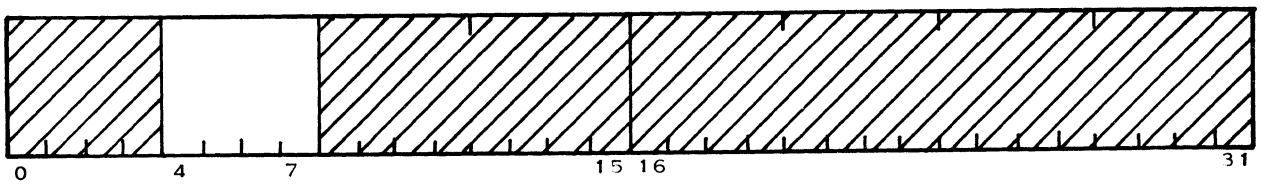


Table 6-6. Specifications for Arithmetic Exception Mask Data Constants

INTERRUPTS	HEXADECIMAL	INTERRUPTS	HEXADECIMAL
none	#0000 0000	D	#0800 0000
U	#0100 0000	D or U	#0900 0000
O	#0200 0000	D or O	#0A00 0000
O or U	#0300 0000	D, O, or U	#0B00 0000
X	#0400 0000	D or X	#0C00 0000
X or U	#0500 0000	D, X, or U	#0D00 0000
X or O	#0600 0000	D, X, or O	#0E00 0000
X, O, or U	#0700 0000	D, X, O, or U	#0F00 0000

6-64. PROGRAM COUNTER

The program counter is a 24-bit field which contains the current instruction address.

The value in this field informs the Central Processor which instruction in the program to begin processing when the signal from the Peripheral Processor (system) starts processing. The capability of the Central Processor to store the program status doubleword and then reinstate it (both on signal from the Peripheral Processor) enables program interruptions by the system without destruction of the currently executing program.

SECTION VII
THE SCALAR INSTRUCTIONS FOR THE CENTRAL PROCESSOR

7-1. INTRODUCTION

This section describes the scalar instructions implemented in the Central Processor. The assembler mnemonic for each instruction is given with the instruction name, and then a description of the instruction with its operands, restrictions, limitations, and other programming information follows.

7-2. LOAD REGISTER INSTRUCTIONS

Table 7-1 lists the load register instructions discussed on the following pages.

Table 7-1. Load Register Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
L	Load, Word	7-3
LLL	Load, Left Halfword from Left	7-4
LRR	Load, Right Halfword from Right	7-5
LRL	Load, Right Halfword from Left	7-5.1
LLR	Load, Left Halfword from Right	7-6
LD	Load, Doubleword	7-7
LI	Load Immediate, Word	7-8
LIH	Load Immediate, Halfword	7-9
LN	Load Negative, Fixed Point Word	7-10
LNH	Load Negative, Fixed Point Halfword	7-11
LNF	Load Negative, Floating Point Word	7-12
LND	Load Negative, Floating Point Doubleword	7-13
LM	Load Magnitude, Fixed Point Word	7-14
LMH	Load Magnitude. Fixed Point Halfword	7-15
LMF	Load Magnitude, Floating Point Word	7-16
LMD	Load Magnitude, Floating Point Doubleword	7-17
LNM	Load Negative Magnitude, Fixed Point Word	7-18
LNMH	Load Negative Magnitude, Fixed Point Halfword	7-19
LNMF	Load Negative Magnitude, Floating Point Word	7-20
LNMD	Load Negative Magnitude, Floating Point Doubleword	7-21
LO	Load Ones Complement, Word	7-22
LF	Load Register File	7-23
LFM	Load Register Files, Multiple	7-24

7-3. LOAD, WORD (L)

The instruction L causes the data in the effective address to replace the contents of the register addressed by the register operand.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	L	ß	r,[@][=]n[,x]

Examples:

	L	B2,(A3)
	L	X1,@NUM,X3

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
BR, AR, XR, or VR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: If base register zero (B0) is addressed by the register operand, the result code is set according to the algebraic value in the effective address although base register zero remains set to zero.

7-4. LOAD, LEFT HALFWORD FROM LEFT (LLL)

The instruction LH causes the data in the effective halfword address to replace the contents of the left half of the arithmetic register addressed by the register operand.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	LLL	⊬	r,[@][=]n[,x]

Examples:

```
        LLL     A1,=#12F3
        LLL     A2,NUM,X3
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-5. LOAD, RIGHT HALFWORD FROM RIGHT (LRR)

The instruction LRR causes the data in the effective halfword address to replace the contents of the right half of the arithmetic register addressed by the register operand.

Terminal index displacement is by halfword increments beginning from the initial right halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	LRR	ß	r,[@][=]n[,x]

Examples:

	LRR	A1,WORD
	LRR	A1,@WORD,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
right half of AR, only	right half of BR, AR XR, VR, or CM	zero	n/a	right half of BR, AR, XR, VR, or CM
		XR	even	
			odd	left half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-5.1 LOAD, RIGHT HALFWORD FROM LEFT (LRL)

The instruction LRL causes the data in the effective halfword address to replace the contents of the right half of the arithmetic register addressed by the register operand.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL	⊬	COMMAND	⊬	OPERANDS
[symbol]	⊬	LRL	⊬	r,[@][=]n[,x]

Examples:

```
        LRL     A1,WORD
        LRL     A1,@WORD,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
right half of AR, only	left half of BR, AR XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-6. LOAD, LEFT HALFWORD FROM RIGHT (LLR)

The instruction LLR causes the data in the effective halfword address to replace the contents of the left half of the arithmetic register addressed by the operand.

Terminal index displacement is by halfword increments beginning from the initial right halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	LLR	⊬	r,[@][=]n[,x]

Examples:

```
        LLR     A1,WORD
        LLR     A1,@WORD,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	right half of BR, AR, XR, VR, or CM	zero	n/a	right half of BR, AR, XR, VR, or CM
		XR	even	
			odd	left half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-7. LOAD, DOUBLEWORD (LD)

The instruction LD causes the data in the effective doubleword address to replace the contents of the even-odd arithmetic register pair addressed by the register operand.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	LD	ß	r,[@][=]n[,x]

Examples:

	LD	A2,ADDR
	LD	A2,@ADDR,X3

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	specification error if AR is odd

Restrictions: Both register and effective addresses must be even valued.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by doubleword increments.

7-8. LOAD IMMEDIATE, WORD (LI)

The instruction LI causes the effective immediate operand to replace the contents of the register addressed by the register operand.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	LI	ß	r, i[, x]

Examples:

	LI	X2, 123
	LI	V1, #3CF9, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
AR, XR, or VR	zero	none
	XR	$-2^{23} \leq m \leq 2^{23}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

7-9. LOAD IMMEDIATE, HALFWORD (LIH)

The instruction LIH causes the effective immediate operand to replace the contents of the left half of the arithmetic register addressed by the register operand.

GENERAL FORM:

LABEL	⊬	COMMAND	⊬	OPERANDS
[symbol]	⊬	LIH	⊬	r, i[, x]

Examples:

	LIH	A2, #36C9
	LIH	A1, 1236, X4

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
left half of AR, only	zero XR right half	none $-2^{15} \leq m \leq 2^{15}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

7-10. LOAD NEGATIVE, FIXED POINT WORD (LN)

The instruction LN causes the algebraic negative (twos complement) of the value in the effective address to replace the contents of the arithmetic register addressed by the register operand.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	LN	⊬	r, [@][=]n[, x]

Examples:

	LN	A1, RSLT
	LN	A1, @ADDR, X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Note: Fixed point overflow and failure to change from negative to positive occur for the case of the algebraically largest negative number; i. e., the twos complement of 8000 0000 (base 16) is 8000 0000.

7-11. LOAD NEGATIVE, FIXED POINT HALFWORD (LNH)

The instruction LNH causes the algebraic negative (two's complement) of the val in the effective halfword address to replace the contents of the left half of the arithmetic register addressed by the register operand.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL	⊬	COMMAND	⊬	OPERANDS
[symbol]	⊬	LNH	⊬	R,[@][=]n[,x]

Examples:

```
        LNH     A1,RSLT
        LNH     A1,@ADDR,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Note: Fixed point overflow and failure to change from negative to positive occur for the case of the algebraically largest negative number; i.e., the twos complement of 8000 (base 16) is 8000.

7-12. LOAD NEGATIVE, FLOATING POINT WORD (LNF)

The instruction LNF causes the algebraic negative (sign change only) of the value in the effective address to replace the contents of the arithmetic register addressed by the register operand.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	LNF	⊬	r,[@][=]n[,x]

Examples:

	LNF	A1, FLOAT
	LNF	A1, (A2)
	LNF	A1, @(A3), X2

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

7-13. LOAD NEGATIVE, FLOATING POINT DOUBLEWORD (LND)

The instruction LND causes the algebraic negative (sign change only) of the value in the effective doubleword address to replace the contents of the even-odd arithmetic register pair addressed by the register operand.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL	⌀	COMMAND	⌀	OPERANDS
[symbol]	⌀	LND	⌀	r, [@][=]n[, x]

Examples:

```
        LND     A2, (A4)
        LND     A2, @ADDR, X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	specification error if AR is odd

Restrictions: Both register and effective addresses must be even valued.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by doubleword increments.

7-14. LOAD MAGNITUDE, FIXED POINT WORD (LM)

The instruction LM causes the absolute value of the data in the effective address to replace the contents of the arithmetic register addressed by the register operand.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	LM	⊬	r,[@][=]n[,x]

Examples:

	LM	A1,WORD
	LM	A1,@ADDR,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Note: A result code of negative and fixed point overflow are possible only for the case of the algebraically largest negative number, i.e., 8000 0000 (base 16).

7-15. LOAD MAGNITUDE, FIXED POINT HALFWORD (LMH)

The instruction LMH causes the absolute value of the data in the effective halfword address to replace the contents of the left half of the arithmetic register addressed by the register operand.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	LMH	⊬	r,[@][=]n[,x]

Examples:

```
        LMH     A1,WORD
        LMH     A1,@ADDR,X2
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Note: A result code of negative and fixed point overflow are possible only for the case of the algebraically largest negative number, i.e., 8000 (base 16).

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-16. LOAD MAGNITUDE, FLOATING POINT WORD (LMF)

The instruction LMF causes the absolute value of the data in the effective address to replace the contents of the arithmetic register addressed by the register operand.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	LMF	ß	r,[@][=]n[,x]

Examples:

```
        LMF     A1,RSLT
        LMF     A1,@ADDR,X3
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+ or 0	none

Note: A negative result is not possible.

7-17. LOAD MAGNITUDE, FLOATING POINT DOUBLEWORD (LMD)

The instruction LMD causes the absolute value of the data in the effective doubleword address to replace the contents of the even-odd arithmetic register pair addressed by the register operand.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	LMD	ß	r,[@][=]n[,x]

Examples:

	LMD	A2, RSLT
	LMD	A4, @ADDR, X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+ or 0	specification error if AR is odd

Restrictions: Both register and effective addresses must be even valued.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by doubleword increments.

7-18. LOAD NEGATIVE MAGNITUDE, FIXED POINT WORD (LNM)

The instruction LNM causes the negative (two's complement) of the absolute value of the data in the effective address to replace the contents of the arithmetic register addressed by the register operand.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	LNM	ß	r,[@][=]n[,x]

Examples:

```
        LNM     A1,(A2)
        LNM     A1,@ADDR,X2
```

Addressing.

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
0 or -	none

Note: A positive result is not possible.

7-19. LOAD NEGATIVE MAGNITUDE, FIXED POINT HALFWORD (LNMH)

The instruction LNMH causes the negative (two's complement) of the absolute value of the data in the effective halfword address to replace the contents of the left half of the arithmetic register addressed by the register operand.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL	⌀	COMMAND	⌀	OPERANDS
[symbol]	⌀	LNMH	⌀	r,[@][=]n[,x]

Examples:

```
        LNMH    A1,RSLT
        LNMH    A1,@ADDR,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
0 or -	none

Note: A positive result is not possible.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-20. LOAD NEGATIVE MAGNITUDE, FLOATING POINT WORD (LNMF)

The instruction LNMF causes the negative (sign change only) of the absolute value of the data in the effective address to replace the contents of the arithmetic register addressed by the register operand.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	LNMF	ß	r,[@][=]n[,x]

Examples:

	LNMF	A1,(A2)
	LNMF	A1,@ADDR,X3

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
0 or -	none

Note: A positive result is not possible.

7-21. LOAD NEGATIVE MAGNITUDE, FLOATING POINT DOUBLEWORD (LNMD)

The instruction LNMD causes the negative (sign change only) of the absolute value of the data in the effective doubleword address to replace the contents of the even-odd arithmetic register pair addressed by the register operand.

Terminal index displaced is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	LNMD	ß	r, [@][=]n[, x]

Examples:

```
        LNMD    A2, RSLT
        LNMD    A4, @(A3), X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
0 or -	specification error if AR is odd

Restrictions: Both register and effective addresses must be even valued.

Note: A positive result is not possible.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by doubleword increments.

7-22. LOAD ONE'S COMPLEMENT, WORD (LO)

The instruction LO causes the one's complement of the data in the effective address to replace the contents of the arithmetic register addressed by the register operand.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	b	LO	b	r, [@][=]n[, x]

Examples:

	LO	A1, (A1)
	LO	A1, @ADDR, X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

7-23. LOAD REGISTER FILE (LF)

The instruction LF causes the contents of the word octet beginning at the effective address to replace the contents of the eight-word register file addressed by the mask operand.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	LF	⊬	m,[@]n[,x]

Examples:

	LF	1,BASEREG,X2
	LF	4,@VECTR,X3

Addressing:

MASK VALUE	REGISTER FILE LOADED	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
0 1 2 3 4 5	BR-A BR-B AR-C AR-D XR-X VR-V	BR, AR, XR, VR, or CM	zero or XR	register file BR-A, BR-B, AR-C, AR-D, XR-X, or VR-V, or CM octet
6,7	no operation, take next instruction			

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Restrictions: The effective address will be forced to a multiple of eight, i.e., to an octet boundary.

Note: With a mask value of six or seven, all parts of this instruction are assembled and it can be made operative by programmed alteration of the R field.

Note: Base register zero is wired to zero and will not be altered if file A is loaded.

7-24. LOAD REGISTER FILES, MULTIPLE (LFM)

The instruction LFM causes the contents of the six central memory octets that begin at the effective address to replace the contents of the six eight-word register files.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	LFM	⊬	[@]n[, x]

Examples:

```
        LFM     SAVEREG
        LFM     @ADDR, X1
```

Addressing:

ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
CM only	zero or XR	CM only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	specification error if effective address is a register address

Restrictions: The effective address must refer to central memory and will be forced to a multiple of eight, i.e., to an octet boundary.

Note: Base register zero will not be altered since it is wired to zero.

7-25. STORE INSTRUCTIONS

Table 7-2 lists the store instructions discussed on the following pages.

Table 7-2. Store Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
ST	Store Word	7-26
STLL	Store Left Halfword Into Left	7-27
STRR	Store Right Halfword Into Right	7-28
STRL	Store Right Halfword Into Left	7-28.1
STLR	Store Left Halfword Into Right	7-29
STD	Store Doubleword	7-30
STZ	Store Zero, Word	7-31
STZH	Store Zero, Halfword	7-32
STZD	Store Zero, Doubleword	7-33
STN	Store Negative, Fixed Point Word	7-34
STNH	Store Negative, Fixed Point Halfword	7-35
STNF	Store Negative, Floating Point Word	7-36
STND	Store Negative, Floating Point Doubleword	7-37
STO	Store Ones Complement, Word	7-38
STOH	Store Ones Complement, Halfword	7-39
STF	Store Register File	7-40
STFM	Store Register Files, Multiple	7-41

7-26. STORE WORD (ST)

The instruction ST causes the data in the register addressed by the register operand to replace the contents of the effective address.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	ST	⊬	r,[@]n[,x]

Examples:

ST	B1, RSLT
ST	X2, SAVEREG, X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
BR, AR, XR, or VR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

7-27. STORE LEFT HALFWORD INTO LEFT (STLL)

The instruction STLL causes the data in the left half of the arithmetic register addressed by the register operand to replace the contents of the effective halfword address.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	STLL	ß	r,[@]n[,x]

Examples:

```
        STLL    A1,STORE
        STLL    A2,@ADDR,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-28. STORE RIGHT HALFWORD INTO RIGHT (STRR)

The instruction STRR causes the data in the right half of the arithmetic register addressed by the register operand to replace the contents of the effective halfword address.

Terminal index displacement is by halfword increment beginning from the initial right halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	STRR	⊬	r,[@]n[,x]

Examples:

	STRR	A3, RSLT
	STRR	A2, @RSLT, X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
right half of AR, only	right half of BR, AR, XR, VR, or CM	zero	n/a	right half of BR, AR, XR, VR, or CM
		XR	even	
			odd	left half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: Indirect address indexing by singleword increments although the terminal indexing is by halfword increments.

7-28.1 STORE RIGHT HALFWORD INTO LEFT (STRL)

The instruction STRL causes the data in the right half of the arithmetic register addressed by the register operand to replace the contents of the effective halfword address.

Terminal index displacement is by halfword increment beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	STRL	ß	r,[@]n[,x]

Examples:

	STRL	A3, RSLT
	STRL	A2, @RSLT, X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
right half of AR, only	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: Indirect address indexing by singleword increments although the terminal indexing is by halfword increments.

7-29. STORE LEFT HALFWORD INTO RIGHT (STLR)

The instruction STLR causes the data in the left half of the arithmetic register addressed by the register operand to replace the contents of the effective halfword address.

Terminal index displacement is by halfword increments beginning from the initial right halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	STLR	⊬	r,[@]n[,x]

Examples:

```
        STLR    A1,(A2)
        STLR    A2,@RSLT,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	right half of BR, AR, XR, VR, or CM	zero	n/a	right half of BR, AR, XR, VR, or CM
		XR	even	
			odd	left half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-30. STORE DOUBLEWORD (STD)

The instruction STD causes the data in the even-odd arithmetic register pair addressed by the register operand to replace the contents of the effective doubleword address.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	STD	⊬	r,[@]n[,x]

Examples:

	STD	A2,RSLT
	STD	A2,@ADDR,X2

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	specification error if AR is odd

Restrictions: Both register and effective addresses must be even valued.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by doubleword increments.

7-31. STORE ZERO, WORD (STZ)

The instruction STZ causes a word of zeros to be stored into the effective address.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	STZ	⊬	[@]n[, x]

Examples:

	STZ	RSLT
	STZ	@(A1), X1

Addressing:

ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
zero	none

7-32. STORE ZERO, HALFWORD (STZH)

The instruction STZH causes a halfword of zeros to be stored into effective halfword address.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	STZH	⊬	[@]n[,x]

Examples:

```
        STZH    RSLT
        STZH    @(A1),X1
```

Addressing:

ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
	XR	even	
		odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
zero	none

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-33. STORE ZERO, DOUBLEWORD (STZD)

The instruction STZD causes a doubleword of zeros to be stored into the effective doubleword address.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	b	STZD	b	[@]n[, x]

Examples:

```
        STZD    RSLT
        STZD    @(A3), X1
```

Addressing:

ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM even-odd pair

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
zero	none

Restrictions: The effective doubleword address must be even valued.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by doubleword increments.

7-34. STORE NEGATIVE, FIXED POINT WORD (STN)

The instruction STN causes the algebraic negative (two's complement) of the value in the arithmetic register addressed by the register operand to be stored into the effective address.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	b	STN	b	r,[@]n[,x]

Examples:

	STN	A1,(A1)
	STN	A1,@ADDR,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Note: Fixed point overflow is possible only for the algebraically largest negative value and the value stored is the same as the value in the register; i.e., the two's complement of 8000 0000 (base 16) is 8000 0000.

7-35. STORE NEGATIVE, FIXED POINT HALFWORD (STNH)

The instruction STNH causes the algebraic negative (two's complement) of the value in the left half of the arithmetic register addressed by the register operand to be stored into the effective halfword address.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL	⌀	COMMAND	⌀	OPERANDS
[symbol]	⌀	STNH	⌀	r,[@]n[,x]

Examples:

```
        STNH    A1,RSLT
        STNH    A1,@ADDR,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Note: Fixed point overflow is possible only for the algebraically largest negative value and the value stored is the same as the value in the register; i.e., the two's complement of 8000 (base 16) is 8000.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-36. STORE NEGATIVE, FLOATING POINT WORD (STNF)

The instruction STNF causes the algebraic negative (sign change only) of the value in the arithmetic register addressed by the register operand to be stored into the effective address.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	STNF	⊬	r,[@]n[,x]

Examples:

	STNF	A1, FLOAT
	STNF	A1,@ADDR, X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

7-37. STORE NEGATIVE, FLOATING POINT DOUBLEWORD (STND)

The instruction STND causes the algebraic negative (sign change only) of the value in the even-odd arithmetic register pair addressed by the register operand to be stored into the effective doubleword address.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	STND	⊬	r,[@]n[,x]

Examples:

```
        STND    A2,FLOAT
        STND    A2,@ADDR,X2
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	specification error if AR is odd

Restrictions: Both register and effective addresses must be even valued.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by doubleword increments.

7-38. STORE ONE'S COMPLEMENT, WORD (STO)

The instruction STO causes the one's complement of the data in the arithmetic register addressed by the register operand to be stored into the effective address.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	STO	ß	r,[@]n[,x]

Examples:

	STO	A1,NEG
	STO	A1,@ADDR,X2

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

7-39. STORE ONE'S COMPLEMENT, HALFWORD (STOH)

The instruction STOH causes the one's complement of the data in the left half of the arithmetic register addressed by the register operand to be stored into the effective halfword address.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↓	STOH	↓	r,[@]n[,x]

Examples:

```
        STOH    A1,RSLT
        STOH    A1,@ADDR,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-40. STORE REGISTER FILE (STF)

The instruction STF causes the contents of the eight-word register file addressed by the mask operand to be stored into the word octet beginning at the effective address.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	STF	ß	m,[@]n[,x]

Examples:

	STF	1,SAVEREG
	STF	5,@ADDR,X1

Addressing:

MASK VALUE	REGISTER FILE STORED	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
0 1 2 3 4 5	BR-A BR-B AR-C AR-D XR-X VR-V	BR, AR, XR, VR, or CM	zero or XR	register file BR-A, BR-B AR-C, AR-D XR-X, or VR-V, or CM octet
6,7	word octet of zeros stored			

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Restrictions: The effective address will be forced to a multiple of eight, i.e., to an octet boundary.

7-41. STORE REGISTER FILES, MULTIPLE (STFM)

The instruction STFM causes the contents of the six eight-word register files to be stored into the six central memory word octets that begin at the effective address.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	STFM	⊬	[@]n[, x]

Examples:

```
        STFM    SAVEREG
        STFM    @ADDR,X1
```

Addressing:

ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
CM only	zero or XR	CM only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	specification error if effective address is a register address

Restrictions: The effective address must refer to central memory and will be forced to a multiple of eight, i. e., to an octet boundary.

7-42. ARITHMETIC INSTRUCTIONS

Table 7-3 lists the arithmetic instructions discussed on the following pages.

Table 7-3. Arithmetic Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
A	Add, Fixed Point Word	7-43
AH	Add, Fixed Point Halfword	7-44
AF	Add, Floating Point Word	7-45
AFD	Add, Floating Point Doubleword	7-46
AI	Add, Immediate, Fixed Point Word	7-47
AIH	Add Immediate, Fixed Point Halfword	7-48
AM	Add Magnitude, Fixed Point Word	7-49
AMH	Add Magnitude, Fixed Point Halfword	7-50
AMF	Add Magnitude, Floating Point Word	7-51
AMFD	Add Magnitude, Floating Point Doubleword	7-52
S	Subtract, Fixed Point Word	7-53
SH	Subtract, Fixed Point Halfword	7-54
SF	Subtract, Floating Point Word	7-55
SFD	Subtract, Floating Point Doubleword	7-56
SI	Subtract Immediate, Fixed Point Word	7-57
SIH	Subtract Immediate, Fixed Point Halfword	7-58
SM	Subtract Magnitude, Fixed Point Word	7-59
SMH	Subtract Magnitude, Fixed Point Halfword	7-60
SMF	Subtract Magnitude, Floating Point Word	7-61
SMFD	Subtract Magnitude, Floating Point Doubleword	7-62
M	Multiply, Fixed Point Word	7-63
MH	Multiply, Fixed Point Halfword	7-64
MF	Multiply, Floating Point Word	7-65
MFD	Multiply, Floating Point Doubleword	7-66
MI	Multiply, Immediate, Fixed Point Word	7-67
MIH	Multiply Immediate, Fixed Point Halfword	7-68
D	Divide, Fixed Point Word	7-69
DH	Divide, Fixed Point Halfword	7-70
DF	Divide, Floating Point Word	7-71
DFD	Divide, Floating Point Doubleword	7-72
DI	Divide Immediate, Fixed Point Word	7-73
DIH	Divide Immediate, Fixed Point Halfword	7-74

7-43. ADD, FIXED POINT WORD (A)

The instruction A causes the value in the effective address to be added to the value in the register addressed by the register operand, and causes the sum to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↓	A	↓	r,[@][=]n[,x]

Examples:

A	B1,=#100
A	X1,@ADDR,X2

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
BR, AR, XR, or VR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

7-44. ADD, FIXED POINT HALFWORD (AH)

The instruction AH causes the value in the effective halfword address to be added to the value in the left half of the arithmetic register addressed by the register operand, and causes the sum to be loaded into the left half of the register.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	AH	⊬	r,[@][=]n[,x]

Examples:

```
        AH      A1,=#FFF
        AH      A1,@ADD,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-45. ADD, FLOATING POINT WORD (AF)

The instruction AF causes the value in the effective address to be added to the value in the arithmetic register addressed by the register operand, and causes the sum to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	AF	⊬	r,[@][=]n[,x]

Examples:

	AF	A1,(A2)
	AF	A1,RSLT,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow floating point exponent underflow

7-46. ADD, FLOATING POINT DOUBLEWORD (AFD)

The instruction AFD causes the value in the effective doubleword address to be added to the value in the even-odd arithmetic register pair addressed by the register operand, and causes the sum to be loaded into the register pair.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL	⊬	COMMAND	⊬	OPERANDS
[symbol]	⊬	AFD	⊬	r,[@][=]n[,x]

Examples:

```
        AFD     A2,(A4)
        AFD     A2,RSLT,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow floating point exponent underflow specification error if AR is odd

Restrictions: Both register and effective addresses must be even valued.

Note: Indirect address indexing is by singleword increments although the terminal indexing is doubleword increments.

7-47. ADD IMMEDIATE, FIXED POINT WORD (AI)

The instruction AI causes the effective immediate operand to be added to the value in the register addressed by the register operand, and causes the sum to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	AI	⊬	r, i[, x]

Examples:

	AI	A1, 13
	AI	B1, 15, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
BR, AR, XR, or VR	zero XR	none $-2^{23} \leq m \leq 2^{23}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

7-48. ADD IMMEDIATE, FIXED POINT HALFWORD (AIH)

The instruction AIH causes the effective immediate operand to be added to the value in the left half of the arithmetic register addressed by the register operand, and causes the sum to be loaded into the left half of the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	AIH	ß	r, i[, x]

Examples:

	AIH	A2, 3
	AIH	A2, 5, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
left half of AR, only	zero	none
	XR right half	$-2^{15} \leq m \leq 2^{15}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

7-49. ADD MAGNITUDE, FIXED POINT WORD (AM)

The instruction AM causes the absolute value of the data in the effective address to be added to the value in the arithmetic register addressed by the register operand, and causes the sum to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	AM	ß	r,[@][=]n[,x]

Examples:

	AM	A1, ABS, X1
	AM	A1, @ADDR, X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Note: Fixed point overflow is possible when the value in the arithmetic register is positive; it will occur when the value in the effective address is the algebraically largest negative number (8000 0000, base 16) and the register contents are zero or positive.

7-50. ADD MAGNITUDE, FIXED POINT HALFWORD (AMH)

The instruction AMH causes the absolute value of the data in the effective halfword address to be added to the value in the left half of the register addressed by the register operand, and causes the sum to be loaded into the left half of the register.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL	␣	COMMAND	␣	OPERANDS
[symbol]	␣	AMH	␣	r, [@][=]n[, x]

Examples:

```
        AMH     A3, RSLT
        AMH     A3, @ADDR, X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Note: Fixed point overflow is possible when the value in the register halfword is positive; it will occur when the value in the effective halfword is the algebraically largest negative number (8000, base 16) and the register halfword contents are zero or positive.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-51. ADD MAGNITUDE, FLOATING POINT WORD (AMF)

The instruction AMF causes the absolute value of the data in the effective address to be added to the value in the arithmetic register addressed by the register operand, and causes the sum to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	AMF	⊬	r,[@][=]n[,x]

Examples:

	AMF	A1,ABS
	AMF	A1,@ADDR,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow

7-52. ADD MAGNITUDE, FLOATING POINT DOUBLEWORD (AMFD)

The instruction AMFD causes the absolute value of the data in the effective doubleword address to be added to the value in the even-odd arithmetic register pair addressed by the register operand, and causes the sum to be loaded into the register pair.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	AMFD	ß	r,[@][=]n[,x]

Examples:

```
        AMFD    A2,ADD
        AMFD    A2,@ADDR,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow specification error if AR is odd

Restrictions: Both register and effective addresses must be even-valued..

Note: Indirect address indexing is by singleword increments although terminal indexing is by doubleword increments.

7-53. SUBTRACT, FIXED POINT WORD (S)

The instruction S causes the value in the effective address to be subtracted from the value in the arithmetic register addressed by the register operand, and causes the difference to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	S	ß	r,[@][=]n[,x]

Examples:

	S	B1,=#100
	S	X1,@ADDR,X2

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

7-54. SUBTRACT, FIXED POINT HALFWORD (SH)

The instruction SH causes the value in the effective halfword address to be subtracted from the value in the left half of the arithmetic register addressed by the register operand, and causes the difference to be loaded into the left half of the register.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↓	SH	↓	r,[@][=]n[,x]

Examples:

	SH	A1,=#FFFF
	SH	A1,@SUB,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-55. SUBTRACT, FLOATING POINT WORD (SF)

The instruction SF causes the value in the effective address to be subtracted from the value in the arithmetic register addressed by the register operand, and causes the difference to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	SF	ß	r,[@][=]n[,x]

Examples:

	SF	A1,(A2)
	SF	A1,RSLT,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow floating point exponent underflow

7-56. SUBTRACT, FLOATING POINT DOUBLEWORD (SFD)

The instruction SFD causes the value in the effective doubleword address to be subtracted from the value in the even-odd arithmetic register pair addressed by the register operand, and causes the difference to be loaded into the register pair.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	SFD	⊬	r,[@][=]n[,x]

Examples:

	SFD	A2,(A4)
	SFD	A2,RSLT,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow floating point exponent underflow specification error if AR is odd

Restrictions: Both register and effective addresses must be even valued.

Note: Indirect address indexing is by singleword increments although terminal indexing is by doubleword increments.

7-57. SUBTRACT IMMEDIATE, FIXED POINT WORD (SI)

The instruction SI causes the effective immediate operand to be subtracted from the value in the arithmetic register addressed by the register operand, and causes the difference to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	SI	⊬	r, i[, x]

Examples:

	SI	A1, 13
	SI	A1, 15, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
AR	zero	none
	XR	$-2^{23} \leq m \leq 2^{23}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

7-58. SUBTRACT IMMEDIATE, FIXED POINT HALFWORD (SIH)

The instruction SIH causes the effective immediate operand to be subtracted from the value in the left half of the arithmetic register addressed by the register operand, and causes the difference to be loaded into the left half of the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	SIH	⊬	r, i[, x]

Examples:

SIH	A2, 3
SIH	A2, 5, x1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
left half of AR, only	zero	none
	XR right half	$-2^{15} \leq m \leq 2^{15}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

7-59. SUBTRACT MAGNITUDE, FIXED POINT WORD (SM)

The instruction SM causes the absolute value of the data in the effective address to be subtracted from the value in the arithmetic register addressed by the register operand, and causes the difference to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	SM	⊬	r,[@][=]n[,x]

Examples:

	SM	A1,ABS,X1
	SM	A1,@ADDR,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Note: Fixed point overflow is possible when the value in the arithmetic register is negative; it will occur when the value in the effective address is the algebraically largest negative number (8000 0000, base 16) and the contents of the arithmetic register are negative.

7-60. SUBTRACT MAGNITUDE, FIXED POINT HALFWORD (SMH)

The instruction SMH causes the absolute value of the data in the effective halfword address to be subtracted from the value in the left half of the arithmetic registe addressed by the register operand, and causes the difference to be loaded into the left half of the register.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	SMH	⊬	r,[@][=]n[,x]

Examples:

	SMH	A3, RSLT
	SMH	A3,@ADDR, X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Note: Indirect address indexing is by singleword increments although terminal indexing is by halfword increments.

7-61. SUBTRACT MAGNITUDE, FLOATING POINT WORD (SMF)

The instruction SMF causes the absolute value of the data in the effective address to be subtracted from the value in the arithmetic register addressed by the register operand, and causes the difference to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	SMF	ß	r,[@][=]n[,x]

Examples:

```
        SMF     A1,ABS
        SMF     A1,@ADDR,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent underflow

7-62. SUBTRACT MAGNITUDE, FLOATING POINT DOUBLEWORD (SMFD)

The instruction SMFD causes the absolute value of the data in the effective doubleword address to be subtracted from the value in the even-odd arithmetic register pair addressed by the register operand, and causes the difference to be loaded into the register pair.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	SMFD	ß	r,[@][=]n[,x]

Examples:

```
        SMFD    A2,SUB
        SMFD    A2,@SUB,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent underflow specification error if AR is odd

Restrictions: Both register and effective addresses must be even valued.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by doubleword increments.

7-63. MULTIPLY, FIXED POINT WORD (M)

The instruction M causes the value in the effective address to be multiplied by the value in the register addressed by the register operand, and causes the product to be loaded into the register or registers.

The full doubleword integer product is loaded into an even-odd register pair if the register operand addresses an even arithmetic register; but only the least significant half of the integer product is saved and loaded if the register operand addresses an odd arithmetic register or a register from any other register file.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	M	⊬	r,[@][=]n[,x]

Examples:

	M	A1,(A2)
	M	A2,RSLT,X1

Addressing:

REGISTER OPERAND	DESTINATION OF RESULT	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
even AR	AR even-odd pair	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM singleword
odd AR	odd AR singleword			
BR, XR, or VR	BR, XR, or VR singleword			

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Limitations: Where only a singleword product is saved, the product, p, must have a value within the range: $-2^{31} \leq p \leq 2^{31}-1$; otherwise, a fixed point overflow occurs.

Restrictions: When an even arithmetic register is addressed in the register operand, the data in the succeeding odd arithmetic register will be replaced by the result of this instruction.

7-64. MULTIPLY, FIXED POINT HALFWORD (MH)

The instruction MH causes the value in the effective halfword address to be multiplied by the value in the left half of the arithmetic register addressed by the register operand, and causes the fullword product (32 bits) to be loaded into the register.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	MH	⊬	r,[@][=]n[,x]

Examples:

	MH	A1,RSLT
	MH	A1,@ADDR,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
AR; left half for data, fullword product	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Restrictions: Any data in the right half of the arithmetic register will be replaced by the result of this operation.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-65. MULTIPLY, FLOATING POINT WORD (MF)

The instruction MF causes the value in the effective address to be multiplied by the value in the arithmetic register addressed by the register operand, and causes the product to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	MF	⊬	r,[@][=]n[,x]

Examples:

	MF	A1,(A2)
	MF	A1,MULT,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow floating point exponent underflow

7-66. MULTIPLY, FLOATING POINT DOUBLEWORD (MFD)

The instruction MFD causes the value in the effective doubleword address to be multiplied by the value in the even-odd arithmetic register pair addressed by the register operand, and causes the product to be loaded into the register pair.

Terminal indexing is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	MFD	ß	r,[@][=]n[,x]

Examples:

	MFD	A2,(A4)
	MFD	A2,RSLT,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow floating point exponent underflow specification error if AR is odd

Restrictions: Both register and effective addresses must be even valued.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by doubleword increments.

7-67. MULTIPLY IMMEDIATE, FIXED POINT WORD (MI)

The instruction MI causes the effective immediate operand to be multiplied by the value in the register addressed by the register operand, and causes the product to be loaded into the register or registers.

The full doubleword integer product is loaded into an even-odd register pair if the register operand addresses an even arithmetic register; but only the least significant half of the integer product is saved and loaded if the register operand addresses an odd arithmetic register or a register from any other register file.

GENERAL FORM:

LABEL	⊬	COMMAND	⊬	OPERANDS
[symbol]	⊬	MI	⊬	r, i, [, x]

Examples:

	MI	B2, #FFCF
	MI	V1, 3, X3

Addressing:

REGISTER OPERAND	DESTINATION OF RESULT	INDEX OPERAND	IMMEDIATE MODIFIER
even AR	AR even-odd pair	zero	none
odd AR	odd AR singleword	XR	$-2^{23} \leq m \leq 2^{23}-1$
BR, XR, or VR	BR, XR, or VR singleword		

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

Limitations: When only a singleword product is saved, the product, p, must have a value within the range; $-2^{31} \leq p \leq 2^{31}-1$; otherwise, a fixed point overflow occurs.

Restrictions: When an even arithmetic register is addressed in the register operand, the data in the succeeding odd arithmetic register will be replaced by the result of this instruction.

7-68. MULTIPLY IMMEDIATE, FIXED POINT HALFWORD (MIH)

The instruction MIH causes the effective immediate operand to be multiplied by the value in the left half of the arithmetic register addressed by the register operand, and causes the fullword product (32 bits) to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	MIH	⊬	r, i[, x]

Examples:

	MIH	A1, 3
	MIH	A1, 5, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
AR; left half for data, fullword product	zero	none
	XR right half	$-2^{15} \leq m \leq 2^{15}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Restrictions: Any data in the right half of the arithmetic register will be replaced by the result of this operation.

7-69. DIVIDE, FIXED POINT WORD (D)

The instruction D causes the value in the effective address to be divided into the value in the arithmetic register addressed by the register operand, and causes the quotient to be loaded into the single register addressed by the operand.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	D	⊬	r,[@][=]n[,x]

Examples:

	D	A1,(A2)
	D	A2,RSLT,X1

Addressing:

REGISTER OPERAND	DIVIDEND	RESULT	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
even AR	AR even-odd pair	AR even singleword	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM singleword
odd AR	AR odd singleword	AR odd singleword			

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow divide check

Limitations: When a quotient is derived from a doubleword dividend, the quotient, q, must have a value within the range: $-2^{31} \leq q \leq 2^{31}-1$; otherwise, a fixed point overflow will occur.

Note: When a doubleword dividend is selected (by addressing an even arithmetic register), the second word of the dividend is not altered when the quotient is loaded into the even singleword arithmetic register.

7-70. DIVIDE, FIXED POINT HALFWORD (DH)

The instruction DH causes the value in the effective halfword address to be divided into the value in the arithmetic register (fullword) addressed by the register operand, and causes the quotient to be loaded into the left half of the register.

Terminal index displacement is by halfword increments beginning from the initial left halfword of the index word set.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	DH	⊬	r, [@][=]n[, x]

Examples:

```
        DH      A1, RSLT
        DH      A2, @RSLT, X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
AR; fullword of data, halfword quotient	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow divide check

Limitations: The quotient, q, must have a value within the range: $-2^{15} \leq q \leq 2^{15}-1$; otherwise, a fixed point overflow will occur.

Note: The right halfword of the dividend is not altered when the halfword quotient is loaded.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by halfword increments.

7-71. DIVIDE, FLOATING POINT WORD (DF)

The instruction DF causes the value in the effective address to be divided into the value in the arithmetic register addressed by the register operand, and causes the quotient to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol2	ß	DF	ß	r,[@][=]n[,x]

Examples:

```
        DF      A1,RSLT
        DF      A1,@RSLT,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow floating point exponent underflow divide check

7-72. DIVIDE, FLOATING POINT DOUBLEWORD (DFD)

The instruction DFD causes the value in the effective doubleword address to be divided into the value in the even-odd arithmetic register pair addressed by the register operand, and causes the quotient to be loaded into the even odd register pair.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	b	DFD	b	r,[@][=]n[,x]

Examples:

```
        DFD     A2,(A4)
        DFD     A2,@RSLT,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow floating point exponent underflow divide check specification error if AR is odd

Restrictions: Both register and effective addresses must be even-valued.

Note: Indirect address indexing is by singleword increments although the terminal indexing is by doubleword increments.

7-73. DIVIDE IMMEDIATE, FIXED POINT WORD (DI)

The instruction DI causes the effective immediate operand to be divided into the value in the arithmetic register addressed by the register operand, and causes the quotient to be loaded into the single register addressed by the operand.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	DI	⊬	r, i[, x]

Examples:

	DI	A1, 3
	DI	A1, #FF, X1

Addressing:

REGISTER OPERAND	DIVIDEND	QUOTIENT	INDEX OPERAND	IMMEDIATE MODIFIER
AR	AR singleword	AR singleword	zero	none
			XR	$-2^{23} \leq m \leq 2^{23}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	divide check

7-74. DIVIDE IMMEDIATE, FIXED POINT HALFWORD (DIH)

The instruction DIH causes the effective immediate operand to be divided into the value in the arithmetic register (fullword) addressed by the register operand, and causes the quotient to be loaded into the left half of the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	DIH	⊬	r, i[, x]

Examples:

	DIH	A3, 1
	DIH	A3, #C, X3

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
AR; fullword dividend, left halfword quotient	zero	none
	XR right half	$-2^{15} \leq m \leq 2^{15}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow divide check

Limitations: The quotient, q, must have a value within the range: $-2^{15} \leq q \leq 2^{15}-1$; otherwise, a fixed point overflow will occur.

Note: The right halfword of the dividend is not altered when the halfword quotient is loaded.

7-75. LOGICAL INSTRUCTIONS

Table 7-4 lists the logical instructions discussed on the following pages.

Table 7-4. Logical Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
AND	AND, Word	7-76
ANDD	AND, Doubleword	7-77
ANDI	AND Immediate, Word	7-78
OR	OR, Word	7-79
ORD	OR, Doubleword	7-80
ORI	OR Immediate, Word	7-81
XOR	Exclusive OR, Word	7-82
XORD	Exclusive OR, Doubleword	7-83
XORI	Exclusive OR Immediate, Word	7-84
EQC	Equivalence, Word	7-85
EQCD	Equivalence, Doubleword	7-86
EQCI	Equivalence Immediate, Word	7-87

7-76. AND, WORD (AND)

The instruction AND causes the data in the effective address to be ANDed, bit by corresponding bit, with the data in the arithmetic register addressed by the register operand, and causes the result to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	AND	ß	r,[@][=]n[,x]

Examples:

```
        AND     A1,@(A3),X1
        AND     A1,RSLT,X2
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none

7-77. AND, DOUBLEWORD (ANDD)

The instruction ANDD causes the data in the effective doubleword address to be ANDed, bit by corresponsing bit, with the data in the even-odd arithmetic register pair addressed by the register operand, and causes the result to be loaded into the register pair.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	ANDD	⊬	r,[@][=]n[,x]

Examples:

	ANDD	A2,(A4)
	ANDD	A2,@RSLT,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	specification error if AR is odd

7-78. AND IMMEDIATE, WORD (ANDI)

The instruction ANDI causes the effective immediate operand to be ANDed, bit by corresponding bit, with the data in the arithmetic register addressed by the register operand, and causes the result to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↓	ANDI	↓	r, i[, x]

Examples:

```
        ANDI    A1,#FFCD
        ANDI    A1,#FF39,X1
```

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
AR	zero	none
	XR	$0 \leq m \leq 2^{23}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none

Restrictions: Modification of the immediate operand by an index value is by one's complement addition only; i. e., there is no sign extension from the 24-bit index and no end-around-carry to the least significant bit.

Note: The effective immediate operand is 24 bits, therefore bits 0-7 of the ANDed result can never be ones.

7-79. OR, WORD (OR)

The instruction OR causes the data in the effective address to be ORed, bit by corresponding bit, with the data in the arithmetic register addressed by the register operand, and causes the result to be loaded into the register.

GENERAL FORM:

LABEL	␣	COMMAND	␣	OPERANDS
[symbol]	␣	OR	␣	r,[@][=]n[,x]

Examples:

	OR	A1,(A4)
	OR	A1,@WORD,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none

7-80. OR, DOUBLEWORD (ORD)

The instruction ORD causes the data in the effective doubleword address to be ORed, bit by corresponding bit, with the data in the even-odd arithmetic register pair addressed by the register operand, and causes the result to be loaded into the register pair.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	ORD	ß	r, [@][=]n[, x]

Examples:

```
        ORD     A2, RSLT
        ORD     A4, @RSLT, X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	specification error if AR is odd

7-81. OR IMMEDIATE, WORD (ORI)

The instruction ORI causes the effective immediate operand to be ORed, bit by corresponding bit, with the data in the arithmetic register addressed by the register operand, and causes the result to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↓	ORI	↓	r, i[, x]

Examples:

	ORI	A1, 3
	ORI	A1, #FF, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
AR	zero	none
	XR	$0 \leq m \leq 2^{23}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none

Restrictions: Modification of the immediate operand by an index value is by one's complement addition only; i. e., there is no sign extension from the 24-bit index and no end-around-carry to the least significant bit.

7-82. EXCLUSIVE OR, WORD (XOR)

The instruction XOR causes the data in the effective address to be exclusive ORed, bit by corresponding bit, with the data in the arithmetic register addressed by the register operand, and causes the result to be loaded into the register.

GENERAL FORM:

LABEL	↓	COMMAND	↓	OPERANDS
[symbol]	↓	XOR	↓	r,[@][=]n[,x]

Examples:

```
        XOR     A1,(A1)
        XOR     A1,@RSLT,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	none or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none

7-83. EXCLUSIVE OR, DOUBLEWORD (XORD)

The instruction XORD causes the data in the effective doubleword address to be exclusive ORed, bit by corresponding bit, with the data in the even-odd arithmetic register pair addressed by the register operand, and causes the result to be loaded into the register pair.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	XORD	⊬	r,[@][=]n[,x]

Examples:

```
        XORD    A2,RSLT
        XORD    A2,RSLT,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	specification error if AR is odd

7-84. EXCLUSIVE OR IMMEDIATE, WORD (XORI)

The instruction XORI causes the effective immediate operand to be exclusive ORed, bit by corresponding bit, with the data in the arithmetic register addressed by the register operand, and causes the result to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	XORI	⊬	r, i[, x]

Examples:

	XORI	A1, 3
	XORI	A1, #00, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
AR	none	none
	XR	$0 \leq x \leq 2^{23}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none

Note: Modification of the immediate operand by an index value is by one's complement addition only. There is no sign extension or end-around carry.

7-85. EQUIVALENCE, WORD (EQC)

The instruction EQC causes the data in the effective address to be equivalenced, bit by corresponding bit, with the data in the arithmetic register addressed by the register operand, and causes the result to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	EQC	⊬	r, [@][=]n[, x]

Examples:

```
        EQC     A1,(A2)
        EQC     A1,@RSLT,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	none or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none

7-86. EQUIVALENCE, DOUBLEWORD (EQCD)

The instruction EQCD causes the data in the effective doubleword address to be equivalenced, bit by corresponding bit, with the data in the even-odd arithmetic register pair addressed by the register operand, and causes the result to be loaded into the register pair.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	EQCD	⊬	r,[@][=]n[,x]

Examples:

```
        EQCD    A2,(A4)
        EQCD    A2,@RSLT
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	specification error if AR is odd

7-87. EQUIVALENCE IMMEDIATE, WORD (EQCI)

The instruction EQCI causes the effective immediate operand to be equivalenced, bit by corresponding bit, with the data in the arithmetic register addressed by the register operand, and causes the result to be loaded into the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↯	EQCI	↯	r,i[,x]

Examples:

```
        EQCI    A1,#FF
        EQCI    A1,#C1,X1
```

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
AR	zero	none
	XR	$0 \le x \le 2^{23}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none

Note: Modification of the immediate operand by an index value is by one's complement addition only. There is no sign extension or end-around carry.

7-88. SHIFT INSTRUCTIONS

Table 7-5 lists the shift instructions discussed on the following pages.

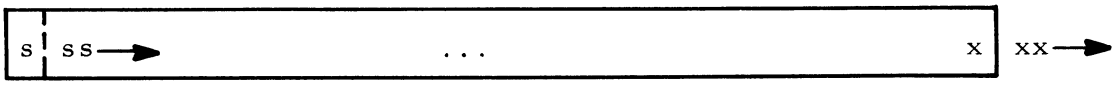
Table 7-5. Shift Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
SA	Arithmetic Shift, Word	7-93
SAH	Arithmetic Shift, Halfword	7-94
SAD	Arithmetic Shift, Doubleword	7-95
SL	Logical Shift, Word	7-96
SLH	Logical Shift, Halfword	7-97
SLD	Logical Shift, Doubleword	7-98
SC	Circular Shift, Word	7-99
SCH	Circular Shift, Halfword	7-100
SCD	Circular Shift, Doubleword	7-101
RVS	Bit Reversal, Word	7-102

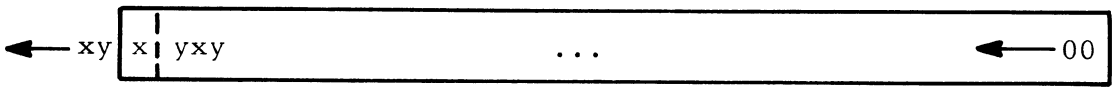
7-89. ARITHMETIC SHIFTS

Arithmetic shifts are implemented to preserve the sign of the shifted fixed point value and to detect fixed point overflow.

An arithmetic right shift (negative shift count) causes the sign bit to be extended, unchanged, so that all replaced bits are ones for negative values and zeros for positive values. Bits shifted out the right end are lost. Illustratively:



During an arithmetic left shift (positive shift count), the sign bit is continuously checked, and, if it changes, a fixed point overflow is indicated in the arithmetic exception condition register. The shift is completed regardless of this condition. Bits shifted through the sign bit are lost and zeros enter the low order bit positions. Illustratively:



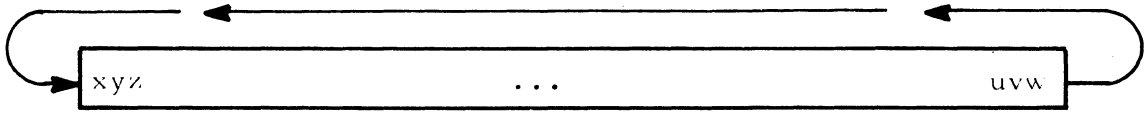
7-90. LOGICAL SHIFTS

Logical shifts preserve no arithmetic information. With either right or left shifts, zeros are entered into the vacated bit positions and bits shifted out the designated end are lost. There are no arithmetic exception conditions.

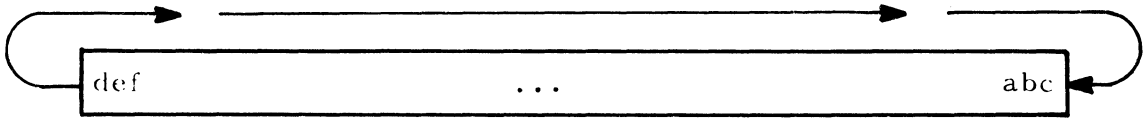
7-91. CIRCULAR SHIFTS

In circular shifts, the bits shifted out the designated end are entered into the bit positions being vacated at the other end.

A circular right shift (negative shift count) causes bits to be shifted out the right end of the word and to be reentered at the left end. Illustratively:



A circular left shift (positive shift count) causes bits to be shifted out the left end of the word and to be reentered at the right end. Illustratively:



The result code is set logically.

Note: The 7-bit shift count is used for all word sizes. If the resulting shift count should exceed the word size of 32-bits for single word shift instructions, then the register result would appear as follows:

Circular right shift - Right shift modulo 32

Circular left shift - Left shift modulo 32

The most significant bit of the T-field is not used, i.e., indirect shift counts are not possible. The M-field must be zero.

7-92. ALGORITHM FOR BIT REVERSAL

Bit reversal is performed by the following steps:

1. If shift count is positive, zero the result field.
2. The word upon which reversal is to be performed is copied twice to produce a doubleword with two identical halves.
3. A circular shift, according to the shift count, is performed on the left singleword.
4. The bits of the right singleword are reversed.
5. A logical shift opposite to that of step 2 is performed on the doubleword.
6. The resultant left singleword is sent to the specified arithmetic register.

This instruction reverses the rightmost bits of an arithmetic register. Other bits remain unchanged. Values greater than zero show results as indicated below. The initial bit assignment for register AR runs from 0 on the left to 31 on the right. Bit Reversal values of 0 and -1 leave register AR unchanged.

Bit Reversal Output

Values of na	Contents of AR After Bit Reversal Instruction	
+ 1 to +Max.	ØØØ . . .	Ø
0	012 . . .	30 31
-1	012 . . .	30 31
-2	012 . . .	28 29 31 30
-3	012 . . .	27 28 31 30 29
-4	012 . . .	26 27 31 30 29 28
.	.	.
.	.	.
.	.	.
-29	0 1 2 31 30 29 . . .	5 4 3
-30	0 1 31 30 29 . . .	5 4 3 2
-31	0 31 30 29 . . .	5 4 3 2 1
-32	31 30 29 . . .	5 4 3 2 1 0
-33	30 29 28 . . .	2 1 0 Ø
-34	29 28 27 . . .	2 1 0 Ø Ø
-35	28 27 26 . . .	2 1 0 Ø Ø Ø
.	.	.
.	.	.
.	.	.
-60	3 2 1 0 Ø . . .	Ø
-61	2 1 0 Ø . . .	Ø
-62	1 0 Ø . . .	Ø
-63	0 Ø . . .	Ø
-64 to -Max.	Ø . . .	Ø

Where Ø represents a bit position that is zero.

7-93. ARITHMETIC SHIFT, WORD (SA)

The instruction SA causes the data in the arithmetic register addressed by the register operand to be arithmetically shifted left or right, the number of bits specifie by the effective immediate operand. If the effective immediate operand is positive, the shift is to the left, and if it is negative, the shift is to the right.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	SA	ß	r, i[, x]

Examples:

	SA	A1, 3, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER	EFFECTIVE IMMEDIATE
AR	zero	none	$-64 \leq i \leq +63$
	XR	right half XR	

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow, left shift only

Restrictions: Regardless of the index value, the effective immediate operand will be within the range: $-64 \leq i \leq +63$. Index overflow is truncated, but not detected.

Note: During left shifts any change of the sign bit causes a fixed point overflow.

7-94. ARITHMETIC SHIFT, HALFWORD (SAH)

The instruction SAH causes the data in the left half of the arithmetic register addressed by the register operand to be arithmetically shifted, left or right, the number of bits specified by the effective immediate operand. If the effective immediate operand is positive, the shift is to the left, and if it is negative, the shift is to the right.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	b	SAH	b	r, i[, x]

Examples:

	SAH	A3, 6, X2

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER	EFFECTIVE IMMEDIATE
left half of AR, only	zero	none	-64≤ i ≤+ 63
	XR	right half XR	

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow, left shift only

Restrictions: Regardless of the index value, the effective immediate operand will be within the range: $-64 \leq i \leq +63$. Index overflow is truncated, but not detected.

Note: The contents of the right half of the register are not affected.

Note: During left shifts any change of the sign bit causes a fixed point overflow, but the total shift is completed regardless of this condition.

7-95. ARITHMETIC SHIFT, DOUBLEWORD (SAD)

The instruction SAD causes the data in the even-odd arithmetic register pair addressed by the register operand to be shifted arithmetically left or right, the number of bits specified by the effective immediate operand. If the effective immediate operand is positive, the shift is to the left, and if it is negative, the shift is to the right.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	SAD	⊬	r, i[, x]

Examples:

	SAD	A2, 10, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER	EFFECTIVE IMMEDIATE
AR; even-odd pair only	zero	none	-64≤ i ≤+ 63
	XR	right half XR	

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow, left shift only specification error if AR is odd

Restrictions: Regardless of the index value, the effective immediate operand will be within the range: $-64 \leq i \leq +63$. Index overflow is truncated, but not detected.

Note: During left shifts any change of the sign bit causes a fixed point overflow, but the total shift is completed regardless of this condition.

7-96. LOGICAL SHIFT, WORD (SL)

The instruction SL causes the data in the arithmetic register addressed by the register operand to be shifted logically, left or right, the number of bits specified by the effective immediate operand. If the effective immediate operand is positive, the shift is to the left, and if it is negative, the shift is to the right.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	b	SL	b	r, i[, x]

Examples:

	SL	A1, 10, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER	EFFECTIVE IMMEDIATE
AR	zero	none	-64≤ i ≤+ 63
	XR	right half XR	

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none

Restrictions: Regardless of the index value, the effective immediate operand will be within the range: $-64 \leq i \leq +63$. Index overflow is truncated, but not detected.

7-97. LOGICAL SHIFT, HALFWORD (SLH)

The instruction SLH causes the data in the left half of the arithmetic register addressed by the register operand to be shifted logically, left or right, the number of bits specified by the effective immediate operand. If the effective immediate operand is positive, the shift is to the left, and if it is negative, the shift is to the right.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	SLH	⊬	r, i[, x]

Examples:

	SLH	A1, 10, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER	EFFECTIVE IMMEDIATE
left half of AR, only	zero	none	-64≤ i ≤+ 63
	XR	right half XR	

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none

Restrictions: Regardless of the index value, the effective immediate operand will be within the range: $-64 \leq i \leq +63$. Index overflow is truncated, but not detected.

Note: The contents of the right half of the register are not affected.

7-98. LOGICAL SHIFT, DOUBLEWORD (SLD)

The instruction SLD causes the data in the even-odd arithmetic register pair addressed by the register operand to be shifted logically, left or right, the number of bits specified by the effective immediate operand. If the effective immediate operand is positive, the shift is to the left, and if it is negative, the shift is to the right.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↓b	SLD	↓b	r, i[, x]

Examples:

	SLD	A2, 10, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER	EFFECTIVE IMMEDIATE
AR; even-odd pair only	zero XR	none right half XR	$-64 \leq i \leq +63$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	specification error if AR is odd

Restrictions: Regardless of the index value, the effective immediate operand will be within the range: $-64 \leq i \leq +63$. Index overflow is truncated, but not detected.

7-99. CIRCULAR SHIFT, WORD (SC)

The instruction SC causes the data in the arithmetic register addressed by the register operand to be shifted circularly, left or right, the number of bits specified by the effective immediate operand. If the effective immediate operand is positive, the shift is to the left, and if it is negative, the shift is to the right.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	SC	⊬	r, i[, x]

Examples:

	SC	A1, 10, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER	EFFECTIVE IMMEDIATE
AR	zero	none	$-64 \leq i \leq +63$
	XR	right half XR	

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none

Restrictions: Regardless of the index value, the effective immediate operand will be within the range: $-64 \leq i \leq +63$. Index overflow is truncated, but not detected.

Note: The shift count is performed modulo 32.

7-100. CIRCULAR SHIFT, HALFWORD (SCH)

The instruction SCH causes the data in the left half of the arithmetic register addressed by the register operand to be shifted circularly, left or right, the number of bits specified by the effective immediate operand. If the effective immediate operand is positive, the shift is to the left, and if it is negative, the shift is to the right.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	SCH	⊬	r, i[, x]

Examples:

	SCH	A1, 3, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER	EFFECTIVE IMMEDIATE
left half of AR, only	zero	none	$-64 \leq i \leq +63$
	XR	right half XR	

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none

Restrictions: Regardless of the index value, the effective immediate operand will be within the range : $-64 \leq i \leq +63$. Index overflow is truncated, but not detected.

Note: The contents of the right half of the register are not affected.

Note: The shift count is performed modulo 16.

7-101. CIRCULAR SHIFT, DOUBLEWORD (SCD)

The instruction SCD causes the data in the even-odd arithmetic register pair addressed by the register operand to be shifted circularly, left or right, the number of bits specified by the effective immediate operand. If the effective immediate operand is positive, the shift is to the left, and if it is negative, the shift is to the rig

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	SCD	⊬	r, i[, x]

Examples:

	SCD	A2, 16, X2

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER	EFFECTIVE IMMEDIATE
AR; even-odd pair only	zero	none	-64≤ i≤ + 63
	XR	right half XR	

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	specification error if AR is odd

Restrictions: Regardless of the index value, the effective immediate operand will be within the range: $-64 \leq i \leq +63$. Index overflow is truncated, but not detected.

7-102. BIT REVERSAL, WORD (RVS)

The instruction RVS causes the reversal of the rightmost or leftmost n bits (where n is the value of the effective immediate operand) of the data in the arithmetic register addressed by the register operand. The rightmost n bits are reversed if n is negative, and if n is positive, the register is cleared. The other bits of the register remain unchanged.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	RVS	⊬	r, i[, x]

Examples:

	RVS	A1, -5, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER	EFFECTIVE IMMEDIATE
AR	zero	none	-64≤ i≤ +63
	XR	right half XR	

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none

Restrictions: Regardless of the index value, the effective immediate operand will be within the range: $-64 \leq i \leq +63$.

Note: Although this operation uses a seven bit shift count, the reversal will be modulo 32; thus, operationally, the shift count is within the range: $-32 \leq sc \leq 32$, and sc values 0 and -1 are effectively no operation.

7-103. COMPARE INSTRUCTIONS

Table 7-6 lists the shift instructions discussed on the following pages.

Table 7-6. Compare Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
C	Compare, Fixed Point Word	7-104
CH	Compare, Fixed Point Halfword	7-105
CF	Compare, Floating Point Word	7-106
CFD	Compare, Floating Point Doubleword	7-107
CI	Compare Immediate, Fixed Point Word	7-108
CIH	Compare Immediate, Fixed Point Halfword	7-109
CAND	Compare Logical AND, Word	7-110
CANDD	Compare Logical AND, Doubleword	7-111
CANDI	Compare Logical AND Immediate, Word	7-112
COR	Compare Logical OR, Word	7-113
CORD	Compare Logical OR, Doubleword	7-114
CORI	Compare Logical OR Immediate, Word	7-115

7-104. COMPARE, FIXED POINT WORD (C)

The instruction C causes the compare code to be set to reflect whether the value in the register addressed by the register operand is greater than, equal to, or less than the value in the effective address. The contents of both the register and the effective address are unchanged.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	b	C	b	r,[@][=]n[,x]

Examples:

	C	V1,(A1)
	C	A1,@RSLT,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR, XR, or VR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

COMPARE CODE REFLECTS	PROGRAM INTERRUPTIONS
>, =, or <	none
RESULT CODE	
not affected	

7-105. COMPARE FIXED POINT HALFWORD (CH)

The instruction CH causes the compare code to be set to reflect whether the value in the left half of the arithmetic register addressed by the register operand is greater than, equal to, or less than the value in the effective halfword address. The contents of both the register and the effective address are unchanged.

Terminal index displacement is by halfword increments beginning from the first left halfword of the index word set.

GENERAL FORM:

LABEL	⌀	COMMAND	⌀	OPERANDS
[symbol]	⌀	CH	⌀	r,[@][=]n[,x]

Examples:

```
        CH      A1,RSLT
        CH      A1,RSLT,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
left half of AR, only	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

COMPARE CODE REFLECTS	PROGRAM INTERRUPTIONS
>, =, or <	none
RESULT CODE	
not affected	

7-106. COMPARE, FLOATING POINT WORD (CF)

The instruction CF causes the compare code to be set to reflect whether the value in the arithmetic register addressed by the register operand is greater than, equal to, or less than the value in the effective address. The contents of both the register and the effective address are unchanged.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	CF	⊬	r,[@][=]n[,x]

Examples:

	CF	A1,RSLT
	CF	A1,@RSLT,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

COMPARE CODE REFLECTS	PROGRAM INTERRUPTIONS
>, =, or <	none
RESULT CODE	
not affected	

Note: The input floating point arguments must be hexadecimally normalized prior to use in a floating point compare instruction.

7-107. COMPARE, FLOATING POINT DOUBLEWORD (CFD)

The instruction CFD causes the compare code to be set to reflect whether the value in the even-odd arithmetic register pair addressed by the register operand is greater than, equal to, or less than the value in the effective doubleword address. The contents of both the register and the effective doubleword are unchanged.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	CFD	⊬	r,[@][=]n[,x]

Examples:

```
        CFD     A2,(A4)
        CFD     A2,RSLT,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

COMPARE CODE REFLECTS	PROGRAM INTERRUPTIONS
>, =, or <	specification error if AR is odd
RESULT CODE	
not affected	

Note: The input floating point arguments must be hexadecimally normalized prior to use in a floating point compare instruction.

7-108. COMPARE IMMEDIATE, FIXED POINT WORD (CI)

The instruction CI causes the compare code to be set to reflect whether the value in the register addressed by the register operand is greater than, equal to, or less than the value of the effective immediate operand. The contents of the register are unchanged.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	CI	ß	r, i[, x]

Example:

```
        CI      A1,#FA
        CI      A1,#C,X2
```

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
AR, XR, or VR	zero	none
	XR	$-2^{23} \leq x \leq 2^{23}-1$

Program Status:

COMPARE CODE REFLECTS	PROGRAM INTERRUPTIONS
>, =, or <	none
RESULT CODE	
not affected	

7-109. COMPARE IMMEDIATE, FIXED POINT HALFWORD (CIH)

The instruction CIH causes the compare code to be set to reflect whether the value in the left half of the arithmetic register addressed by the register operand is greater than, equal to, or less than the value of the effective immediate operand. The contents of the register are unchanged.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	CIH	⊬	r, i[, x]

Examples:

	CIH	A1, 10, X1

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
left half of AR, only	zero	none
	XR	right half XR

Program Status:

COMPARE CODE REFLECTS	PROGRAM INTERRUPTIONS
>, =, or <	none
RESULT CODE	
not affected	

7-110. COMPARE LOGICAL AND, WORD (CAND)

The instruction CAND causes the compare code to be set to reflect whether the logical ANDing of the data in the arithmetic register addressed by the register operand with that in the effective address produces a result that is all ones, all zeros, or mixed. The contents of both the register and the effective address are unchanged.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	CAND	⊬	r,[@][=]n[,x]

Examples:

	CAND	A1,(A2)
	CAND	A1,RSLT,X2

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

COMPARE CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none
RESULT CODE	
not affected	

7-111. COMPARE LOGICAL AND, DOUBLEWORD (CANDD)

The instruction CANDD causes the compare code to be set to reflect whether the logical ANDing of the data in the even-odd arithmetic register pair addressed by the register operand with the data in the effective doubleword address produces a result that is all ones, all zeros, or mixed. The contents of both the register pair and the doubleword address are unchanged.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	CANDD	⊬	r,[@][=]n[,x]

Examples:

```
        CANDD   A2,(A4)
        CANDD   A2,RSLT,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

COMPARE CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	specification error if AR is odd
RESULT CODE	
not affected	

7-112. COMPARE LOGICAL AND IMMEDIATE, WORD (CANDI)

The instruction CANDI causes the compare code to be set to reflect whether the logical ANDing of the data in the arithmetic register addressed by the register operand with the effective immediate operand produces a result that is all ones, all zeros, or mixed. The contents of the register are unchanged.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	CANDI	⊬	r, i[, x]

Examples:

```
        CANDI   A1,#FFCD
        CANDI   A1,#0,X1
```

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
AR	zero	none
	XR	$0 \le x \le 2^{23}-1$

Program Status:

COMPARE CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none
RESULT CODE	
not affected	

Note: Modification of the immediate operand by an index value is by one's complement addition only. There is no sign extension or end-around carry.

Note: The effective immediate address is only 24 bits, therefore bits 0-7 of the anded result can never be ones.

7-113. COMPARE LOGICAL OR, WORD (COR)

The instruction COR causes the compare code to be set to reflect whether the logical ORing of the data in the arithmetic register addressed by the register operand with that in the effective address produces a result that is all ones, all zeros, or mixed. The contents of both the register and the effective address are unchanged.

GENERAL FORM:

Examples:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	COR	⊬	r,[@][=]n[,x]
		COR		A1,(A2)
		COR		A1,@(A3),X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

COMPARE CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none
RESULT CODE	
not affected	

7-114. COMPARE LOGICAL OR, DOUBLEWORD (CORD)

The instruction CORD causes the compare code to be set to reflect whether the logical ORing of the data in the even-odd arithmetic register pair addressed by the register operand with the data in the effective doubleword address produces a result that is all ones, all zeros, or mixed. The contents of both the register pair and the doubleword address are unchanged.

Terminal index displacement is by doubleword increments of even-odd word pairs.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	CORD	⊬	r,[@][=]n[,x]

Examples:

	CORD	A2,(A4)
	CORD	A2,@(A1),X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

COMPARE CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	specification error if AR is odd
RESULT CODE	
not affected	

7-115. COMPARE LOGICAL OR IMMEDIATE, WORD (CORI)

The instruction CORI causes the compare code to be set to reflect whether the logical ORing of the data in the arithmetic register addressed by the register operand with the effective immediate operand produces a result that is all ones, all zeros, or mixed. The contents of the register are unchanged.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	CORI	⊬	r, i[, x]

Examples:

```
        CORI    A1, 3
        CORI    A1, #F, X2
```

Addressing:

REGISTER OPERAND	INDEX OPERAND	IMMEDIATE MODIFIER
AR	zero	none
	XR	$0 \leq x \leq 2^{23}-1$

Program Status:

COMPARE CODE REFLECTS	PROGRAM INTERRUPTIONS
ones, zeros, or mixed	none
RESULT CODE	
not affected	

Note: Modification of the immediate operand by an index value is by one's complement addition only. There is no end around carry or sign extension.

7-116. INCREMENT OR DECREMENT, TEST AND SKIP INSTRUCTIONS

Table 7-7 lists the increment or decrement, test and skip instructions discussed on the following pages.

Table 7-7. Increment or Decrement, Test and Skip Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
ISE	Increment, Test and Skip on Equal	7-117
ISNE	Increment, Test and Skip on Not Equal	7-118
DSE	Decrement, Test and Skip on Equal	7-119
DSNE	Decrement, Test and Skip on Not Equal	7-120

7-117. INCREMENT, TEST AND SKIP ON EQUAL (ISE)

The instruction ISE causes the fixed point value in the arithmetic register addressed by the register operand to be incremented by unity and the result to be stored in the register and causes this value to be arithmetically compared to the value in the effective address. If the two values are equal, the next instruction is skipped; if the two values are not equal, the next instruction is executed in turn.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	ISE	ß	r,[@][=]n[,x]

Examples:

	ISE	A1,(X1)
	ISE	A1,@(A2),X1
	ISE	A1,@SAVE,X2

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: If $\alpha \leq 2F$ and M=0, where α is specified to be the same register address as defined by the R-field, then the next instruction is taken.

7-118. INCREMENT, TEST AND SKIP ON NOT EQUAL (ISNE)

The instruction ISNE causes the fixed point value in the arithmetic register addressed by the register operand to be incremented by unity and the result to be stored in the register and causes this value to be arithmetically compared to the value in the effective address. If the two values are not equal, the next instruction is skipped; if the two values are equal, the next instruction is executed in turn.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	ISNE	ß	r,[@][=]n[,x]

Examples:

	ISNE	A1,(X1)
	ISNE	A1,@(A2),X1
	ISNE	A1,@SAVE,X2

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: If $\alpha \leq 2F$ and M=0, where α is specified to be the same register address as defined by the R-field, then the next instruction is skipped.

7-119. DECREMENT, TEST AND SKIP ON EQUAL (DSE)

The instruction DSE causes the fixed point value in the arithmetic register addressed by the register operand to be decremented by unity and the result to be stored in the register, causes this value to be arithmetically compared to the value in the effective address, and, if the two values are equal, causes the next instruction to be skipped; if the two values are not equal, the next instruction is executed in turn.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	DSE	ß	r,[@][=]n[,x]

Examples:

	DSE	A1,(X1)
	DSE	A1,@(A2),X1
	DSE	A1,@SAVE,X2

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: If $\alpha \leq 2^F$ and M=0, where α is specified to be the same register address as defined by the R-field, then the next instruction is taken.

7-120. DECREMENT, TEST AND SKIP ON NOT EQUAL (DSNE)

The instruction DSNE causes the fixed point value in the arithmetic register addressed by the register operand to be decremented by unity and the result to be stored in the register and causes this value to be arithmetically compared to the value in the effective address. If the two values are no equal, the next instruction is skipped; if the two values are equal, the next instruction is executed in turn.

GENERAL FORM:

LABEL	ß	COMMAND	ß	OPERANDS
[symbol]	ß	DSNE	ß	r,[@][=]n[,x]

Examples:

	DSNE	A1,(X1)
	DSNE	A1,@(A2),X1
	DSNE	A1,@SAVE,X2

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

Note: If $\alpha \leq 2F$ and M=0, where α is specified to be the same register address as defined by the R-field, then the next instruction is skipped.

7-121. INCREMENT OR DECREMENT, TEST AND BRANCH INSTRUCTIONS

Table 7-8 lists the increment or decrement, test and branch instructions discussed on the following pages.

Table 7-8. Increment or Decrement, Test and Branch Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
IBZ	Increment, Test and Branch on Zero	7-122
IBNZ	Increment, Test and Branch on Not Zero	7-123
DBZ	Decrement, Test and Branch on Zero	7-124
DBNZ	Decrement, Test and Branch on Not Zero	7-125

7-122. INCREMENT, TEST AND BRANCH ON ZERO (IBZ)

The instruction IBZ causes the fixed point value in the register addressed by the register operand to be incremented by unity and the result stored in the register, and, if the new value is zero, causes a branch to the effective branch address; if the new value is not zero, the next instruction in sequence is executed in turn.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	IBZ	ß	r,[@[=]]n[,x]

Examples:

```
        IBZ     A1,OUT
        IBZ     X1,@SAVE,X2
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR, XR, or VR	CM only	zero or XR	CM only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

7-123. INCREMENT, TEST AND BRANCH ON NOT ZERO (IBNZ)

The instruction IBNZ causes the fixed point value in the register addressed by the register operand to be incremented by unity and the result to be stored in the register, and, if the new value is not zero, causes a branch to the effective branch address; if the new value is zero, the next instruction in sequence is executed in turn.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	IBNZ	ß	r,[@[=]]n[,x]

Examples:

	IBNZ	A1,OUT
	IBNZ	X1,@SAVE,X2

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR, XR, or VR	CM only	zero or XR	CM only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

7-124. DECREMENT, TEST AND BRANCH ON ZERO (DBZ)

The instruction DBZ causes the fixed point value in the register addressed by the register operand to be decremented by unity and the result to be stored in the register, and, if the new value is zero, causes a branch to the effective branch address; if the new value is not zero, the next instruction in sequence is executed in turn.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	DBZ	⊬	r,[@[=]]n[,x]

Examples:

```
        DBZ     A1,OUT
        DBZ     X1,@SAVE,X2
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR, XR, or VR	CM only	zero or XR	CM only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

7-125. DECREMENT, TEST AND BRANCH ON NOT ZERO (DBNZ)

The instruction DBNZ causes the fixed point value in the register addressed by the register operand to be decremented by unity and the result to be stored in the register, and, if the new value is not zero, causes a branch to the effective branch address; if the new value is zero, the next instruction in sequence is executed in turn.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	DBNZ	ß	r,[@[=]]n[,x]

Examples:

	DBNZ	A1,OUT
	DBNZ	X1,@SAVE,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR, XR, or VR	CM only	zero or XR	CM only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

7-126. INDEX, TEST AND BRANCH INSTRUCTIONS

Table 7-9 lists the test and branch instructions discussed on the following pages.

Table 7-9. Index, Test and Branch Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
BCLE	Branch on Less Than or Equal	7-128
BCG	Branch on Greater Than	7-129

7-127. ALGORITHM FOR INDEX, TEST AND BRANCH

An index, test and branch operation proceeds by the following steps:

1. The value in the register addressed by the first operand is added to the value in the even register addressed by the second operand.
2. The sum from step 1 is stored in the first operand register.
3. The sum from step 1 is compared to the value in the odd register addressed by the second operand, and
   a. if the comparison condition is true, a branch is taken to the branch address derived from the third operand, or
   b. if the comparison condition is false, the instruction next in sequence is executed.

Illustratively, for branch on greater than (BCG):

1. REG1 [value, v] + REG2 [increment, i / limit, t]
2. REG1 ⇐ [value + increment]
3. REG1 [value + increment] ≤ REG2 [increment / limit]
   a. if v + i > t, PROGRAM COUNTER ⇐ [operand three]
   b. if v + i ≯ t, PROGRAM COUNTER ⇐ [$+1]

7-128. BRANCH ON LESS THAN OR EQUAL (BCLE)

The instruction BCLE causes the value in the register addressed by the first operand to be added to the increment value in the even-odd register pair addressed by the second operand, causes the result to be stored in the first operand register, causes the result to be arithmetically compared to the limit value in the second operand register pair, and, if the result is less than or equal to the limit, causes a branch to the branch address. If the result is greater than the limit, BCLE causes the instruction next in sequence to be executed.

The increment value is the value in the first word of the even-odd arithmetic register pair addressed by the second operand, and the limit value is the value in the second word of the arithmetic register pair. The branch address is developed from the third operand.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	b	BCLE	b	r, r, n

Examples:

```
        BCLE    A1,A2,OUT
        BCLE    X1,A2,LOOP
```

Addressing:

FIRST REGISTER OPERAND	SECOND REGISTER OPERAND	BRANCH ADDRESS OPERAND
AR, XR, or VR	even-odd AR pair only	CM only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	specification error if second register operand is odd

Restrictions: This instruction uses fixed point arithmetic only.

Limitations: There is no index operand for address or value modification.

7-129. BRANCH ON GREATER THAN (BCG)

The instruction BCG causes the value in the register addressed by the first register operand to be added to the increment value in the even-odd arithmetic register pair addressed by the second operand, causes the result to be stored in the first operand register, causes the result to be arithmetically compared to the limit value in the second operand register pair, and, if the result is greater than the limit, causes a branch to the branch address. If the result is not greater than the limit, the instruction next in sequence to the BCG instruction is executed.

The increment value is the value in the first word of the even-odd arithmetic register pair addressed by the second operand, and the limit value is the value in the second word of the arithmetic register pair. The branch address is developed from the third operand.

GENERAL FORM:

Examples:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	BCG	⊬	r, r, n
		BCG		A1, A2, OUT
		BCG		X1, A2, LOOP

Addressing:

FIRST REGISTER OPERAND	SECOND REGISTER OPERAND	BRANCH ADDRESS OPERAND
AR, XR, or VR	even-odd AR pair only	CM only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	specification error if second register operand is odd

Restrictions: This instruction uses fixed point arithmetic only.

Limitations: There is no index operand for address or value modification.

7-130. CONDITIONAL BRANCH INSTRUCTIONS

Table 7-10 lists the branch instructions discussed on the following pages.

Table 7-10. Conditional Branch Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
BCC	Branch on Comparison Code True	7-132
NOP	No operation (Branch on no conditions)	7-132
B	Unconditional Branch (Branch on any condition)	7-132
BE	Branch on Compare Code of Equal	7-132
BG	Branch on Compare Code of Greater Than	7-132
BGE	Branch on Compare Code of Greater Than or Equal	7-132
BL	Branch on Compare Code of Less Than	7-132
BLE	Branch on Compare Code of Less Than or Equal	7-132
BNE	Branch on Compare Code of Not Equal	7-132
BCZ	Branch on Compare Code of All Bits Are Zero	7-132
BCO	Branch on Compare Code of All Bits Are One	7-132
BCNM	Branch on Compare Code of Not Mixed	7-132
BCM	Branch on Compare Code of Mixed Zeros and Ones	7-132
BCNO	Branch on Compare Code of Not All Ones	7-132
BCNZ	Branch on Compare Code of Not All Zeros	7-132
BRC	Branch on Result Code True	7-133
BZ	Branch on Result Code of Zero	7-133
BPL	Branch on Result Code of Positive	7-133
BZP	Branch on Result Code of Zero or Positive	7-133
BMI	Branch on Result Code of Negative	7-133
BZM	Branch on Result Code of Zero or Negative	7-133
BNZ	Branch on Result Code of Not Zero	7-133
BRZ	Branch on Result Code of All Bits Are Zero	7-133
BRO	Branch on Result Code of All Bits Are One	7-133
BRNM	Branch on Result Code of Bits Not Mixed Zeros and Ones	7-133

Table 7-10. Conditional Branch Instructions (Continued)

MNEMONIC	INSTRUCTION NAME	TOPIC
BRM	Branch on Result Code of Bits Mixed Zeros And Ones	7-133
BRNO	Branch on Result Code of Not All Bits Ones	7-133
BRNZ	Branch on Result Code of Not All Bits Zeros	7-133
BAE	Branch on Arithmetic Exception	7-134
BU	Branch on Floating Point Exponent Underflow	7-134
BO	Branch on Floating Point Exponent Overflow	7-134
BUO	Branch on Floating Point Exponent Underflow or Overflow	7-134
BX	Branch on Fixed Point Overflow	7-134
BXU	Branch on Fixed Point Overflow or Floating Point Exponent Underflow	7-134
BXO	Branch on Fixed Point Overflow or Floating Point Exponent Overflow	7-134
BXUO	Branch on Fixed Point Overflow or Floating Point Exponent Overflow or Underflow	7-134
BD	Branch on Divide Check	7-134
BDU	Branch on Divide Check or Floating Point Exponent Underflow	7-134
BDO	Branch on Divide Check or Floating Point Exponent Overflow	7-134
BDUO	Branch on Divide Check or Floating Point Exponent Overflow or Underflow	7-134
BDX	Branch on Divide Check or Fixed Point Overflow	7-134
BDXU	Branch on Divide Check or Fixed Point Overflow or Floating Point Exponent Underflow	7-134
BDXO	Branch on Divide Check or Fixed Point Overflow or Floating Point Exponent Overflow	7-134
BDXUO	Branch on Divide Check or Fixed Point Overflow or Floating Point Exponent Overflow or Underflow	7-134
BXEC	Branch on Execute Branch Condition True	7-135

7-131. CONDITION ALGORITHMS FOR CONDITIONAL BRANCHES

The conditional branch instructions determine whether the branch path is to be taken according to whether the result is true for a logical operation between the bits in a program status field and the R field mask (mask operand) of the conditional branch instruction. Let the value of the R-field be represented as r_1, r_2, r_3, r_4.

The branch on comparison code true operation branches if, in the logical equation cond = cl · r_1 + cg · r_2 + ce · r_3, cond is true (one). Refer to Topic 6-58.

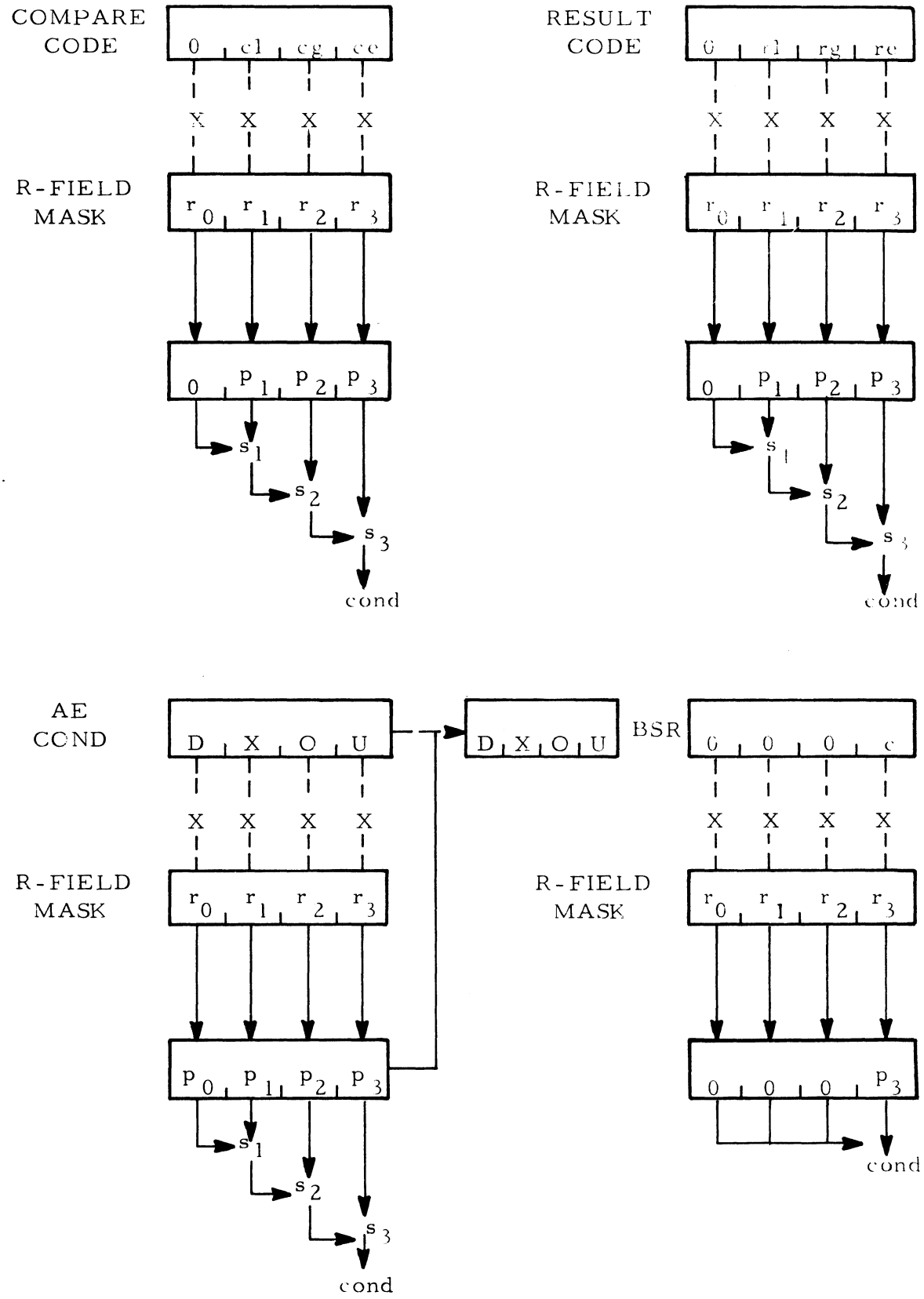
The branch on result code true operation branches if, in the logical equation cond = rl · r_1 + rg · r_2 + re · r_3, cond is true (one). Refer to Topic 6-59.

The branch on arithmetic exception operation branches if, in the logical equation cond = D · r_0 + X · r_1 + O · r_2 + U · r_3, cond is true (one). The arithmetic exception mask is also reset, bit by bit, according to the logical equations: D = D - D · r_0, X = X - X · r_1, O = O - O · r_2, and U = U - U · r_3. Refer to Topic 6-60 and 6-61.

The branch on execute branch condition operation branches if, in the logical equation cond = c · r_3, cond is true (one). Refer to Topic 6-57.

Note: In the proceding equations, · represents a logical AND, + represents a logical OR, and - represents a logical exclusive OR.

Illustratively, these matching operations may be represented:


COMPARE CODE
0 cl cg ce
X X X X
R-FIELD MASK
r0 r1 r2 r3
0 p1 p2 p3
s1
s2
s3
cond
RESULT CODE
0 rl rg re
X X X X
R-FIELD MASK
r0 r1 r2 r3
0 p1 p2 p3
s1
s2
s3
cond
AE COND
D X O U
D X O U
BSR
0 0 0 c
X X X X
R-FIELD MASK
r0 r1 r2 r3
p0 p1 p2 p3
s1
s2
s3
cond
R-FIELD MASK
r0 r1 r2 r3
0 0 0 p3
cond

7-132. BRANCH ON COMPARISON CODE TRUE (BCC)

The instruction BCC causes a branch to the effective branch address if the compare code reflects the condition(s) for branching specified by the mask operand; otherwise, the instruction next in sequence to the BCC instruction is executed in turn.

This same machine instruction has an extended set of mnemonics within the assembler whereby the assembler provides the mask value for the desired branch conditions.

In either method of coding, the compare code condition must have been set in the compare code field of the program status doubleword by the previous execution of an arithmetic or logical compare instruction.

GENERAL FORMS:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	BCC	⊬	m,[@[=]]n[,x]
[symbol]	⊬	code	⊬	[@[=]n[,x]

Examples:

	BCC	LT,LSTHN,X1
	NOP	NONE,X1
	B	ALWAYS
	BE	STOP,X2
	BG	GO
	BGE	HERE,X4
	BL	THERE
	BLE	@GONE,X6
	BNE	NTEQUAL,X1
	BCZ	HERE
	BCO	THERE,X3
	BCNM	GO
	BCM	OUT
	BCNO	IN,X6
	BCNZ	GOING,X7

EXTENDED CODE AND MASK DEFINITIONS: The masks for specifying branch conditions are defined in the following table with the assembler mnemonics that will provide the masks without coding them in the operand field:

MASK VALUE	ARITH CODE	DEFINITIONS		LOGIC CODE
0	NOP	no operation, take next instruction		NOP
		BRANCH IF COMPARE CODE REFLECTS COMPARE		
		ARITHMETIC RESULT OF	LOGICAL RESULT OF	
1	BE	equal	all zeros	BCZ
2	BG	greater than	all ones	BCO
3	BGE	greater than or equal	not mixed ones and zeros	BCNM
4	BL	less than	mixed ones and zeros	BCM
5	BLE	less than or equal	not all ones	BCNO
6	BNE	not equal	not all zeros	BCNZ
7	B	unconditional branch, go to effective branch address		

Addressing:

ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
CM only	zero or XR	CM only

Program Status:

COMPARE CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none
RESULT CODE	
not affected	

7-133. BRANCH ON RESULT CODE TRUE (BRC)

The instruction BRC causes a branch to the effective branch address if the result code reflects the condition(s) for branching specified by the mask operand; otherwise, the instruction next in sequence to the BRC instruction is executed in turn.

This same machine instruction has an extended set of mnemonics within the assembler whereby the assembler provides the mask value for the desired branch conditions.

In either method of coding, the result code condition must have been set in the result code field of the program status doubleword by the previous execution of an instruction which affects the result code.

GENERAL FORMS:

LABEL		COMMAND		OPERANDS
[symbol]	b	BRC	b	m,[@[=]]n[,x]
[symbol]	b	code	b	[@[=]]n[,x]

Examples:

BRC	NEG, INCOMP, X3
BZ	HERE, X2
BPL	THERE, X3
BZP	AGAIN, X3
BMI	OUT, X5
BZM	NOPE, X6
BNZ	AROUND, X7
BRZ	AGAIN, X2
BRO	AROUND, X3
BRNM	LOOP, X4
BRM	HERE, X5
BRNO	THERE, X6
BRNZ	OUT, X7

EXTENDED CODE AND MASK DEFINITIONS: The masks for specifying branch conditions are defined in the following table with the assembler mnemonics that will provide the masks without coding them in the operand field:

MASK VALUE	ARITH CODE	DEFINITIONS		LOGIC CODE
0		no operation, take next instruction		
		BRANCH IF RESULT CODE REFLECTS		
		ARITHMETIC RESULT OF	LOGICAL RESULT OF	
1	BZ	zero	all bits zero	BRZ
2	BPL	positive	all bits one	BRO
3	BZP	zero or positive	not mixed	BRNM
4	BMI	negative	mixed zeros and ones	BRM
5	BZM	zero or negative	not all ones	BRNO
6	BNZ	not zero	not all zeros	BRNZ
7		unconditional branch, go to effective branch address		

Addressing:

ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
CM only	zero or XR	CM only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Note: Although the mask values of zero and seven with the BRC instruction produce a no operation and an unconditional branch, respectively, there is no corresponding mnemonic such as NOP or B to produce those masks as there is for the BCC instruction. The assembler is implemented to produce a machine instruction with the operation code of 91 only for NOP and B.

7-134. BRANCH ON ARITHMETIC EXCEPTION (BAE)

The instruction BAE causes a branch to the effective branch address if the arithmetic exception code reflects the occurrence of the arithmetic exception(s) specified as the branch condition by the mask operand; otherwise, the instruction next in sequence to the BAE instruction is executed in turn. The conditions tested by the BAE instruction are reset to zero in the arithmetic exception code; those not tested are not reset thus permitting cumulative arithmetic exception detection.

This same machine instruction has an extended set of mnemonics within the assembler whereby the assembler provides the mask value for the desired branch conditions.

In either method of coding, the arithmetic exception code must have been set in the arithmetic exception code field of the program status doubleword by the previous execution of an instruction which affects the arithmetic exception code.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	BAE	⊬	m,[@[=]]n[,x]
[symbol]	⊬	code	⊬	[@[=]]n[,x]

Examples:

	BAE	AEX,MESG,X4
	BU	ERROR,X1
	BO	STOP,X2
	BUO	END,X3
	BX	AGAIN,X4
	BXU	WHY,X5
	BXO	@WHERE,X6
	BXUO	HERE,X7
	BD	THERE,X1
	BDU	AROUND,X2
	BDO	@LOOP,X3
	BDUO	OUT,X4
	BDX	GO,X5
	BDXU	GONE,X6
	BDXO	GOING,X7
	BDUXO	ENOUGH,X1

EXTENDED CODE AND MASK DEFINITIONS: The masks for specifying branch conditions are defined in the following table with the assembler mnemonics that will provide the masks without coding them in the operand field.

MASK VALUE	BRANCH IF ARITHMETIC EXCEPTION CODE REFLECTS	CODE
0	no operation, take next instruction	
1	floating point exponent underflow	BU
2	floating point exponent overflow	BO
3	floating point exponent underflow or overflow	BUO
4	fixed point overflow	BX
5	fixed point overflow or floating point exponent underflow	BXU
6	fixed point overflow or floating point exponent overflow	BXO
7	fixed point overflow or floating point exponent underflow or overflow	BXUO
8	divide check	BD
9	divide check or floating point exponent underflow	BDU
10	divide check or floating point exponent overflow	BDO
11	divide check or floating point exponent underflow or overflow	BDUO
12	divide check or fixed point overflow	BDX
13	divide check or fixed point overflow or floating point exponent underflow	BDXU
14	divide check or fixed point overflow or floating point exponent overflow	BDXO
15	divide check or fixed point overflow or floating point exponent underflow or overflow	BDXUO

Addressing:

ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
CM only	zero or XR	CM only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

7-135. BRANCH ON EXECUTE BRANCH CONDITION (BXEC)

The instruction BXEC causes a branch to the effective branch address if the branch or skip register reflects the previous occurrence of an execute instruction (XEC) (see Topic 7-161) which caused execution of a conditional branch or skip instruction wherein the condition for branching or skipping was satisfied; otherwise, the instruction next in sequence to the BXEC instruction is executed in turn. The branch or skip condition bit of the branch or skip register in the program status doubleword is reset to zero whenever the BXEC instruction is executed.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↓	BXEC	↓	[@]n[,x]

Examples:

```
        BXEC    HERE,X1
        BXEC    @SAVE,X2
```

Addressing:

ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
CM only	zero or XR	CM only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

7-136. UNCONDITIONAL BRANCH INSTRUCTIONS

Table 7-11 lists the unconditional branch instructions discussed on the following pages. See the conditional branch instructions for unconditional branches and skips that may be derived from them.

Table 7-11. Unconditional Branch Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
BLB	Branch and Load Base Register With Program Counter	7-137
BLX	Branch and Load Index or Vector Register With Program Counter	7-138

7-137. BRANCH AND LOAD BASE REGISTER WITH PROGRAM COUNTER (BLB)

The instruction BLB causes the program counter value incremented by unity to be loaded into the base register addressed by the register operand, and causes an unconditional branch to the effective branch address.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	BLB	⊬	r,[@[=]]n[,x]

Examples:

	BLB	B1,0
	BLB	B2,@SAVE,X2

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
BR	CM only	zero or XR	CM only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Note: Since it is the second word of the program status doubleword that is being loaded into the base register, the register will contain in its eight most significant bits the arithmetic exception condition code and the arithmetic exception condition mask. See Topic 6-56 for details of the program status doubleword.

This program status information can be reinstated by use of the LAM and LAC instructions.

7-138. BRANCH AND LOAD INDEX OR VECTOR REGISTER WITH PROGRAM COUNTER (BLX)

The instruction BLX causes the program counter value incremented by unity to be loaded into the index register or vector register addressed by the register operand, and causes an unconditional branch to the effective branch address.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	BLX	ß	r,[@[=]]n[,x]

Examples:

	BLX	X1,0
	BLX	V1,@SAVE,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
XR or VR	CM only	zero or XR	CM only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Note: Since it is the second word of the program status doubleword that is being loaded into the index or vector register, the register will contain in its eight most significant bits the arithmetic exception condition code and the arithmetic exception condition mask. See Topic 6-56 for details of the program status doubleword.

This program status information can be reinstated by use of the LAM and LAC instructions.

7-139. STACK INSTRUCTIONS

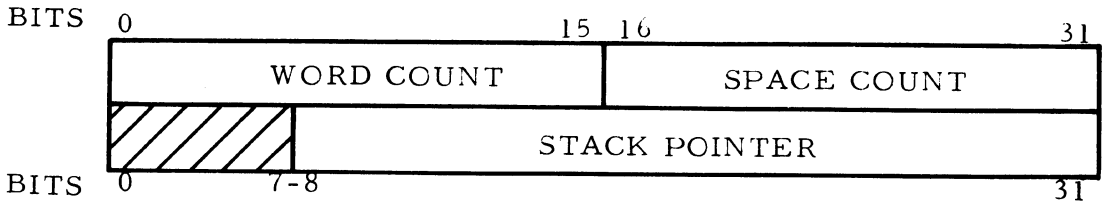
Table 7-12 lists the stack instructions discussed on the following pages. See Topic 7-140 for the definition of a stack operation.

Table 7-12. Stack Instructions

MNEMONIC	INSTRUCTION NAMES	TOPIC
PSH	Push Word Into Last-In-First-Out Stack	7-141
PUL	Pull Word From Last-In-First-Out Stack	7-142
MOD	Modify Stack Parameter Doubleword	7-143

7-140. STACK INSTRUCTION DEFINITION

The stack instructions are for use on push down stacks. They push, pull, and modify pointers on a stack by manipulating a pair of adjacent central memory words (called the stack parameter doubleword). The effective address of the instruction addresses the first word of the pair. The stack parameter doubleword has the following format:



The word count (WC) is the number of words currently in the stack ($2^{15}-1$ words maximum) and the space count (SC) is the count of unused words remaining in the stack ($2^{15}-1$ words maximum).

The stack pointer (SP) is the 24-bit central memory address of the next available location in the stack. Modification of the stack pointer is by full 32-bit two's complement addition.

If the stack acted upon by the stack instruction permits the specified operation, the instruction next in sequence to the stack instruction is skipped; if the operation exceeds the stack parameters (i.e., the stack is already full for PSH, is already empty for PUL, or does not have enough entries or enough spaces for MOD), the next instruction is executed. Thus, the instruction following a stack instruction would be a pointer to a routine that handles the case of a full stack, of an empty stack, or of a stack with inadequate parameters for modification.

Note: The programmer must provide the actual stack and the stack parameter doubleword. The stack parameter doubleword might, for example, be created by DATA directives or by store instructions.

7-141. PUSH WORD INTO LAST-IN-FIRST-OUT STACK (PSH)

The instruction PSH causes the data in the arithmetic register addressed by the register operand to be stored in the next available location in a stack and causes the stack parameter doubleword to be updated. The effective address specifies the address of the stack parameter doubleword that describes the stack.

If the stack is already full (i.e., the space count is zero), the parameters are not changed and the instruction next in sequence to the PSH instruction is executed; otherwise, the next instruction is skipped.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	PSH	⊬	r,[@]n[,x]

Examples:

	PSH	A1,STACK
	PSH	A1,@ADDR,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Limitations: The maximum number of stack entries addressable by this instruction is $2^{15}-1$.

Note: Updating the stack parameter doubleword includes incrementing the word count by unity, decrementing the space count by unity, and incrementing the stack pointer by unity.

7-142. PULL WORD FROM LAST-IN-FIRST-OUT STACK (PUL)

The instruction PUL causes the data in the last created entry in a stack to be loaded into the arithmetic register addressed by the register operand and causes the stack parameter doubleword to be updated. The effective address specifies the address of the stack parameter doubleword that describes the stack.

If the stack is already empty (i. e., the word count is zero), the parameters are not changed and the instruction next in sequence to the PUL instruction is executed; otherwise, the next instruction is skipped.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↓	PUL	↓	r,[@]n[,x]

Examples:

	PUL	A1,STACK
	PUL	A1,@ADDR,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Limitations: The maximum number of stack entries addressable by this instruction is $2^{15}-1$.

Note: Updating the stack parameter doubleword includes decrementing the word count by unity, incrementing the space count by unity, and decrementing the stack pointer by unity.

7-143. MODIFY STACK PARAMETER DOUBLEWORD (MOD)

The instruction MOD causes the stack parameter doubleword addressed by the effective address to be modified by the value in the left half of the arithmetic register addressed by the register operand. A negative (two's complement) modification value causes deletion of the most recent stack entries, and a positive modification value causes creation of a gap of unused stack locations.

If the modification would cause either the word count (exceed the number of present entries for negative modification) or the space count (exceed the maximum table length for positive modification) to become negative, the stack parameter doubleword is not modified and the instruction next in sequence to the MOD instruction is executed; otherwise, the next instruction is skipped.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	MOD	ß	r,[@]n[,x]

Examples:

	MOD	A1,STACK
	MOD	A1,@ADDR,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
left half of AR, only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; even-odd pair only

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Limitations: The modification value, x, is limited to the range: $-2^{15} \leq x \leq 2^{15}-1$.

Note: Modification includes, if x is the modification value, the algebraic addition of x to the word count, algebraic subtraction of x from the space count, and algebraic addition of x to the stack pointer.

Note: This instruction does not alter the stack; it alters the stack parameters and, therefore, the entry to which the stack parameter doubleword presently points.

7-144. CONVERSION AND NORMALIZATION INSTRUCTIONS

Table 7-13 lists the instructions for converting floating point numbers to fixed point numbers and vice versa, and the floating point normalization instructions that are discussed on the following pages.

Table 7-13. Floating Point/Fixed Point Conversion Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
FLFX	Convert Floating Point Word to Fixed Point Word	7-148
FLFH	Convert Floating Point Word to Fixed Point Halfword	7-149
FDFX	Convert Floating Point Doubleword to Fixed Point Word	7-150
FXFL	Convert Fixed Point Word to Floating Point Word	7-151
FHFL	Convert Fixed Point Halfword to Floating Point Word	7-152
FXFD	Convert Fixed Point Word to Floating Point Doubleword	7-153
FHFD	Convert Fixed Point Halfword to Floating Point Doubleword	7-154
NFX	Normalize Fixed Point Word	7-155
NFH	Normalize Fixed Point Halfword	7-156

7-145. ALGORITHM FOR FLOATING POINT TO FIXED POINT CONVERSIONS

The conversion of floating point values to fixed point values is performed by the following steps:

1. Record the sign of the floating point fraction.

2. Subtract 40 base 16 from the biased hexadecimal exponent to obtain the unbiased hexadecimal exponent.

3. Multiply the unbiased hexadecimal exponent by 4 (shift left two bit positions) to obtain the equivalent binary exponent, ie., (nine bits including sign).

4. Align the most significant bit of the floating point fraction (bit position 8) into bit position 1 of the fixed point output register.

5. Insert a zero into the sign bit (bit position 0) of the fixed point output register.

6. Obtain the shift factor, h, for the value in the fixed point output register:

   a. Add 31 to the scale factor, sf, (obtained from the effective halfword address), and

   b. subtract the binary exponent (obtained in step 3), so that

   c. h = 31 + sf - be.

7. Shift the contents of the fixed point output register:

   a. right h bit positions if h is positive

   b. left h bit positions if h is negative, or

   c. not at all if h is zero.

8. If and only if the sign (recorded in step 1) was negative, take the two's complement of the number in the fixed point output register.

9. Send the content of the fixed point output register to its single word or halfword, as appropriate, destination.

7-146. ALGORITHM FOR FIXED POINT TO FLOATING POINT CONVERSIONS

The conversion of fixed point values to floating point values is performed by the following steps:

1. Determine the sign of the fixed point integer to be converted, and
   a. record the sign information, and
   b. if and only if it is negative, take the two's complement of the fixed point number.
2. Add 32 to the binary scale factor, sf, obtained from the effective halfword address (move the decimal from the right to the left end of the value).
3. Perform a floating point normalization on the fixed point fraction by
   a. shifting the fraction left a multiple of four-bit units until at least one of the four most significant bit positions contains a one, and
   b. subtracting four from the binary exponent for each four-bit unit, h, shifted, so that: be = sf + 32 - 4h.
4. Convert the binary exponent to a hexadecimal exponent by
   a. shifting the fraction according to the two least significant bits of the binary exponent. [If these bits are (00) no shift occurs.]
      (1) if no overflow would occur, shift left one bit for (01), two bits for (10), and three bits for (11), or
      (2) if overflow would occur on left shift, add four to the binary exponent and shift the fraction right one bit for (11), two bits for (10), and three bits for (01); and
   b. shifting the binary exponent right two bits (i.e., divide by four).
5. Produce the biased hexadecimal exponent by adding 64 to the hexadecimal exponent modulo 128.
6. Assemble the floating point number by
   a. placing the normalized fraction in the fraction portion of the floating point output,
   b. placing the biased hexadecimal exponent in the exponent portion of the floating point output, and
   c. placing the sign information saved in step 1a in the most significant bit of the floating point output.
7. Send the result to its singleword or doubleword, as appropriate, destination

Note: The operation of both conversion algorithms takes place in the arithmetic unit in registers that are not accessible to the programmer.

7-147. FIXED POINT NORMALIZATION

A fixed point number is said to be normalized when it differs in its two most significant bits, i.e., when the bit pattern is 01 or 10. A fixed point zero is considered to be normalized.

A normalization instruction causes the fixed point value to be shifted left until the two most significant bits differ (unless the value was initially zero), counts the number of bit positions shifted, and stores the shift count as a negative number (two's complement) or zero.

7-148. CONVERT FLOATING POINT WORD TO FIXED POINT WORD (FLFX)

The instruction FLFX causes the floating point word in the arithmetic register addressed by the register operand to be converted to a fixed point word according to the scale factor in the effective halfword address and causes the result to be loaded into the arithmetic register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	FLFX	⊬	r,[@]n[,x]

Examples:

	FLFX	A1, FACTOR
	FLFX	A1,@CM,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
AR, fullword	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

7-149. CONVERT FLOATING POINT WORD TO FIXED POINT HALFWORD (FLFH)

The instruction FLFH causes the floating point word in the arithmetic register addressed by the register operand to be converted to a fixed point halfword according to the scale factor in the effective halfword address and causes the result to be loaded into the left half of the arithmetic register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	b	FLFH	b	r,[@]n[,x]

Examples:

	FLFH	A1,FACTOR
	FLFH	A1,@WORD,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
AR, fullword	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow

7-150. CONVERT FLOATING POINT DOUBLEWORD TO FIXED POINT WORD (FDFX)

The instruction FDFX causes the floating point doubleword in the even-odd arithmetic register pair addressed by the register operand to be converted to a fixed point word according to the scale factor in the effective halfword address and causes the result to be loaded into the even arithmetic register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	FDFX	⊬	r,[@]n[,x]

Examples:

```
        FDFX    A2,FACTOR
        FDFX    A4,@HERE,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
even-odd AR pair	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	fixed point overflow specification error if AR is odd

7-151. CONVERT FIXED POINT WORD TO FLOATING POINT WORD (FXFL)

The instruction FXFL causes the fixed point word in the arithmetic register addressed by the register operand to be converted to a floating point word according to the scale factor in the effective halfword address and causes the normalized result to be loaded into the arithmetic register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	FXFL	ß	r,[@]n[,x]

Examples:

```
        FXFL    A1,FACTOR
        FXFL    A1,@HERE,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
AR, fullword	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow

7-152. CONVERT FIXED POINT HALFWORD TO FLOATING POINT WORD (FHFL)

The instruction FHFL causes the fixed point halfword in the left half of the arithmetic register addressed by the register operand to be converted to a floating point singleword according to the scale factor in the effective halfword address and causes the normalized result to be loaded into the arithmetic register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	FHFL	⊬	r,[@]n[,x]

Examples:

```
        FHFL    A1,FACTOR
        FHFL    A1,@(A2),X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
AR, fullword	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow

7-153. CONVERT FIXED POINT WORD TO FLOATING POINT DOUBLEWORD (FXFD)

The instruction FXFD causes the fixed point word in the even arithmetic register of the even-odd pair addressed by the register operand to be converted to a floating point doubleword according to the scale factor in the effective halfword address and causes the normalized result to be loaded into the even-odd arithmetic register pair.

GENERAL FORM:

LABEL	↓	COMMAND	↓	OPERANDS
[symbol]	↓	FXFD	↓	r,[@]n[,x]

Examples:

	FXFD	A2, FACTOR
	FXFD	A4, @HERE, X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
even-odd AR	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow specification error if AR is odd

7-154. CONVERT FIXED POINT HALFWORD TO FLOATING POINT DOUBLEWORD (FHFD)

The instruction FHFD causes the fixed point halfword in the left half of the even register of the even-odd arithmetic register pair addressed by the register operand to be converted to a floating point doubleword according to the scale factor in the effective halfword address and causes the normalized result to be loaded into the even-odd arithmetic register pair.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	FHFD	⊬	r,[@]n[,x]

Examples:

	FHFD	A1,FACTOR
	FHFD	A1,@HERE,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
even-odd AR pair	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	floating point exponent overflow specification error if AR is odd

7-155. NORMALIZE FIXED POINT WORD (NFX)

The instruction NFX causes the fixed point word in the effective address to be normalized and causes the normalized fixed point result to be loaded into the even register of the even-odd arithmetic register pair addressed by the register operand and the scale factor to be loaded into the right half of the odd register. The left half of the odd register is filled with zeros.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	NFX	ß	r,[@]n[,x]

Examples:

```
        NFX     A2,PNPT
        NFX     A2,@INPT,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR; even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM; singleword

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	specification error if AR is odd

7-156. NORMALIZE FIXED POINT HALFWORD (NFH)

The instruction NFH causes the fixed point halfword in the effective halfword address to be normalized and causes the normalized fixed point result to be loaded into the left half of the arithmetic register addressed by the register operand and the scale factor to be loaded into the right half of the register.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	b	NFH	b	r,[@]n[,x]

Examples:

```
        NFH     A1,INPTH
        NFH     A1,@INPTH,X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	INDEX VALUE	EFFECTIVE ADDRESS
AR fullword	left half of BR, AR, XR, VR, or CM	zero	n/a	left half of BR, AR, XR, VR, or CM
		XR	even	
			odd	right half of BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
+, 0, or -	none

7-157. MISCELLANEOUS INSTRUCTIONS

Table 7-14 lists the miscellaneous instructions discussed on the following pages.

Table 7-14. Miscellaneous Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
XCH	Exchange Words	7-158
LLA	Load Look Ahead	7-159
PB	Prepare to Branch	7-159.1
LEA	Load Effective Address	7-160
XEC	Execute	7-161
INT	Interpret	7-162
FORK	Allow mix of scalar and vector	7-162.1
JOIN	Allow only scalars or only vectors	7-162.2
MCP	Monitor Call and Proceed	7-163
MCW	Monitor Call and Wait	7-164

7-158. EXCHANGE WORDS (XCH)

The instruction XCH causes the current data in the effective address to replace the data in the arithmetic register addressed by the register operand while the current data in the arithmetic register replaces the contents of the effective address; i.e., the current contents of the locations are exchanged.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	XCH	⊬	r,[@]n[,x]

Examples:

```
        XCH     A1,RSLT
        XCH     A1,(A2),X1
```

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

7-159. LOAD LOOK AHEAD (LLA)

The instruction LLA provides the lookahead unit in the Central Processor control hardware with information about a branch instruction that is to be executed and that will most frequently take the branch path. The instruction LLA does not influence the branch decision; it only increases the execution speed of a closed instruction loop.

The value of the immediate operand i must be equal to the number of executable instructions from the instruction LLA through the branch instruction that is expedited; e.g., if LLA were in location 401 and the branch instruction were in location 429, the value of i would be 28.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	LLA	⊬	i

Examples:

```
        LLA     28
        LLA     BRCH-$
```

Addressing: This instruction permits no programmer addressing; the registers into which LLA enters values are internal to the Central Processor control hardware, and index modification of the immediate operand is not performed.

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Restrictions: This instruction may have only an immediate operand without index modification.

Limitations: The maximum applicable program loop size, including the LLA instruction, is 255 instructions, i.e., $0 < i \leq 255$.

Note: The instruction LLA must be included within the program loop defined by the expedited branch instruction so that the branch counter and branch address registers (internal to the central processor) will be reinitialized each time the program reenters the loop.

Note: If the execution of a branch instruction between the LLA and the expedited branch instruction results in its branch path being taken, the branch look ahead information is discarded; if the branch path is not taken, the look ahead information remains current for the intended branch instruction.

Note: The execution of a skip instruction does not alter the number of executable instructions within the domain of the LLA instruction and its related branch instruction; i. e., the instruction immediately following the skip instruction must be counted regardless of whether execution results in the instruction's being skipped.

Note: The LLA and the PB instructions use the source registers, and hence cannot both be effective at once.

7-159.1 PREPARE TO BRANCH (PB)

The instruction PB provides the lookahead unit in the Central Processor control hardware with information about a branch instruction that is to be executed and that will most frequently take the branch path. The instruction PB does not influence the branch decision; it only increases the execution speed of a program branch.

The PB instruction develops a β address from its T-, M-, and N-fields in the same way that a standard branch instruction (BCC or BRC) would do if it were placed at the instruction address of the PB instruction. The R-field of the PB instruction should be set to the difference between the instruction address of the PB instruction and the intended branch instruction. This count may not exceed 15 since the R-field is only four bits. Counts of "0" and "1" are not used.

The internal IPU hardware saves both the β address developed by the PB instruction and the length count specified by the R-field. The length count is decremented by one as each new instruction is entered into the instruction register (IR). At the octet boundary where the look-ahead would normally request the next octet past the octet containing the branch, it recalls the β address saved by the PB instruction and requests it instead of the normal look-ahead octet. In this manner the instruction at the branch address of the target branch instruction will be available for immediate processing following the execution of the target branch instruction.

Should the target branch fail to take the branch, the hardware will realign itself to take the downstream instructions. This is done by rerequesting the branch instruction's octet if necessary, plus the next octet of look-ahead instructions beyond the branch octet.

Note: The LLA and PB instructions use the source registers, and hence cannot both be effective at once.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	PB	⊬	r, @[$ + x]

Examples:

	PB	7, @ $ + 8

In the above example the R-field is "7," designating seven instruction locations from the PB to the branch instruction. The branch address developed by the PB is indirect to the PB instruction address, plus eight (Program counter + 8).

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Limitations: The maximum difference between the PB and the branch address cannot exceed 15 due to the size of the R-field. Values of zero and one should not be used.

Note: The execution of a skip instruction does not alter the number of executable instructions within the domain of the PB instruction and its related branch instruction; i. e., the instruction immediately following the skip instruction must be counted regardless of whether execution results in the instruction's being skipped.

7-160. LOAD EFFECTIVE ADDRESS (LEA)

The instruction LEA causes its own effective address (24 least significent bits) to be developed and causes that address to be loaded into the register addressed by the register operand. The eight most significant bits are zero.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	LEA	⊬	r,[@][=]n[,x]

Examples:

	LEA	X1,$
	LEA	V1,@HERE,X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
BR, XR, or VR	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Restrictions: When used to produce an indirect address constant, this instruction does not produce a T field and, thus, no terminal indexing. See Topic 6-27 for details of the indirect address word format.

Limitations: When this instruction (operation code) is used as an address tracing element, the 24-bit address developed leaves no traces of its mode or path of development; therefore, it will not be possible to distinguish between low memory and registers when the effective address is less than of equal to 2F; neither will it give the correct terminal address if substituted for an indexed halfword or doubleword instruction since it is always indexed as a singleword instruction.

7-161. EXECUTE (XEC)

The instruction XEC causes the instruction at the effective address to be executed and the program counter to be incremented by one; i.e., the next instruction executed is the one following the XEC instruction.

If the instruction at the effective address is a branch or skip instruction and the condition(s) for branching or skipping are true, no actual branch or skip is executed but rather the branch or skip condition bit of the branch or skip register of the program status doubleword is set to one to indicate that the condition(s) were true. The next instruction executed is still the one following the XEC instruction.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	XEC	ß	[@[=]]n[,x]

Examples:

	XEC	$+1,X2
	XEC	@HERE,X1

Addressing:

ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

COMPARE CODE EFFECT	PROGRAM INTERRUPTIONS
that of instruction executed	those of instruction executed
RESULT CODE EFFECT	
that of instruction executed	

Note: If the XEC instruction executes a branch on execute branch condition instruction (BXEC), the branch or skip condition bit of the branch or skip register of the program status doubleword will be reset to zero if it is presently true (one) and will remain zero if it is presently false (zero).

7-162. INTERPRET (INT)

The instruction INT causes the operation code and register (R) field of the instruction in the effective address of this instruction to be loaded (right-justified) into the even arithmetic register of the even-odd arithmetic register pair addressed by the register operand of this instruction, and causes the index, base, and displacement (T, M, and N) fields of that instruction to be loaded (right-justified) into the odd arithmetic register of the even-odd register pair. See Topic 6-2 for a detailed description of the machine code format of instructions.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	INT	⊬	r,[@][=]n[,x]

Examples:

	INT	A4, BRANCH
	INT	A2, @=THERE, X1

Addressing:

REGISTER OPERAND	ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
AR, even-odd pair only	BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	specification error if AR is odd

Note: The 20 most significant bits of the even register and the 12 most significant bits of the odd register are set to zero.

7-162.1 FORK (FORK)

The FORK instruction is an advisory type instruction to the IPU control. Execution of the FORK instruction sets the fork indicator bit within the IPU control and allows subsequent vector or scalar instructions to proceed to execution independently. In the times-four CP, this means that any combination of vector or scalar instructions can be in execution simultaneously in each of the four MBU-AU pairs.

A FORK instruction with the fork indicator already "on" results in the equivalent of a JOIN followed by a FORK.

7-162.2 JOIN (JOIN)

The JOIN instruction is an advisory type instruction to the IPU control. Execution of the JOIN instruction resets a control bit which then disallows parallel pipeline processing of subsequent mixtures of vector and scalar instructions. In the times-four CP, this means that only scalars can be in execution at a time or only a singular vector at a time. Combinations of vectors and scalars cannot be in execution simultaneously.

7-163. MONITOR CALL AND PROCEED (MCP)

The instruction MCP causes a monitor service request signal to be issued to the Peripheral Processor (PP) via the Central Processor/Peripheral Processor communication link, and causes the Central Processor (CP) to proceed with execution of the next instruction.
turned, and causes the Central Processor (CP) to proceed with execution of the next instruction when the storage is complete.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	MCP	ß	i[,x]

Examples:

	MCP	#FFC,X1

Addressing:

INDEX OPERAND	IMMEDIATE MODIFIER
zero	none
XR	$-2^{23} \leq x \leq 2^{23}-1$

Program Status:

RESULT CODE	PROGRAM INTERRUPTIONS
not affected	none

7-164. MONITOR CALL AND WAIT (MCW)

The instruction MCW causes a monitor service request signal to be issued to the Peripheral Processor (PP) via the Central Processor/Peripheral Processor communication link, and causes program execution to begin with the next instruction when the proper context switch returns the program to Central Processor (CP) control.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	MCW	ß	i[, x]

Examples:

	MCW	#F, X2

Addressing:

INDEX OPERAND	IMMEDIATE MODIFIER
zero	none
XR	$-2^{23} \leq x \leq 2^{23}-1$

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Note: There is no hardware protection to ensure that the program is not returned to the Central Processor before the monitor service is completed; this is a function of system programming.

7-165. PROGRAM STATUS INSTRUCTIONS

Table 7-15 list the program status instructions discussed on the following pages.

Table 7-15. Program Status Instructions

MNEMONIC	INSTRUCTION NAME	TOPIC
LAM	Load Arithmetic Exception Mask	7-166
LAC	Load Arithmetic Exception Condition	7-167
LEM	Load Arithmetic Exception Mask and Condition	7-167.1
SCLK	Store 32 Bit, Fixed Point Clock	7-167.2
SPS	Store Program Status Word	7-168

7-166. LOAD ARITHMETIC EXCEPTION MASK (LAM)

The instruction LAM causes the contents of bits four through seven of the effective address to replace the contents of the arithmetic exception mask register (AEM) of the program status doubleword.

The effect is to enable an interrupt signal from the Central Processor (CP) to the Peripheral Processor (PP) for any arithmetic exception condition detected whose corresponding bit is now set to one in the mask. A mask bit of zero disables the interrupt for the corresponding arithmetic exception. See Topics 6-60 through 6-63 for a more detailed description of maskable interrupts.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	ß	LAM	ß	[@[=]]n[, x]

Examples:

	LAM	(B1)
	LAM	RSLT, X1

Addressing:

ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
BR, AR, XR, VR; or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Note: This instruction may be used to reinstate an arithmetic exception mask that was stored in a base, index, or vector register by a previously executed branch and load base register (BLB) or branch and load index register (BLX) instructions. See Topic 7-137, 7-138 and 6-63.

7-167. LOAD ARITHMETIC EXCEPTION CONDITION (LAC)

The instruction LAC causes the contents of bits zero through three of the effective address to replace the contents of the arithmetic exception code register (AEC) of the program status doubleword.

A bit setting of one in the corresponding bit of the code indicates that a specific arithmetic exception condition has been detected in the execution of an instruction, and a bit setting of zero indicates that no corresponding arithmetic exception has been detected. See Topic 6-60 for a more detailed discussion of the maskable interrupts.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↓	LAC	↓	[@[=]]n[,x]

Examples:

	LAC	(B1)
	LAC	RSLT, X1

Addressing:

ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Note: Since the arithmetic exception code (AEC) is changed by this instruction, record of any arithmetic exceptions prior to the execution will be lost unless the current arithmetic exception code is saved through execution of a branch and load base register (BLB) or, branch and load index register (BLX). The arithmetic exception condition code loaded by this instruction may be the reinstatement of such a previously stored arithmetic exception code. See Topics 7-137, 7-138, and 6-61.

7-167.1 LOAD ARITHMETIC EXCEPTION MASK AND CONDITION (LEM)

The instruction LEM loads bits 0 through 3 of the contents of location α into the four-bit arithmetic exception code register (AEC) and loads bits 4 through 7 of the contents of location α into the four-bit arithmetic exception mask register (AEM) of the program status doubleword.

Bits 0 through 3 load the arithmetic exception condition code register as follows:

Bit	
0	Divide check
1	Fixed point overflow
2	Floating point overflow
3	Floating point underflow

Bits 4 through 7 load the arithmetic exception mask as follows:

Bit	
4	Divide check
5	Fixed point overflow
6	Floating point overflow
7	Floating point underflow

Result Code: Not set.

Programming Notes: An interrupt signal from the CP to the PPU is activated if an arithmetic exception is detected and if the mask bit corresponding to that arithmetic exception has been set to a "one." An interrupt is not possible for that arithmetic exception if the mask bit is set to "zero."

Note: Alteration of the AE condition register and AE mask register by a LEM instruction will cause an arithmetic exception program interruption if the corresponding bits of the AE condition register and the AE mask register are both "one" after the LEM instruction has passed through the CP pipeline. This implies that a program interruption will occur after completion of a LEM instruction if any of the following pairs of bits from the contents of location α are both "one":

(0, 4)

(1, 5)

(2, 6)

(3, 7)

This instruction is paired with the BLB and BLX instructions in that the bit positions (bits 0 through 7) agree with the position of the AE condition and AE mask bits stored as a result of a previous BLB or BLX instruction.

7-167.2 STORE CLOCK (SCLK)

The instruction SCLK stores the current value of the 32-bit, fixed-point CP clock into singleword location α. This clock is incremented by "one" every CP clock pulse. It cycles modulo 2^{32} approximately once every four minutes (based on a 60 ns clock rate).

Result Code: Set arithmetically.

7-168. STORE PROGRAM STATUS WORD (SPS)

The instruction SPS causes the second halfword of the program status doubleword to be stored into the effective address. The left half is loaded with zeros. Refer to Topic 6-56 for a more detailed discussion of the program status doubleword.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	⊬	SPS	⊬	[@]n[,x]

Examples:

	SPS	(A1)
	SPS	SAVE,X1

Addressing:

ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
BR, AR, XR, VR, or CM	zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not affected	none

Note: The instruction SPS stores only the second halfword of the program status doubleword; the full program status doubleword is stored in central memory only on a special signal from the Peripheral Processor. The second word of the program status doubleword is loaded into a register by the branch and load base register (BLB) and branch and load index register (BLX) instructions. Refer to Topics 7-137 and 7-138.

SECTION VIII
THE VECTOR INSTRUCTIONS FOR THE CENTRAL PROCESSOR

8-1. INTRODUCTION

Data for vector instructions may be organized into one, two, or three dimensional arrays. A one-dimensional array with one element is called a scalar. A vector is a sequence of elements (a one-dimensional array). A two-dimensional array is a matrix. All other dimensioned arrays are called n-arrays.

The vector instructions cause specific operations which are performed on two series of input data and produce one series of output data. Data is structured as vectors in memory in order to utilize the vector instructions. The definition of a specific vector operation is contained in an octet of data called the Vector Parameter File. The set of instructions which define a vector parameter file are found in Figure 8-1. There are two means of executing the operation defined by the vector parameter file: (1) the file is loaded into the vector registers from memory and executed, (2) the file residing in the vector registers is executed. The vector parameter file is described in Figure 8-1 and should be referred to often during the study of the following sections.

8-2. DEFINITION AND FORMATS OF VECTOR INSTRUCTIONS

The Vector instructions in the ASC Central Processor define vector operations on an ordered set of data. This data may be organized into one-, two-, or three-dimensional arrays. The execution of a vector instruction requires the building of a vector parameter file and the execution of a vector execute instruction. The vector operation, the argument vectors, and the resultant vector are described in the vector parameter file. In Figure 8-1, the contents of the vector registers are described.

In vector operation on three dimensional arrays, the vector looping structure is somewhat analogous to a triple nested DO structure in FORTRAN. The self

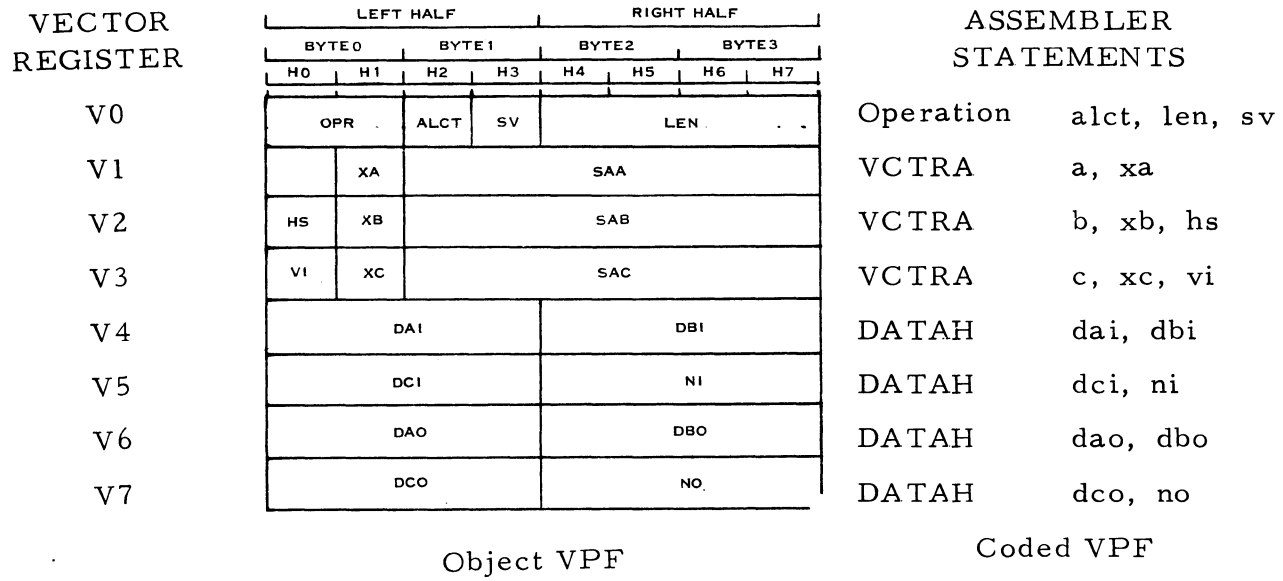
inner, and outer loops of the vector operation correspond to the innermost, middle and outermost loops of a triple nested FORTRAN DO. As in FORTRAN, the outer loop determines the number of iterations of the inner loop, and the inner loop determines the number of iterations of the self loop. The analogy to FORTRAN breaks down when discussing the increment values for each loop. The vector instructions provide more power by allowing variable, positive, or negative increments in the inner and outer loops.

The self loop is analogous to a vector operation such as the operation:

$$\vec{A} \circ \vec{B} \longrightarrow \vec{C}$$

where $\circ$ is the operation to be performed between vectors $\vec{A}$ and $\vec{B}$. The resultant vector is $\vec{C}$. In reference to the vector parameter file (Figure 8-1), the starting address of vectors $\vec{A}$, $\vec{B}$, and $\vec{C}$ are determined by SAA, SAB, and SAC respectively. Each vector in the self loop is a contiguous array. At the end of each self loop, for two- or three-dimensional arrays, SAA, SAB and SAC will have been incremented by the length of the self loop (LEN) -1. Also the increments DAI, DBI and DCI are added to SAA, SAB and SAC respectively. The inner loop count NI, is decremented. When the inner loop has been iterated NI times, for a three-dimensional array, the outer loop increments DAO, DBO, and DCO are added to SAA, SAB, and SAC respectively. The inner loop count (NI) is reset to the original value and the outer loop count (NO) is decremented. When the outer loop count (NO) becomes zero, the vector operation is terminated.

After a vector operation is described in the vector parameter file, the vector operation is performed when the Vector Load and Execute Instruction (VECTL) or the Vector Execute Instruction (VECT) is executed. The instruction VECTL causes the vector parameter file to be loaded into the vector register file, and causes the vector parameter file to be executed. The instruction VECT causes the vector parameter file presently residing in the vector register file to be executed.



REGISTER		SPECIFIES
NUMBER	FIELD	
28	OPR	type of vector operation
	ALCT	arithmetic or logical comparison condition
	SV	single-valued vector
	LEN	vector length (self loop count)
29	XA	vector $\vec{A}$ starting address index
	SAA	starting address of vector $\vec{A}$ or immediate $\vec{A}$
2A	HS	right or left halfword starting addresses
	XB	vector $\vec{B}$ starting address index
	SAB	starting address of vector $\vec{B}$ or immediate $\vec{B}$
2B	VI	self loop increment directions
	XC	vector $\vec{C}$ starting address index
	SAC	starting address of vector $\vec{C}$
2C	DAI	inner loop increment for vector $\vec{A}$ address
	DBI	inner loop increment for vector $\vec{B}$ address
2D	DCI	inner loop increment for vector $\vec{C}$ address
	NI	inner loop count
2E	DAO	outer loop increment for vector $\vec{A}$ address
	DBO	outer loop increment for vector $\vec{B}$ address
2F	DCO	outer loop increment for vector $\vec{C}$ address
	NO	outer loop count

Figure 8-1. The Vector Parameter File

8-3. EXECUTE VECTOR PARAMETER FILE INSTRUCTIONS

The following instructions cause execution of a vector parameter file that exists in the program.

Note that a vector parameter file is not executed from central memory, but must first be loaded into the vector register file.

8-4. VECTOR LOAD AND EXECUTE (VECTL)

The instruction VECTL causes the vector parameter file addressed by the effective octet address to be loaded into the vector register file, and causes the vector parameter file to be executed.

GENERAL FORM:

LABEL		COMMAND		OPERANDS
[symbol]	↓b	VECTL	↓b	[@]n[, x] [n[, x]

Examples:

```
        VECTL   MATX
        VECTL   @VADSET, X1
```

Addressing:

ADDRESS OPERAND	INDEX OPERAND	EFFECTIVE ADDRESS
BR, AR, XR, VR, or CM	Zero or XR	BR, AR, XR, VR, or CM

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not useful after vector operation	those of vector operation being executed

Restrictions: The effective address vector parameter file must be an octet boundary; i.e., a multiple of eight.

Note: The assembler places zero in the R field which specifies that the vector parameter file must be loaded to be executed.

8-5. VECTOR EXECUTE (VECT)

The instruction VECT causes the vector parameter file presently residing in the vector register file to be executed.

GENERAL FORM:

Examples:

LABEL		COMMAND		OPERANDS
[symbol]	ß	VECT	ß	[@]n[, x]
		VECT		MATX
		VECT		@VADSET, X1

Addressing: Addresses are not used by this instruction.

Program Status:

RESULT CODE REFLECTS	PROGRAM INTERRUPTIONS
not useful after vector operation	those of vector operation being executed

Note: Although the assembler generates the T, M, and N fields specified in the operand of the VECT instruction, these fields are ignored. The assembler also places one in the R field which specifies that the vector parameter file to be executed is already loaded.

8-6. THE VECTOR PARAMETER FILE

The vector parameter file specifies a complete vector operation. When a vector parameter file is loaded into (or already exists in) the vector register file, an execute vector parameter file instruction (see Topic 8-3) will cause the specified vector operation to be executed.

Figure 8-1 illustrates the vector parameter file and describes its fields as they exist in the vector register file. An example of one of the general assembler statement sets that would produce a vector parameter file is also included.

Note: The command VCTRA is a CPU procedure designed to permit symbolic coding of the three vector words of the VPF.

Figure 8-2 illustrates the flow of execution of a typical vector parameter file wherein neither of the argument vectors is single-valued.

Note: As a convention, the vector addressed or contained in the second word of the vector parameter file will be called argument vector $\vec{A}$, the vector addressed or contained in the third word of the vector parameter file will be called argument vector $\vec{B}$, and the vector addressed in the fourth word of the vector parameter file will be called the resultant vector $\vec{C}$.

8-7. VECTOR OPERATION SPECIFICATION

The type of vector operation is specified by the assembler mnemonic in the first instruction of the vector parameter file.

The various operations available are described in Topics 8-26 through 8-51.

8-8. ARITHMETIC AND LOGICAL COMPARISON CONDITION SPECIFICATION

The ALCT field specifies the comparison condition to be used between elements of argument vectors and specifies when the vector operation is to terminate. Both the comparison condition and terminate condition are used in vector compare instructions. For vector peak picking instructions, only the terminate condition is

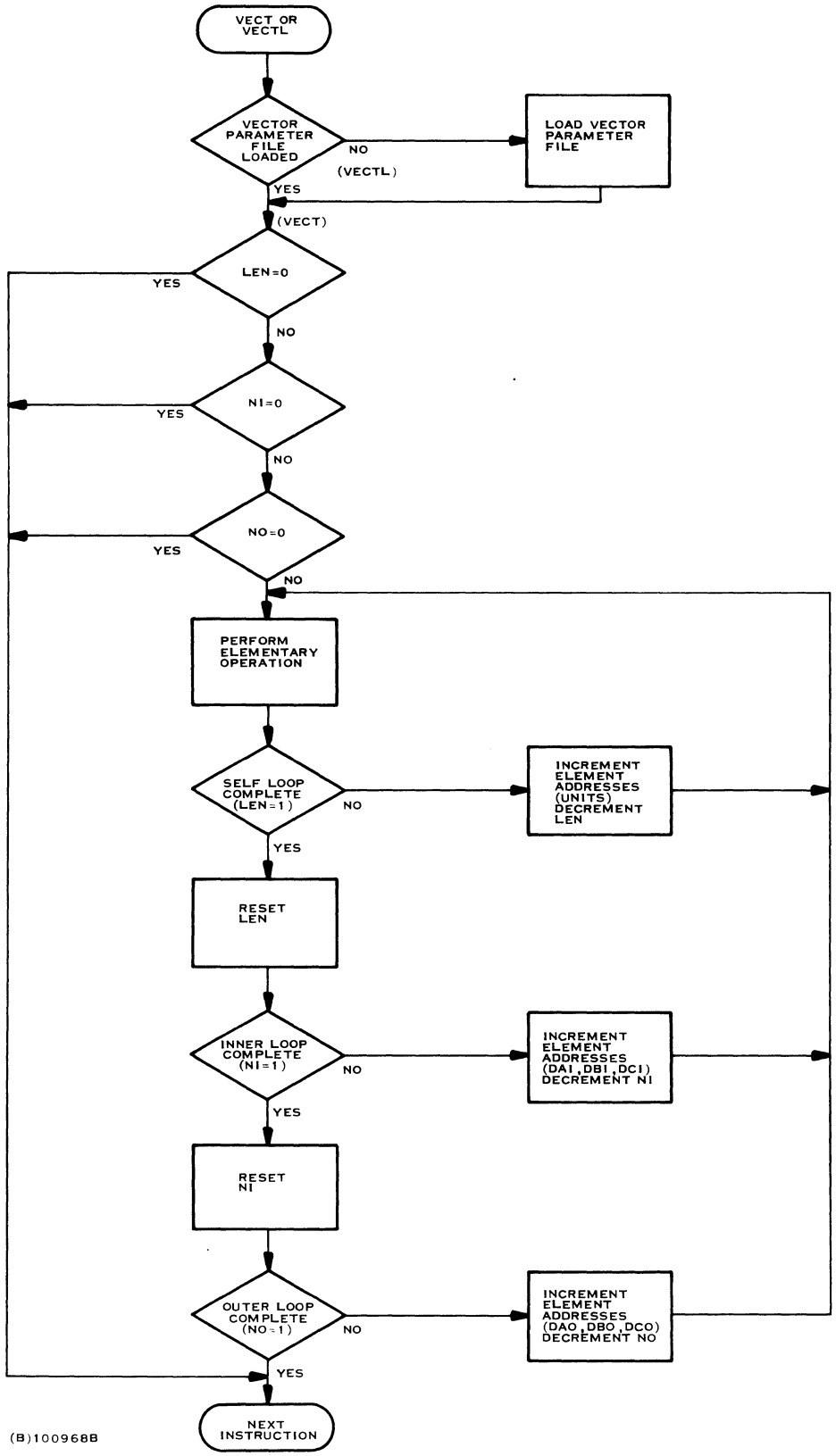


Figure 8-2. Flow of Execution of a Vector Parameter File

checked. The terminate condition specifies whether the operation is to stop at the first true condition or to record all true conditions found in all loops of the vector parameter file.

See Table 8-6 of Topic 8-38 for a complete specification of the ALCT field.

8-9. VECTOR LENGTH SPECIFICATION (SELF LOOP COUNT)

The LEN field specifies the number of times the self-loop operation of the vector parameter file is to be executed. A value of one for the LEN field effectively makes the self loop dimension a scalar operation. Given a self loop on a one-dimensional contiguous argument vector $\vec{A}$ with m elements, the value of the LEN field is m.

Range: The value of the LEN field must be within the range: $0 \leq LEN \leq 2^{16} - 1$. A zero value for LEN makes the vector parameter file a no operation; the inner and outer loop conditions will not be examined.

8-10. SINGLE-VALUED VECTOR AND WORD SIZE SPECIFICATION

The SV field specifies the attributes of the input vectors, i.e., directly addressed, immediate, or directly addressed single-valued.

For multiply and dot product operations with fixed point values as the argument vector elements, the SV field also specifies the element length of the elements of the resultant vector; for divide operations with fixed point values as the argument vector elements, the SV field specifies the length of the dividend elements of the input vector $\vec{A}$. All other operations and all floating point products, dot products, and dividends contain the element length specification as an integral part of the instruction; e.g., the instruction VAH specifies fixed point halfword addition with fixed point halfword results and ignores any word size specification in the SV field.

Table 8-1 contains the complete specifications of the SV field.

8-11. SINGLE-VALUED VECTORS

A single-valued vector is effectively a scalar operand for the self loop.

SV FIELD VALUE	VECTOR TYPE $\vec{A} \circ \vec{B} = \vec{C}$			LOOP USAGE
0 - 3 8 - 11	n	n	n	all loop increments active for all input vectors
4, 12	k	n	n	self loop increment inactive for single-valued input vector
5, 13	n	k	n	
6, 14	i	n	n	all loop increments inactive for immediate vector
7, 15	n	i	n	

n = a vector of any LEN

k = a single-valued vector

i = immediate

ELEMENT LENGTH OF RESULT

SV FIELD VALUE	DIVISOR OR MULTIPLICAND INPUT OF:	
	SINGLEWORD	HALFWORD
$0 \leq SV \leq 7$	doubleword	singleword
$8 \leq SV \leq 15$	singleword	halfword

Note: These length options apply only to products, dot products, and dividends in fixed point operations; floating point operations do not change precision.

Table 8-1. Specifications of the SV Field

For purposes of execution of a vector parameter file, a single-valued vector is stored as a single-element vector, i.e., this vector address is not incremented during the self loop execution, but may be incremented by either or both the inner and outer loop increments in their turns. During the execution of the self loop, only one value will be acquired from central memory for the single-valued vector. The SV field specifies which argument vector is single-valued (see Table 8-1).

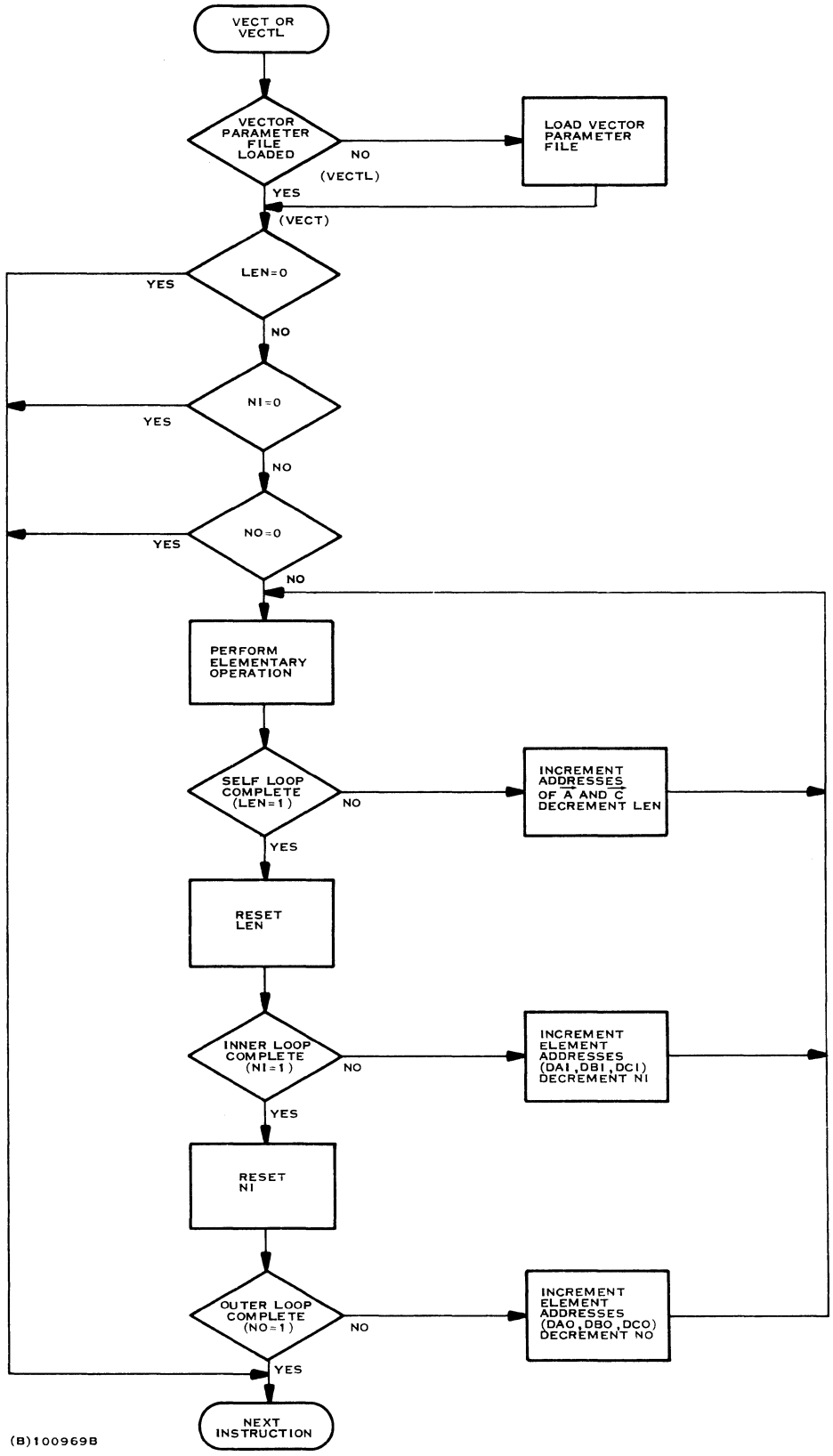


Figure 8-3. Flow of Execution with $\vec{B}$ Single-Valued

Example: Given $\vec{A} \circ \vec{B} = \vec{C}$,

$$\vec{A} = [a_1\ a_2\ \ldots\ a_m] \text{ and } \vec{B} = [B_1\ B_2\ \ldots\ B_r]$$

where each B_i is a single-valued vector with m elements.

The LEN value in the vector parameter file is m to encompass $\vec{A}$ in the self loop. The SV value is five or thirteen to denote $\vec{B}$ as single-valued within the self loop only. The inner loop count is r to encompass $\vec{B}$ in the inner loop. The inner loop increment values are -m + 1 for $\vec{A}$ to reposition the counter for the next self loop, and positive unity for B and C to position to the next column.

In performing an operation on $\vec{A}$ and $\vec{B}$, $\vec{B}$ is treated as a vector having one element b_i in any given self loop. However, on each successive inner loop i = 1, 2, . . . r. Thus, a different b_i is used only when the inner loop increment is used.

8-12. IMMEDIATE VECTORS

An immediate single-valued vector is a single-valued vector that has the additional restriction that none of the loop increments are active. The vector value resides in the vector parameter file itself; thus, the immediate vector is obtained from the vector parameter file for all loops and never from central memory. The argument vector that is to be immediate is specified by the SV field (see Table 8-1).

A halfword immediate vector is contained in the right half of the second or the third word, as specified. The value of such a vector is within the range: $-2^{15} \leq i \leq 2^{15} - 1$.

A singleword fixed point immediate vector occupies the entire vector parameter file word; i.e., the second or third word as specified. The value of such a vector is within the range: $-2^{31} < i < 2^{31} - 1$. If the HS field is used to denote a starting address in the right halfword, then the single-valued vector must be $\vec{A}$. A single-valued vector $\vec{B}$ would override the HS field.

A floating point immediate vector occupies the entire vector parameter file word; i.e., the second or third word as specified. This applies to both singleword

and doubleword operations; the second word of such a vector in a doubleword operation is treated as being filled with zeros.

GENERAL FORMS: The general forms of the assembler coded vector parameter files for halfword immediate vectors are as follows:

LABEL		COMMAND		OPERAND
[symbol]	␣	mnemonic	␣	alct, len, sv
[symbol]	␣	VCTRA	␣	a, xa or i
[symbol]	␣	VCTRA	␣	i, hs or b, xb, hs
[symbol]	␣	VCTRA	␣	c, xc, vi
[symbol]	␣	DATAH	␣	dai, dbi
[symbol]	␣	DATAH	␣	dci, ni
[symbol]	␣	DATAH	␣	dao, dbo
[symbol]	␣	DATAH	␣	deo, no

For singleword and doubleword immediate vectors the general form is shown below.

LABEL		COMMAND		OPERAND
[symbol]	␣	mnemonic	␣	alct, len, sv
[symbol]	␣	VCTRA or DATA	␣	a, xa or i
[symbol]	␣	DATA or VCTRA	␣	i or b, xb, hs
[symbol]	␣	VCTRA	␣	c, xc, vi
[symbol]	␣	DATAH	␣	dai, dbi
[symbol]	␣	DATAH	␣	dci, ni
[symbol]	␣	DATAH	␣	dao, dbo
[symbol]	␣	DATAH	␣	dco, no

Restrictions: Only one of the two argument vectors can be an immediate value.

A doubleword immediate vector cannot be fully specified; only the first word is created and the other (least significant) word is treated as zero.

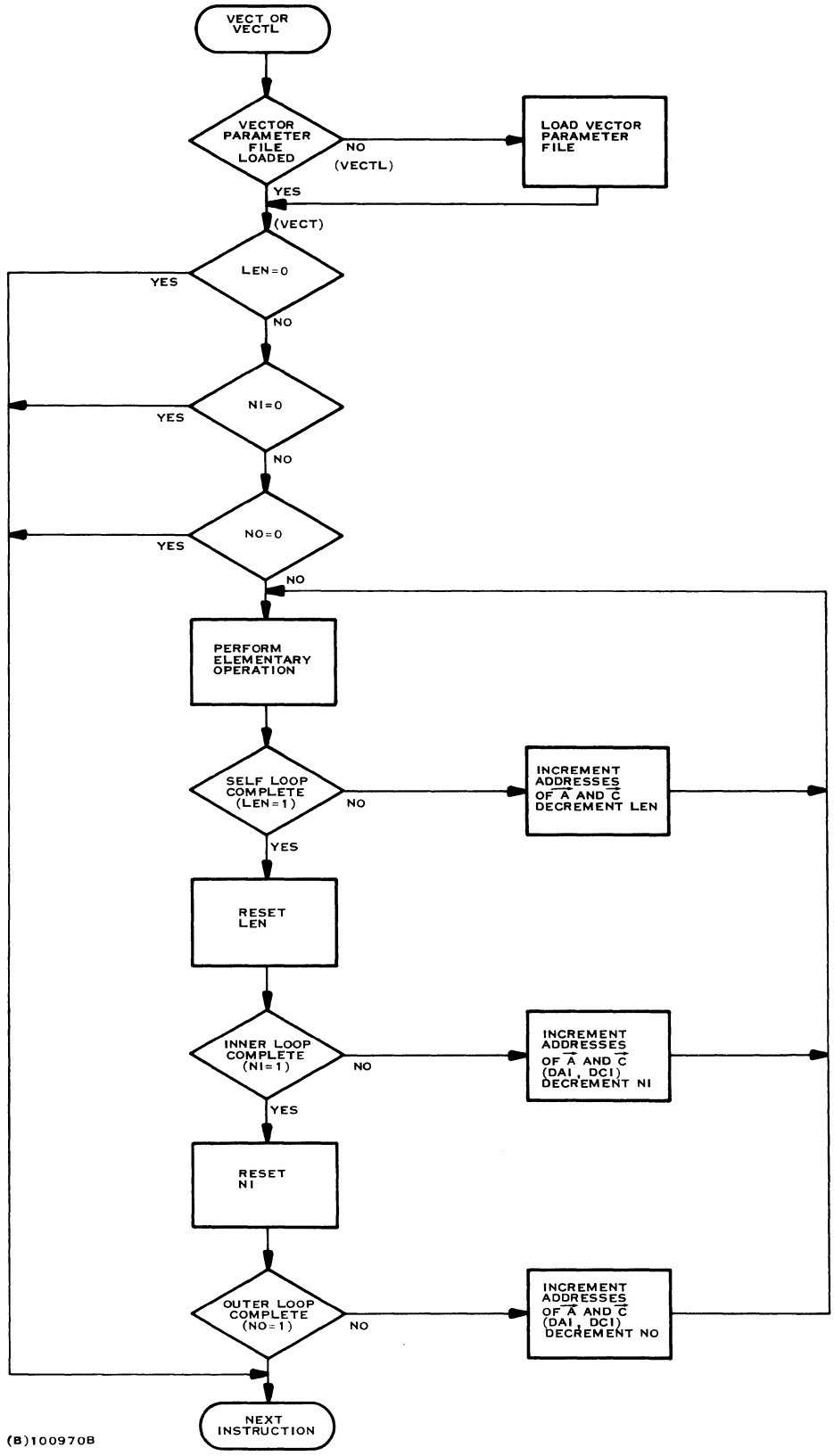


Figure 8-4. Flow of Execution with $\overrightarrow{B}$ Single-Valued Immediate

Immediate vector values cannot be modified by an index value.

Example: Given an operation on argument vectors:

$$\vec{A} \circ \vec{B} = \vec{C}$$

such that a_{mgt} $\vec{A}$ and $\vec{B}$ is an immediate vector then the vector parameter file is described as follows. The LEN field is m to denote the self loop. The SV field is seven or fifteen to specify that $\vec{B}$ is an immediate vector and no loops are active for $\vec{B}$. The inner loop count is q + 1, and the outer loop count is t. Assuming that all the elements are stored sequentially, both inner and outer loop increments for vectors $\vec{A}$ and $\vec{C}$ are positive unity.

Note: $\vec{A}$ might be a 3-array, three vectors, or one vector with (m) (q) (t) elements.

8-13. VECTOR ADDRESS DEVELOPMENT

The starting addresses of the argument vectors, $\vec{A}$ and $\vec{B}$, and the resultant vector, $\vec{C}$, and their index registers, if any, are specified in the second, third, and fourth words of the vector parameter file. Fields in the third and fourth words also specify left or right halfword index word sets and self loop address increment directions.

8-14. DIRECTLY ADDRESSED VECTORS

Directly addressed vectors are those whose elements are acquired from central memory and whose element starting addresses are developed from the address fields and index register fields of the vector parameter file.

The a, b, and c operands of the second, third, and fourth words, respectively, of the assembler coded vector parameter file are translated into the 24 bit address fields SAA, SAB, and SAC, respectively, of the object vector parameter file.

The xa, xb, and xc operands of the assembler coded vector parameter file are translated into the address index fields XA, XB, and XC, respectively, of the object vector parameter file. These three fields address registers in the index register file (viz, registers X1 through X7).

For any given vector $\vec{A}$, $\vec{B}$, or $\vec{C}$, the address developed is the sum of the 24 bit address and the contents of the index register if one is specified. The index displacement produced is halfword, singleword, or doubleword as appropriate to the instruction (see Topics 6-19 and 6-23 for index displacement development). Halfword index word sets can be started from an initial left halfword or an initial right halfword by specifications of the hs operand (see Topic 8-5).

Restrictions: No indirect addressing of vectors is permitted; thus, the most significant bits of the address index fields are ignored.

The address increments of the vectors in the self loop are ±1 as specified in the VI field. The specification of positive or negative incremental direction is through the VI field described in Topic 8-17.

The effective starting address of doubleword vectors must be in an even word boundary.

Note: During execution of a vector parameter file, the addresses are not incremented in the vector registers. The current element address is contained in the hardware unit only. The current element addresses within the hardware unit are cumulative for all loops - self, inner, and outer.

8-15. HALFWORD INDEX START SPECIFICATION

The HS field specifies in which halfword of a central memory word a vector(s) begins. Bits 1, 2, and 3 of the HS field refer to $\vec{A}$, $\vec{B}$, and $\vec{C}$. A bit which is on indicates an initial right halfword address. A bit which is off indicates an initial left halfword address.

For a halfword vector instruction, the contents of index registers are used to compute the starting addresses of the vectors. The indexed halfword starting addresses of vectors $\vec{A}$, $\vec{B}$, and $\vec{C}$ are developed as follows:

$$IA = 2 \cdot SAA + (XA) + HSA$$

$$IB = 2 \cdot SAB + (XB) + HSB$$

$$IC = 2 \cdot SAC + (XC) + HSC$$

If the sum of the last two elements of the preceding equations is even, the vector begins in the left halfword. An odd sum specifies the right halfword. If an index register is not specified, that element is treated as 0. The final element of the equation is zero (left half) or one (right half).

The most significant bit of the HS field is used to delete indices from the output array for the eight Vector Compare (VC) instructions and the four Vector Peak Pick (VPP) instructions.

If this bit is off, the actual element index values are used for the output array. If this bit is on, the index values for the elements which satisfied the operation are suppressed.

8-16. SELF LOOP INCREMENT DIRECTION

The VI field specifies which vector(s), $\overrightarrow{A}$, $\overrightarrow{B}$, or $\overrightarrow{C}$, if any are arranged with their element addresses in decreasing order. Bits 1, 2, and 3 of the VI field correspond to vectors $\overrightarrow{A}$, $\overrightarrow{B}$, and $\overrightarrow{C}$, respectively (bit 1 - $\overrightarrow{A}$, etc.). An off bit indicates positive unity, and an on bit indicates negative unity.

The most significant bit of the VI field refers to the count of the number of elements which satisfy the vector operation in process. If this bit is off, the output count is the total count for each self loop. If this bit is on, the output count is the total count for the vector operation. The MSB of the VI field is also used to compute the dot product involving non-contiguous input vectors. When this bit is on the dot product can be computed using non-contiguous elements in memory. The summation of the products is stored upon completion of each inner loop.

Restrictions: The self loop element addresses are always incremented or decremented by unity where units are the word size specified by the instruction; i.e., halfword, singleword, or doubleword units. There is no provision in the self loop for variable increments.

HS FIELD VALUE	VECTOR INDEX BEGINS FROM HALFWORD		
	$\vec{A}$	$\vec{B}$	$\vec{C}$
0	L	L	L
1	L	L	R
2	L	R	L
3	L	R	R
4	R	L	L
5	R	L	R
6	R	R	L
7	R	R	R

Where L indicates an initial left halfword and R indicates an initial right halfword for the index word set.

Table 8-2. Specifications of the HS Field Values

VI FIELD VALUE	SELF LOOP ELEMENT ADDRESS INCREMENT		
	$\vec{A}$	$\vec{B}$	$\vec{C}$
0	+1	+1	+1
1	+1	+1	-1
2	+1	-1	+1
3	+1	-1	-1
4	-1	+1	+1
5	-1	+1	-1
6	-1	-1	+1
7	-1	-1	-1

Table 8-3. Specifications of the VI Field Values

8-17. INNER LOOP SPECIFICATION

All the inner loop specification fields are halfword fields and can be conveniently coded with DATAH directives.

The DAI, DBI, DCI, and NI fields in the vector parameter file specify the inner loop conditions.

8-18. Inner Loop Vector Address Increments

The DAI, DBI, and DCI fields specify the inner loop address increments of the vectors A, B, and C, respectively. They each are added to the current values of the respective vector element addresses at the completion of each but the last self loop within the inner loop. This addition occurs in the hardware; the addresses in the vector registers are not altered.

Range: The increment values may be either positive or negative fixed point halfword values. The values of the DAI, DBI, and DCI fields must be within the range: $-2^{15} \leq$ DAI, DBI, DCI $\leq 2^{15} - 1$.

Note: A value of zero in an increment field will cause the first element of the next self loop to have the same address as the last element of the currently completed self loop. The addresses are not incremented after the last elementary operation of the self loop.

8-19. Inner Loop Count

The NI field in the vector parameter file specifies the number of times the self loop is to be executed.

Range: The inner loop count is treated as a positive value only, and must be within the range: $0 \leq ni \leq 2^{16} - 1$.

Note: A value of zero as the inner loop count causes a NO OPERATION. The outer loop specifications will not be examined, and the self loop will not be executed.

A value of one as the inner loop count causes the outer loop count to be checked for a value greater than one. The vector increments for the inner loop are not to be made.

8-20. OUTER LOOP SPECIFICATION

The DAO, DBO, DCO, and NO fields in the vector parameter file specify the outer loop conditions. All the outer loop specification fields are halfword fields and can be conveniently coded with DATAH directives.

8-21. Outer loop Vector Address Increments

The DAO, DBO, and DCO fields specify the outer loop address increments of the vectors A, B, and C, respectively. They each are added to the current values of the respective vector element addresses at the completion of each but the last inner loop within the outer loop. This addition occurs in the hardware; the addresses in the vector registers are not altered.

Range: The increment values may be either positive or negative fixed point halfword values, and must be within the range: $-2^{15} \leq$ DAO, DBO, DCO $\leq 2^{15} - 1$.

Note: A value of zero in an increment field causes the first element of the next self loop to have the same address as the last element of the currently completed self loop. The addresses are not incremented by either the self loop increment or the inner loop increment after the last elementary operation of the self loop in any given loop phase.

8-22. Outer Loop Count

The NO field in the vector parameter file specifies the number of times the inner loop is to be executed.

Range: The outer loop count is treated as a positive value only, and must be within the range: $0 \leq$ NO $\leq 2^{16} - 1$.

Note: A value of zero or one as the outer loop count causes the vector to be a NO OPERATION. The self loop and inner loop will not be executed.

8-23. PROGRAM INTERRUPTS

The elementary operations between any two elements of the argument vectors are subject to the same program interrupts as their scalar counterparts.

The maskable interrupts (the arithmetic exception conditions: fixed point overflow, floating point exponent overflow, floating point exponent underflow, and divide check) when masked off permit the vector operation to run to completion; when masked on, the vector operation terminates on occurrence of the exception condition. See Topics 6-56 and 6-62 for a discussion of the program status doubleword and the arithmetic exception condition mask.

The unmaskable interrupts, such as specification errors for doubleword elements addressed on odd-even word boundaries, will cause termination of the vector operation whenever they are encountered.

8-24. VECTOR HAZARD

Because of the buffering scheme used to implement the vector operations, a condition known as the vector hazard condition exists. Essentially, it involves those cases in which it is desired to have the elements of the resultant vector stored in the same locations as the elements of one of the argument vectors and immediately reusing the new data in the same vector operation.

The Vector Hazard Rule may be stated:

> A Vector hazard condition exists when the addresses of the current octet of elements or the addresses of the next two octets of elements of argument vectors $\vec{A}$ or $\vec{B}$ are the same as the addresses of the current octet of elements or the addresses of the past three octets of elements of the resultant vector $\vec{C}$.

When the vector hazard condition occurs, the old data in the elements of $\vec{A}$ or $\vec{B}$ are used rather than the new (processed) data represented by the elements of the resultant vector $\vec{C}$.

8-25. VECTOR ARITHMETIC INSTRUCTIONS

A vector arithmetic instruction causes the specified arithmetic operation to be performed on two correlated elements (one each from two argument vectors), and causes the result to be stored as the element of a resultant vector. The order in

which the $\vec{A}$ and $\vec{B}$ elements are operated upon and the order in which the resulting values are stored are dependent upon the respective self, inner, and outer loop increments. The other parameters in the vector parameter field cause identical operations to be performed on specified elements of the vectors until the self, inner, and outer loops are satisfied.

Note: In the following instruction descriptions, the following notation is used:

$\vec{A}$ is a vector with elements $a_1, a_2, \ldots, a_n$;

The elements appear sequentially in memory although not necessarily contiguously.

The subscripts appear in sequence in the examples for convenience only. This is not required in the actual vector operation.

Fortran instructions are used to describe more easily the vector operations. The resulting vector parameter file is also included.

8-26. VECTOR ADD INSTRUCTIONS

A vector add instruction, with argument vectors $\vec{A}$ and $\vec{B}$, produces a self loop resultant vector $\vec{C}$ where $\vec{C} = \vec{A} + \vec{B}$.

The alignment of the various elements for addition in the inner and outer loops may be displaced by the inner and outer loop increments.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VA	vector add, fixed point singleword elements
VAH	vector add, fixed point halfword elements
VAF	vector add, floating point singleword elements
VAFD	vector add, floating point doubleword elements

Example:

DIMENSION C(10,20), A(10,20), B(10,20)
$\vec{C} = \vec{A} + \vec{B}$

VECTOR PARAMETER FILE

VAF	0,10
VCTRA	A
VCTRA	B
VCTRA	C
DATAH	1,1
DATAH	1,20
DATAH	0,0
DATAH	0,1

8-27. VECTOR ADD MAGNITUDE INSTRUCTIONS

A vector add magnitude instruction, with argument vectors $\vec{A}$ and $\vec{B}$ produces a self loop resultant vector $\vec{C}$, where $\vec{C} = \vec{A} + |\vec{B}|$.

The alignment of the various elements for addition in the inner and outer loops may be displaced by the inner and outer loop increments.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VAM	vector add magnitude, fixed point singleword elements
VAMH	vector add magnitude, fixed point halfword elements
VAMF	vector add magnitude, floating point singleword elements
VAMFD	vector add magnitude, floating point doubleword elements

Example:

DIMENSION C(10,20) A(10,20), B(10,20)
$\vec{C} = \vec{A} + \vec{B}$

VECTOR PARAMETER FILE

VAF	0,10
VCTRA	A
VCTRA	B
VCTRA	C

Example: (continued)

VECTOR PARAMETER FILE

```
DATAH   1,1
DATAH   1,20
DATAH   0,0
DATAH   0,1
```

8-28. VECTOR SUBTRACT INSTRUCTIONS

A vector subtract instruction, with argument vectors $\vec{A}$ and $\vec{B}$, produces a self loop resultant vector $\vec{C}$ where $\vec{C} = \vec{A} - \vec{B}$.

The alignment of the various elements for subtraction in the inner and outer loops may be displaced by the inner and outer loop increments.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VS	vector subtract, fixed point singleword elements
VSH	vector subtract, fixed point halfword elements
VSF	vector subtract, floating point singleword elements
VSFD	vector subtract, floating point doubleword elements

Example:

DIMENSION C(10,20), A(10,20), B(10,20)

$\vec{C} = \vec{A} - \vec{B}$

VECTOR PARAMETER FILE

```
VSF     0,10
VCTRA   A
VCTRA   B
VCTRA   C
DATAH   1,1
DATAH   1,20
DATAH   0,0
DATAH   0,1
```

8-29. VECTOR SUBTRACT MAGNITUDE INSTRUCTIONS

A vector subtract magnitude instruction with argument vectors $\vec{A}$ and $\vec{B}$, produces a self-loop resultant vector $\vec{C}$ where $\vec{C} = \vec{A} - |\vec{B}|$.

The alignment of the various elements for subtraction in the inner and outer loops may be displaced by the inner and outer loop increments.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VSM	vector subtract magnitude, fixed point singleword elements
VSMH	vector subtract magnitude, fixed point halfword elements
VSMF	vector subtract magnitude, floating point singleword elements
VSMFD	vector subtract magnitude, floating point doubleword elements

Example:

DIMENSION C(10,20), A(10,20), B(10,20)
$\vec{C} = \vec{A} - |\vec{B}|$

VECTOR PARAMETER FILE

```
VSMF    0,10
VCTRA   A
VCTRA   B
VCTRA   C
DATAH   1,1
DATAH   1,20
DATAH   0,0
DATAH   0,1
```

8-30. VECTOR MULTIPLY INSTRUCTIONS

A vector multiply instruction, with argument vectors $\vec{A}$ and $\vec{B}$, produces a self loop resultant vector $\vec{C}$ where $\vec{C} = \vec{A} \times \vec{B}$.

The alignment of the various elements for multiplication in the inner and outer loops may be displaced by the inner and outer loop increments.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VM	vector multiply, fixed point singleword argument elements
VMH	vector multiply, fixed point halfword argument elements
VMF	vector multiply, floating point singleword elements
VMFD	vector multiply, floating point doubleword elements

Limitations: The instructions VM and VMH do not specify the element length of the result elements. See Topic 8-10 for the method of specification.

Note: The instruction VM causes 64 bit products to be generated in the arithmetic unit. If the MSB of the SV field is off, indexing on the resultant vector $\vec{C}$ is treated as a doubleword. If singleword results are specified (MSB of SV field is on), the least significant 32 bits of that product are stored in the resultant vector $\vec{C}$. Overflow is detected.

Note: The instruction VMH causes 32 bit products to be generated in the arithmetic unit. If the MSB of the SV field is off, indexing on the resultant vector $\vec{C}$ is treated as a fullword. If halfword results are specified (MSB of SV field is on), the least significant 16 bits of that product are stored in the resultant vector $\vec{C}$. Overflow is detected.

Example:

DIMENSION C(10,20), A(10,20), B(10,20)

$\vec{C} = \vec{A} * \vec{B}$

VECTOR PARAMETER FILE

VMF	0,10
VCTRA	B
VCTRA	A
VCTRA	C
DATAH	1,1
DATAH	1,20
DATAH	0,0
DATAH	0,1

8-31. VECTOR DOT PRODUCT INSTRUCTIONS

The vector dot product operates on two vectors and produces a scalar output. The operation sums the products of the elements of $\vec{A}$ and $\vec{B}$ addressed in the self loop and stores the scalar result in the address designated by $\vec{C}$. Following the execution of the self loop the $\vec{A}$, $\vec{B}$ and $\vec{C}$ addresses are modified by the specified inner and outer loop increments. Since the self loop is used in the dot product, the elements of $\vec{A}$ and $\vec{B}$ must be contiguous. The self loop is never applied to $\vec{C}$ as $\vec{C}$ is single-valued.

Algebraically, the dot product is:

$$C_i = \sum_{j=1}^{m} a_j \cdot b_j \qquad C_i \in \vec{C},\ i = 1,n$$

where n and m are integers denoting the number of elements in the vectors.

The sparse dot product is the dot product of non-contiguous elements of $\vec{A}$ and $\vec{B}$. In order to obtain the sum of non-contiguous elements, the MSB of the VI field is set. This allows accumulation of the products over the whole vectors rather than storing the results at the completion of each self loop. This produces a scalar result and can never result in a vector.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VDP	vector dot product, fixed point singleword argument elements
VDPH	vector dot product, fixed point halfword argument elements
VDPF	vector dot product, floating point singleword elements
VDPFD	vector dot product, floating point doubleword elements

Limitations: The instructions VDP and VDPH do not specify the element length of the result elements. See Topic 8-10 for the method of specification.

Note: The instruction VDP causes 64 bit dot products to be generated in the arithmetic unit. If the MSB of the SV field is off, indexing on the resultant vector $\vec{C}$ is treated as a doubleword. If singleword results are specified (MSB of SV field is on), the least significant 32 bits of that dot product are stored in vector $\vec{C}$. Overflow is detected.

Note: The instruction VDPH causes 32 bit dot products to be generated in the arithmetic unit. If the MSB of the SV field is off, indexing on the resultant vector is treated as a fullword. If halfword results are specified (MSB of SV field is on), the least significant 16 bits of that dot product are stored in the resultant vector. Overflow is detected.

Example 1: The following illustrates a dot product:

```
      DIMENSION X(10), Y(10)
      DOTPR = 0
      DO  10  I = 1,10
10    DOTPR = DOTPR + X(I) * Y(I)
```

VECTOR PARAMETER FILE

```
VDPF    0,10,8
VCTRA   Y
VCTRA   X
VCTRA   DOTPR
DATAH   0,0
DATAH   0,1
DATAH   0,0
DATAH   0,1
```

Example 2: The following illustrates a spare dot product:

```
      DIMENSION X(10), Y(10)
      DOTPR = 0
      DO 30 I = 1,10,2
30    DOTPR = DOTPR + X(I) * Y(I)
```

VECTOR PARAMETER FILE

```
VDPF    0,1,13
VCTRA   Y
VCTRA   X
VCTRA   DOTPR,,8
DATAH   2,2
DATAH   0,5
DATAH   0,0
DATAH   0,1
```

MATRIX-MATRIX MULTIPLY

```
      DIMENSION D(20,20)A(10,20)B(10,20)
      DO 20 J = 1,20
      DO 20 I = 1,20
      D(I,J) = 0
      DO 20, K = 1,10
20    D(I,J) = D(I,J) + A(K,J) * B(K,I)
```

VECTOR PARAMETER FILE

```
VDPF    0,10,8
VCTRA   B
VCTRA   A
VCTRA   D
DATAH   1,-9
DATAH   1,20
DATAH   -199,1
DATAH   1,20
```

8-32. VECTOR DIVIDE INSTRUCTIONS

A vector divide instruction, with argument vectors $\vec{A}$ and $\vec{B}$, produces a self loop resultant vector $\vec{C}$ where $\vec{C} = \vec{A}/\vec{B}$.

The alignment of the various elements for division in the inner and outer loops may be displaced by the inner and outer loop increments.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VD	vector divide, fixed point singleword argument elements
VDH	vector divide, fixed point halfword argument elements
VDF	vector divide, floating point singleword elements
VDFD	vector divide, floating point doubleword elements

Limitations: The instructions VD and VDH do not specify the element length of the dividend elements (vector $\vec{A}$). See Topic 8-10 for the method of specifications.

Note: If the relative values of dividend and divisor are such that the quotient cannot be expressed in 32 bits, overflow occurs and the output result is unpredictable.

Note: The instruction VDH causes 16 bit quotients to be generated from 16 bit divisors (vector $\vec{B}$ elements) and either 32 bit dividends (MSB of SV field is on) or 16 bit dividends (MSB of SV field is off). Indexing of the resultant vector is by fullwords if the MSB is on and by halfwords if the MSB is off. If the relative values of dividend and divisor are such that the quotient cannot be expressed in 16 bits, overflow occurs and the output result is unpredictable.

Example:

DIMENSION C(10, 20), A(10, 20), B(10, 20)

$\vec{C} = \vec{A}/\vec{B}$

VECTOR PARAMETER FILE

```
VDF     0, 10, 8
VCTRA   A
VCTRA   B
VCTRA   C
DATAH   1, 1
DATAH   1, 20
DATAH   0, 0
DATAH   0, 1
```

8-33. VECTOR LOGICAL INSTRUCTIONS

A vector logical instruction causes the specified logical operation to be performed on two correlated elements (one from each argument vectors), and causes the result to be stored as an element of a resultant vector. The other parameters in the vector parameter file cause identical operations to be performed on sequential elements of the vectors until the self, inner, and outer loops are satisfied.

In terms of the self loop only, given argument vectors $\vec{A}$ and $\vec{B}$ and a resultant vector $\vec{C}$, where $\vec{C} = \vec{A} \circledast \vec{B}$.

Any given elementary operation is defined the same as its scalar counterpart.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VAND	vector AND, singleword elements
VANDD	vector AND, doubleword elements
VOR	vector OR, singleword elements
VORD	vector OR, doubleword elements
VXOR	vector exclusive OR, singleword elements
VXORD	vector exclusive OR, doubleword elements
VEQC	vector equivalence, singleword elements
VEQCD	vector equivalence, doubleword elements

Example:

```
DIMENSION LC(10), LA(10), LB(10)
LC = AND (LA, LB)
```

VECTOR PARAMETER FILE

```
VAND    0, 10, 8
VCTRA   LB
VCTRA   LA
VCTRA   LC
DATAH   0, 0
DATAH   0, 1
DATAH   0, 0
DATAH   0, 1
```

LC = OR (LA, LB)	VOR	0, 10, 8
	VCTRA	LB
	VCTRA	LA
	VCTRA	LC
	DATAH	0, 0
	DATAH	0, 1
	DATAH	0, 0
	DATAH	0, 1
LC = XOR (LA, LB)	VXOR	0, 10, 8
	VCTRA	LB
	VCTRA	LA
	VCTRA	LC
	DATAH	0, 0
	DATAH	0, 1
	DATAH	0, 0
	DATAH	0, 1
LC = EQU (LA, LB)	VEQC	0, 10, 8
	VCTRA	LB
	VCTRA	LA
	VCTRA	LC
	DATAH	0, 0
	DATAH	0, 1
	DATAH	0, 0
	DATAH	0, 1

8-34. VECTOR SHIFT INSTRUCTIONS

A vector shift instruction causes the elements of the argument vector $\vec{A}$ to be shifted left or right the number of bit positions specified in $\vec{B}$ and causes the results to be stored as elements of a resultant vector. Vector $\vec{B}$ can be an immediate value or a vector composed of halfwords containing the shift counts.

In terms of the self loop only, given argument vector $\vec{A}$ and a shift count sc, where $-2^6 \leq sc \leq 2^6 - 1$, $\vec{C} = \vec{A}$ where the elements of $\vec{A}$ are shifted sc positions to produce $\vec{C}$.

A negative shift count causes a right shift and a positive shift count causes a left shift.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VSA	vector arithmetic shift, fixed point singleword elements
VSAH	vector arithmetic shift, fixed point halfword elements
VSAD	vector arithmetic shift, fixed point doubleword elements
VSL	vector logical shift, singleword elements
VSLH	vector logical shift, halfword elements
VSLD	vector logical shift, doubleword elements
VSC	vector circular shift, singleword elements
VSCH	vector circular shift, halfword elements
VSCD	vector circular shift, doubleword elements

Example:

```
DIMENSION LC(10), LA(10), LB(10)
LC = ASHF (LA, -3)
```

VECTOR PARAMETER FILE

```
VSA     0, 10, 7
VCTRA   LA
VCTRA   #FFFFFFFD
VCTRA   LC
DATAH   0, 0
DATAH   0, 1
DATAH   0, 0
DATAH   0, 1
```

	VECTOR PARAMETER FILE	
LC = LSHF (LB, 20)	VSL	0, 10, 7
	VCTRA	LB
	VCTRA	#00000014
	VCTRA	LC
	DATAH	0, 0
	DATAH	0, 1
	DATAH	0, 0
	DATAH	0, 1
LC = SLHF (LB, 10)	VSL	0, 10, 7
	VCTRA	LB
	VCTRA	#0000000A
	VCTRA	LC
	DATAH	0, 0
	DATAH	0, 1
	DATAH	0, 0
	DATAH	0, 1

8-35. VECTOR MERGE INSTRUCTIONS

A vector merge instruction causes elements to be acquired alternately from the two argument vectors $\vec{A}$ and $\vec{B}$, and causes them to be merged into a resultant vector $\vec{C}$.

In terms of the self loop only and given arguments vectors $\vec{A}$ and $\vec{B}$, with n elements, respectively, $\vec{C} = (a_1, b_1, a_2, b_2, \ldots, a_n, b_n)$

Note: Use of a single-valued vector as either argument vector would produce alternate identical elements in the resultant vector; e.g., given $\vec{A}$ and single-valued $\vec{B}$ then $\vec{C} = (a_1, b_1, a_2, b_1, \ldots, a_m b_1)$.

Note: The lengths of the input vectors must be the same if not a single valued or immediate valued vector.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VMG	vector merge, singleword elements
VMGH	vector merge, halfword elements
VMGD	vector merge, doubleword elements

Example:

$\vec{C} = \vec{A} \circ \vec{B}$

$\vec{A} = (0, 2, 4, 6, 8)$

$\vec{B} = (1, 3, 5, 7, 9)$

Then,

$\vec{C} = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$

VECTOR PARAMETER FILE

```
VMG     0,5,0
VCTRA   A
VCTRA   B
VCTRA   C
DATAH   0,0
DATAH   0,1
DATAH   0,0
DATAH   0,1
```

8-36. VECTOR ORDER INSTRUCTIONS

A vector order instruction performs an arithmetic comparison of the elements of vectors $\vec{A}$ and $\vec{B}$, such that the smaller element, whether from $\vec{A}$ or $\vec{B}$, is the next element to be stored in the output vector $\vec{C}$. If two equal values are compared from $\vec{A}$ and $\vec{B}$, the value from $\vec{A}$ is stored in $\vec{C}$ and the value in $\vec{B}$ is retained for comparison with the next element of $\vec{A}$.

The input vectors $\vec{A}$ and $\vec{B}$ must be of a special form, $(e_1, e_2, \ldots, e_n, \ell)$ where e_i are the elements of the vector and ℓ is the boundary limit. The boundary limit is the largest positive number possible in the designated form (see following table). The boundary limits at the end of the argument vectors are necessary to prevent overrunning the end of the first vector exhausted. For vectors $\vec{A}$ with m elements and $\vec{B}$ with n elements, vector $\vec{C}$ may contain as many as m + n - 1 elements. Thus, LEN $\leq$ m + n - 1. If A or B is immediate single valued, LEN may be any length desired.

Note: The boundary limits for $\vec{A}$ and $\vec{B}$ are equal, thus only the boundary limit for $\vec{A}$ is stored in $\vec{C}$ which then contains one element less than the total of elements in $\vec{A}$ and $\vec{B}$ (i.e., m + n - 1).

ASSEMBLER MNEMONIC	OPERATION SPECIFIED	BOUNDARY LIMIT
VO	vector order, fixed point singleword elements	7FFF FFFF
VOH	vector order, fixed point halfword elements	7FFF
VOF	vector order, floating point singleword elements	7FFF FFFF
VOFD	vector order, floating point doubleword elements	FFFF FFFF

Restriction: The resultant vector must not be written over either argument vector; the restart condition for a vector order instruction begins from the initial argument elements.

Floating point vectors $\vec{A}$ and $\vec{B}$ must be normalized prior to use in a vector order instruction.

Limitations: Vector order instructions operate on only the self loop; values for the inner and outer loops are ignored.

If the HS field is to be used and an immediate single-valued vector is used, the immediate vector must be vector $\vec{A}$.

Example:

$\vec{C} = \vec{A} \circ \vec{B}$

$A = (0, 3, 7, 2, 5, \ell)$

$B = (2, 9, 1, 4, \ell)$

Then,

$C = (0, 2, 3, 7, 2, 5, 9, 1, 4, \ell)$

VECTOR PARAMETER FILE

```
VO      0,10,0
VCTRA   A
VCTRA   B
VCTRA   C
DATAH   0,0
DATAH   0,1
DATAH   0,0
DATAH   0,1
```

Example: $\vec{B}$ is Immediate Single-Valued Vector

$\vec{C} = \vec{A} \circ \vec{B}$

$\vec{A} = (0, 1, 2, 3, 4, 5, 4, 7, 1, \ell)$

$\vec{B} = (5)$

Then,

$\vec{C} = (0, 1, 2, 3, 4, 5, 4, 5, 5)$

VECTOR PARAMETER FILE

VO	0, 9, 15
VCTRA	A
VCTRA	5
VCTRA	C
DATAH	0, 0
DATAH	0, 1
DATAH	0, 0
DATAH	0, 1

Example: $\vec{A}$ is Immediate Single-Valued Vector

$\vec{C} = \vec{A} \circ \vec{B}$

$\vec{A} = (5)$

$\vec{B} = (0, 1, 2, 3, 4, 5, 7, 1, \ell)$

$\vec{C} = (0, 1, 2, 3, 4, 5, 5, 5, 5)$

If $\vec{A}$ is the single-valued vector, the element value (5) in $\vec{B}$ is continuously compared to the single-valued vector.

8-37. VECTOR COMPARE INSTRUCTIONS

A vector compare instruction causes the specified comparison to be made between corresponding elements of two argument vectors, and causes the index numbers of the true comparisons to be stored as halfword elements of a resultant vector, beginning in the second halfword.

At the completion of each self loop the count of the number of true conditions is stored in the first halfword of the resultant vector.

Various information is obtained using the most significant bits of the ALCT, VI, and HS fields.

The high order bit of ALCT field, if on, indicates that the vector operation is to terminate on the first true compare.

The high order bit of the VI field, if on, indicates that item counts will not be stored for the index vector produced during each self loop. A total item count (representing the sum of the item counts which would have been stored if the MSB had been on) is still recorded at the beginning of the $\vec{C}$ vector immediately prior to termination of the vector operation.

The high order bit of the HS field, if on, indicates that no index values are to be stored into the output vector.

High order bit of:			Results in:
VI	HS	ALCT	
0	0	0	Store count and indices for each self loop.
0	0	1	Terminate on first true compare-store count and index.
0	1	0	Store count only for each self loop.
0	1	1	Terminate on first true compare-store count only.
1	0	0	Store indices only for each self loop, total count stored at termination.
1	0	1	Effectively same as 001.
1	1	0	Store total count only at termination.
1	1	1	Effectively same as 011.

Table 8-4. Specifications of the VI, HS, and ALCT Fields

Low Order 3 Bits of:	Comparison True if:	
ALCT	Arithmetic	Logical
0	Never true	Never true
1	Aj = Bj	All bits zero
2	Aj > Bj	All bits ones
3	Aj ≥ Bj	Not mixed zeros and ones
4	Aj < Bj	Mixed zeros and ones
5	Aj ≤ Bj	Not all ones
6	Aj ≠ Bj	Not all zeros
7	Always true	Always true

Table 8-5. Specifications of the ALCT Values

Restrictions: Floating point argument vector elements must be normalized prior to the vector comparison.

Limitations: The elements of the resultant vector $\vec{C}$ are limited to halfword positive integers; i.e., the values of the indices of the argument vectors are limited to the range: $0 \leq j \leq 2^{16} - 1$.

If the HS field is to be used and one of the argument vectors is to be an immediate single-valued vector, the immediate vector must be vector $\vec{A}$; i.e., the second word of the vector parameter file.

Note: A zero will be stored as the first and only element, c_1, of vector $\vec{C}$ for any self loop in which a true comparison is not found.

8-38. VECTOR COMPARE ARITHMETIC INSTRUCTIONS

A vector compare arithmetic instruction compares elements of two argument vectors, $\vec{A}$ and $\vec{B}$, for $aj > bj$, $aj = bj$, or $aj < bj$. The result of this comparison is tested for true against the ALCT field, and the indices of the elements satisfying the comparison are stored as elements of a resultant vector $\vec{C}$. See Table 8-5 for the ALCT field.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VC	vector compare arithmetic, fixed point singleword elements
VCH	vector compare arithmetic, fixed point halfword elements
VCF	vector compare arithmetic, floating point singleword elements
VCFD	vector compare arithmetic, floating point doubleword elements

8-39. VECTOR COMPARE LOGICAL INSTRUCTIONS

A vector compare logical instruction compares the result of a logical operation on elements of two argument vectors, $\vec{A}$ and $\vec{B}$, to be checked for all bits on, all bits off, or bits mixed on and off. The result is compared to the ALCT field, and the indices of the elements which satisfy the comparison are stored as elements of a resultant vector $\vec{C}$. See Table 8-4 and Table 8-5.

Restrictions: There are no vector compare logical halfword instructions.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VCAND	vector logical comparison using AND, singleword
VCANDD	vector logical comparison using AND, doubleword
VCOR	vector logical comparison using OR, singleword
VCORD	vector logical comparison using OR, doubleword

8-40. VECTOR PEAK PICKING INSTRUCTIONS

A vector peak picking instruction records the sign changes between elements of a vector by storing the respective index in the resultant vector. When all peaks and valleys (sign changes) have been found and their indexes stored, the count is stored as the first element of the resultant vector.

The algorithm for the vector peak picking instruction is as follows:

$$y_i = a_{i-1} - a_i \qquad i = 1, 2, \ldots, n$$

$$y_{i+1} = a_i = a_{i+1}$$

If the sign of y_i is different than the sign of y_{i+1}, then the index value, i, is stored. If the signs are the same, no value is stored. When the value of y_{i+1} is zero, y_{i+1} is considered to retain the sign of the last non-zero value of y_i.

The MSB of the ALCT field determines whether the operation is to continue for all elements of the input vector or to terminate after the first change is detected. If the bit is off, all elements are tested; if the bit is on, the operation terminates after the first valley or peak is detected.

The MSB of the HS and VI fields are used as described previously. Thus, the self loop count and index values may be suppressed.

Restrictions: When a floating point argument vector is specified, the elements of the argument vector must be normalized prior to the peak picking operation.

Limitations: The values of the indices of the argument vector are limited to the range: $0 \leq i \leq 2^{16} - 1$ (positive halfword integers).

Inflection points and the leading edges of plateaus are not detected.

Irrelevant Fields: The $\vec{B}$ address and $\vec{B}$ address index fields are ignored.

Note: A zero is stored as the first and only element, c_1, of vector $\vec{C}$ for any self loop in which a peak is not found.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VPP	find vector peaks, fixed point singleword elements
VPPH	find vector peaks, fixed point halfword elements
VPPF	find vector peaks, floating point singleword elements
VPPFD	find vector peaks, floating point doubleword elements

8-41. VECTOR SEARCH INSTRUCTIONS

A vector search instruction causes every element of the argument vector to be compared to every other element of the argument vector, and causes the index number of the largest element or the smallest element, as specified, to be stored as a single-valued halfword resultant vector for each self loop.

Restrictions: The elements of the argument vector must be normalized prior to the search operation when a floating point search is specified.

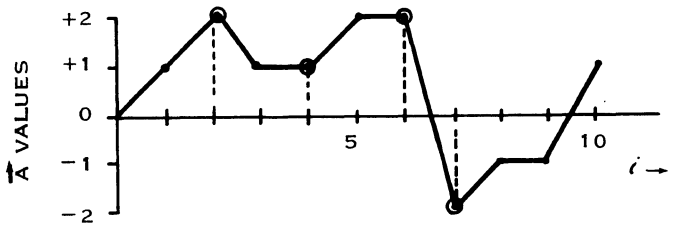
The HS field is deactivated if the SV field specifies $\vec{B}$ as an immediate vector.

Limitations: The values of the indices of the argument vector are limited to the range: $0 \leq i \leq 2^{16} - 1$ (positive halfword integers).

Irrelevant Fields: The ALCT, SAB, and SB fields are ignored.

Note: If there are two or more equal elements which are the largest (or smallest) elements, the index number stored is that of the first encountered in the search.

Example: The following illustration shows the vector peak picking instruction.



◉ INDICATES PEAK PICK ITEM POINT

THE FIRST AND LAST ITEMS OF $\vec{A}$ CAN NEVER BE PEAK PICK ITEM POINTS

$\vec{A}$ (singlewords)	$\vec{C}$ (halfwords)
0	4
1	2
2	4
1	6
1	7
2	
2	
-2	
-1	
-1	
1	

8-42. VECTOR SEARCH FOR LARGEST ELEMENT INSTRUCTIONS

A vector search for the largest element instruction causes the index value of the algebraically largest element of the argument vector to be stored as a single-valued halfword resultant vector in the self loop.

An inner loop count may be applied, thus resulting in a vector $\vec{C}$ containing elements each being the index of the algebraically largest element in each self loop.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VL	find largest element, fixed point singleword elements
VLH	find largest element, fixed point halfword elements
VLF	find largest element, floating point singleword elements
VLFD	find largest element, floating point doubleword elements

Note: If there are two or more equal elements that are larger than all others, the index of the first encountered is stored as the self loop element of $\vec{C}$.

Example: The following is an example of a vector search for largest element instruction.

Example 1. VL

$\vec{A}$ (singlewords)	$\vec{C}$ (halfwords)
-33	6
0	
5	
-10	
7	
5	
10	
-45	

8-43. VECTOR SEARCH FOR LARGEST MAGNITUDE INSTRUCTIONS

The vector search for largest magnitude instruction searches the argument vector for an element of the largest absolute value. The index of this element is stored as a single-valued halfword resultant vector in the self loop.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VLM	find element of largest absolute value, fixed point singleword elements
VLMH	find element of largest absolute value, fixed point halfword elements
VLMF	find element of largest absolute value, floating point singleword elements
VLMFD	find element of largest absolute value, floating point doubleword elements

Note: If there are two or more equal elements whose absolute values are larger than all others, the index of the first encountered is stored as the self loop element of $\vec{C}$.

Example: The following assembler code and illustration shows a simple search for largest magnitude.

Example 2. VLM

$\vec{A}$ (singlewords)	$\vec{C}$ (halfwords)
-33	7
0	
5	
-10	
7	
5	
10	
-45	

8-44. VECTOR SEARCH FOR SMALLEST ELEMENT INSTRUCTIONS

A vector search for the smallest element instruction causes the index value of the algebraically smallest element of the argument vector to be stored as a single-valued halfword resultant vector in the self loop.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VSS	find smallest element, fixed point singleword elements
VSSH	find smallest element, fixed point halfword elements
VSSF	find smallest element, floating point singleword elements
VSSFD	find smallest element, floating point doubleword elements

Note: If there are two or more elements that are smaller than all others, the index of the first encountered is stored as the self loop element of $\vec{C}$.

Example: The following assembler code and illustration shows a simple search for smallest elements.

Example 3. VSS

$\vec{A}$ (singlewords)	$\vec{C}$ (halfwords)
-33	7
0	
5	
-10	
7	
5	
10	
-45	

8-45. VECTOR SEARCH FOR SMALLEST MAGNITUDE INSTRUCTIONS

A vector search for the smallest magnitude instruction causes the index value of the smallest absolute value element of the argument vector to be stored as a single-valued halfword resultant vector.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VSSM	find element of smallest absolute value, fixed point single-word elements
VSSMH	find element of smallest absolute value, fixed point half-word elements
VSSMF	find element of smallest absolute value, floating point single-word elements
VSSMFD	find element of smallest absolute value, floating point double-word elements

Note: If there are two or more elements whose absolute values are smaller than all others, the index of the first encountered is stored as the self loop element of $\vec{C}$.

Example: The following assembler code and illustration shows a search for smallest magnitude.

Example 4. VSM

$\vec{A}$ (Singlewords)	$\vec{C}$ (halfwords)
-33	1
0	
5	
-10	
7	
5	
10	
-45	

8-46. VECTOR CONVERSION INSTRUCTIONS

A vector conversion instruction causes the fixed or floating point elements of the argument vector $\vec{A}$ to be converted to floating or fixed point elements of vector $\vec{C}$ according to the scale factor(s) of vector $\vec{B}$.

See Topics 7-145 and 7-146 for the algorithms for the elementary conversion operations.

8-47. CONVERT FLOATING POINT ELEMENTS TO FIXED POINT ELEMENTS

A convert floating point elements to fixed point elements instruction causes the floating point elements of vector $\vec{A}$ to be converted to fixed point values according to the halfword scale factors (fixed point values) of vector $\vec{B}$, and causes the fixed point values to be stored as the elements of vector $\vec{C}$.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VFLFX	convert floating point singleword elements to fixed point singleword elements
VFLFH	convert floating point singleword elements to fixed point halfword elements
VFDFX	convert floating point doubleword elements to fixed point singleword elements

Restrictions: The scale factors are the right halfwords of the singleword elements of vector $\vec{B}$; i.e., vector $\vec{B}$ loops and indexes are by singleword increments.

Limitations: The scale factors (elements of vector $\vec{B}$) are restricted to the range: $-2^{15} \leq sf \leq 2^{15} - 1$.

Note: These instructions all differ from their scalar counterparts in that these scale factors are in right halfwords of singlewords, whereas scalar scale factors are normally in the left halfword with halfword addressing (i.e., left halfword index word sets).

8-48. CONVERT FIXED POINT ELEMENTS TO FLOATING POINT ELEMENTS

A convert fixed point elements to floating point elements instruction causes the fixed point elements of vector $\vec{A}$ to be converted to floating point values according to the halfword scale factors (fixed point values) of vector $\vec{B}$, and causes the floating point values to be stored as the elements of vector $\vec{C}$.

ASSEMBLER MNEMONIC	OPERATION SPECIFIED
VFXFL	convert fixed point singleword elements to floating point singleword elements
VFXFD	convert fixed point singleword elements to floating point doubleword elements
VFHFL	convert fixed point halfword elements to floating point singleword elements
VFHFD	convert fixed point halfword elements to floating point doubleword elements

Restrictions: For the fixed point singleword arguments (instructions VFXFL and VFXFD), the scale factors are the right halfwords of the singleword elements of vector $\vec{B}$; i.e., vector $\vec{B}$ loops and indexes are by singleword increments. The instructions operating on fixed point halfword arguments have halfword elements in vector $\vec{B}$, also; i.e., the HS field is effective for both vectors $\vec{A}$ and $\vec{B}$.

Limitations: The scale factors are restricted to the range: $-2^{15} \leq sf \leq 2^{15} - 1$.

Note: These instructions all differ from their scalar counterparts in that these scale factors are in right halfwords of singlewords, whereas scalar scale factors are normal in the left halfword with halfword addressing (i.e., left halfword index word sets).

8-49. VECTOR NORMALIZE INSTRUCTIONS

A vector normalize instruction shifts each of the fixed point elements of the argument vector left until the two most significant bits of each element differ (01 or 10), counts the number of bits positions each is shifted, and causes the normalized fixed point values and the negatives (two's complement) of their respective shift counts to be stored as the elements of the resultant vector. An element having a value of zero is considered to be normalized and to have a scale factor of -32.

ASSEMBLER MNEMONICS	OPERATION SPECIFIED
VNFX	normalize fixed point singleword elements
VNFH	normalize fixed point halfword elements

NORMALIZE SINGLEWORDS (VNEX): In terms of the self loop only, given argument vector $\vec{A}$, a resultant vector $\vec{C}$ is produced where each element of $\vec{C}$ is the corresponding element of $\vec{A}$. Each element in $\vec{C}$ is a doubleword element with the normalized element of $\vec{A}$ in the first word of the doubleword and with the two's complement scale factor in the (right) halfword of the second word. The unused halfword is filled with zeros.

NORMALIZE HALFWORDS (VNFH): In terms of the self loop only, given argument vector $\vec{A}$, a resultant vector $\vec{C}$ is produced where each element of $\vec{C}$ is the corresponding element of $\vec{A}$. The resulting element is a singleword element with the normalized element of $\vec{A}$ in the left halfword and with the two's complement scale factor in the right halfword.

Limitations: The scale factors are restricted to the range: $-2^{15} \leq sf \leq 0$.

Irrelevant Fields: The $\vec{B}$ address and $\vec{B}$ address index fields in the third word of the vector parameter file are ignored.

Note: Zeros are entered at the right as the value is shifted left.

8-50 VECTOR MAP INSTRUCTIONS

A vector map instruction replaces elements of one vector with the corresponding elements of another vector. The elements that are mapped are determined by a third index vector.

ASSEMBLER MNEMONICS	OPERATION SPECIFIED
VMAP	Vector Map Singlewords
VMAPH	Vector Map Halfwords
VMAPD	Vector Map Doublewords

VECTOR MAP ON EQUAL, SV_{msb} OPTION BIT EQUAL TO ZERO

A Vector-Map-on-Equal instruction accepts as inputs a contiguous list of indices from vector $\vec{A}$ and a set of source mapping elements from vector $\vec{B}$. The elements of vector $\vec{B}$ that are mapped are those elements for which the index location in $\vec{B}$ corresponds to the index value given by the elements of vector $\vec{A}$. Elements from source mapping vector $\vec{B}$ replace previously existing elements in a central memory region defined as the $\vec{C}$ output vector. Elements of the $\vec{C}$ output vector that are replaced are those elements for which the index location in $\vec{C}$ corresponds to the index value given by the elements of vector $\vec{A}$. Elements of vector $\vec{B}$ do not replace elements of vector $\vec{C}$ for those index locations in $\vec{C}$ that are not represented in the index list given by vector $\vec{A}$.

This instruction differs from the Vector Replace instruction in the manner in which elements of vector $\vec{B}$ are used. Vector Replace uses consecutive elements of vector $\vec{B}$, whereas Vector Map uses only those elements of Vector $\vec{B}$ that are mapped by the specification of vector $\vec{A}$.

Self Loop Programming Notes

1) The length specification of the self-loop (L-field) for a Vector-Map-on-Equal instruction should be set equal to the number of elements of a self-loop of the $\vec{C}$ vector. Or, if the $\vec{B}$ vector is the greater in length, then set the L-field equal to the number of elements of vector $\vec{B}$.

2) It is possible to shorten the vector operation and still obtain the same result vector $\vec{C}$ by setting the self-loop length equal to one plus the value of the last index in vector $\vec{A}$.

3) If the vector length is specified according to 1 above, then an index boundary limit equal to the largest positive number (7FFF hex) must be placed in the data location following the last index value of vector $\vec{A}$.

   If the vector length is specified according to 2 above, then the index boundary limit is not necessary.

4) Each index value given by vector $\vec{A}$ is a positive, fixed-point halfword. Vector $\vec{A}$ should be a contiguous list of monotone increasing halfwords.

5) An index value of zero maps the first element of vector $\vec{B}$ into the first element of vector $\vec{C}$.

Example 1: A singleword Vector-Map-on-Equal ($SV_{msb}=0$) instruction using a self-loop of length 8.

Singleword Index Vector $\vec{A}$	Singleword Vector $\vec{B}$	Singleword Vector $\vec{C}$ Before Replacement	Singleword Vector $\vec{C}$ After Replacement
2, 3	-14	16	16
5, 6	-70	82	82
7FFF, -	-25	27	-25
	-34	36	-34
	-69	71	71
	-30	32	-30
	- 6	8	- 6
	-12	14	14

VECTOR MAP ON NOT EQUAL, SV_{msb} OPTION BIT EQUAL TO ONE

A Vector-Map-on-Not-Equal instruction accepts as inputs a contiguous list of indices from vector $\vec{A}$ and a set of source mapping elements from vector $\vec{B}$. The

elements of vector $\vec{B}$ that are mapped are those elements for which the index location in $\vec{B}$ is not represented in the index list given by vector $\vec{A}$. Elements from source mapping vector $\vec{B}$ replace previously existing elements in a central memory region defined as the $\vec{C}$ output vector. Elements of the $\vec{C}$ output vector that are replaced are those elements for which the index location in $\vec{C}$ is not represented in the index list given by vector $\vec{A}$. Elements of vector $\vec{B}$ do not replace elements of vector $\vec{C}$ for those index locations that correspond to the index value given by the elements of vector $\vec{A}$.

Self Loop Programming Notes

1) The length specification of the self-loop (L-field) for a Vector-Map-on-Not-Equal instruction should be set equal to the number of elements of a self-loop of the $\vec{C}$ vector. Or, if the $\vec{B}$ vector is the greater in length, then set the L-field equal to the number of elements of vector $\vec{B}$.

2) An index boundary limit equal to the largest positive number (7FFF hex) must be placed in the data location following the last index value of vector $\vec{A}$.

3) Each index value given by vector $\vec{A}$ is a positive, fixed-point halfword. Vector $\vec{A}$ should be a contiguous list of monotone increasing halfwords.

4) An index list beginning with a value of "one" maps the first elements of vector $\vec{B}$ into $\vec{C}$ but not the second element (element C_0 is replaced with B_0 but not C_1 by B_1).

Example 2: A singleword Vector-Map-on-Not-Equal (SV_{msb}=1) instruction using a self-loop of length 8.

Singleword Index Vector $\vec{A}$	Singleword Vector $\vec{B}$	Singleword Vector $\vec{C}$ Before Replacement	Singleword Vector $\vec{C}$ After Replacement
2, 3	-14	16	-14
5, 6	-70	82	-70
7FFF, -	-25	27	27

Singleword Index Vector $\vec{A}$	Singleword Vector $\vec{B}$	Singleword Vector $\vec{C}$ Before Replacement	Singleword Vector $\vec{C}$ After Replacement
	-34	36	36
	-69	71	-69
	-30	32	32
	- 6	8	8
	-12	14	-12

8-51 VECTOR SELECT BOOLEAN INSTRUCTIONS

The Vector Select Boolean instructions generate an output vector from elements of an input vector. The elements of the input vector are mapped into the output vector depending upon the logical state ("1" or "0") of bits in a control vector.

ASSEMBLER MNEMONICS	OPERATION SPECIFIED
VSELB	Vector Select Singleword Boolean
VSELHB	Vector Select Halfword Boolean
VSELDB	Vector Select Doubleword Boolean

VECTOR SELECT ON ONE, SV_{msb} OPTION BIT EQUAL TO ZERO

A Vector-Select-on-One instruction generates an output vector $\vec{C}$ composed of elements from vector $\vec{B}$. The elements selected from vector $\vec{B}$ are those for which the location in vector $\vec{B}$ corresponds to the location on nonzero elements of vector $\vec{A}$. Selected elements are stored into contiguous locations of vector $\vec{C}$.

Example 1: A singleword Vector-Select-on-One (SV_{msb}=0) instruction using a self-loop of length 8.

Halfword Boolean Vector $\vec{A}$	Singleword Vector $\vec{B}$	Singleword Selected Vector $\vec{C}$
0, 0	+16	-54
1, 1	+82	-75
0, 1	-54	-64
1, 0	-75	-15
	+71	
	-64	
	-15	
	+14	

VECTOR SELECT ON ZERO, SV_{msb} OPTION BIT EQUAL TO ONE

A Vector-Select-on-Zero instruction generates an output vector $\vec{C}$ composed of elements from vector $\vec{B}$. The elements selected from vector $\vec{B}$ are those for which the location in vector $\vec{B}$ corresponds to the location of zero elements of vector $\vec{A}$. Selected elements are stored into contiguous locations of vector $\vec{C}$.

Example 2: A singleword Vector-Select-on-Zero ($SV_{msb}=1$) instruction using a self-loop of length 8.

Halfword Boolean Vector $\vec{A}$	Singleword Vector $\vec{B}$	Singleword Selected Vector $\vec{C}$
0, 0	+16	+16
1, 1	+82	+82
0, 1	-54	+71
	-75	+14
	+71	
	-64	
	-15	
	+14	

Vector Select Programming Notes for Self Loops

1) The length specification of the self-loop (L-field) for a Vector-Select-Boolean instruction is set equal to the number of elements of vector $\vec{A}$ or $\vec{B}$.

2) Each element of vector $\vec{A}$ is a halfword that assumes one of two Boolean values. "Zero" is assumed if the value is zero, and "one" is assumed if the value is nonzero.

8-52 VECTOR REPLACE BOOLEAN INSTRUCTIONS

The Vector Replace Boolean instructions replace elements of one vector with corresponding elements of another vector. The replacement is enabled by the logic state ("1" or "0") of elements in a third control vector.

ASSEMBLER MNEMONICS	OPERATION SPECIFIED
VREPB	Vector Replace Singleword Boolean
VREPHB	Vector Replace Halfword Boolean
VREPDB	Vector Replace Doubleword Boolean

VECTOR REPLACE ON ONE, SV_{msb} OPTION BIT EQUAL TO ZERO

A Vector-Replace-on-One instruction accepts as inputs a continuous list of replacement elements from vector $\vec{B}$ and a continuous list of Boolean elements from vector $\vec{A}$. Elements from vector $\vec{B}$ replace previously existing elements in a central memory region defined as the $\vec{C}$ output vector. Elements of the $\vec{C}$ output vector that are replaced with elements of vector $\vec{B}$ are those elements for which the location in the $\vec{C}$ output vector corresponds to the location of nonzero elements of vector $\vec{A}$. Elements of the $\vec{C}$ output vector remain unchanged in those locations for which the corresponding location in vector $\vec{A}$ is zero.

Example 1: A singleword Vector-Replace-on-One (SV_{msb}=0) instruction using a self-loop of length 8.

Halfword Boolean Vector $\vec{A}$	Singleword Vector $\vec{B}$	Singleword Vector $\vec{C}$ Before Replacement	Singleword Vector $\vec{C}$ After Replacement
0, 0	-54	16	16
1, 1	-72	82	82
0, 1	-64	27	-54
1, 0	-15	36	-72
		71	71
		32	-64
		8	-15
		14	14

VECTOR REPLACE ON ZERO, SV_{msb} OPTION BIT EQUAL TO ONE

A Vector-Replace-on-Zero instruction accepts as inputs a continuous list of replacement elements from vector $\vec{B}$ and a continuous list of Boolean elements from vector $\vec{A}$. Elements from vector $\vec{B}$ replace previously existing elements in a central memory region defined as the $\vec{C}$ output vector. Elements of the $\vec{C}$ output vector that are replaced with elements of vector $\vec{B}$ are those elements for which the location in the $\vec{C}$ output vector corresponds to the location of zero elements of vector $\vec{A}$. Elements of the $\vec{C}$ output vector remain unchanged in those locations for which the corresponding location in vector $\vec{A}$ is nonzero.

Example 2: A singleword Vector Replace on Zero (SV_{msb}=1) instruction using a self-loop length of 8.

Halfword Boolean Vector $\vec{A}$	Singleword Vector $\vec{B}$	Singleword Vector $\vec{C}$ Before Replacement	Singleword Vector $\vec{C}$ After Replacement
0, 0	-54	16	-54
1, 1	-72	82	-72
0, 1	-64	27	27
1, 0	-15	36	36

Halfword Boolean Vector $\vec{A}$	Singleword Vector $\vec{B}$	Singleword Vector $\vec{C}$ Before Replacement	Singleword Vector $\vec{C}$ After Replacement
		71	-64
		32	32
		8	8
		14	-15

Programming Notes for Self Loops

1) The length specification of the self-loop (L-field) for a Vector-Replace-Boolean instruction is set equal to the number of elements of vector $\vec{A}$ or $\vec{C}$.

2) Each element of vector $\vec{A}$ is a halfword that assumes one of two Boolean values. "Zero" is assumed if the value is zero, and "one" is assumed if the value is nonzero.

8-53 VECTOR MAP BOOLEAN INSTRUCTIONS

The Vector Map Boolean instructions map the elements of a continuing vector into element locations of an output vector. The logic state ("1" or "0") of elements in a third control vector determine which elements will be mapped.

ASSEMBLER MNEMONICS	OPERATION SPECIFIED
VMAPB	Vector Map Singleword Boolean
VMAPHB	Vector Map Halfword Boolean
VMAPDB	Vector Map Doubleword Boolean

VECTOR MAP ON ONE, SV_{msb} OPTION BIT EQUAL TO ZERO

A Vector-Map-on-One instruction accepts as inputs a continuous list of Boolean elements from vector $\vec{A}$ and a set of source mapping elements from vector $\vec{B}$. The elements of vector $\vec{B}$ that are mapped are those elements for which

the location in $\vec{B}$ corresponds to the location of nonzero elements of vector $\vec{A}$. Elements from source mapping vector $\vec{B}$ replace previously existing elements in a central memory region defined as the $\vec{C}$ output vector.

Elements of the $\vec{C}$ output vector that are replaced are those elements for which the location in $\vec{C}$ corresponds to the location of nonzero elements of vector $\vec{A}$. Elements of the $\vec{C}$ output vector remain unchanged in those locations for which the corresponding location in vector $\vec{A}$ is zero.

Example 1: A singleword Vector-Map-on-One ($SV_{msb}=0$) instruction using a self-loop of length 8.

Halfword Boolean Vector $\vec{A}$	Singleword Vector $\vec{B}$	Singleword Vector $\vec{C}$ Before Mapping	Singleword Vector $\vec{C}$ After Mapping
0, 0	-54	16	16
1, 1	-72	82	82
0, 1	-64	27	-64
1, 0	-15	36	-15
	-29	71	71
	- 5	32	- 5
	-47	8	-47
	- 2	14	14

VECTOR MAP ON ZERO, SV_{msb} OPTION BIT EQUAL TO ONE

A Vector-Map-on-Zero instruction accepts as inputs a continuous list of Boolean elements from vector $\vec{A}$ and a set of source mapping elements from vector $\vec{B}$. The elements of vector $\vec{B}$ that are mapped are those elements for which the location in $\vec{B}$ corresponds to the location of zero elements of vector $\vec{A}$. Elements from source mapping vector $\vec{B}$ replaced previously existing elements in a central memory region defined as the $\vec{C}$ output vector.

Elements of the $\vec{C}$ output vector that are replaced are those elements for which the location in $\vec{C}$ corresponds to the location of zero elements of vector $\vec{A}$.

Elements of the $\vec{C}$ output vector remain unchanged in those locations for which the corresponding location in vector $\vec{A}$ is nonzero.

Example 2: A singleword Vector-Map-on-Zero ($SV_{msb}=1$) instruction using a self-loop of length 8.

Halfword Boolean Vector $\vec{A}$	Singleword Vector $\vec{B}$	Singleword Vector $\vec{C}$ Before Mapping	Singleword Vector $\vec{C}$ After Mapping
0, 0	-54	16	-54
1, 1	-72	82	-72
0, 1	-64	27	27
1, 0	-15	36	36
	-29	71	-29
	- 5	32	32
	-47	8	8
	- 2	14	- 2

Programming Notes for Self Loops

1) The length specification of the self-loop (L-field) for a Vector-Map-Boolean instruction is set equal to the number of elements of vector $\vec{A}$. Vectors $\vec{B}$ and $\vec{C}$ should be of this same length.

2) Each element of vector $\vec{A}$ is a halfword that assumes one of two Boolean values. "Zero" is assumed if the value is zero, and "one" is assumed if the value is nonzero.

8-54 VECTOR MAXIMUM/MINIMUM INSTRUCTIONS

These instructions compare the elements of two input vectors and form an output vector from the results of the comparison.

ASSEMBLER MNEMONICS	OPERATION SPECIFIED
VMAX	Vector Max/Min Fixed-Point Singleword
VMAXH	Vector Max/Min Fixed-Point Halfword
VMAXF	Vector Max/Min Floating-Point Singleword
VMAXD	Vector Max/Min Floating-Point Doubleword

VECTOR MAXIMUM, SV_{msb} OPTION BIT EQUAL TO ZERO

A Vector-Maximum instruction forms an output vector $\vec{C}$ composed of the larger of the elements from either vector $\vec{A}$ or vector $\vec{B}$. That is, element c_i assumes the larger arithmetic value of the elements a_i or b_i.

$$c_i = MAX(a_i, b_i)$$

Example 1: A Fixed-Point, Singleword-Vector-Maximum instruction with a self-loop of length 8.

Vector $\vec{A}$	Vector $\vec{B}$	Vector $\vec{C}$
40	72	72
75	20	75
-11	45	45
56	56	56
32	- 9	32
16	64	64
97	28	97
21	20	21

VECTOR MINIMUM, SV_{msb} OPTION BIT EQUAL TO ONE

Vector Minimum, SV_{msb} option bit = 1

A Vector-Minimum instruction forms an output vector $\vec{C}$ composed of the smaller of the elements from either vector $\vec{A}$ or vector $\vec{B}$. That is, element c_i assumes the smaller arithmetic value of the elements a_i or b_i.

$$c_i = MIN(a_i, b_i)$$

Example 2: A Fixed-Point, Singleword-Vector-Minimum instruction with a self-loop length of 8.

Vector $\vec{A}$	Vector $\vec{B}$	Vector $\vec{C}$
40	72	40
75	20	20
-11	45	-11
56	56	56
32	- 9	- 9
16	64	16
97	28	28
21	20	20

8-55 VECTOR COMPARE BOOLEAN INSTRUCTIONS

All ALCT options of the arithmetic-compare instructions can be used to generate Boolean vector outputs. A Boolean vector is a vector containing elements having a value of either "zero" or "one." For the arithmetic compare instructions, a "one" is placed in the $\vec{C}$ output vector in each halfword location corresponding to the location of true comparisons of elements in the input vectors $\vec{A}$ and $\vec{B}$. A "zero" is placed in the halfword location corresponding to the location of false comparisons of the $\vec{A}$ and $\vec{B}$ input vectors.

No item count is stored for any of the Boolean vector compare instructions.

ASSEMBLER MNEMONICS	OPERATION SPECIFIED
VCB	Vector Compare Fixed-Point Singleword Boolean
VCHB	Vector Compare Fixed-Point Halfword Boolean
VCFB	Vector Compare Floating-Point Singleword Boolean
VCFDB	Vector Compare Floating-Point Doubleword Boolean

Example 1: A Vector Compare Fixed-Point Singleword Boolean instruction with ALCT comparison option set to search for "greater than or equal to".

Singleword Vector $\vec{A}$	Singleword Vector $\vec{B}$	Halfword Vector $\vec{C}$
40	72	0
75	20	1
-11	45	0
56	56	1
32	- 9	1
16	64	0
97	28	1
21	20	1

8-56 VECTOR COMPARE AND/OR BOOLEAN INSTRUCTIONS

All ALCT options of the logical compare instructions can be used to generate Boolean vector outputs. A Boolean vector is a vector containing elements having a value of either "zero" or "one". For the logical compare instructions, a "one" is placed in the $\vec{C}$ output vector in each halfword location corresponding to the location of true comparisons of elements in the input vectors $\vec{A}$ and $\vec{B}$. A "zero" is placed in the halfword location corresponding to the location of false comparisons of the $\vec{A}$ and $\vec{B}$ input vectors.

No item count is stored for any of the Boolean vector compare instructions.

ASSEMBLER MNEMONICS	OPERATION SPECIFIED
VCAB	Vector Compare AND, Singleword Boolean
VCADB	Vector Compare AND, Doubleword Boolean
VCORB	Vector Compare OR, Singleword Boolean
VCORDB	Vector Compare OR, Doubleword Boolean

Example 1: A Vector Compare OR Singleword Boolean instruction with ALCT comparison option set to search for "mixed zeros and ones".

Singleword Vector $\vec{A}$	Singleword Vector $\vec{B}$	Halfword Vector $\vec{C}$
005A	0000	01
0000	0000	00
0048	0024	01
5A5A	A5A5	01

8-57 VECTOR SELECT

A vector select instruction generates an output vector $\vec{C}$ composed of elements from vector $\vec{B}$. The elements selected from vector $\vec{B}$ are those for which the index location in vector $\vec{B}$ corresponds to the index value given by the elements of vector $\vec{A}$.

ASSEMBLER MNEMONICS	OPERATION SPECIFIED
VSEL	Select singlewords from vector $\vec{B}$
VSELH	Select halfwords from vector $\vec{B}$
VSELD	Select doublewords from vector $\vec{B}$

Programming Notes:

1) Input vectors $\vec{A}$ and $\vec{B}$ are read from contiguous memory and the output is stored into contiguous memory for a given self loop.

2A) The length specification of the self loop (L-field) for a vector select instruction is normally set equal to the number of elements of vector $\vec{B}$.

2B) It is possible to shorten the vector operation and still obtain the same result vector $\vec{C}$ by setting the self loop length equal to one plus the value of the last index in vector $\vec{A}$.

3A) If the vector length is specified according to 2A above, then an index boundary limit equal to the largest positive number ($7FFF_{16}$) must be placed in the data location following the last index value of vector $\vec{A}$.

3B) If the vector length is specified according to 2B above, then the index boundary limit is not necessary.

4) Each index value given by vector $\vec{A}$ is a positive fixed point halfword. Vector A should be a contiguous list of monotone increasing halfwords.

5) An index value of zero selects the first element of vector $\vec{B}$.

6) If inner or outer loops are employed, then a dummy value should be placed at the end of each self loop vector $\vec{B}$ and the index of this dummy value should be placed at the end of each self loop index vector $\vec{A}$. Each successive index list must be in contiguous memory, i.e., DBI and DBØ must be equal to one. Vector $\vec{B}$ may use delta increments not equal to one for inner or outer loops if desired. However, the resultant vector $\vec{C}$ of selected elements should use delta increments, DCI and DCØ equal to one if the number of selected elements varies from self loop to self loop. Delta increments for vector $\vec{C}$ are added to the address of the last element selected for each self loop.

Example: A singleword select instruction using one self loop of length 8.

Halfword Vector $\vec{A}$	Singleword Index Vector $\vec{B}$	Singleword Selected Vector $\vec{C}$
2, 3	+16	-54
5, 6	+72	-75
7FFF, -	-54	-64
	-75	-15
	+71	
	-64	
	-15	
	+14	

8-58 VECTOR REPLACE

A vector replace instruction accepts as inputs a contiguous list of replacement elements from vector $\vec{B}$ and a contiguous list of indices from vector $\vec{A}$.

Elements from vector $\vec{B}$ replace previously existing elements in a central memory region defined as the $\vec{C}$ output array. Elements of the $\vec{C}$ output array that are replaced with elements of vector $\vec{B}$ are those elements for which the index location in the $\vec{C}$ output array corresponds to the index value given by the elements of vector $\vec{A}$.

ASSEMBLER MNEMONICS	OPERATION SPECIFIED
VREP	Replace singlewords in vector $\vec{C}$
VREPH	Replace halfwords in vector $\vec{C}$
VREPD	Replace doublewords in vector $\vec{C}$

Programming Notes:

1) The length specification of the self loop (L-field) for a vector replace instruction should be set equal to the number of replacement elements in vector $\vec{B}$. This value is also equal to the number of indices of vector $\vec{A}$.

2) Each index value given by vector $\vec{A}$ is a positive fixed point halfword. Vector $\vec{A}$ should be a contiguous list of monotone increasing halfwords.

3) An index value of zero selects the first element of vector $\vec{B}$.

4) If inner or outer loops are employed, then it becomes a requirement that each self loop be of the same length. In general, the length of the data replacement vectors throughout all of the inner and outer loops are not the same length. In order to obtain meaningful results using inner and outer loops, a dummy region of memory must be established at the end of the $\vec{C}$ data output array for each self loop. The size of the dummy region for each self loop $\vec{C}$ output array is equal to one plus the difference between the sizes of the maximum and minimum data replacement vectors as found by searching the data replacement lists throughout all inner and outer loops.

For the case of a self loop passing over the maximum data replacement vector, one dummy element is picked up one location past the end of the data replacement vector $\vec{B}$ and is placed in the final address available to the dummy output region of that self loop.

For the case of a self loop passing over the minimum data replacement vector, the first dummy replacement element after the last data replacement element is picked up and placed in the first location past the data output array, which is at the beginning of the dummy output region. The last dummy element is placed in the final address available to the dummy output region of that self loop.

This procedure establishes a constant number of replacement elements and indices for each self loop. The number of elements of the data output array is assumed to be constant for each self loop.

Example: A singleword replace instruction using one self loop of length 4.

Halfword Vector $\vec{A}$	Singleword Index Vector $\vec{B}$	Singleword Vector $\vec{C}$ Before Replacement	Singleword Vector $\vec{C}$ After Replacement
2, 3	-54	16	16
5, 6	-72	72	72
	-64	27	-54
	-15	36	-72
		71	71
		32	-64
		8	-15
		14	14

APPENDIX A: SCALAR INSTRUCTIONS BY LOGICAL GROUPING

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
ST	Store arithmetic register, singleword	24	r,[@]n[,x]	7-26
ST	Store base register, singleword	28	r,[@]n[,x]	7-26
ST	Store index register or vector parameter register, singleword	2C	r,[@]n[,x]	7-26
STLL	Store arithmetic left halfword into memory left halfword, indexed	25	r,[@]n[,x]	7-27
STRL	Store arithmetic right halfword into memory left halfword, indexed	26	r,[@]n[,x]	7-28.1
STRR	Store arithmetic register right halfword into memory right halfword, indexed	2D	r,[@]n[,x]	7-28
STLR	Store arithmetic register left halfword into memory right halfword, indexed	29	r,[@]n[,x]	7-29
STD	Store arithmetic register, doubleword	27	r,[@]n[,x]	7-30
STZ	Store zero, word	20	[@]n[,x]	7-31
STZH	Store zero, halfword	21	[@]n[,x]	7-32
STZD	Store zero, doubleword	23	[@]n[,x]	7-33
STN	Store negative, fixed point word	34	r,[@]n[,x]	7-34
STNH	Store negative, fixed point halfword	35	r,[@]n[,x]	7-35
STNF	Store negative, floating point word	36	r,[@]n[,x]	7-36
STND	Store negative, floating point doubleword	37	r,[@]n[,x]	7-37
STO	Store ones complement, word	2E	r,[@]n[,x]	7-38
STOH	Store ones complement, halfword	2A	r,[@]n[,x]	7-39
STF	Store base register file A, M=0	2B	m,[@]n[,x]	7-40
STF	Store base register file B, M=1	2B	m,[@]n[,x]	7-40
STF	Store arithmetic register file C, M=2	2B	m,[@]n,[,x]	7-40
STF	Store arithmetic register file D, M=3	2B	m,[@]n[,x]	7-40
STF	Store index register file X, M=4	2B	m,[@]n[,x]	7-40
STF	Store vector parameter register file V, M=5	2B	m,[@]n[,x]	7-40
STFM	Store all six eight-word register files	2F	[@]n[,x]	7-41
L	Load arithmetic register, singleword	14	r,[@][=]n[,x]	7-3
L	Load base register, singleword	18	r,[@][=]n[,x]	7-3
L	Load index register or vector parameter register, singleword	1C	r,[@][=]n[,x]	7-3
LLL	Load arithmetic register left halfword from memory right halfword, indexed	15	r,[@][=]n[,x]	7-4
LRL	Load memory left halfword, indexed, into arithmetic register right halfword	10	r,[@][=]n[,x]	7-5.1

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
LRR	Load memory right halfword, indexed, into arithmetic register right halfword	1D	r,[@][=]n[,x]	7-5
LLR	Load memory right halfword, indexed, into arithmetic register left halfword	19	r,[@][=]n[,x]	7-6
LD	Load arithmetic register, doubleword	17	r,[@][=]n[,x]	7-7
LI	Load immediate into arithmetic register singleword	54	r,i[,x]	7-8
LI	Load immediate into index register, or vector parameter register, singleword	5C	r,i[,x]	7-8
LIH	Load immediate into arithmetic register, halfword	55	r,i[,x]	7-9
LN	Load negative, fixed point singleword, arithmetic register	30	r,[@][=]n[,x]	7-10
LNH	Load negative, fixed point halfword, arithmetic register	31	r,[@][=]n[,x]	7-11
LNF	Load negative, floating point singleword, arithmetic register	32	r,[@][=]n[,x]	7-12
LND	Load negative, floating point doubleword, arithmetic register	33	r,[@][=]n[,x]	7-13
LM	Load magnitude, fixed point singleword, arithmetic register	3C	r,[@][=]n[,x]	7-14
LMH	Load magnitude, fixed point halfword, arithmetic register	3D	r,[@][=]n[,x]	7-15
LMF	Load magnitude, floating point singleword, arithmetic register	3E	r,[@][=]n[,x]	7-16
LMD	Load magnitude, floating point doubleword, arithmetic register	3F	r,[@][=]n[,x]	7-17
LNM	Load negative magnitude, fixed point singleword, arithmetic register	38	r,[@][=]n[,x]	7-18
LNMH	Load negative magnitude, fixed point halfword, arithmetic register	39	r,[@][=]n[,x]	7-19
LNMF	Load negative magnitude, floating point singleword, arithmetic register	3A	r,[@][=]n[,x]	7-20
LNMD	Load negative magnitude, floating point doubleword, arithmetic	3B	r,[@][=]n[,x]	7-21
LO	Load arithmetic register with ones complement, singleword	1E	r,[@][=]n[,x]	7-22
LF	Load base register file A, M=0	1B	m,[@]n[,x]	7-23
LF	Load base register file B, M=1	1B	m,[@]n[,x]	7-23
LF	Load arithmetic register file C, M=2	1B	m,[@]n[,x]	7-23
LF	Load arithmetic register file D, M=3	1B	m,[@]n[,x]	7-23
LF	Load index register file X, M=4	1B	m,[@]n[,x]	7-23

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
LF	Load vector parameter register file V, M=5	1B	m,[@]n[,x]	7-23
LFM	Load all six eight-word register files	1F	[@]n[,x]	7-24
A	Add to arithmetic register, fixed point single-word	40	r,[@][=]n[,x]	7-43
A	Add to base register, fixed point singleword	60	r,[@][=]n[,x]	7-43
A	Add to index or vector parameter register, fixed point singleword	62	r,[@][=]n[,x]	7-43
AH	Add to arithmetic register, fixed point halfword	41	r,[@][=]n[,x]	7-44
AF	Add to arithmetic register, floating point single-word	42	r,[@][=]n[,x]	7-45
AFD	Add to arithmetic register, floating point doubleword	43	r,[@][=]n[,x]	7-46
AI	Add immediate to arithmetic register, fixed point singleword	50	r,i[,x]	7-47
AI	Add immediate to base register, fixed point singleword	70	r,i[,x]	7-47
AI	Add immediate to index or vector parameter register, fixed point singleword	72	r,i[,x]	7-47
AIH	Add immediate to arithmetic register, fixed point halfword	51	r,i[,x]	7-48
AM	Add magnitude to arithmetic register, fixed point singleword	44	r,[@][=]n[,x]	7-49
AMH	Add magnitude to arithmetic register, fixed point halfword	45	r,[@][=]n[,x]	7-50
AMF	Add magnitude to arithmetic register, floating point singleword	46	r,[@][=]n[,x]	7-51
AMFD	Add magnitude to arithmetic register, floating point doubleword	47	r,[@][=]n[,x]	7-52
S	Subtract from arithmetic register, fixed point singleword	48	r,[@][=]n[,x]	7-53
SH	Subtract from arithmetic register, fixed point halfword	49	r,[@][=]n[,x]	7-54
SF	Subtract from arithmetic register, floating point singleword	4A	r,[@][=]n[,x]	7-55
SFD	Subtract from arithmetic register, floating point doubleword	4B	r,[@][=]n[,x]	7-56
SI	Subtract immediate from arithmetic register, fixed point singleword	58	r,i[,x]	7-57

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
SIH	Subtract immediate from arithmetic register, fixed point halfword	59	r, i[, x]	7-57
SM	Subtract magnitude from arithmetic register, fixed point singleword	4C	r, [@][=]n[, x]	7-59
SMH	Subtract magnitude from arithmetic register, fixed point halfword	4D	r, [@][=]n[, x]	7-60
SMF	Subtract magnitude from arithmetic register, floating point singleword	4E	r, [@][=]n[, x]	7-61
SMFD	Subtract magnitude from arithmetic register, floating point doubleword	4F	r, [@][=]n[, x]	7-62
M	Multiply, fixed point singleword - arithmetic register	6C	r, [@][=]n[, x]	7-63
M	Multiply, fixed point singleword - base register	68	r, [@][=]n[, x]	7-63
M	Multiply, fixed point singleword - index or vector parameter register	6A	r, [@][=]n[, x]	7-63
MH	Multiply, fixed point halfword - arithmetic register	6D	r, [@][=]n[, x]	7-64
MF	Multiply, floating point singleword - arithmetic register	6E	r, [@][=]n[, x]	7-65
MFD	Multiply, floating point doubleword - arithmetic register	6F	r, [@][=]n[, x]	7-66
MI	Multiply immediate, fixed point singleword - arithmetic register	7C	r, i[, x]	7-67
MI	Multiply immediate, fixed point singleword - base register	78	r, i[, x]	7-67
MI	Multiply immediate, fixed point singleword- index or vector parameter register	7A	r, i[, x]	7-67
MIH	Multiply immediate, fixed point halfword - arithmetic register	7D	r, i[, x]	7-68
D	Divide into arithmetic register, fixed point singleword	64	r, [@][=]n[, x]	7-69
DH	Divide into arithmetic register, fixed point halfword	65	r, [@][=]n[, x]	7-70
DF	Divide into arithmetic register, floating point singleword	66	r, [@][=]n[, x]	7-71
DFD	Divide into arithmetic register, floating point doubleword	67	r, [@][=]n[, x]	7-72
DI	Divide immediate into arithmetic register, fixed point singleword	74	r, i[, x]	7-73

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
DIH	Divide immediate into arithmetic register, fixed point halfword	75	r, i[, x]	7-74
AND	AND, singleword - arithmetic register	E0	r, [@][=]n[, x]	7-76
ANDD	AND, doubleword - arithmetic register	E1	r, [@][=]n[, x]	7-77
ANDI	AND immediate, singleword - arithmetic register	F0	r, i[, x]	7-78
OR	OR, singleword - arithmetic register	E4	r, [@][=]n[, x]	7-79
ORD	OR, doubleword - arithmetic register	E5	r, [@][=]n[, x]	7-80
ORI	OR immediate, singleword - arithmetic register	F4	r, i[, x]	7-81
XOR	Exclusive OR, singleword - arithmetic register	E8	r, [@][=]n[, x]	7-82
XORD	Exclusive OR, doubleword - arithmetic register	E9	r, [@][=]n[, x]	7-83
XORI	Exclusive OR immediate, singleword- arithmetic register	F8	r, i[, x]	7-84
EQC	Equivalence, singleword - arithmetic register	EC	r, [@][=]n[, x]	7-85
EQCD	Equivalence, doubleword - arithmetic register	ED	r, [@][=]n[, x]	7-86
EQCI	Equivalence immediate, singleword - arithmetic register	FC	r, i[, x]	7-87
SA	Arithmetic shift, fixed point singleword - arithmetic register	C0	r, i[, x]	7-93
SAH	Arithmetic shift, fixed point halfword - arithmetic register	C1	r, i[, x]	7-94
SAD	Arithmetic shift, fixed point doubleword - arithmetic register	C3	r, i[, x]	7-95
SL	Logical shift, singleword - arithmetic register	C4	r, i[, x]	7-96
SLH	Logical shift, halfword - arithmetic register	C5	r, i[, x]	7-97
SLD	Logical shift, doubleword - arithmetic register	C7	r, i[, x]	7-98
SC	Circular shift, singleword - arithmetic register	CC	r, i[, x]	7-99
SCH	Circular shift, halfword - arithmetic register	CD	r, i[, x]	7-100
SCD	Circular shift, doubleword - arithmetic register	CF	r, i[, x]	7-101
RVS	Bit reversal, singleword - arithmetic register	C6	r, i[, x]	7-102
C	Compare arithmetic register, fixed point singleword	C8	r, [@][=]n[, x]	7-104
C	Compare index or vector register, fixed point singleword	CE	r, [@][=]n[, x]	7-104
CH	Compare arithmetic register, fixed point halfword	C9	r, [@][=]n[, x]	7-105

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
CF	Compare arithmetic register, floating point singleword	CA	r,[@][=]n[,x]	7-106
CFD	Compare arithmetic register, floating point doubleword	CB	r,[@][=]n[,x]	7-107
CI	Compare immediate arithmetic register, fixed point singleword	D8	r,i[,x]	7-108
CI	Compare index or vector register with immediate, singleword	DE	r,i[,x]	7-108
CIH	Compare arithmetic register immediate, fixed point halfword	D9	r,i[,x]	7-109
CAND	Compare logical AND, singleword - arithmetic register	E2	r,[@][=]n[,x]	7-110
CANDD	Compare logical AND, doubleword - arithmetic register	E3	r,[@][=]n[,x]	7-111
CANDI	Compare immediate logical AND, singleword - arithmetic register	F2	r,i[,x]	7-112
COR	Compare logical OR, singleword - arithmetic register	E6	r,[@][=]n[,x]	7-113
CORD	Compare logical OR, doubleword - arithmetic register	E7	r,[@][=]n[,x]	7-114
CORI	Compare immediate logical OR, singleword - arithmetic register	F6	r,i[,x]	7-115
ISE	Increment arithmetic register, test, and skip on equal	80	r,[@][=]n[,x]	7-117
ISNE	Increment arithmetic register, test, and skip on not equal	81	r,[@][=]n[,x]	7-118
DSE	Decrement arithmetic register, test, and skip on equal	82	r,[@][=]n[,x]	7-119
DSNE	Decrement arithmetic register, test, and skip on not equal	83	r,[@][=]n[,x]	7-120
BCC	Branch on compare code true	91	m,[@[=]]n[,x]	7-132
NOP	Take next instruction, Assembler supplies R field of zero	91	[@[=]]n[,x]	7-132
BE	Branch on compare code of equal, Assembler supplies R field of one	91	[@[=]]n[,x]	7-132
BG	Branch on compare code of greater than, Assembler supplies R field of 2	91	[@[=]]n[,x]	7-132
BGE	Branch on compare code of greater than or equal, Assembler supplies R field of 3	91	[@[=]]n[,x]	7-132

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
BL	Branch on compare code of less than, Assembler supplies R field of 4	91	[@[=]]n[,x]	7-132
BLE	Branch on compare code of less than or equal, Assembler supplies R field of 5	91	[@[=]]n[,x]	7-132
BNE	Branch on compare code of not equal, Assembler supplies R field of 6	91	[@[=]]n[,x]	7-132
B	Unconditional branch, Assembler supplies R field of 7	91	[@[=]]n[,x]	7-132
BCZ	Branch on compare code of all bits are zero, Assembler supplies R field of one	91	[@[=]]n[,x]	7-132
BCO	Branch on compare code of all bits are one, Assembler supplies R field of 2	91	[@[=]]n[,x]	7-132
BCNM	Branch on compare code of not mixed, Assembler supplies R field of 3	91	[@[=]]n[,x]	7-132
BCM	Branch on compare code of mixed zeros and ones, Assembler supplies R field of 4	91	[@[=]]n[,x]	7-132
BCNO	Branch on compare code of not all ones, Assembler supplies R field of 5	91	[@[=]]n[,x]	7-132
BCNZ	Branch on compare code of not all zeros, Assembler supplies the R field of 6	91	[@[=]]n[,x]	7-132
BRC	Branch on result code true	95	m,[@[=]]n[,x]	7-133
BZ	Branch on result code of zero, Assembler supplies the R field of one	95	[@[=]]n[,x]	7-133
BPL	Branch on result code of positive, Assembler supplies the R field of 2	95	[@[=]]n[,x]	7-133
BZP	Branch on result code of zero or positive, Assembler supplies the R field of 3	95	[@[=]]n[,x]	7-133
BMI	Branch on result code of negative, Assembler supplies the R field of 4	95	[@[=]]n[,x]	7-133
BZM	Branch on result code of zero or negative, Assembler supplies the R field of 5	95	[@[=]]n[,x]	7-133
BNZ	Branch on result code of not zero, Assembler supplies the R field of 6	95	[@[=]]n[,x]	7-133
BLR	Branch on logical result	95	m,[@[=]]n[,x]	7-133
BRZ	Branch on result code of all bits are zero, Assembler supplies the R field of one	95	[@[=]]n[,x]	7-133
BRO	Branch on result code of all bits are one, Assembler supplies the R field of 2	95	[@[=]]n[,x]	7-133
BRNM	Branch on result code of bits not mixed zeros and ones, Assembler supplies the R field of 3	95	[@[=]]n[,x]	7-133

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
BRM	Branch on result code of bits mixed zeros and ones, Assembler supplies the R field of 4	95	[@[=]]n[,x]	7-133
BRNO	Branch on result code of not all bits ones, Assembler supplies the R field of 5	95	[@[=]]n[,x]	7-133
BRNZ	Branch on result code of not all bits zeros, Assembler supplies the R field of 6	95	[@[=]]n[,x]	7-133
BAE	Branch on arithmetic exception condition true	9D	m,[@[=]]n[,x]	7-134
BU	Branch on floating point exponent underflow, Assembler supplies R field of one	9D	[@[=]]n[,x]	7-134
BO	Branch on floating point exponent overflow, Assembler supplies R field of 2	9D	[@[=]]n[,x]	7-134
BUO	Branch on floating point exponent underflow or overflow, Assembler supplies R field of 3	9D	[@[=]]n[,x]	7-134
BX	Branch on fixed point overflow, Assembler supplies R field of 4	9D	[@[=]]n[,x]	7-134
BXU	Branch on fixed point overflow or floating point exponent underflow, Assembler supplies R field of 5	9D	[@[=]]n[,x]	7-134
BXO	Branch on fixed point overflow or floating point exponent overflow, Assembler supplies R field of 6	9D	[@[=]]n[,x]	7-134
BXUO	Branch on fixed point overflow or floating point exponent overflow or underflow, Assembler supplies R field of 7	9D	[@[=]]n[,x]	7-134
BD	Branch on divide check, Assembler supplies R field of 8	9D	[@[=]]n[,x]	7-134
BDU	Branch on divide check or floating point exponent underflow, Assembler supplies R field of 9	9D	[@[=]]n[,x]	7-134
BDO	Branch on divide check or floating point exponent overflow, Assembler supplies R field of A	9D	[@[=]]n[,x]	7-134
BDUO	Branch on divide check or floating point exponent overflow or underflow, Assembler supplies R field of B	9D	[@[=]]n[,x]	7-134
BDX	Branch on divide check or fixed point overflow, Assembler supplies R field of C	9D	[@[=]]n[,x]	7-134
BDXU	Branch on divide check or fixed point overflow or floating point exponent underflow, Assembler supplies R field of D	9D	[@[=]]n[,x]	7-134

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
BDXO	Branch on divide check or fixed point overflow or floating point exponent overflow, Assembler supplies R field of E	9D	[@[=]]n[,x]	7-134
BDXUO	Branch on divide check or fixed point overflow or floating point exponent overflow or underflow, Assembler supplies R field of F	9D	[@[=]]n[,x]	7-134
BXEC	Branch on Execute branch condition true, Assembler supplies R field of one or odd	9C	[@]n[,x]	7-135
IBZ	Increment arithmetic register, test, and branch on zero	88	r,[@[=]]n[,x]	7-122
IBZ	Increment index or vector register, test, and branch on zero	8C	r,[@[=]]n[,x]	7-122
IBNZ	Increment arithmetic register, test, and branch on not zero	89	r,[@[=]]n[,x]	7-123
IBNZ	Increment index or vector register, and branch on not zero	8D	r,[@[=]]n[,x]	7-123
DBZ	Decrement arithmetic register, test, and branch zero	8A	r,[@[=]]n[,x]	7-124
DBZ	Decrement index or vector register, test, and branch on zero	8E	r,[@[=]]n[,x]	7-124
DBNZ	Decrement arithmetic register, test, and branch on not zero	8B	r,[@[=]]n[,x]	7-124
DBNZ	Decrement index or vector register, test, and branch on not zero	8F	r,[@[=]]n[,x]	7-124
BCLE	Branch on arithmetic register less than or equal	84	r,r,n	7-128
BCLE	Branch on index or vector register less than or equal	86	r,r,n	7-128
BCG	Branch on arithmetic register greater than	85	r,r,n	7-129
BCG	Branch on index or vector register greater than	87	r,r,n	7-129
BLB	Branch and load base register with program counter	98	r,[@[=]]n[,x]	7-137
BLX	Branch and load index or vector register with program counter	99	r,[@[=]]n[,x]	7-138
PSH	Push word into last-in-first-out stack	93	r,[@]n[,x]	7-141
PUL	Pull word from last-in-first-out stack	97	r,[@]n[,x]	7-142
MOD	Modify stack parameter doubleword	9F	r,[@]n[,x]	7-143
FLFX	Convert floating point singleword to fixed point singleword	A0	r,[@]n[,x]	7-148

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
FLFH	Convert floating point singleword to fixed point halfword	A1	r,[@]n[,x]	7-149
FDFX	Convert floating point doubleword to fixed point singleword	A2	r,[@]n[,x]	7-150
FXFL	Convert fixed point singleword to floating point singleword	A8	r,[@]n[,x]	7-151
FHFL	Convert fixed point halfword to floating point singleword	A9	r,[@]n[,x]	7-152
FXFD	Convert fixed point singleword to floating point doubleword	AA	r,[@]n[,x]	7-153
FHFD	Convert fixed point halfword to floating point doubleword	AB	r,[@]n[,x]	7-154
NFX	Normalize fixed point singleword	AC	r,[@]n[,x]	7-155
NFH	Normalize fixed point halfword	AD	r,[@]n[,x]	7-156
XCH	Exchange - arithmetic register with effective address	1A	r,[@]n[,x]	7-158
LLA	Load look ahead	16	i	7-159
PB	Prepare to branch	9E	---	7-159.1
LEA	Load effective address into base register	52	r,[@][=]n[,x]	7-160
LEA	Load effective address into index or vector register	56	r,[@][=]n[,x]	7-160
XEC	Execute addressed instruction in line	96	[@[=]]n[,x]	7-161
INT	Interpret - arithmetic register	92	r,[@][=]n[,x]	7-162
FORK	Fork	9A	---	7-162.1
JOIN	Join	9B	---	7-162.2
MCP	Monitor call and proceed	90	i[,x]	7-163
MCW	Monitor call and wait	94	i[,x]	7-164
LAM	Load arithmetic mask	12	[@[=]]n[,x]	7-166
LAC	Load arithmetic exception condition	13	[@[=]]n[,x]	7-167
LEM	Load arithmetic exception mask and condition	11	[@[=]]n[,x]	7-167.1
SCLK	Store Clock	AE	---	7-167.2
SPS	Store program status word	22	[@]n[,x]	7-168
VECTL	Load and execute vector parameter file, Assembler supplies R field of zero	B0	[@]n[,x]	8-4
VECT	Execute vector parameter file, Assembler supplies R field of one	B0	[@]n[,x]	8-5

APPENDIX B: SCALAR INSTRUCTIONS IN ALPHABETICAL ORDER BY ASSEMBLER CODE

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
A	Add to arithmetic register, fixed point singleword	40	r,[@][=]n[,x]	7-43
A	Add to base register, fixed point singleword	60	r,[@][=]n[,x]	7-43
A	Add to index or vector parameter register,	62	r,[@][=]n[,x]	7-43
AF	Add to arithmetic register, floating point singleword	42	r,[@][=]n[,x]	7-45
AFD	Add to arithmetic register, floating point doubleword	43	r,[@][=]n[,x]	7-46
AH	Add to arithmetic register, fixed point halfword	41	r,[@][=]n[,x]	7-44
AI	Add immediate to arithmetic register, fixed point singleword	50	r,i[,x]	7-47
AI	Add immediate to base register, fixed point singleword	70	r,i[,x]	7-47
AI	Add immediate to index or vector parameter register, fixed point singleword	72	r,i[,x]	7-47
AIH	Add immediate to arithmetic register, fixed point halfword	51	r,i[,x]	7-48
AM	Add magnitude to arithmetic register, fixed point singleword	44	r,[@][=]n[,x]	7-49
AMF	Add magnitude to arithmetic register, floating point singleword	46	r,[@][=]n[,x]	7-51
AMFD	Add magnitude to arithmetic register, floating point doubleword	47	r,[@][=]n[,x]	7-52
AMH	Add magnitude to arithmetic register, fixed point halfword	45	r,[@][=]n[,x]	7-50
AND	AND, singleword - arithmetic register	E0	r,[@][=]n[,x]	7-76
ANDD	AND, doubleword - arithmetic register	E1	r,[@][=]n[,x]	7-77
ANDI	AND immediate, singleword - arithmetic register	F0	r,i[,x]	7-78
B	Unconditional branch, Assembler supplies R field of 7	91	[@[=]]n[,x]	7-132
BAE	Branch on arithmetic exception condition true	9D	m,[@[=]]n[,x]	7-134
BCC	Branch on compare code true	91	m,[@[=]]n[,x]	7-132
BCG	Branch on arithmetic register greater than	85	r,r,n	7-129
BCG	Branch on index or vector register greater than	87	r,r,n	7-129
BCLE	Branch on arithmetic register less than or equal	84	r,r,n	7-128

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
BCLE	Branch on index or vector register less than or equal	86	r, r, n	7-128
BCM	Branch on compare code of mixed zeros and ones, Assembler supplies R field of 4	91	[@[=]]n[, x]	7-132
BCNM	Branch on compare code of not mixed, Assembler supplies R field of 3	91	[@[=]]n[, x]	7-132
BCNO	Branch on compare code of not all ones, Assembler supplies R field of 5	91	[@[=]]n[, x]	7-132
BCNZ	Branch on compare code of not all zeros, Assembler supplies the R field of 6	91	[@[=]]n[, x]	7-132
BCO	Branch on compare code of all bits are one, Assembler supplies R field of 2	91	[@[=]]n[, x]	7-132
BCZ	Branch on compare code of all bits are zero, Assembler supplies R field of one	91	[@[=]]n[, x]	7-132
BD	Branch on divide check, Assembler supplies R field of 8	9D	[@[=]]n[, x]	7-134
BDO	Branch on divide check or floating point exponent overflow, Assembler supplies R field of A	9D	[@[=]]n[, x]	7-134
BDU	Branch on divide check or floating point exponent underflow, Assembler supplies R field of 9	9D	[@[=]]n[, x]	7-134
BDUO	Branch on divide check or floating point exponent overflow or underflow, Assembler supplies R field of B	9D	[@[=]]n[, x]	7-134
BDX	Branch on divide check or fixed point overflow, Assembler supplies R field of C	9D	[@[=]]n[, x]	7-134
BDXO	Branch on divide check or fixed point overflow or floating point exponent overflow, Assembler supplies R field of E	9D	[@[=]]n[, x]	7-134
BDXU	Branch on divide check or fixed point overflow or floating point exponent underflow, Assembler supplies R field of D	9D	[@[=]]n[, x]	7-134
BDXUO	Branch on divide check or fixed point overflow or floating point exponent overflow or underflow, Assembler supplies R field of F	9D	[@[=]]n[, x]	7-134
BE	Branch on compare code of equal, Assembler supplies R field of one	91	[@[=]]n[, x]	7-132
BG	Branch on compare code of greater than, Assembler supplies R field of 2	91	[@[=]]n[, x]	7-132

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
BGE	Branch on compare code of greater than or equal, Assembler supplies R field of 3	91	[@[=]]n[,x]	7-132
BL	Branch on compare code of less than, Assembler supplies R field of 4	91	[@[=]]n[,x]	7-132
BLB	Branch and load base register with program counter	98	r,[@[=]]n[,x]	7-137
BLE	Branch on compare code of less than or equal, Assembler supplies R field of 5	91	[@[=]]n[,x]	7-132
BLR	Branch on logical result	95	m,[@[=]]n[,x]	7-133
BLX	Branch and load index or vector register with program counter	99	r,[@[=]]n[,x]	7-138
BMI	Branch on result code of negative, Assembler supplies the R field of 4	95	[@[=]]n[,x]	7-133
BNE	Branch on compare code of not equal, Assembler supplies R field of 6	91	[@[=]]n[,x]	7-132
BNZ	Branch on result code of not zero, Assembler supplies the R field of 6	95	[@[=]]n[,x]	7-133
BO	Branch on floating point exponent overflow, Assembler supplies R field of 2	9D	[@[=]]n[,x]	7-134
BPL	Branch on result code of positive, Assembler supplies the R field of 2	95	[@[=]]n[,x]	7-133
BRC	Branch on result code true	95	m,[@[=]]n[,x]	7-133
BRM	Branch on result code of bits mixed zeros and ones, Assembler supplies the R field of 4	95	[@[=]]n[,x]	7-133
BRNM	Branch on result code of bits not mixed zeros and ones, Assembler supplies the R field of 3	95	[@[=]]n[,x]	7-133
BRNO	Branch on result code of not all bits ones, Assembler supplies the R field of 5	95	[@[=]]n[,x]	7-133
BRNZ	Branch on result code of not all bits zeros, Assembler supplies the R field of 6	95	[@[=]]n[,x]	7-133
BRO	Branch on result code of all bits are one, Assembler supplies the R field of 2	95	[@[=]]n[,x]	7-133
BRZ	Branch on result code of all bits are zero, Assembler supplies the R field of one	95	[@[=]]n[,x]	7-133
BU	Branch on floating point exponent underflow, Assembler supplies R field of one	9D	[@[=]]n[,x]	7-134
BUO	Branch on floating point exponent underflow or overflow, Assembler supplies R field of 3	9D	[@[=]]n[,x]	7-134
BX	Branch on fixed point overflow, Assembler supplies R field of 4	9D	[@[=]]n[,x]	7-134

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
BXEC	Branch on Execute branch condition true, Assembler supplies R field of one or odd	9C	[@]n[,x]	7-135
BXO	Branch on fixed point overflow or floating point exponent overflow, Assembler supplies R field of 6	9D	[@[=]]n[,x]	7-134
BXU	Branch on fixed point overflow or floating point exponent underflow, Assembler supplies R field of 5	9D	[@[=]]n[,x]	7-134
BXUO	Branch on fixed point overflow or floating point exponent overflow or underflow, Assembler supplies R field of 7	9D	[@[=]]n[,x]	7-134
BZ	Branch on result code of zero, Assembler supplies the R field of one	95	[@[=]]n[,x]	7-133
BZM	Branch on result code of zero or negative, Assembler supplies the R field of 5	95	[@[=]]n[,x]	7-133
BZP	Branch on result code of zero or positive, Assembler supplies the R field of 3	95	[@[=]]n[,x]	7-133
C	Compare arithmetic register, fixed point singleword	C8	r,[@][=]n[,x]	7-104
C	Compare index or vector register, fixed point singleword	CE	r,[@][=]n[,x]	7-104
CAND	Compare logical AND, singleword - arithmetic register	E2	r,[@][=]n[,x]	7-110
CANDD	Compare logical AND, doubleword - arithmetic register	E3	r,[@][=]n[,x]	7-111
CANDI	Compare immediate logical AND, singleword - arithmetic register	F2	r,i[,x]	7-112
CF	Compare arithmetic register, floating point singleword	CA	r,[@][=]n[,x]	7-106
CFD	Compare arithmetic register, floating point doubleword	CB	r,[@][=]n[,x]	7-107
CH	Compare arithmetic register, fixed point halfword	C9	r,[@][=]n[,x]	7-105
CI	Compare immediate arithmetic register, fixed point singleword	D8	r,i[,x]	7-108
CI	Compare index or vector register with immediate singleword	DE	r,i[,x]	7-108
CIH	Compare arithmetic register immediate, fixed point halfword	D9	r,i[,x]	7-109

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
COR	Compare logical OR, singleword - arithmetic register	E6	r,[@][=]n[,x]	7-113
CORD	Compare logical OR, doubleword - arithmetic register	E7	r,[@][=]n[,x]	7-114
CORI	Compare immediate logical OR, singleword - arithmetic register	F6	r,i[,x]	7-115
D	Divide into arithmetic register, fixed point singleword	64	r,[@][=]n[,x]	7-69
DBNZ	Decrement arithmetic register, test, and branch on not zero	8B	r,[@[=]]n[,x]	7-125
DBNZ	Decrement index or vector register, test, and branch on not zero	8F	r,[@[=]]n[,x]	7-125
DBZ	Decrement arithmetic register, test, and branch on zero	8A	r,[@[=]]n[,x]	7-124
DBZ	Decrement index or vector register, test, and branch on zero	8E	r,[@[=]]n[,x]	7-124
DF	Divide into arithmetic register, floating point singleword	66	r,[@][=]n[,x]	7-71
DFD	Divide into arithmetic register, floating point doubleword	67	r,[@][=]n[,x]	7-72
DH	Divide into arithmetic register, fixed point halfword	65	r,[@][=]n[,x]	7-70
DI	Divide immediate into arithmetic register, fixed point singleword	74	r,i[,x]	7-73
DIH	Divide immediate into arithmetic register, fixed point halfword	75	r,i[,x]	7-74
DSE	Decrement arithmetic register, test, and skip on equal	82	r,[@][=]n[,x]	7-119
DSNE	Decrement arithmetic register, test, and skip on not equal	83	r,[@][=]n[,x]	7-120
EQC	Equivalence, singleword - arithmetic register	EC	r,[@][=]n[,x]	7-85
EQCD	Equivalence, doubleword - arithmetic register	ED	r,[@][=]n[,x]	7-86
EQCI	Equivalence immediate, singleword - arithmetic register	FC	r,i[,x]	7-87
FDFX	Convert floating point doubleword to fixed point singleword	A2	r,[@]n[,x]	7-150
FHFD	Convert fixed point halfword to floating point doubleword	AB	r,[@]n[,x]	7-154

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
FHFL	Convert fixed point halfword to floating point singleword	A9	r,[@]n[,x]	7-152
FLFH	Convert floating point singleword to fixed point halfword	A1	r,[@]n[,x]	7-149
FLFX	Convert floating point singleword to fixed point singleword	A0	r,[@]n[,x]	7-148
FORK	Fork	9A	---	7-162.1
FXFD	Convert fixed point singleword to floating point doubleword	AA	r,[@]n[,x]	7-153
FXFL	Convert fixed point singleword to floating point singleword	A8	r,[@]n[,x]	7-151
IBNZ	Increment arithmetic register, test, and branch on not zero	89	r,[@[=]]n[,x]	7-123
IBNZ	Increment index or vector register, and branch on not zero	8D	r,[[[=]]n[,x]	7-123
IBZ	Increment arithmetic register, test, and branch on zero	88	r,[@[=]]n[,x]	7-122
IBZ	Increment index or vector register, test, and branch on zero	8C	r,[@[=]]n[,x]	7-122
INT	Interpret - arithmetic register	92	r,[@][=]n[,x]	7-162
ISE	Increment arithmetic register, test and skip on equal	80	r,[@][=]n[,x]	7-117
ISNE	Increment arithmetic register, test, and skip on not equal	81	r,[@][=]n[,x]	7-118
JOIN	Join	9B	---	7-162.2
L	Load arithmetic register, singleword	14	r,[@][=]n[,x]	7-3
L	Load base register, singleword	18	r,[@][=]n[,x]	7-3
L	Load index register or vector parameter register, singleword	1C	r,[@][=]n[,x]	7-3
LAC	Load arithmetic exception condition	13	[@[=]]n[,x]	7-167
LAM	Load arithmetic mask	12	[@[=]]n[,x]	7-166
LD	Load arithmetic register doubleword	17	r,[@][=]n[,x]	7-7
LEA	Load effective address into base register	52	r,[@][=]n[,x]	7-160
LEA	Load effective address into index or vector register	56	r,[@][=]n[,x]	7-160
LEM	Load arithmetic exception mask and condition	11	[@[=]]n[,x]	7-167.1
LF	Load base register file A, M=0	1B	m,[@]n[,x]	7-23
LF	Load base register file B, M=1	1B	m,[@]n[,x]	7-23
LF	Load arithmetic register file C, M=2	1B	m,[@]n[,x]	7-23
LF	Load arithmetic register file D, M=3	1B	m,[@]n[,x]	7-23

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
LF	Load index register file X, M=4	1B	m,[@]n[,x]	7-23
LF	Load vector parameter register file V, M=5	1B	m,[@]n[,x]	7-23
LFM	Load all six eight-word register files	1F	[@]n[,x]	7-24
LI	Load immediate into arithmetic register singleword	54	r,i[,x]	7-8
LI	Load immediate into index register, or vector parameter register, singleword	5C	r,i[,x]	7-8
LIH	Load immediate into arithmetic register, halfword	55	r,i[,x]	7-9
LLA	Load look ahead	16	i	7-159
LLL	Load memory left halfword, indexed, into arithmetic register left halfword	15	r,[@][=]n[,x]	7-4
LLR	Load memory right halfword, indexed, into arithmetic register left halfword	19	r,[@][=]n[,x]	7-6
LM	Load magnitude, fixed point singleword, arithmetic register	3C	r,[@][=]n[,x]	7-14
LMD	Load magnitude, floating point doubleword, arithmetic register	3F	r,[@][=]n[,x]	7-17
LMF	Load magnitude, floating point singleword, arithmetic register	3E	r,[@][=]n[,x]	7-16
LMH	Load magnitude, fixed point halfword, arithmetic register	3D	r,[@][=]n[,x]	7-15
LN	Load negative, fixed point singleword, arithmetic register	30	r,[@][=]n[,x]	7-10
LND	Load negative, floating point doubleword, arithmetic register	33	r,[@][=]n[,x]	7-13
LNF	Load negative, floating point singleword, arithmetic register	32	r,[@][=]n[,x]	7-12
LNH	Load negative, fixed point halfword, arithmetic register	31	r,[@][=]n[,x]	7-11
LNM	Load negative magnitude, fixed point singleword, arithmetic register	38	r,[@][=]n[,x]	7-18
LNMD	Load negative magnitude, floating point doubleword, arithmetic register	3B	r,[@][=]n[,x]	7-21
LNMF	Load negative magnitude, floating point singleword, arithmetic register	3A	r,[@][=]n[,x]	7-20
LNMH	Load negative magnitude, fixed point halfword, arithmetic register	39	r,[@][=]n[,x]	7-19
LO	Load arithmetic register with ones complement singleword	1E	r,[@][=]n[,x]	7-22

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
LRL	Load memory left halfword, indexed, into arithmetic register right halfword	10	r,[@][=]n[,x]	7-5.1
LRR	Load memory right halfword, indexed, into arithmetic register right halfword	1D	r,[@][=]n[,x]	7-5
M	Multiply fixed point singleword - arithmetic register	6C	r,[@][=]n[,x]	7-63
M	Multiply, fixed point singleword base register	68	r,[@][=]n[,x]	7-63
M	Multiply, fixed point singleword - index or vector parameter register	6A	r,[@][=]n[,x]	7-63
MCP	Monitor call and proceed	90	i[,x]	
MCW	Monitor call and wait	94	i[,x]	7-164
MF	Multiply, floating point singleword - arithmetic register	6E	r,[@][=]n[,x]	7-65
MFD	Multiply, floating point doubleword - arithmetic register	6F	r,[@][=]n[,x]	7-66
MH	Multiply, fixed point halfword - arithmetic register	6D	r,[@][=]n[,x]	7-64
MI	Multiply immediate, fixed point singleword - arithmetic register	7C	r,i[,x]	7-67
MI	Multiply immediate, fixed point singleword - base register	78	r,i[,x]	7-67
MI	Multiply immediate, fixed point singleword - index or vector parameter register	7A	r,i[,x]	7-67
MIH	Multiply immediate, fixed point halfword - arithmetic register	7D	r,i[,x]	7-68
MOD	Modify stack parameter doubleword	9F	r,[@]n[,x]	7-143
NFH	Normalize fixed point halfword	AD	r,[@]n[,x]	7-156
NFX	Normalize fixed point singleword	AC	r,[@]n[,x]	7-155
NOP	Take next instruction, Assembler supplies R field of zero	91	[@[=]]n[,x]	7-132
OR	OR, singleword - arithmetic register	E4	r,[@][=]n[,x]	7-79
ORD	OR, doubleword - arithmetic register	E5	r,[@][=]n[,x]	7-80
ORI	OR immediate, singleword - arithmetic register	F4	r,i[,x]	7-81
PB	Prepare to Branch	9E	---	7-159.1
PSH	Push word into last-in-first-out stack	93	r,[@]n[,x]	7-141
PUL	Pull word from last-in-first-out stack	97	r,[@]n[,x]	7-142
RVS	Bit reversal, singleword - arithmetic register	C6	r,i[,x]	7-102
S	Subtract from arithmetic register, fixed	48	r,[@][=]n[,x]	7-53

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
SA	Arithmetic shift, fixed point singleword - arithmetic register	C0	r,i[,x]	7-93
SAD	Arithmetic shift, fixed point doubleword - arithmetic register	C3	r,i[,x]	7-95
SAH	Arithmetic shift, fixed point halfword - arithmetic register	C1	r,i[,x]	7-94
SC	Circular shift, singleword - arithmetic register	CC	r,i[,x]	7-99
SCD	Circular shift, doubleword - arithmetic register	CF	r,i[,x]	7-101
SCH	Circular shift, halfword - arithmetic register	CD	r,i[,x]	7-100
SCLK	Store clock	AE	---	7-167.2
SF	Subtract from arithmetic register, floating point singleword	4A	r,[@][=]n[,x]	7-55
SFD	Subtract from arithmetic register, floating point doubleword	4B	r,[@][=]n[,x]	7-56
SH	Subtract from arithmetic register, fixed point halfword	49	r,[@][=]n[,x]	7-54
SI	Subtract immediate from arithmetic register, fixed point singleword	58	r,i[,x]	7-57
SIH	Subtract immediate from arithmetic register, fixed point halfword	59	r,i[,x]	7-58
SL	Logical shift, singleword - arithmetic register	C4	r,i[,x]	7-96
SLD	Logical shift, doubleword - arithmetic register	C7	r,i[,x]	7-98
SLH	Logical shift, halfword - arithmetic register	C5	r,i[,x]	7-97
SM	Subtract magnitude from arithmetic register, fixed point singleword	4C	r,[@][=]n[,x]	7-59
SMF	Subtract magnitude from arithmetic register, floating point singleword	4E	r,[@][=]n[,x]	7-61
SMFD	Subtract magnitude from arithmetic register, floating point doubleword	4F	r,[@][=]n[,x]	7-62
SMH	Subtract magnitude from arithmetic register, fixed point halfword	4D	r,[@][=]n[,x]	7-60
SPS	Store program status word	22	[@]n[,x]	7-168
ST	Store arithmetic register, singleword	24	r,[@]n[,x]	7-26
ST	Store base register, singleword	28	r,[@]n[,x]	7-26
ST	Store index register or vector parameter register, singleword	2C	r,[@]n[,x]	7-26

ASSMB CODE	INSTRUCTION	MCHN CODE	OPERAND FORMAT	TOPIC
STD	Store arithmetic register, doubleword	27	r,[@]n[,x]	7-30
STF	Store base register file A, M=0	2B	m,[@]n[,x]	7-39
STF	Store base register file B, M=1	2B	m,[@]n[,x]	7-39
STF	Store arithmetic register file C, M=2	2B	m,[@]n[,x]	7-39
STF	Store arithmetic register file D, M=3	2B	m,[@]n[,x]	7-39
STF	Store index register file X, M=4	2B	m,[@]n[,x]	7-39
STF	Store vector parameter register file V, M=5	2B	m,[@]n[,x]	7-39
STFM	Store all six eight word register files	2F	[@]n[,x]	7-41
STLL	Store arithmetic left halfword into memory left halfword, indexed	25	r,[@]n[,x]	7-27
STLR	Store arithmetic register left halfword into memory right halfword, indexed	29	r,[@]n[,x]	7-29
STN	Store negative, fixed point word	34	r,[@]n[,x]	7-34
STND	Store negative, floating point doubleword	37	r,[@]n[,x]	7-37
STNF	Store negative, floating point word	36	r,[@]n[,x]	7-36
STNH	Store negative, fixed point halfword	35	r,[@]n[,x]	7-35
STO	Store ones complement, word	2E	r,[@]n[,x]	7-38
STOH	Store ones complement, halfword	2A	r,[@]n[,x]	7-39
STRL	Store arithmetic right halfword into memory left halfword, indexed	26	r,[@]n[,x]	7-28.1
STRR	Store arithmetic register right halfword into memory right halfword, indexed	2D	r,[@]n[,x]	7-28
STZ	Store zero, word	20	[@]n[,x]	7-31
STZD	Store zero, doubleword	23	[@]n[,x]	7-33
STZH	Store zero, halfword	21	[@]n[,x]	7-32
VECT	Execute vector parameter file, Assembler supplies R field of one	B0	[@]n[,x]	8-5
VECTL	Load and execute vector parameter file, Assembler supplies R field of zero	B0	[@]n[,x]	8-4
XCH	Exchange - arithmetic register with effective address	1A	r,[@]n[,x]	7-158
XEC	Execute addressed instruction in line	96	[@[=]]n[,x]	7-161
XOR	Exclusive OR, singleword - arithmetic register	E8	r,[@][=]n[,x]	7-82
XORD	Exclusive OR, doubleword - arithmetic register	E9	r,[@][=]n[,x]	7-83
XORI	Exclusive OR immediate, singleword - arithmetic register	F8	r,i[,x]	7-84

APPENDIX C: SCALAR INSTRUCTIONS IN NUMERIC ORDER BY MACHN CODE

MCHN CODE	ASSMB CODE	INSTRUCTION	OPERAND FORMAT	TOPIC
10	LRL	Load memory left halfword, indexed, into arithmetic register right halfword	r,[@][=]n[,x]	7-5.1
11	LEM	Load arithmetic exception mask and condition	[@[=]]n[,x]	7-167.1
12	LAM	Load arithmetic mask	[@[=]]n[,x]	7-166
13	LAC	Load arithmetic exception condition	[@[=]]n[,x]	7-167
14	L	Load arithmetic register, singleword	r,[@][=]n[,x]	7-3
15	LLL	Load memory left halfword, indexed, into arithmetic register left halfword	r,[@][=]n[,x]	7-4
16	LLA	Load look ahead	i	7-159
17	LD	Load arithmetic register, doubleword	r,[@][=]n[,x]	7-7
18	L	Load base register, singleword	r,[@][=]n[,x]	7-3
19	LLR	Load memory right halfword, indexed, into arithmetic register left halfword	r,[@][=]n[,x]	7-6
1A	XCH	Exchange - arithmetic register with effective address	r,[@]n[,x]	7-158
1B	LF	Load base register file A, M=0	m,[@]n[,x]	7-23
1B	LF	Load base register file B, M=1	m,[@]n[,x]	7-23
1B	LF	Load arithmetic register file C, M=2	m,[@]n[,x]	7-23
1B	LF	Load arithmetic register file D, M=3	m,[@]n[,x]	7-23
1B	LF	Load index register file X, M=4	m,[@]n[,x]	7-23
1B	LF	Load vector parameter register file V, M=5	m,[@]n[,x]	7-23
1C	L	Load index register or vector parameter register, singleword	r,[@][=]n[,x]	7-3
1D	LRR	Load memory right halfword, indexed, into arithmetic register right halfword	r,[@][=]n[,x]	7-5
IE	LO	Load arithmetic register with ones complement, singleword	r,[@][=]n[,x]	7-22
1F	LFM	Load all six eight-word register files	[@]n[,x]	7-24
20	STZ	Store zero, word	[@]n[,x]	7-31
21	STZH	Store zero, halfword	[@]n[,x]	7-32
22	SPS	Store program status word	[@]n[,x]	7-168
23	STZD	Store zero, doubleword	[@]n[,x]	7-33
24	ST	Store arithmetic register, singleword	r,[@]n[,x]	7-26
25	STLL	Store arithmetic register left halfword into memory left halfword	r,[@]n[,x]	7-27
26	STRL	Store arithmetic register right halfword into memory left halfword	r,[@]n[,x]	7-28.1
27	STD	Store arithmetic register, doubleword	r,[@]n[,x]	7-30
28	ST	Store base register, singleword	r,[@]n[,x]	7-26

MCHN CODE	ASSMB CODE	INSTRUCTION	OPERAND FORMAT	TOPIC
29	STLR	Store arithmetic register left halfword into memory right halfword, indexed	r,[@]n[,x]	7-29
2A	STOH	Store ones complement, halfword	r,[@]n[,x]	7-39
2B	STF	Store base register file A, M=0	m,[@]n[,x]	7-40
2B	STF	Store base register file B, M=1	m,[@]n[,x]	7-40
2B	STF	Store arithmetic register file C, M=2	m,[@]n[,x]	7-40
2B	STF	Store arithmetic register file D, M=3	m,[@]n[,x]	7-40
2B	STF	Store index register file X, M=4	m,[@]n[,x]	7-40
2B	STF	Store vector parameter register file V, M=5	m,[@]n[,x]	7-40
2C	ST	Store index register or vector parameter register	r,[@]n[,x]	7-26
2D	STRR	Store arithmetic register right halfword into memory right halfword, indexed	r,[@]n[,x]	7-28
2E	STO	Store ones complement, word	r,[@]n[,x]	7-38
2F	STFM	Store all six eight-word register files	[@]n[,x]	7-41
30	LN	Load negative, fixed point single word, arithmetic register	r,[@][=]n[,x]	7-10
31	LNH	Load negative, fixed point halfword, arithmetic register	r,[@][=]n[,x]	7-11
32	LNF	Load negative, floating point singleword, arithmetic register	r,[@][=]n[,x]	7-12
33	LND	Load negative, floating point doubleword, arithmetic register	r,[@][=]n[,x]	7-13
34	STN	Store negative, fixed point word	r,[@]n[,x]	7-34
35	STNH	Store negative, fixed point halfword	r,[@]n[,x]	7-35
36	STNF	Store negative, floating point word	r,[@]n[,x]	7-36
37	STND	Store negative, floating point doubleword	r,[@]n[,x]	7-37
38	LNM	Load negative magnitude, fixed point single-word, arithmetic register	r,[@][=]n[,x]	7-18
39	LNMH	Load negative magnitude, fixed point half-word, arithmetic register	r,[@][=]n[,x]	7-18
3A	LNMF	Load negative magnitude, floating point singleword, arithmetic register	r,[@][=]n[,x]	7-20
3B	LNMD	Load negative magnitude, floating point doubleword, arithmetic register	r,[@][=]n[,x]	7-21
3C	LM	Load magnitude, fixed point singleword, arithmetic register	r,[@][=]n[,x]	7-14
3D	LMH	Load magnitude, fixed point halfword, arithmetic register	r,[@][=]n[,x]	7-15

MCHN CODE	ASSMB CODE	INSTRUCTION	OPERAND FORMAT	TOPIC
3E	LMF	Load magnitude, floating point singleword, arithmetic register	r,[@][=]n[,x]	7-16
3F	LMD	Load magnitude, floating point doubleword, arithmetic register	r,[@][=]n[,x]	7-17
40	A	Add to arithmetic register, fixed point singleword	r,[@][=]n[,x]	7-43
41	AH	Add to arithmetic register, fixed point halfword	r,[@][=]n[,x]	7-44
42	AF	Add to arithmetic register, floating point singleword	r,[@][=]n[,x]	7-45
43	AFD	Add to arithmetic register, floating point doubleword	r,[@][=]n[,x]	7-46
44	AM	Add magnitude to arithmetic register, fixed point singleword	r,[@][=]n[,x]	7-49
45	AMH	Add magnitude to arithmetic register, fixed point halfword	r,[@][=]n[,x]	7-50
46	AMF	Add magnitude to arithmetic register, floating point singleword	r,[@][=]n[,x]	7-51
47	AMFD	Add magnitude to arithmetic register, floating point doubleword	r,[@][=]n[,x]	7-52
48	S	Subtract from arithmetic register, fixed point singleword	r,[@][=]n[,x]	7-53
49	SH	Subtract from arithmetic register, fixed point halfword	r,[@][=]n[,x]	7-54
4A	SF	Subtract from arithmetic register, floating point singleword	r,[@][=]n[,x]	7-55
4B	SFD	Subtract from arithmetic register, floating point doubleword	r,[@][=]n[,x]	7-56
4C	SM	Subtract magnitude from arithmetic register, fixed point singleword	r,[@][=]n[,x]	7-59
4D	SMH	Subtract magnitude from arithmetic register, fixed point halfword	r,[@][=]n[,x]	7-60
4E	SMF	Subtract magnitude from arithmetic register, floating point singleword	r,[@][=]n[,x]	7-61
4F	SMFD	Subtract magnitude from arithmetic register, floating point doubleword	r,[@][=]n[,x]	7-62
50	AI	Add immediate to arithmetic register, fixed point singleword	r,i[,x]	7-47
51	AIH	Add immediate to arithmetic register, fixed point halfword	r,i[,x]	7-48

MCHN CODE	ASSMB CODE	INSTRUCTION	OPERAND FORMAT	TOPIC
52	LEA	Load effective address into base register	r,[@][=]n[,x]	7-160
54	LI	Load immediate into arithmetic register singleword	r,i[,x]	7-8
55	LIH	Load immediate into arithmetic register, halfword	r,i[,x]	7-9
56	LEA	Load effective address into index or vector register	r,[@][=]n[,x]	7-160
58	SI	Subtract immediate from arithmetic register, fixed point singleword	r,i[,x]	7-57
59	SIH	Subtract immediate from arithmetic register, fixed point halfword	r,i[,x]	7-58
5C	LI	Load immediate into index register, or vector parameter register, singleword	r,i[,x]	7-8
60	A	Add to base register, fixed point singleword	r,[@][=]n[,x]	7-43
62	A	Add to index or vector parameter register, fixed point singleword	r,[@][=]n[,x]	7-43
64	D	Divide into arithmetic register, fixed point singleword	r,[@][=]n[,x]	7-69
65	DH	Divide into arithmetic register, fixed point halfword	r,[@][=]n[,x]	7-70
66	DF	Divide into arithmetic register, floating point singleword	r,[@][=]n[,x]	7-71
67	DFD	Divide into arithmetic register, floating point doubleword	r,[@][=]n[,x]	7-72
68	M	Multiply, fixed point singleword - base register	r,[@][=]n[,x]	7-63
6A	M	Multiply, fixed point singleword - index or vector parameter register	r,[@][=]n[,x]	7-63
6C	M	Multiply, fixed point singleword - arithmetic register	r,[@][=]n[,x]	7-63
6D	MH	Multiply, fixed point halfword - arithmetic register	r,[@][=]n[,x]	7-64
6E	MF	Multiply, floating point singleword - arithmetic register	r,[@][=]n[,x]	7-65
6F	MFD	Multiply, floating point doubleword - arithmetic register	r,[@][=]n[,x]	7-66
70	AI	Add immediate to base register, fixed point singleword	r,i[,x]	7-47

MCHN CODE	ASSMB CODE	INSTRUCTION	OPERAND FORMAT	TOPIC
72	AI	Add immediate to index or vector parameter register, fixed point singleword	r,i[,x]	7-47
74	DI	Divide immediate into arithmetic register, fixed point singleword	r,i[,x]	7-73
75	DIH	Divide immediate into arithmetic register, fixed point halfword	r,i[,x]	7-74
78	MI	Multiply immediate, fixed point singleword - base register	r,i[,x]	7-67
7A	MI	Multiply immediate, fixed point singleword - index or vector parameter register	r,i[,x]	7-67
7C	MI	Multiply immediate, fixed point singleword - arithmetic register	r,i[,x]	7-67
7D	MIH	Multiply immediate, fixed point halfword - arithmetic register	r,i[,x]	7-68
8D	ISE	Increment, test and skip on equal	r,[@][=]n[,x]	7-117
81	ISNE	Increment, test and skip on not equal	r,[@][=]n[,x]	7-118
82	DSE	Decrement, test and skip on equal	r,[@][=]n[,x]	7-119
83	DSNE	Decrement, test and skip on not equal	r,[@][=]n[,x]	7-120
84	BCLE	Branch on arithmetic register less than or equal	r,r,n	7-128
85	BCG	Branch on arithmetic register greater than	r,r,n	7-129
86	BCLE	Branch on index less than or equal	r,r,n	7-128
87	BCG	Branch on index greater than	r,r,n	7-129
88	IBZ	Increment, test and branch on zero	r,[@[=]]n[,x]	7-122
89	IBNZ	Increment, test and branch on not zero	r,[@[=]]n[,x]	7-123
8A	DBZ	Decrement, test and branch on zero	r,[@[=]]n[,x]	7-124
8B	DBNZ	Decrement, test and branch on not zero	r,[@[=]]n[,x]	7-125
8C	IBZ	Increment, test and branch on zero	r,[@[=]]n[,x]	7-122
8D	IBNZ	Increment index or vector register, and branch on not zero	r,[@[=]]n[,x]	7-123
8E	DBZ	Decrement index or vector register, test, and branch on zero	r,[@[=]]n[,x]	7-124
8F	DBNZ	Decrement index or vector register, test, and branch on not zero	r,[@[=]]n[,x]	7-125
90	MCP	Monitor call and proceed	i,[,x]	7-163
91	BCC	Branch on compare code true	m,[@[=]]n[,x]	7-132
91	NOP	Take next instruction, Assembler supplies R field of zero	[@[=]]n[,x]	7-132

MCHN CODE	ASSMB CODE	INSTRUCTION	OPERAND FORMAT	TOPIC
91	BE	Branch on compare code of equal, Assembler supplies R field of one	[@[=]]n[,x]	7-132
91	BG	Branch on compare code of greater than, Assembler supplies R field of 2	[@[=]]n[,x]	7-132
91	BGE	Branch on compare code of greater than or equal, Assembler supplies R field of 3	[@[=]]n[,x]	7-132
91	BL	Branch on compare code of less than, Assembler supplies R field of 4	[@[=]]n[,x]	7-132
91	BLE	Branch on compare code of less than or equal, Assembler supplies R field of 5	[@[=]]n[,x]	7-132
91	BNE	Branch on compare code of not equal, Assembler supplies R field of 6	[@[=]]n[,x]	7-132
91	B	Unconditional branch, Assembler supplies R field of 7	[@[=]]n[,x]	7-132
91	BCZ	Branch on compare code of all bits are zero, Assembler supplies R field of one	[@[=]]n[,x]	7-132
91	BCO	Branch on compare code of all bits are one, Assembler supplies R field of 2	[@[=]]n[,x]	7-132
91	BCNM	Branch on compare code of not mixed, Assembler supplies R field of 3	[@[=]]n[,x]	7-132
91	BCM	Branch on compare code of mixed zeros and ones, Assembler supplies R field of 4	[@[=]]n[,x]	7-132
91	BCNO	Branch on compare code of not all ones, Assembler supplies R field of 5	[@[=]]n[,x]	7-132
91	BCNZ	Branch on compare code of not all zeros, Assembler supplies the R field of 6	[@[=]]n[,x]	7-132
92	INT	Interpret - arithmetic register	r,[@][=]n[,x]	7-162
93	PSH	Push word into last-in-first-out stack	r,[@]n[,x]	7-141
94	MCW	Monitor call and wait	i[,x]	7-164
95	BRC	Branch on result code true	m,[@[=]]n[,x]	7-133
95	BZ	Branch on result code of zero, Assembler supplies the R field of one	[@[=]]n[,x]	7-133
95	BPL	Branch on result code of positive, Assembler supplies the R field of 2	[@[=]]n[,x]	7-133
95	BZP	Branch on result code of zero or positive, Assembler supplies the R field of 3	[@[=]]n[,x]	7-133
95	BMI	Branch on result code or negative, Assembler supplies the R field of 4	[@[=]]n[,x]	7-133
95	BZM	Branch on result code of zero or negative, Assembler supplies the R field of 5	[@[=]]n[,x]	7-133

MCHN CODE	ASSMB CODE	INSTRUCTION	OPERAND FORMAT	TOPIC
95	BNZ	Branch on result code of not zero, Assembler supplies the R field of 6	[@[=]]n[,x]	7-133
95	BLR	Branch on logical result	m[@[=]]n[,x]	7-133
95	BRZ	Branch on result code of all bits zero zero, Assembler supplies the R field of one	[@[=]]n[,x]	7-133
95	BRO	Branch on result code of all bits are one, Assembler supplies the R field of 2	[@[=]]n[,x]	7-133
95	BRNM	Branch on result code of bits not mixed zeros and ones, Assembler supplies the R field of 3	[@[=]]n[,x]	7-133
95	BRM	Branch on result code of bits mixed zeros and ones, Assembler supplies the R field of 4	[@[=]]n[,x]	7-133
95	BRNO	Branch on result code of not all bits ones, Assembler supplies the R field of 5	[@[=]]n[,x]	7-133
95	BRNZ	Branch on result code of not all bits zeros, Assembler supplies the R field of 6	[@[=]]n[,x]	7-133
96	XEC	Execute addressed instruction in line	[@[=]]n[,x]	7-161
97	PUL	Pull word from last-in-first-out stack	r,[@]n[,x]	7-142
98	BLB	Branch and load base register with program counter	r,[@[=]]n[,x]	7-137
99	BLX	Branch and load index or vector register with program counter	r,[@[=]]n[,x]	7-138
9A	FORK	Fork	---	7-162.1
9B	JOIN	Join	---	7-162.2
9C	BXEC	Branch on Execute branch condition true, Assembler supplies R field of one or odd	[@]n[,x]	7-135
9D	BAE	Branch on arithmetic exception condition true	m,[@[=]]n[,x]	7-134
9D	BU	Branch on floating point exponent underflow, Assembler supplies R field of one	[@[=]]n[,x]	7-134
9D	BO	Branch on floating point exponent overflow, Assembler supplies R field of 2	[@[=]]n[,x]	7-134
9D	BUO	Branch on floating point exponent underflow or overflow, Assembler supplies R field of 3	[@[=]]n[,x]	7-134
9D	BX	Branch on fixed point overflow, Assembler supplies R field of 4	[@[=]]n[,x]	7-134
9D	BXU	Branch on fixed point overflow or floating point exponent underflow, Assembler supplies R field of 5	[@[=]]n[,x]	7-134
9D	BXO	Branch on fixed point overflow or floating point exponent overflow, Assembler supplies R field of 6	[@[=]]n[,x]	7-134
9D	BXUO	Branch on fixed point overflow or floating point exponent overflow or underflow, Assembler supplies R field of 7	[@[=]]n[,x]	7-134

MCHN CODE	ASSMB CODE	INSTRUCTION	OPERAND FORMAT	TOPIC
9D	BD	Branch on divide check, Assembler supplies R field of 8	[@[=]]n[,x]	7-134
9D	BDU	Branch on divide check or floating point exponent underflow, Assembler supplies R field of 9	[@[=]]n[,x]	7-134
9D	BDO	Branch on divide check or floating point exponent overflow, Assembler supplies R field of A	[@[=]]n[,x]	7-134
9D	BDUO	Branch on divide check or floating point exponent overflow or underflow, Assembler supplies R field of B	[@[=]]n[,x]	7-134
9D	BDX	Branch on divide check or fixed point overflow, Assembler supplies R field of C	[@[=]]n[,x]	7-134
9D	BDXU	Branch on divide check or fixed point overflow or floating point exponent underflow, Assembler supplies R field of D	[@[=]]n[,x]	7-134
9D	BDXO	Branch on divide check or fixed point overflow or floating point exponent overflow, Assembler supplies R field of E	[@[=]]n[,x]	7-134
9D	BDXUO	Branch on divide check or fixed point overflow or floating point exponent overflow or underflow, Assembler supplies R field of F	[@[=]]n[,x]	7-134
9E	PB	Prepare to branch	---	7-159.1
9F	MOD	Modify stack parameter doubleword	r,[@]=[,x]	7-143
A0	FLFX	Convert floating point singleword to fixed point singleword	r,[@]n[,x]	7-148
A1	FLFH	Convert floating point singleword to fixed point halfword	r,[@]n[,x]	7-149
A2	FDFX	Convert floating point doubleword to fixed point singleword	r,[@]n[,x]	7-150
A8	FXFL	Convert fixed point singleword to floating point singleword	r,[@]n[,x]	7-151
A9	FHFL	Convert fixed point halfword to floating point singleword	r,[@]n[,x]	7-152
AA	FXFD	Convert fixed point singleword to floating point doubleword	r,[@]n[,x]	7-153
AB	FHFD	Convert fixed point halfword to floating point doubleword	r,[@]n[,x]	7-154
AC	NFX	Normalize fixed point singleword	r,[@]n[,x]	7-155
AD	NFH	Normalize fixed point halfword	r,[@]n[,x]	7-156
AE	SCLK	Store clock	---	7-167.2
B0	VECTL	Load and execute vector parameter file, Assembler supplies R field of zero	[@]n[,x]	8-4

MCHN CODE	ASSMB CODE	INSTRUCTION	OPERAND FORMAT	TOPIC
B0	VECT	Execute vector parameter file, Assembler supplies R field of one	[@]n[,x]	8-5
C0	SA	Arithmetic shift, fixed point singleword - arithmetic register	r,i[,x]	7-93
C1	SAH	Arithmetic shift, fixed point halfword - arithmetic register	r,i[,x]	7-94
C3	SAD	Arithmetic shift, fixed point doubleword - arithmetic register	r,i[,x]	7-95
C4	SL	Logical shift, singleword - arithmetic register	r,i[,x]	7-96
C5	SLH	Logical shift, halfword - arithmetic register	r,i[,x]	7-97
C6	RVS	Bit reversal, singleword - arithmetic register	r,i[,x]	7-102
C7	SLD	Logical shift, doubleword - arithmetic register	r,i[,x]	7-98
C8	C	Compare arithmetic register, fixed point singleword	r,[@][=]n[,x]	7-104
C9	CH	Compare arithmetic register, fixed point halfword	r,[@][=]n[,x]	7-105
CA	CF	Compare arithmetic register, floating point singleword	r,[@][=]n[,x]	7-106
CB	CED	Compare arithmetic register, floating point doubleword	r,[@][=]n[,x]	7-107
CC	SC	Circular shift, singleword - arithmetic register	r,i[,x]	7-99
CD	SCH	Circular shift, halfword - arithmetic register	r,i[,x]	7-100
CE	C	Compare index or vector register, fixed point singleword	r,[@][=]n[,x]	7-104
CF	SCD	Circular shift, doubleword - arithmetic register	r,i[,x]	7-101
D8	CI	Compare immediate arithmetic register, fixed point singleword	r,i[,x]	7-108
D9	CIH	Compare arithmetic register immediate, fixed point halfword	r,i[,x]	7-109
DE	CI	Compare index or vector register with immediate, singleword	r,i[,x]	7-108
E0	AND	AND, singleword - arithmetic register	r,[@][=]n[,x]	7-76
E1	ANDD	AND, doubleword - arithmetic register	r,[@][=]n[,x]	7-77
E2	CAND	Compare logical AND, singleword - arithmetic register	r,[@][=]n[,x]	7-110

MCHN CODE	ASSMB CODE	INSTRUCTION	OPERAND FORMAT	TOPIC
E3	CANDD	Compare logical AND, doubleword - arithmetic register	r,[@][=]n[,x]	7-111
E4	OR	OR, singleword - arithmetic register	r,[@][=]n[,x]	7-79
E5	ORD	OR, doubleword - arithmetic register	r,[@][=]n[,x]	7-80
E6	COR	Compare logical OR, singleword - arithmetic register	r,[@][=]n[,x]	7-113
E7	CORD	Compare logical OR, doubleword - arithmetic register	r,[@][=]n[,x]	7-114
E8	XOR	Exclusive OR, singleword - arithmetic register	r,[@][=]n[,x]	7-82
E9	XORD	Exclusive OR, doubleword - arithmetic register	r,[@][=]n[,x]	7-83
EC	EQC	Equivalence, singleword - arithmetic register	r,[@][=]n[,x]	7-85
ED	EQCD	Equivalence, doubleword - arithmetic register	r,[@][=]n[,x]	7-86
F0	ANDI	AND immediate, singleword - arithmetic register	r,i[,x]	7-78
F2	CANDI	Compare immediate logical AND, singleword - arithmetic register	r,i[,x]	7-112
F4	ORI	OR immediate, singleword - arithmetic register	r,i[,x]	7-81
F6	CORI	Compare immediate logical OR, singleword - arithmetic register	r,i[,x]	7-115
F8	XORI	Exclusive OR immediate, singleword - arithmetic register	r,i[,x]	7-84
FC	EQCI	Equivalence immediate, singleword - arithmetic register	r,i[,x]	7-87

APPENDIX D: VECTOR INSTRUCTIONS BY LOGICAL GROUPING

ASSMB CODE	MCHN CODE	INSTRUCTION	TOPIC
VA	40	Vector add, fixed point singleword elements	8-26
VAH	41	Vector add, fixed point halfword elements	8-26
VAF	42	Vector add, floating point singleword	8-26
VAFD	43	Vector add, floating point doubleword	8-26
VAM	44	Vector add magnitude, fixed point singleword	8-27
VAMH	45	Vector add magnitude, fixed point halfword	8-27
VAMF	46	Vector add magnitude, floating point singleword	8-27
VAMFD	47	Vector add magnitude, floating point doubleword	8-27
VS	48	Vector subtract, fixed point singleword	8-28
VSH	49	Vector subtract, fixed point halfword	8-28
VSF	4A	Vector subtract, floating point singleword	8-28
VSFD	4B	Vector subtract, floating point doubleword	8-28
VSM	4C	Vector subtract magnitude, fixed point singleword	8-29
VSMH	4D	Vector subtract magnitude, fixed point halfword	8-29
VSMF	4E	Vector subtract magnitude, floating point singleword	8-29
VSMFD	4F	Vector subtract magnitude, floating point doubleword	8-29
VM	6C	Vector multiply, fixed point singleword	8-30
VMH	6D	Vector multiply, fixed point halfword	8-30
VMF	6E	Vector multiply, floating point singleword	8-30
VMFD	6F	Vector multiply, floating point doubleword	8-30
VDP	68	Vector dot product, fixed point singleword	8-31
VDPH	69	Vector dot product, fixed point halfword	8-31
VDPF	6A	Vector dot product, floating point singleword	8-31
VDPFD	6B	Vector dot product, floating point doubleword	8-31
VD	64	Vector divide, fixed point singleword	8-32
VDH	65	Vector divide, fixed point halfword	8-32
VDF	66	Vector divide, floating point singleword	8-32
VDFD	67	Vector divide, floating point doubleword	8-32
VAND	E0	Vector logical AND, singleword	8-33
VOR	E4	Vector logical OR, singleword	8-33
VXOR	E8	Vector logical Exclusive OR, singleword	8-33
VEQC	EC	Vector logical Equivalence, singleword	8-33
VANDD	E1	Vector logical AND, doubleword	8-33

ASSMB CODE	MCHN CODE	INSTRUCTION	TOPIC
VORD	E5	Vector logical OR, doubleword	8-33
VXORD	E9	Vector Exclusive OR, doubleword	8-33
VEQCD	ED	Vector Equivalence, doubleword	8-33
VSA	C0	Vector arithmetic shift, fixed point singleword	8-34
VSAH	C1	Vector arithmetic shift, fixed point halfword	8-34
VSAD	C3	Vector arithmetic shift, fixed point doubleword	8-34
VSL	C4	Vector logical shift, singleword	8-34
VSLH	C5	Vector logical shift, halfword	8-34
VSLD	C7	Vector logical shift, doubleword	8-34
VSC	CC	Vector circular shift, singleword	8-34
VSCH	CD	Vector circular shift, halfword	8-34
VSCD	CF	Vector circular shift, doubleword	8-34
VMG	D8	Vector merge singlewords	8-35
VMGH	D9	Vector merge halfwords	8-35
VMGD	DB	Vector merge doublewords	8-35
VO	D4	Vector order singlewords, fixed point	8-36
VOH	D5	Vector order halfwords, fixed point	8-36
VOF	D6	Vector order singlewords, floating point	8-36
VOFD	D7	Vector order doublewords, floating point	8-36
VC	D0	Vector arithmetic comparison, fixed point singleword	8-38
VCH	D1	Vector arithmetic comparison, fixed point halfword	8-38
VCF	D2	Vector arithmetic comparison, floating point singleword	8-38
VCFD	D3	Vector arithmetic comparison, floating point doubleword	8-38
VCAND	E2	Vector logical comparison using AND, singleword	8-39
VCANDD	E3	Vector logical comparison using AND, doubleword	8-39
VCOR	E6	Vector logical comparison using OR, singleword	8-39
VCORD	E7	Vector logical comparison using OR, doubleword	8-39
VPP	DC	Vector peak, fixed point singleword	8-40
VPPH	DD	Vector peak, fixed point halfword	8-40
VPPF	DE	Vector peak, floating point singleword	8-40
VPPFD	DF	Vector peak, floating point doubleword	8-40
VL	50	Vector search for largest arithmetic element, fixed point singleword	8-42

ASSMB CODE	MCHN CODE	INSTRUCTION	TOPIC
VLH	51	Vector search for largest arithmetic element, fixed point halfword	8-42
VLF	52	Vector search for largest arithmetic element, floating point singleword	8-42
VLFD	53	Vector search for largest arithmetic element, floating point doubleword	8-42
VLM	54	Vector search for largest magnitude, fixed point singleword	8-43
VLMH	55	Vector search for largest magnitude, fixed point halfword	8-43
VLMF	56	Vector search for largest magnitude, floating point singleword	8-43
VLMFD	57	Vector search for largest magnitude, floating point doubleword	8-43
VSS	58	Vector search for smallest arithmetic element, fixed point singleword	8-44
VSSH	59	Vector search for smallest arithmetic element, fixed point halfword	8-44
VSSF	5A	Vector search for smallest arithmetic element, floating point singleword	8-44
VSSFD	5B	Vector search for smallest arithmetic element, floating point doubleword	8-44
VSSM	5C	Vector search for smallest magnitude, fixed point singleword	8-45
VSSMH	5D	Vector search for smallest magnitude, fixed point halfword	8-45
VSSMF	5E	Vector search for smallest magnitude, floating point singleword	8-45
VSSMFD	5F	Vector search for smallest magnitude, floating point doubleword	8-45
VFLFX	A0	Vector convert floating point singleword to fixed point singleword elements	8-47
VFLFH	A1	Vector convert floating point singleword to fixed point halfword elements	8-47
VFDFX	A2	Vector convert floating point doubleword to fixed point singleword elements	8-47
VFXFL	A8	Vector convert fixed point singleword to floating point singleword elements	8-48
VFXFD	AA	Vector convert fixed point singleword to floating point doubleword elements	8-48
VFHFL	A9	Vector convert fixed point halfword to floating point singleword elements	8-48
VFHFD	AB	Vector convert fixed point halfword to floating point doubleword elements	8-48
VNFX	AC	Vector normalize fixed point singleword	8-49
VNFH	AD	Vector normalize fixed point halfword	8-49

ASSMB CODE	MCHN CODE	INSTRUCTION	TOPIC
VMAP	F8	Vector map singleword	8-50
VMAPH	F9	Vector map halfword	8-50
VMAPD	FB	Vector map doubleword	8-50
VSEL	B0	Select singlewords from vector B	8-57
VSELH	B1	Select halfwords from vector B	8-57
VSELD	B3	Select doublewords from vector B	8-57
VSELB	B4	Vector select singleword boolean	8-51
VSELHB	B5	Vector select halfword boolean	8-51
VSELDB	B7	Vector select doubleword boolean	8-51
VREP	B8	Replace singlewords in vector C	8-58
VREPH	B9	Replace halfwords in vector C	8-58
VREPD	BB	Replace doublewords in vector C	8-58
VREPB	BC	Vector replace singleword boolean	8-52
VREPHB	BD	Vector replace halfword boolean	8-52
VREPDB	BF	Vector replace doubleword boolean	8-52
VMAPB	FC	Vector map singleword boolean	8-52
VMAPHB	FD	Vector map halfword boolean	8-53
VMAPDB	FF	Vector map doubleword boolean	8-53
VMAX	F4	Vector maximum/minimum fixed point singleword	8-54
VMAXH	F5	Vector maximum/minimum fixed point halfword	8-54
VMAXF	F6	Vector maximum/minimum floating point singleword	8-54
VMAXFD	F7	Vector maximum/minimum floating point doubleword	8-54
VCB	F0	Vector compare fixed point singleword boolean	8-55
VCHB	F1	Vector compare fixed point halfword boolean	8-55
VCFB	F2	Vector compare floating point singleword boolean	8-55
VCFDB	F3	Vector compare floating point doubleword boolean	8-55
VCAB	EA	Vector compare AND singleword boolean	8-56
VCADB	EB	Vector compare AND doubleword boolean	8-56
VCORB	EE	Vector compare OR singleword boolean	8-56
VCORDB	EF	Vector compare OR doubleword boolean	8-56

APPENDIX E: VECTOR INSTRUCTIONS IN ALPHABETICAL ORDER BY ASSEMBLER CODE

ASSMB CODE	MCHN CODE	INSTRUCTION	TOPIC
VA	40	Vector add, fixed point singleword	8-26
VAF	42	Vector add, floating point singleword	8-26
VAFD	43	Vector add, floating point doubleword	8-26
VAH	41	Vector add, fixed point halfword	8-26
VAM	44	Vector add magnitude, fixed point singleword	8-27
VAMF	46	Vector add magnitude, floating point singleword	8-27
VAMFD	47	Vector add magnitude, floating point doubleword	8-27
VAMH	45	Vector add magnitude, fixed point singleword	8-27
VAND	E0	Vector logical AND, singleword	8-32
VANDD	E1	Vector logical AND, doubleword	8-32
VC	D0	Vector arithmetic comparison, fixed point singleword	8-38
VCAB	EA	Vector compare AND singleword boolean	8-56
VCADB	EB	Vector compare AND doubleword boolean	8-56
VCAND	E2	Vector logical comparison using AND, singleword	8-39
VCANDD	E3	Vector logical comparison using AND, doubleword	8-39
VCB	F0	Vector compare fixed point singleword boolean	8-55
VCF	D2	Vector arithmetic comparison, floating point singleword	8-38
VCFB	F2	Vector compare floating point singleword boolean	8-55
VCFD	D3	Vector arithmetic comparison, floating point doubleword	8-38
VCFDB	F3	Vector compare floating point doubleword boolean	8-55
VCH	D1	Vector arithmetic comparison, fixed point halfword	8-38
VCHB	F1	Vector compare fixed point halfword boolean	8-55
VCOR	E6	Vector logical comparison using OR, singleword	8-39
VCORB	EE	Vector compare OR singleword boolean	8-56
VCORD	E7	Vector logical comparison using OR, doubleword	8-39
VCORDB	EF	Vector compare OR doubleword boolean	8-56
VD	64	Vector divide fixed point, singleword	8-32
VDF	66	Vector divide floating point, singleword	8-32
VDFD	67	Vector divide floating point, doubleword	8-32
VDH	65	Vector divide fixed point, halfword	8-32
VDP	68	Vector dot product, fixed point singleword	8-31
VDPF	6A	Vector dot product, floating point singleword	8-31

ASSMB CODE	MCHN CODE	INSTRUCTION	TOPIC
VDPFD	6B	Vector dot product, floating point doubleword	8-31
VDPH	69	Vector dot product, fixed point halfword	8-31
VEQC	EC	Vector logical Equivalence, singleword	8-33
VEQCD	ED	Vector logical Equivalence, doubleword	8-33
VFDFX	A2	Vector convert floating point doubleword to fixed point singleword	8-47
VFHFD	AB	Vector convert fixed point halfword to floating point doubleword	8-49
VFHFL	A9	Vector convert fixed point half length to floating point singleword	8-49
VFLFH	A1	Vector convert floating point singleword to fixed point halfword	8-47
VFLFX	A0	Vector convert floating point singleword to fixed point singleword	8-47
VFXFD	AA	Vector convert fixed point singleword to fixed point doubleword	8-48
VFXFL	A8	Vector convert fixed point singleword to floating point singleword	8-48
VL	50	Vector search for largest arithmetic element, fixed point singleword	8-42
VLF	52	Vector search for largest arithmetic element, floating point singleword	8-42
VLFD	53	Vector search for largest arithmetic element, floating point doubleword	8-42
VLH	51	Vector search for largest arithmetic element, fixed point halfword	8-42
VLM	54	Vector search for largest magnitude, fixed point singleword	8-43
VLMF	56	Vector search for largest magnitude, floating point singleword	8-43
VLMFD	57	Vector search for largest magnitude, floating point doubleword	8-43
VLMH	55	Vector search for largest magnitude, fixed point halfword	8-43
VM	6C	Vector multiply, fixed point singleword	8-30
VMAP	F8	Vector map singleword	8-48
VMAPB	FC	Vector map singleword boolean	8-53
VMAPD	FB	Vector map doubleword	8-48
VMAPDB	FF	Vector map doubleword boolean	8-53
VMAPH	F9	Vector Map Halfword	8-48
VMAPHB	FD	Vector map halfword boolean	8-53
VMAX	F4	Vector maximum/minimum fixed point singleword	8-54
VMAXF	F6	Vector maximum/minimum floating point singleword	8-54
VMAXFD	F7	Vector maximum/minimum floating point doubleword	8-54

ASSMB CODE	MCHN CODE	INSTRUCTION	TOPIC
VMAXH	F5	Vector maximum/minimum fixed point halfword	8-54
VMF	6E	Vector multiply, floating point singleword	8-30
VMFD	6F	Vector multiply, floating point doubleword	8-30
VMG	D8	Vector merge singlewords	8-35
VMGD	DB	Vector merge doublewords	8-35
VMGH	D9	Vector merge halfwords	8-35
VMH	6D	Vector multiply, fixed point halfword	8-30
VNFH	AD	Vector normalize fixed point halfword	8-49
VNFX	AC	Vector normalize fixed point singleword	8-49
VO	D4	Vector order singlewords, fixed point	8-36
VOF	D6	Vector order singlewords, floating point	8-36
VOFD	D7	Vector order doublewords, floating point	8-36
VOH	D5	Vector order halfwords, fixed point	8-36
VOR	E4	Vector logical OR, singleword	8-33
VORD	E5	Vector logical OR, doubleword	8-33
VPP	DC	Vector peak, fixed point singleword	8-40
VPPF	DE	Vector peak, floating point singleword	8-40
VPPFD	DF	Vector peak, floating point doubleword	8-40
VPPH	DD	Vector peak, fixed point halfword	8-40
VREP	B8	Replace singlewords in vector $\vec{C}$	8-58
VREPB	BC	Vector replace singleword boolean	8-52
VREPD	BB	Replace doublewords in vector $\vec{C}$	8-58
VREPDB	BF	Vector replace doubleword boolean	8-52
VREPH	B9	Replace halfwords in vector $\vec{C}$	8-58
VREPHB	BD	Vector replace halfword boolean	8-52
VS	48	Vector subtract, fixed point singleword	8-28
VSA	C0	Vector arithmetic shift, fixed point singleword	8-34
VSAD	C3	Vector arithmetic shift, fixed point doubleword	8-34
VSAH	C1	Vector arithmetic shift, fixed point halfword	8-34
VSC	CC	Vector circular shift, singleword	8-34
VSCD	CF	Vector circular shift, doubleword	8-34
VSCH	CD	Vector circular shift, halfword	8-34

ASSMB CODE	MCHN CODE	INSTRUCTION	TOPIC
VSEL	B0	Select singlewords from vector $\vec{B}$	8-57
VSELB	B4	Vector select singleword boolean	8-51
VSELH	B1	Select halfwords from vector $\vec{B}$	8-57
VSELHB	B5	Vector select halfword boolean	8-51
VSELD	B3	Select doublewords from vector $\vec{B}$	8-57
VSELDB	B7	Vector select doubleword boolean	8-51
VSF	4A	Vector subtract, floating point singleword	8-28
VSFD	4B	Vector subtract, floating point doubleword	8-28
VSH	49	Vector subtract, fixed point halfword	8-28
VSL	C4	Vector logical shift, singleword	8-34
VSLD	C7	Vector logical shift, doubleword	8-34
VSLH	C5	Vector logical shift, halfword	8-34
VSM	4C	Vector subtract magnitude, fixed point singleword	8-29
VSMF	4E	Vector subtract magnitude, floating point singleword	8-29
VSMFD	4F	Vector subtract magnitude, floating point doubleword	8-29
VSMH	4D	Vector subtract magnitude, fixed point halfword	8-29
VSS	58	Vector search for smallest arithmetic element, fixed point singleword	8-44
VSSF	5A	Vector search for smallest arithmetic element, floating point singleword	8-44
VSSFD	5B	Vector search for smallest arithmetic element, floating point doubleword	8-44
VSSH	59	Vector search for smallest arithmetic element, fixed point halfword	8-44
VSSM	5C	Vector search for smallest magnitude, fixed point singleword	8-45
VSSMF	5E	Vector search for smallest magnitude, floating point singleword	8-45
VSSNFD	5F	Vector search for smallest magnitude, floating point doubleword	8-45
VSSMH	5D	Vector search for smallest magnitude, fixed point halfword	8-45
VXOR	E8	Vector logical Exclusive OR, singleword	8-33
VXORD	E9	Vector logical Exclusive OR, doubleword	8-33

APPENDIX F: VECTOR INSTRUCTIONS IN NUMERIC ORDER BY MACHINE CODE

MCHN CODE	ASSMB CODE	INSTRUCTION	TOPIC
40	VA	Vector add, fixed point singleword	8-26
41	VAH	Vector add, fixed point halfword	8-26
42	VAF	Vector add, floating point singleword	8-26
43	VAFD	Vector add, floating point doubleword	8-26
44	VAM	Vector add magnitude, fixed point singleword	8-27
45	VAMH	Vector add magnitude, fixed point halfword	8-27
46	VAMF	Vector add magnitude, floating point singleword	8-27
47	VAMFD	Vector add magnitude, floating point doubleword	8-27
48	VS	Vector subtract, fixed point singleword	8-28
49	VSH	Vector subtract, fixed point halfword	8-28
4A	VSF	Vector subtract, floating point singleword	8-28
4B	VSFD	Vector subtract, floating point doubleword	8-28
4C	VSM	Vector subtract magnitude, fixed point singleword	8-29
4D	VSMH	Vector subtract magnitude, fixed point halfword	8-29
4E	VSMF	Vector subtract magnitude, floating point singleword	8-29
4F	VSMFD	Vector subtract magnitude, floating point doubleword	8-29
50	VL	Vector search for largest arithmetic element, fixed point singleword	8-42
51	VLH	Vector search for largest arithmetic element, fixed point halfword	8-42
52	VLF	Vector search for largest arithmetic element, floating point singleword	8-42
53	VLFD	Vector search for largest arithmetic element, floating point doubleword	8-42
54	VLM	Vector search for largest magnitude, fixed point singleword	8-43
55	VLMH	Vector search for largest magnitude, fixed point halfword	8-43
56	VLMF	Vector search for largest magnitude, floating point singleword	8-43
57	VLMFD	Vector search for largest magnitude, floating point doubleword	8-43
58	VSS	Vector search for smallest arithmetic element, fixed point singleword	8-44
59	VSSH	Vector search for smallest arithmetic element, fixed point halfword	8-44
5A	VSSF	Vector search for smallest arithmetic element, floating point singleword	8-44
5B	VSSFD	Vector search for smallest arithmetic element, floating point doubleword	8-44

MCHN CODE	ASSMB CODE	INSTRUCTION	TOPIC
5C	VSSM	Vector search for smallest magnitude, fixed point singleword	8-45
5D	VSSMH	Vector search for smallest magnitude, fixed point halfword	8-45
5E	VSSMF	Vector search for smallest magnitude, floating point singleword	8-45
5F	VSSMFD	Vector search for smallest magnitude, floating point doubleword	8-45
64	VD	Vector divide, fixed point singleword	8-32
65	VDH	Vector divide, fixed point halfword	8-32
66	VDE	Vector divide, floating point singleword	8-32
67	VDFD	Vector divide, floating point doubleword	8-32
68	VDP	Vector dot product, fixed point singleword	8-31
69	VDPH	Vector dot product, fixed point halfword	8-31
6A	VDPF	Vector dot product, floating point singleword	8-31
6B	VDPFD	Vector dot product, floating point doubleword	8-31
6C	VM	Vector multiply, fixed point singleword	8-30
6D	VMH	Vector multiply, fixed point halfword	8-30
6E	VMF	Vector multiply, floating point singleword	8-30
6F	VMFD	Vector multiply, floating point doubleword	8-30
A0	VFLFX	Vector convert floating point singleword to fixed point halfword	8-47
A1	VELFH	Vector convert floating point singleword to fixed point halfword elements	8-47
A2	VFDFX	Vector convert floating point doubleword to fixed point singleword	8-47
A8	VFXFL	Vector convert fixed point singleword to floating point singleword	8-48
A9	VFHFL	Vector convert fixed point halfword to floating point singleword	8-48
AA	VFXFD	Vector convert fixed point singleword to floating point doubleword	8-48
AB	VFHFD	Vector convert fixed point halfword to floating point doubleword	8-48
AC	VNFX	Vector normalize, fixed point singleword	8-49
AD	VNFH	Vector normalize, fixed point halfword	8-49
B0	VSEL	Select singlewords from vector $\vec{B}$	8-57
B1	VSELH	Select halfwords from vector $\vec{B}$	8-57
B3	VSELD	Select doublewords from vector $\vec{B}$	8-57
B4	VSELB	Vector select singleword boolean	8-51
B5	VSELHB	Vector select halfword boolean	8-51
B7	VSELDB	Vector select doubleword boolean	8-51
B8	VREP	Replace singlewords in vector $\vec{C}$	8-58
B9	VREPH	Replace halfwords in vector $\vec{C}$	8-58

MCHN CODE	ASSMB CODE	INSTRUCTION	TOPIC
BB	VREPD	Replace doublewords in vector $\vec{C}$	8-58
BC	VREPB	Vector replace singleword boolean	8-52
BD	VREPHB	Vector replace halfword boolean	8-52
BF	VREPDB	Vector replace doubleword boolean	8-52
C0	VSA	Vector arithmetic shift, fixed point singleword	8-34
C1	VSAH	Vector arithmetic shift, fixed point halfword	8-34
C3	VSAD	Vector arithmetic shift, fixed point doubleword	8-34
C4	VSL	Vector logical shift, singleword	8-34
C5	VSLH	Vector logical shift, halfword	8-34
C7	VSLD	Vector logical shift, doubleword	8-34
CC	VSC	Vector circular shift, singleword	8-34
CD	VSCH	Vector circular shift, halfword	8-34
CF	VSCD	Vector circular shift, doubleword	8-34
D0	VC	Vector arithmetic comparison, fixed point singleword	8-38
D1	VCH	Vector arithmetic comparison, fixed point halfword	8-38
D2	VCF	Vector arithmetic comparison, floating point singleword	8-38
D3	VCFD	Vector arithmetic comparison, floating point doubleword	8-38
D4	VO	Vector order singlewords, fixed point	8-36
D5	VOH	Vector order halfwords, fixed point	8-36
D6	VOF	Vector order singlewords, floating point	8-36
D7	VOFD	Vector order doublewords, floating point	8-36
D8	VMG	Vector merge singlewords	8-35
D9	VMGH	Vector merge halfwords	8-35
DB	VMGD	Vector merge doublewords	8-35
DC	VPP	Vector peak, fixed point singleword	8-40
DD	VPPH	Vector peak, fixed point halfword	8-40
DE	VPPF	Vector peak, floating point singleword	8-40
DF	VPPFD	Vector peak, floating point doubleword	8-40
E0	VAND	Vector logical AND, singleword	8-33
E1	VANDD	Vector logical AND, doubleword	8-33
E2	VCAND	Vector logical comparison using AND, singleword	8-39
E3	VCANDD	Vector logical comparison using AND, doubleword	8-39
E4	VOR	Vector logical OR, singleword	8-33
E5	VORD	Vector logical OR, doubleword	8-33

MCHN CODE	ASSMB CODE	INSTRUCTION	TOPIC
E6	VCOR	Vector logical comparison using OR, singleword	8-39
E7	VCORD	Vector logical comparison using OR, doubleword	8-39
E8	VXOR	Vector logical Exclusive OR, singleword	8-33
E9	VXORD	Vector logical Exclusive OR, doubleword	8-33
EA	VCAB	Vector compare AND singleword boolean	8-56
EB	VCADB	Vector compare AND doubleword boolean	8-56
EC	VEQC	Vector logical Equivalence, singleword	8-33
ED	VEQCD	Vector logical Equivalence, doubleword	8-33
EE	VCORB	Vector compare OR singleword boolean	8-56
EF	VCORDB	Vector compare OR doubleword boolean	8-56
F0	VCB	Vector compare fixed point singleword boolean	8-55
F1	VCHB	Vector compare fixed point halfword boolean	8-55
F2	VCFB	Vector compare floating point singleword boolean	8-55
F3	VCFDB	Vector compare floating point doubleword boolean	8-55
F4	VMAX	Vector maximum/minimum fixed point singleword	8-54
F5	VMAXH	Vector maximum/minimum fixed point halfword	8-54
F6	VMAXF	Vector maximum/minimum floating point singleword	8-54
F7	VMAXFD	Vector maximum/minimum floating point doubleword	8-54
F8	VMAP	Vector map singleword	8-50
F9	VMAPH	Vector map halfword	8-50
FB	VMAPD	Vector map doubleword	8-50
FC	VMAPB	Vector map singleword boolean	8-53
FD	VMAPHB	Vector map halfword boolean	8-53
FF	VMAPDB	Vector map doubleword boolean	8-53

APPENDIX G: SCALAR INSTRUCTION TIME REQUIREMENTS

INSTRUCTION		CLOCK TIMES
LOAD	L	1
	LI	1
	LH	1
	LIH	1
	LRL	1
	LRR	1
	LLL	1
	LLR	1
	LD	1
	LM	2
	LMH	2
	LMF	1
	LMD	1
	LN	2
	LNH	2
	LNF	1
	LND	1
	LNM	2
	LNMH	2
	LNMF	1
	LNMD	1
	LF	1
	LFM	*
	XCH	2
	LAM	1
	LAE	1
	LEM	1
	LLA	1
	LO	1
STORE	ST	1
	STH	1
	STRL	1
	STRR	1
	STLL	1
	STLR	1
	STD	1
	SPS	1
	SCLK	1
	STZ	1
	STZH	1
	STZD	1
	STN	2
	STNH	2
	STNF	1
	STND	1
	STO	1
	STOH	1
	STF	1
	STFM	*
ARITHMETIC	A	2
	AI	2
	AH	2
	AIH	2
	AF	5
	AFD	5
	AM	2
	AMH	2
	AMF	5
	AMFD	5
	S	2
	SI	2
	SH	2
	SIH	2
	SF	5
	SFD	5
	SM	2
	SMH	2
	SMF	5
	SMFD	5
	M	3
	MI	3
	MH	3
	MIH	3
	MF	4
	MFD	6

*Determined by memory access time.

INSTRUCTION		CLOCK TIMES	INSTRUCTION		CLOCK TIMES
ARITHMETIC (Continued)	D	16		CORI	1
	DI	16		CANDD	1
	DH	16		CORD	1
	DIH	16			
	DF	16			
	DFD	16			
LOGICAL	AND	1	TEST & BRANCH	BCC	-
	ANDI	1		BRC	-
	OR	1		BXEC	-
	ORI	1		BAE	-
	XOR	1		PB	-
	XORI	1	REG MOD & TESTING	IBZ	2
	EQC	1		IBNZ	2
	EQCI	1		DBZ	2
	ANDD	1		DBNZ	2
	ORD	1		ISE	3
	XORD	1		ISNE	3
	EQCD	1		DSE	3
				DSNE	3
SHIFT	SA	3		BCLE	2
	SAH	3		BCG	2
	SAD	3			
	SL	3	STACK	PSH	3
	SLH	3		PUL	3
	SLD	3		MOD	2
	SC	3	SUB-ROUTINE	BLB	1
	SCH	3		BLX	1
	SCD	3			
	RVS	6	ANALYZE	LEA	1
				INT	1
ARITH COMPARE	C	2		XEC	-
	CI	2	CONVER-SION	FLFX	5
	CH	2		FLFH	5
	CIH	2		FDFX	5
	CF	2		FXFL	4
	CFD	2		FXFD	4
				FHFL	4
LOGICAL COMPARE	CAND	1		FHFD	4
	CANDI	1			
	COR	1			

INSTRUCTION		CLOCK TIMES	INSTRUCTION		CLOCK TIMES
NORMALIZE	NFX	3	VECTOR	VECT	(See Vector Timing)
	NFH	3			
				VECTL	(See Vector Timing)
CALL	MCP	1			
	MCW	1			
MISCELLANEOUS					
	FORK	1			
	JOIN	1			

G-1. SCALAR INSTRUCTION TIMING GROUPS

The following lists are of Central Processor instructions grouped according to those which may follow each other without a delay in the Arithmetic Unit.

GROUP 1A			GROUP 1B	
L	LAM	AND	ST	SCLK
LI	LAC	ANDI	STH	SPS
LH	LEM	OR	STRL	SPS
LIH	CAND	ORI	STRR	XCH
LRL	CANDI	XOR	STLL	
LRR	COR	XORI	STLR	
LLL	CORI	EQC	STD	
LLR	CANDD	EQCI	STZ	
LD	CORD	ANDD	STZH	
LMF		ORD	STZD	
LMD		XORD	STNF	
LNF		EQCD	STND	
LND		LEA	STO	
LNMF		INT	STOH	
LNMD			MCP	
LO			MCW	

Note: For instructions in group 1B, multiple store instruction delay occurs when sequential store instructions write into different Central Memory octets.

GROUP 2				
LM	A	S	IBZ	ISE
LMH	AI	SI	IBNZ	ISNE
LN	AH	SH	DBZ	DSE
LNH	AIH	SIH	DBNZ	DSNE
LNM	AM	SM	C	STN
LNMH	AMH	SMH	CI	STNA
			CH	BLB
			CIH	BLX
			CF	
			CD	

GROUP 3	GROUP 4	GROUP 5	GROUP 6	GROUP 7	
M (A)	FXFL	FLFXQ	NFX	AF	SF
MI (A)	FXFD	FLFH	NFH	AFD	SFD
	FHFL	FDFX		AMX	SMF
	FHFD			AMFD	SMFD

Groups 8 and 9 are lists of instructions which cannot immediately follow each other on the next clock into the Arithmetic Unit pipeline.

GROUP 8							GROUP 9
MF	D	DF	SA	SL	SC	RVS	PSH
MFD	DI	DFD	SAH	SLH	SCH	BCLE	PUL
	DH		SAD	SLD	SCD	BCG	MOD
	DIH						XCH
							MCW

Group 10 is a list of instructions which do not use the Arithmetic Unit.

GROUP 10				
LAM	LF	STF	SPS	BCC
LAE	LFM	STFM	XEC	BRC
LLA	FORK	LEM		BAE
PB	JOIN	SCLK		BEC

Note: Any instruction in groups 2 through 8 may immediately follow any instruction in groups 1A or 1B without creating a delay.

APPENDIX H: VECTOR INSTRUCTION TIME REQUIREMENTS

Table H-1. Vector Execution Rates in Clocks/Element

VECTOR INSTRUCTION		RATE	VECTOR INSTRUCTION		RATE	VECTOR INSTRUCTION		RATE
ADD	VA	1	SHIFT	VSA	2	SEARCH	VL	1
	VAH	1		VSAH	2	(Input Rate)	VLH	1
	VAF	1		VSAD	2		VLF	1
	VAFD	2					VLFD	2
				VSL	2			
ADD	VAM	1		VSLH	2		VLM	1
MAGNITUDE	VAMH	1		VSLD	2		VLMH	1
	VAMF	1					VLMF	1
	VAMFD	2		VSC	2		VLMFD	2
				VSCH	2			
				VSCD	2		VSS	1
DOT	VDP	1					VSSH	1
PRODUCT	VDPH	1	MERGE	VMGH	2		VSSF	1
(Input Rate)	VDPF	1	(Input Rate)	VMG	2		VSSFD	2
	VDPFD	4		VMGD	2		VSSM	1
							VSSMH	1
DIVIDE	VD	16	VECTOR	VO	6		VSSMF	1
	VDH	16	ORDER	VOD	6		VSMFD	2
	VDF	16	(Output Rate)	VOF	6			
	VDFD	16		VOFD	6			
						PEAK	VPP	1
SUBTRACT	VS	1	ARITHMETIC	VC	1	PICKING	VPPH	1
	VSH	1	COMPARISON	VCH	1	(Input Rate)	VPPF	1
	VSF	1	(Input Rate)	VCF	1		VPPFD	2
	VSFD	2		VCFD	2			
						CONVERSION	VFLFX	2
SUBTRACT	VSM	1	LOGICAL	VCAND	1		VFLFH	2
MAGNITUDE	VSMH	1	COMPARISON	VCANDD	2		VFDFX	2
	VSMF	1	(Input Rate)	VCOR	1		VFXFL	2
	VSMFD	2		VCORD	2		VFXFD	2
							VFHFL	2
MULTIPLY	VM	1	LOGICAL	VAND	1		VFHFD	2
	VMH	1		VOR	1			
	VMF	1		VXOR	1	NORMALIZE	VNFX	2
	VMFD	3		VEQC	1		VNFH	2
				VANDD	2			
				VORD	2			
				VSORD	2			
				VEQCD	2			

Table H-1. Vector Execution Rates in Clocks/Element (Continued)

VECTOR INSTRUCTION		RATE	VECTOR INSTRUCTION		RATE	VECTOR INSTRUCTION		RATE
ARITHMETIC	VCB	1	SELECT	VSELB	1	MAP	VMAP	3
COMPARE	VCHB	1	BOOLEAN	VSELHB	1		VMAPH	3
BOOLEAN	VCFB	1		VSELDB	1.75		VMAPD	3
(Input Rate)	VCFDB	1.75						
			REPLACE	VREP	3	MAP	VMAPB	1
LOGICAL	VCAB	1		VREPH	3	BOOLEAN	VMAPHB	1
COMPARE	VCADB	1.75		VREPD	3		VMAPDB	1.75
BOOLEAN	VCORB	1						
(Input Rate)	VCORDB	1.75	REPLACE	VREPB	2			
			BOOLEAN	VREPHB	2			
SELECT	VSEL	3		VREPDB	2			
	VSELH	3						
	VSELD	3						

H-1. TIME REQUIREMENTS FOR COMPLETE VECTOR OPERATION

The total time, t for execution of a vector instruction can be computed approximately from the formula:

$$t = P + (R \cdot L \cdot NI \cdot NO)$$

where:

NO = outer loop count

NI = inner loop count

L = vector dimension

R = vector rate in clocks/element

P = approximates 26 + previous scalar time.

The rate, R, is defined as the number of clock times required to obtain each element of the result.

Timing for vector dot products, peak picking, searches, and comparisons are based on element input rate rather than output rate because the Arithmetic Unit outputs of these instructions are infrequent.

TEXAS INSTRUMENTS
INCORPORATED
EQUIPMENT GROUP
AUSTIN, TEXAS

PUBLICATION UPDATE

PUBLICATION

PROGRAM ASC PUBLICATION NO. 930039-2

TITLE Programmer's Guide To The Central Processor

DATE May 1976 JOB NO. _____

TYPE OF CHANGE

☐ IMMEDIATE
(MAY CAUSE PERSONAL INJURY OR EQUIPMENT DAMAGE/FAILURE)

☐ ROUTINE
(BATCH PROCESSED)

SUBMITTED BY

NAME _____ PHONE _____

ADDRESS _____

MAIL STATION _____ DATE _____

LIST PAGE AND PARAGRAPH OR FIGURE NUMBERS AND DESCRIBE RECOMMENDED CHANGES.

FORWARD CHANGES BY FOLDING THIS SHEET AND STAPLING. RETURN ADDRESS IS ON BACK OF SHEET.

TEXAS INSTRUMENTS INCORPORATED
EQUIPMENT GROUP
P.O. BOX 2909
AUSTIN, TEXAS 78767

ATTENTION: TECHNICAL DATA BRANCH
MAIL STATION 2146

TI-22005